Evaluación de algoritmos de planificación sobre un prototipo de sistema multicore asimétrico

Adrian Pousa[1], Juan Carlos Saez[2], Armando De Giusti[1], Manuel Prieto[2]

[1] Instituto de Investigación en Informatica LIDI, UNLP*, Argentina,
{apousa,degiusti}@lidi.info.unlp.edu.ar
[2] Facultad de Informática, Universidad Complutense*, Madrid, Spain,
{jcsaezal,mpmatias}@ucm.es

Resumen Los procesadores multicore asimétricos con repertorio común de instrucciones –AMPs (*Asymmetric Multicore Processors*)– han sido propuestos como alternativa de bajo consumo a los procesadores multicore simétricos convencionales. Los AMPs combinan *cores rápidos* y complejos, de alto rendimiento, con *cores lentos* de consumo reducido. Trabajos previos han propuesto distintos algoritmos de planificación para AMPs, que en su mayoría han sido evaluados empleando plataformas asimétricas emuladas o simuladores. En este artículo se lleva a cabo un análisis experimental de algunas de las estrategias de planificación más relevantes para AMPs sobre el Intel QuickIA, un prototipo de sistema multicore asimétrico. Para el análisis se han realizado implementaciones de estas estrategias en un sistema operativo real y se lleva a cabo una evaluación completa de las mismas empleando cargas de trabajo multiprogramadas. Un aspecto clave de la implementación de los algoritmos es la metodología que se propone para aproximar en tiempo de ejecución el beneficio relativo que una aplicación obtiene al ejecutar en un core rápido con respecto a un core lento.

Keywords: AMP, procesadores multicore asimétricos, planificación, sistema operativo

1. Introducción

Los procesadores multicore asimétricos (AMPs) integran en un mismo chip diversos tipos de cores con distintas características (frecuencia, microarquitectura o consumo), pero con el mismo repertorio de instrucciones. Investigaciones previas han demostrado que los AMPs ofrecen un mayor rendimiento por vatio y unidad de área que los multicores simétricos [9,1]. Además se ha demostrado que la integración de sólo dos tipos de core ("rápidos" y "lentos") en el chip permite una utilización muy eficiente del área del procesador y simplifica considerablemente el diseño [9]. La reciente aparición del procesador big.LITTLE

* Este trabajo se encuentra en el marco del convenio existente entre la UNLP y la UCM, proyecto B/024918/09 – Formación en Computación Avanzada aprobado por la Agencia Española de Cooperación en Investigación y Desarrollo (AECID).

de ARM [2] y del prototipo Quick-IA de Intel [4] demuestra que los principales fabricantes están apostando por este tipo de diseños asimétricos.

A pesar de las ventajas de los AMPs, éstos plantean retos importantes para el sistema operativo. Uno de los más significativos es cómo distribuir eficientemente los cores rápidos entre las aplicaciones que comparten el sistema. La mayor parte de propuestas al respecto persiguen maximizar el rendimiento global o productividad [9,18,13,8]. Para ello, el planificador debe mapear a los cores rápidos aquellas aplicaciones que obtienen un mayor beneficio al ejecutarse en cores rápidos frente a cores lentos [18,8,13]. En este artículo nos referiremos a este beneficio como *speedup* de la aplicación en el AMP.

Otros objetivos relevantes, como proporcionar justicia en el reparto de los cores rápidos o dar soporte a prioridades, han recibido menor atención. Las escasas propuestas que persiguen este objetivo, buscan mejorar la justicia en entornos AMP repartiendo equitativamente los cores rápidos entre las distintas aplicaciones [3,10] o bien intentando que todas las aplicaciones experimenten la misma degradación en rendimiento (*slowdown*) al compartir el sistema, con respecto a cuando ejecutan solas en el mismo.

La mayor parte de las estrategias de planificación propuestas para AMPs han sido evaluadas sobre plataformas asimétricas emuladas (p.ej., cores con distintas frecuencias) o empleando simuladores. En este artículo se lleva a cabo un análisis de la justicia y el rendimiento que ofrecen algunos de los algoritmos de planificación más relevantes para AMPs sobre el Intel QuickIA [4]. Uno de los grandes desafíos al realizar una implementación de estos algoritmos en un sistema operativo real, como la realizada en este trabajo, es diseñar un mecanismo para estimar el speedup en tiempo de ejecución. En este artículo se propone una metodología sistemática para obtener modelos de estimación del speedup.

El resto del artículo se organiza del siguiente modo. La sección 2 describe las métricas utilizadas para el análisis experimental. La sección 3 describe los planificadores estudiados. En la sección 4 se describe el proceso de generación del modelo de estimación del speedup. En la sección 5 se muestran y analizan los resultados experimentales obtenidos sobre el sistema asimétrico real. Finalmente, la sección 6 presenta el trabajo relacionado y la sección 7 expone las conclusiones derivadas de nuestro trabajo.

2. Métricas

Para cuantificar la productividad optamos por emplear el Speedup Agregado (*aggregate speedup*) que se define del siguiente modo:

$$\textit{Speedup Agregado} = \sum_{i=1}^{n} \left(\frac{CT_{slow,i}}{CT_{sched,i}} - 1 \right) \tag{1}$$

donde n es el número de aplicaciones que contiene la carga de trabajo, $CT_{slow,i}$ es el tiempo de ejecución (o de retorno) de la aplicación i cuando se ejecuta sola

en el sistema y sólo usa cores lentos, y $CT_{sched,i}$ es el tiempo de ejecución de la aplicación i bajo un planificador determinado.

Por otra parte, consideramos que un planificador es completamente justo, si el incremento en el tiempo de ejecución o degradación (*slowdown*) que experimenta una aplicación al compartir el sistema con otras –con respecto a la situación en la que la aplicación se ejecuta sola en el sistema– es el mismo para todas las aplicaciones de la misma prioridad [11,5][1]. Para evaluar el comportamiento de una estrategia de planificación en cuanto a justicia emplearemos la Injusticia (*unfairness*), métrica propuesta en [11,5] y definida como sigue:

$$Injusticia = \frac{MAX(Slowdown_1, ..., Slowdown_n)}{MIN(Slowdown_1, ..., Slowdown_n)} \quad (2)$$

donde $Slowdown_i = CT_{sched,i}/CT_{fast,i}$, y $CT_{fast,i}$ es el tiempo de ejecución de la aplicación i cuando se ejecuta sola en el sistema (todos los cores rápidos están disponibles para ella en todo momento).

3. Algoritmos de Planificación para AMPs evaluados

En este trabajo se lleva a cabo un análisis experimental de algunos de los algoritmos de planificación más relevantes para AMPs, como son: RR (Round Robin), Prop-SP (Proportional-SPeedup), HSP (High-SPeedup) y A-DWRR. Para el análisis hemos realizado una implementación de estos algoritmos en el kernel Linux.

RR [3,13] reparte de forma equitativa los cores rápidos entre todas las aplicaciones de la carga de trabajo sin conocer sus speedups. Investigaciones previas han demostrado que este algoritmo tiene un comportamiento más determinista y ofrece un rendimiento superior en AMPs al de los planificadores por defecto de los sistemas operativos actuales [14]. Por este motivo, RR ha sido utilizado ampliamente como *baseline* para comparación con otros algoritmos.

Prop-SP [17] es un algoritmo diseñado para garantizar un equilibrio entre la productividad extraída del AMP y la justicia. Para ello, Prop-SP asigna a cada aplicación una fracción de tiempo de core rápido que es proporcional a su *peso dinámico*. El peso dinámico de una aplicación se define como el producto de su peso estático y su speedup neto (speedup menos uno). El peso estático se deriva de la prioridad de la aplicación establecida por el usuario. El speedup se aproxima en tiempo de ejecución por el planificador, que emplea modelos de estimación (sin interacción del usuario). Nos referiremos a esta implementación del algoritmo con el nombre de *Prop-SP (dynamic)*. Por completitud, se compara la efectividad de este algoritmo, sujeto a fallos de predicción, con una versión estática del mismo –Prop-SP (static)–, que usa speedups medios medidos antes

[1] Debido a la existencia de recursos compartidos en un sistema multicore (niveles de cache, cores de distinto tipo, ...), esta definición de justicia resulta más apropiada para multicore que las definiciones clásicas de justicia que se basan únicamente en el tiempo de CPU otorgado a las aplicaciones [11,5].

de la ejecución (ratio de los tiempos de ejecución de la aplicación en el core lento y en el rápido).

HSP (High-SPeedup) [13,8] intenta maximizar el rendimiento del sistema mapeando a cores rápidos aquellas aplicaciones que obtienen el mayor beneficio de éstos. Obviamente, este algoritmo es inherentemente injusto, dado que no hace ningún esfuerzo por repartir los cores rápidos entre las aplicaciones. Al igual que con Prop-SP, se estudian dos variantes del algoritmo: *HSP (static)*, donde se alimenta al planificador con speedups medios para cada aplicación obtenidos antes de la ejecución, y *HSP (dynamic)*, donde el speedup se estima en tiempo de ejecución.

A-DWRR [10] persigue proporcionar justicia en un sistema asimétrico. El algoritmo emplea un concepto especial de tiempo de CPU, extendido para AMPs, conocido como tiempo de CPU escalado. Usando este concepto, los ciclos de CPU consumidos por una aplicación en un core rápido tienen mayor peso que los consumidos en un core lento. Para asegurar justicia, A-DWRR intenta equiparar el tiempo de CPU escalado consumido entre los hilos teniendo también en cuenta sus prioridades. Es importante mencionar que A-DWRR asume un ratio de rendimiento fast-slow constante, a la hora de computar el tiempo de CPU escalado de cada hilo y no el speedup real de cada aplicación.

Cabe destacar, que todas las implementaciones que hemos realizado de estos algoritmos de planificación presentan algunos aspectos comunes. En primer lugar todas ellas mantienen los cores rápidos ocupados para maximizar su utilización. Por otro lado, para realizar las asignaciones de hilos a cores, todos los algoritmos desencadenan intercambios de hilos (*swaps*) entre distintos tipos de cores. Por último, mencionar que todos los algoritmos se han implementado como clases de planificación independientes en el kernel Linux.

4. Estimación del speedup

En esta sección, detallamos el proceso de generación de un modelo de estimación para determinar el speedup de una aplicación secuencial sobre el prototipo QuickIA de Intel [4]. Este prototipo es un sistema de *socket* dual que integra dos procesadores multicore diferentes: un Intel Atom N330 –dos cores a 1.6Ghz– y un Intel Xeon E5450 –cuatro cores a 1.2Ghz–. Para minimizar el efecto de la contención por recursos compartidos en el análisis experimental, se ha optado por desactivar dos cores en el Intel Xeon, para que ningún core comparta el último nivel de cache con otro. Por lo tanto, se dispone de una configuración multicore asimétrica con dos cores "rápidos" de alto rendimiento (Xeon) y dos cores "lentos" de bajo consumo (Atom).

Nuestro objetivo es obtener un modelo de rendimiento para que el planificador pueda aproximar el speedup de una aplicación secuencial utilizando los contadores hardware del core donde la aplicación está actualmente ejecutandose. Para una aplicación secuencial, el speedup se conoce también como *Speedup Factor* (SF) y se define como $\frac{IPS_{fast}}{IPS_{slow}}$, donde IPS_{fast} e IPS_{slow} son los ratios de

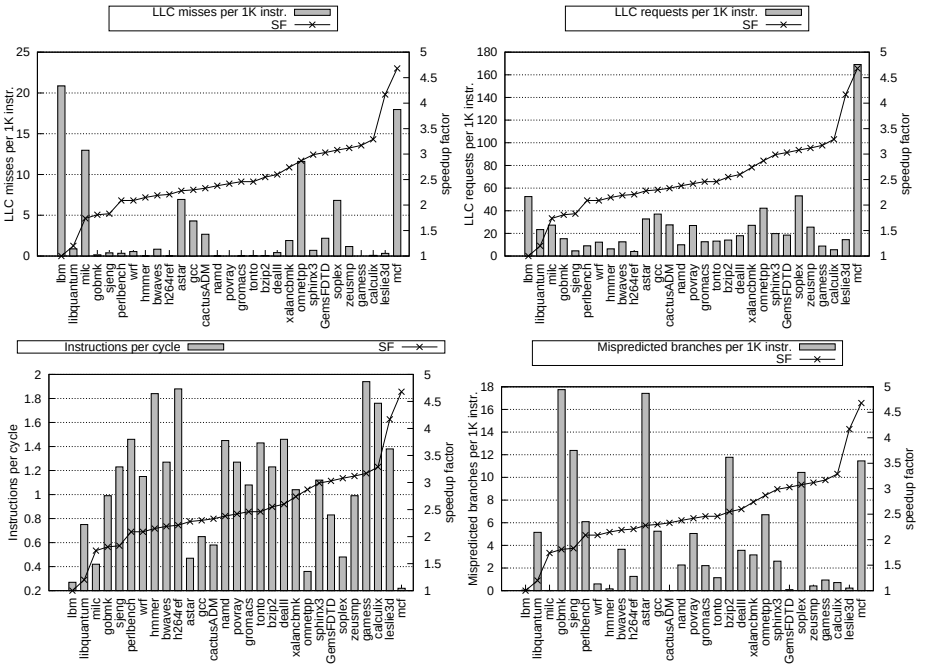
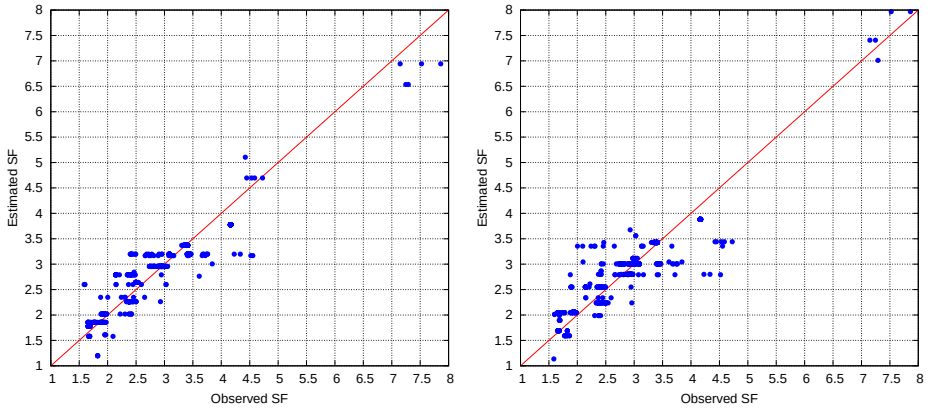


Figura 1: Relación entre el SF y algunas métricas sobre el Intel Xeon.

instrucciones por segundo obtenidos por la aplicación en un core rápido y en un core lento respectivamente.

Trabajos previos han intentado estimar el SF buscando correlaciones entre el SF y distintas métricas de rendimiento que pueden capturarse usando los contadores hardware del procesador. En particular, se ha demostrado que cuando los cores del sistema difieren únicamente en frecuencia, el SF guarda una correlación negativa con la tasa de fallos de último nivel de cache [13,8]. De manera similar Koufaty y otros [8] detectaron que en sistemas donde los cores presentan distinta tasa de retirada de instrucciones, existen dos factores que exhiben una correlación negativa con el SF [8]. El primer factor es la intensidad en memoria que exhibe la aplicación, que puede aproximarse mediante su tasa de fallos de cache de último nivel. El segundo factor esta relacionado con las paradas producidas en el front-end del pipeline del procesador debido a los fallos de predicción de saltos. Tanto las aplicaciones intensivas en memoria (alta tasa de fallos de cache de último nivel) como aquellas con frecuentes fallos de predicción de saltos suelen experimentar *speedup factors* bajos.

Para ilustrar la complejidad asociada a la estimación del SF en el Intel QuickIA, la figura 1 muestra la relación entre el SF y algunas de las métricas de rendimiento a las que nos referimos anteriormente. A diferencia de lo que ocurre en sistemas asimétricos emulados, en esta plataforma no se observan claras correlaciones entre las métricas y el SF.



(a) Estimación en el core rápido (b) Estimación en el core lento

Figura 2: SFs observados y predichos para un subconjunto de benchmarks de SPEC CPU2006. Cada punto representa una fase de SF de un benchmark. A mayor cercanía de un punto a la diagonal, mejor estimación del SF. Los coeficientes de correlación de las estimaciones en el core rápido y lento son 0.95 y 0.94 respectivamente.

En la búsqueda de metodologías sistemáticas para diseñar modelos de estimación en escenarios complejos, como el que se estudia en este trabajo, exploramos distintas técnicas de minería de datos usando la herramienta WEKA [6]. WEKA proporciona numerosos métodos para inferir relaciones entre un conjunto de observaciones (o atributos de entrada) y una variable objetivo. En el problema de estimación de SFs, las observaciones se corresponden con distintos parámetros de rendimiento y la variable objetivo es el SF. Entre los métodos explorados, detectamos que la regresión aditiva permite derivar un modelo de estimación relativamente preciso para esta plataforma.

La metodología propuesta para la generación del modelo de estimación en la plataforma asimétrica puede resumirse en los siguientes pasos:

(1) Seleccionar un conjunto AP de aplicaciones secuenciales representativas y un conjunto M de métricas de rendimiento.
(2) Ejecutar las aplicaciones de AP en ambos tipos de core para obtener su SF y las métricas de rendimiento en M a lo largo del tiempo usando los contadores hardware[2] del procesador. Para que sea posible la monitorización del SF a lo largo del tiempo, muestreamos el número de instrucciones por ciclo (IPC) en cada tipo de core cada cierta ventana de instrucciones.
(3) Para cada aplicación A en AP dividir su ejecución en fases de SF. Esta tarea es una de las partes más complejas de este proceso y se detalla en trabajo previo [12].

[2] Para desempeñar esta tarea desarrollamos PMCtrack, una herramienta de uso de contadores hardware. Las implementaciones de los planificadores estudiados en este trabajo hacen uso intensivo del API a nivel de kernel de PMCTrack, que permite al planificador monitorizar el rendimiento de las aplicaciones en tiempo de ejecución.

(4) Para cada una de las fases de SF obtenidas en el paso anterior, calcular las medias geométricas de las muestras de cada fase para cada métrica del conjunto M. Estas medias geométricas constituyen un "resumen" de cada fase de SF.
(5) Los resultados del paso anterior son proporcionados a WEKA, que genera dos modelos de estimación de SF basados en regresión aditiva: uno para estimar el SF desde el core rápido y otro desde el core lento. Es importante destacar que, al generar los modelos, WEKA descarta automáticamente aquellas métricas de M menos relevantes para el modelo.

La figura 2 muestra la comparación entre el SF observado y el estimado en ambos tipos de cores para un subconjunto de aplicaciones de SPEC CPU2006 en el Intel QuickIA. Los modelos generados requieren que el planificador monitorice las siguientes métricas en tiempo de ejecución: instrucciones por ciclo, fallos y accesos al último nivel de cache por cada 1K instrucciones, fallos de predicción de saltos por cada 1K instrucciones y fallos de ITLB/DTLB por cada 1M instrucciones. Dado el escaso número de contadores hardware en la plataforma ha sido necesario recurrir a la multiplexación de eventos en la implementación.

5. Experimentos

En esta sección se analiza la productividad y justicia de los algoritmos de planificación descritos en la sección 3 sobre el prototipo QuickIA de Intel.

Para realizar un estudio exhaustivo de las distintas políticas de planificación construimos diversas cargas de trabajo multiprogramadas, constituidas por aplicaciones secuenciales de la suite SPEC CPU2006. Para la elección de cargas de trabajo representativas clasificamos las distintas aplicaciones teniendo en cuenta su SF – H (*high*), M (*medium*) o L (*low*)-. La primera columna del cuadro 1 muestra la composición de las cargas seleccionadas. Por ejemplo, la carga de trabajo 3H-1M está compuesta por tres aplicaciones secuenciales con SF alto (`calculix`, `GemsFDTD` y `bzip2`) y una con SF medio (`h264ref`).

La figura 3 muestra los resultados para las cargas de trabajo exploradas. Los resultados ilustran que el planificador HSP obtiene los speedups agregados más altos, pero a costa de degradar la justicia para todas las cargas de trabajo. RR, por el contrario, ofrece los mejores valores de justicia en muchos casos. No obstante, realizar un reparto equitativo de los cores rápidos no siempre garantiza la misma degradación en rendimiento para todas las aplicaciones [16]. El planificador A-DWRR se comporta de forma muy similar a RR porque en el escenario considerado (un hilo por core), garantizar un equilibrio entre los tiempos de CPU escalados se consigue realizando un reparto justo de los cores rápidos entre aplicaciones (tal como lo hace RR). En términos generales, los resultados revelan que los algoritmos RR y A-DWRR no son capaces de obtener speedups agregados tan elevados como en el resto de los algoritmos. Esta degradación en el rendimiento global se debe al hecho de no tener en cuenta la diversidad de *speedups* entre aplicaciones [3,13].

Cuadro 1: Cargas de trabajo multiaplicación constituidas por aplicaciones secuenciales

Categorías	**Benchmarks**
4H	calculix, gamess, GemsFDTD, bzip2
3H-1M	calculix, GemsFDTD, bzip2, h264ref
3H-1L_A	gamess, GemsFDTD, bzip2, sjeng
3H-1L_B	calculix, gamess, sphinx3, sjeng
2H-2M	gamess, soplex, povray, h264ref
2H-2L_A	mcf, calculix, sjeng, gobmk
2H-2L_B	gamess, sphinx3, gobmk, libquantum
1H-1M-2L_A	mcf, h264ref, sjeng, gobmk
1H-1M-2L_B	calculix, gromacs, sjeng, libquantum
2M-2L_A	namd, h264ref, gobmk, libquantum
2M-2L_B	gromacs, povray, sjeng, libquantum

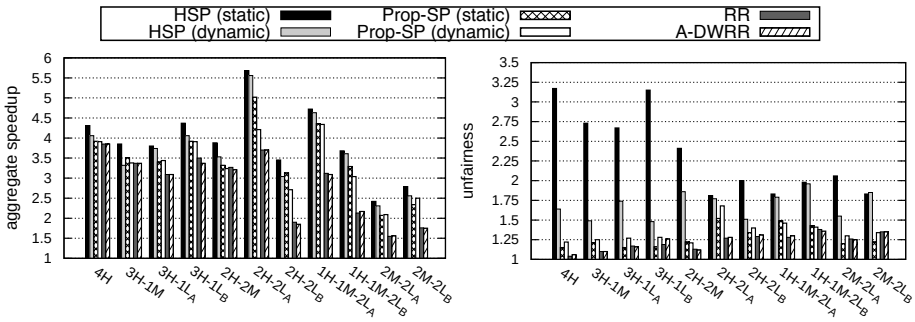


Figura 3: Speedup agregado e injusticia para los distintos algoritmos de planificación.

En general, puede observarse que Prop-SP consigue un buen equilibrio entre productividad (speedup agregado) y justicia. Este algoritmo consigue mayor speedup agregado que RR y A-DWRR para todas las cargas de trabajo. Además para algunas cargas como 3H-1L_A, 3H-1M o 2M-2L_A, Prop-SP ofrece un mejor compromiso productividad-justicia que estos algoritmos (mayor speedup agregado, para justicia similar). Los resultados también revelan que Prop-SP exhibe un comportamiento mucho más justo que HSP y obtiene valores de productividad más próximos a HSP que el resto de algoritmos.

Por último, centramos nuestra atención en las versiones dinámicas de HSP y Prop-SP. En la mayoría de los casos, la versión dinámica de HSP consigue mejores valores de justicia (menor unfairness) que su versión estática. Esto se debe a que HSP (dinámico) se encarga de asegurar que, en todo momento, los hilos que exhiben las fases de mayor SF se mapean a core rápidos. Como una aplicación puede atravesar distintas fases de SF durante su ciclo de vida, HSP (dinámico) migra aplicaciones entre cores cada cierto tiempo. Indirectamente, esto se traduce en mejor justicia en muchos casos ya que múltiples aplicaciones tienen oportunidad de ejecutarse en un core rápido durante algún tiempo. Por otra parte, hemos observado que en la versión dinámica de Prop-SP los valores de justicia y rendimiento pueden empeorar en presencia de fallos de estimación.

6. Trabajo Relacionado

La mayoría de los planificadores existentes para AMPs persiguen maximizar el rendimiento global de la plataforma. Aquellos algoritmos diseñados para cargas de trabajo que incluyen sólo aplicaciones secuenciales [9,3,8,18,14,19] intentan optimizar el rendimiento ejecutando en los cores rápidos aquellos programas con un mayor SF. Para extraer un mayor rendimiento en cargas de trabajo que incluyen programas multihilo, algunos planificadores hacen uso de los cores rápidos del AMP como aceleradores de las fases secuenciales de las aplicaciones paralelas [1,13].

Muchas de las propuestas de planificación para AMPs se han evaluado utilizando simuladores [3,19,7]. Otros autores, en cambio, han recurrido a implementaciones de sus propuestas en un sistema operativo real pero realizando la evaluación de las mismas sobre plataformas asimétricas emuladas [15,8,10]. En nuestro trabajo se lleva a cabo la evaluación de algoritmos implementados en un sistema operativo real sobre una plataforma asimétrica real. Investigaciones previas en el área han mostrado que las implementaciones en sistemas reales permiten detectar problemas significativos en algoritmos que han sido evaluados previamente mediante simuladores. Este es el caso, por ejemplo, de las conclusiones extraídas por Shelepov y otros [18,14] al llevar a cabo la implementación del algoritmo IPC-driven [3]. Los autores demostraron que este tipo de algoritmos, que requieren el muestreo del IPC en ambos tipos de core [9,3], están sujetos a imprecisiones en el cálculo del SF e introducen una mayor sobrecarga en tiempo de ejecución que las estrategias de planificación que se basan en modelos de estimación de SF, como HSP [8,13] o Prop-SP [17].

7. Conclusiones

En este artículo se ha llevado a cabo un estudio experimental de algunos de los algoritmos de planificación más relevantes para AMPs: RR, HSP, Prop-SP y A-DWRR. Para ello hemos realizado implementaciones de estos algoritmos en el kernel Linux y analizado la productividad y la justicia ofrecida por éstos para distintas cargas de trabajo multiprogramadas sobre el prototipo QuickIA de Intel. Se propone además una metodología para estimar el speedup de una aplicación secuencial en AMPs, que representa el beneficio relativo que la aplicación obtiene al ejecutar en un core rápido con respecto a un core lento.

Los resultados de nuestro estudio revelan que el planificador HSP, que destina los cores rápidos a ejecutar las aplicaciones con mayor speedup, obtiene los mejores resultados en cuanto a rendimiento pero falla a la hora de proporcionar justicia. Por otra parte, RR y A-DWRR se comportan bien en términos de justicia pero degradan el rendimiento global de forma significativa al no tener en cuenta el speedup de las aplicaciones a la hora de planificar. Finalmente, Prop-SP es capaz de hacer un uso eficiente del AMP y ofrecer un mejor compromiso productividad-justicia que el resto de algoritmos para un amplio rango de cargas de trabajo multiprogramadas.

Agradecimientos

El presente trabajo ha sido financiado por el proyecto TIN2012-32180 y la Red Europea de Excelencia HIPEAC[3]. Agracecemos a David Koufaty e Intel Labs el habernos permitido experimentar con el prototipo de sistema asimétrico.

Referencias

1. Annavaram, M., et al.: Mitigating Amdahl's Law through EPI Throttling. In: Proc. of ISCA'05. pp. 298–309 (2005)
2. ARM: Benefits of the big.LITTLE Architecture (2012)
3. Becchi, M., Crowley, P.: Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In: Proc. of CF '06. pp. 29–40 (2006)
4. Chitlur, N., et al.: QuickIA: Exploring heterogeneous architectures on real prototypes. In: in Proc. of HPCA '12. pp. 1 –8 (feb 2012)
5. Ebrahimi, E., et al.: Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. ASPLOS '10 (2010)
6. Hall, M., et al.: The WEKA data mining software: an update. SIGKDD Explor. Newsl. 11, 10–18 (2009)
7. Joao, J.A., et al.: Utility-based acceleration of multithreaded applications on asymmetric CMPs. In: Proc. of ISCA '13. pp. 154–165 (2013)
8. Koufaty, D., Reddy, D., Hahn, S.: Bias Scheduling in Heterogeneous Multi-core Architectures. In: Proc. of Eurosys '10 (2010)
9. Kumar, R., et al.: Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In: Proc. of ISCA '04 (2004)
10. Li, T., et al.: Operating system support for overlapping-ISA heterogeneous multi-core architectures. In: HPCA'10. pp. 1–12 (2010)
11. Mutlu, O., Moscibroda, T.: Stall-time fair memory access scheduling for chip multiprocessors. In: Proc. of MICRO '07 (2007)
12. Pousa, A., Saez, J.C.: Modelo de Estimación de Speedup Factor Mediante Umbralización en Multicores Asimétricos. Tech. rep., III-Lidi, Universidad Nacional de La Plata, Argentina - Facultad de Informática, Universidad Complutense de Madrid, España, `http://www.lidi.info.unlp.edu.ar/techreport/2014_04_01.pdf`
13. Saez, J.C., et al.: A Comprehensive Scheduler for Asymmetric Multicore Systems. In: Proc. of ACM Eurosys '10 (2010)
14. Saez, J.C., et al.: Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. J. Parallel Distrib. Comput. 71, 114–131 (January 2011)
15. Saez, J.C., et al.: Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. ACM TOCS 30(2) (Apr 2012)
16. Saez, J.C., et al.: Delivering fairness and priority enforcement on asymmetric multicore systems via OS scheduling. In: Proceedings of the ACM SIGMETRICS'13. pp. 343–344 (2013)
17. Saez, J.C., et al.: Exploring the throughput-fairness trade-off on asymmetric multicore systems. In: To appear in 2nd Workshop on Runtime and Operating Systems for the Many-core Era (ROME), held in conjunction with Euro-Par 2014 (2014)
18. Shelepov, D., et al.: HASS: a Scheduler for Heterogeneous Multicore Systems. ACM SIGOPS OSR 43(2) (2009)
19. Van Craeynest, K., et al.: Scheduling heterogeneous multi-cores through performance impact estimation (PIE). pp. 213–224. ISCA '12 (2012)