ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
Grado en Ingeniería de Computadores

Ejecución concurrente de aplicaciones con Memoria Transaccional en procesadores heterogéneos.

Concurrent Execution of Transactional Memory applications on heterogeneous processors.

Realizado por

Ernesto Rittwagen Martinez

Tutorizado por

Oscar Plata González

y

Alejandro Villegas Fernández

Departamento

Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA

MÁLAGA, Julio de 2017

Fecha defensa:

El Secretario del Tribunal

Resumen: En la actualidad, para encontrar un buen equilibrio entre rendimiento y consumo energético, los fabricantes están empezando a ofrecer procesadores heterogéneos. Estos presentan 2 tipos diferentes de núcleos: unos están enfocados a la eficiencia energética y otros al rendimiento. Un ejemplo de estos procesadores son los big.LITTLE de ARM. Para obtener un buen provecho de este tipo de procesadores, los programadores deben escribir programas multi-hilo. En este tipo de programas, los mecanismos de exclusión mútua son los encargados de garantizar que, en un instante dado, únicamente uno de los hilos acceda a los datos compartidos para asegurar la consistencia de dichos datos. Aunque de forma tradicional estos problemas se han resuelto con cerrojos, la memoria transaccional (TM por sus siglas en inglés) está cobrando importancia. En un trabajo previo se han caracterizado un conjunto de aplicaciones que utilizan una popular librería de TM por software, midiendo rendimiento y consumo de energía en ambos tipos de núcleos. Una vez caracterizadas las aplicaciones individualmente, es interesante el diseño de un sistema de planificación automático que pueda asignar las aplicaciones a aquellos núcleos en los que se estima un mejor rendimiento o menor consumo energético.

En este trabajo presentamos ScHeTM, un planificador de aplicaciones que utilizan TM sobre procesadores multi-núcleo heterogéneos. Atendiendo a las características de las aplicaciones en cuanto a tiempo de cómputo, consumo de energía, y el número de conflictos en la zona de exclusión mútua, ScHeTM realiza una planificación adecuada con el objetivo de mejorar el rendimiento del sistema. Además, el diseño paremetrizado de ScHeTM permite seleccionar, de las características anteriormente mencionadas, aquellas a optimizar. En el conjunto de aplicaciones estudiado, una parametrización correcta de ScHeTM permite lograr una reducción en el consumo de energía en un 15% y el tiempo de cómputo en un 40%, disminuyendo además el número de transacciones con conflictos.

Palabras clave: Memoria transaccional, Procesadores heterogéneos, Eficiencia energética, Planificación.

Abstract: Currently, hardware vendors are developing heterogeneous processors in order to obtain a balance between performance and energy consumption. These processors integrate two different kind of computing cores: ones are focused on performance, while others are more energy-efficient. One example of these processors are the big.LITTLE design proposed by ARM. In order to use all the hardware available in these processors, applications should be multi-threaded. One of the problems with multi-threading is the need of a mutual exclusion mechanism that ensures that only one of the threads accesses shared data at a given time. This problem is usually solved by using locks, but recently transactional memory (TM) solutions are becoming popular. Prior work analyzed the impact of using software TM on heterogeneous processors, measuring energy consumption and performance on each kind of cores. Now, it is interesting to design a scheduling mechanism that assigns applications to those cores where we can achieve better performance or lower energy consumption.

In this work we present ScHeTM, a scheduler designed for TM applications running on heterogeneous processors. ScHeTM schedules applications on the different cores attending to the expected performance, energy consumptions, and other TM-related characteristics in order to improve the behavior of such applications. In addition, parameters can be given to ScHeTM to select which of the aforementioned characteristics is optimized. Provided a good choice of these parameters, ScHeTM reduces energy consumption in up to 15% and execution time in up to 40% for the evaluated set of applications.

Keywords: Transactional Memory, Heterogeneous Processors, Energy Efficiency, Scheduling.

Índice general

1. Introducción **9**

1.1. Introducción 9

1.1.1. Estructura de la memoria 12

1.2. Objetivos 13

1.3. Antecedentes 13

1.3.1. Arquitectura big.LITTLE de ARM 13

1.3.2. Planificación en procesadores heterogéneos 14

1.4. Estudio del arte 15

1.5. Tecnologías utilizadas 17

1.5.1. TinySTM 17

1.5.2. STAMP 17

1.5.3. Lenguajes de script 19

1.5.4. Latex 20

2. Estudio de ejecución concurrente **21**

2.1. Caracterización de las aplicaciones 21

3. Planificador **25**

3.1. Planificador ScHeTM 25

3.1.1. Visión general 25

3.1.2. Función de Idoneidad 28

3.1.3. Planificación 30

3.1.4. Reparto de carga 30

4. Evaluación **33**

4.1. Metodología experimental 33

4.1.1. Implementación y automatización de las pruebas 33
4.1.2. Configuración del hardware 36
4.1.3. Configuración de los experimentos 37
4.2. Resultados experimentales 37
4.2.1. Planificador ávido 37
4.2.2. Planificador ScHeTM 38
4.2.3. Parametrización de ScHeTM: Tiempo de computo 39
4.2.4. Parametrización de ScHeTM: Transacciones abortadas 41
4.2.5. Parametrización de ScHeTM: Consumo energético 41

5. Conclusiones 43

5.1. Conclusiones de la planificación 43
5.2. Conclusiones del TFG 45
5.3. Trabajo futuro 45

Capítulo 1

Introducción

En este primer capítulo se exponen la motivación de este trabajo de fin de grado, el estado del arte, las diversas tecnologías utilizadas y una breve explicación de como se estructura la memoria.

1.1. Introducción

Un problema que surge de la gran utilización de procesadores multi-núcleo es el consumo energético. Muchos de los dispositivos que utilizan este tipo de procesadores necesitan baterías para funcionar. Por ello, es interesante maximizar su rendimiento, duración y durabilidad. En el caso de los dispositivos móviles también hay que tener en cuenta que la temperatura puede ser un problema teniendo que reducirla lo más posible debido a su dificultad de refrigeración y a que deben usarse de una manera cómoda. A causa de esto, los fabricantes de dispositivos móviles están enfocando el diseño de sus nuevos procesadores a minimizar el consumo energético sin perder de vista el rendimiento, puesto que este es otro factor importante a la hora de diseñar procesadores. Para conseguir el mejor rendimiento y un bajo consumo, los fabricantes han optado por usar procesadores multi-núcleo heterogéneos como el *big.LITTLE* diseñado por *ARM*. La característica principal de estos procesadores es la inclusión de 2 tipos de *clúster*, donde se agrupan sus núcleos: el clúster big y el clúster little. El clúster big está enfocado a aportar mejor rendimiento a las aplicaciones, mientras que el clúster little tiene la función de minimizar el consumo de energía. Los dos clústers comparten la misma arquitectura pudiendo ejecutar el mismo conjunto de aplicaciones sin necesidad de compilar nuevamente los ejecutables. El siste-

ma operativo puede decidir donde se ejecutan las aplicaciones según las restricciones de consumo o la demanda de rendimiento.

Para aprovechar los procesadores multi-núcleo es importante utilizar varios hilos de ejecución. Si estos hilos modifican concurrentemente datos compartidos puede producirse un problema de consistencia: se ha de garantizar que los datos son consistentes, siendo las modificaciones temporales o intermedias por parte de uno de los hilos invisibles para el resto. Este problema se solventa utilizando mecanismos de exclusión mútua. De esta manera nos aseguramos que varios hilos no accedan al mismo dato de forma simultánea. La parte del código que accede a estos datos se denomina sección crítica, donde se debe garantizar la exclusión mútua entre los hilos de ejecución. La implementación de la exclusión mútua se ha realizado tradicionalmente con cerrojos de grano fino o cerrojos de grano grueso. Los cerrojos de grano fino proporcionan un mejor rendimiento ya que se basan en proteger individualmente las posiciones de memoria para que se pueda acceder de forma paralela a los datos críticos, siempre y cuando los demas hilos no quieran acceder a la misma posición de memoria ya bloqueada por otro. Esto incrementa la dificultad en la implementación para el programador, ya que se deben evitar los *deadlocks* y los *livelocks*.

La implementación con cerrojos de grano grueso, por otro lado, evita que varios hilos entren en la sección critica al mismo tiempo. Si es necesario acceder a la sección crítica, aunque sean en posiciones de memoria diferentes, este tipo de cerrojos ocasiona un impacto negativo en el rendimiento puesto que el resto de hilos se quedan esperando a el que está ejecutando la sección crítica. En otras palabras, el acceso a la sección crítica se realiza en serie.

```
void Insert(Object elem, Int pos)
{
  while(!get(Locks[pos])){;}
  //Sección crítica
  GlobalArray[pos] = elem;
  release(Locks[pos]);
}
```

Figura 1.1: Cerrojo de grano fino

En la Figura 1.1 se muestra un ejemplo de código para un cerrojo de grano fino, mientras que en la Figura 1.2 podemos observar el otro tipo de cerrojo mencionado. Como podemos observar, el cerrojo de grano fino protege una única posición de memoria. Si la posición de memoria no está siendo accedida por ningún otro hilo, entonces se pueden

```
void Insert(Object elem, Int pos)
{
  while(!get(GlobalLock)){;}
  //Sección crítica
  GlobalArray[pos] = elem;
  release(GlobalLock);
}
```

Figura 1.2: Cerrojo de grano grueso

modificar los datos. En el caso de un cerrojo de grano grueso, tendríamos una solución válida al problema pero ineficiente.

Si utilizando cerrojos de grano fino se acceden de forma simultánea a varias posiciones de memoria, usando cada uno de sus cerrojos correspondientes, el esfuerzo de programación es mucho mayor ya que pueden aparecer deadlocks. Un ejemplo de deadlock se produce cuando dos hilos A y B quieren acceder a las mismas posiciones de memoria. El hilo A quiere acceder a las direcciones de memoria 0 y 1 en ese orden, mientras que el hilo B quiere acceder a las direcciones de memoria 1 y 0 en este orden. Puede darse el caso de que el hilo A obtenga el cerrojo de la posición 0 mientras que el hilo B obtenga el de la posición de memoria 1. Cuando el hilo A intente obtener el cerrojo de la posición de memoria 1 se encontrará que este está en uso y esperará a que esté libre. Esto no ocurrirá puesto que está siendo usado por el hilo B, el cual necesita los datos de la posición de memoria 0 y espera a que este sea liberado por el hilo A. En esta situación ninguno de los dos hilos A y B pueden progresar correctamente y se produce un deadlock.

La Memoria Transaccional (TM) [16] se propone como una solución eficiente para solucionar estos problemas con los cerrojos a la hora de implementar secciones críticas. Con TM, los programadores definen transacciones que se ejecutan de forma paralela y especulativa. Las transacciones garantizan la exclusión mútua mediante mecanismos de gestión de versiones y de detección de conflicto. Cuando dos o más transacciones acceden a la misma posición de memoria, creando conflicto, sólo una de ellas puede continuar. El resto deben deshacer los cambios especulativos en memoria y reiniciar su ejecución. Como puede deducirse, cuando hay que deshacer cambios se produce un impacto negativo en el rendimiento y de consumo de energía.

Existen diversos estudios para la estimación del consumo energético [22, 13, 25], pero dichos estudios siempre han utilizado simuladores para la estimación del consumo energético. Un trabajo previo [26] si ha utilizado una plataforma real para caracterizar un conjunto

de aplicaciones que utilizan TM por software, obteniendo mediciones de energía y rendimiento. Las aplicaciones estudidadas provienen del conjuto de benchmarks STAMP [21] y la libreria TM utilizada es TinySMT [12, 11]. El estudio se ha realizado sobre un dispositivo ODROID XU-3 el cual incorpora un procesador big.LITTLE de ARM. De este trabajo se han derivado varias conclusiones. En primer lugar las aplicaciones funcionan mejor (tanto en cuestión de energía como de rendimiento) en los procesadores de bajo consumo, exceptuando en las aplicaciones que requieren una alta potencia de cómputo. En segundo lugar, la utilización del clúster entero (4 núcleos en cada clúster en nuestro caso) genera mejores resultados. Por último, en los procesadores de alto rendimiento las aplicaciones deben reinicar más transacciones dado que la probabilidad de conflicto aumenta.

Dado que las aplicaciones que utilizan TM se pueden comportar de forma distinta en ambos tipos de núcleos, la manera en la que se planifican tiene un papel fundamental en el rendimiento del sistema. Por ejemplo, un buen planificador debe planificar aplicaciones sobre el clúster little si se requiere reducir el consumo energético. Además, las aplicaciones deben planificarse sobre los núcleos en los que se producen menos transacciones abortadas, con el objetivo de minimizar la energía desperdiciada. En este trabajo exploramos el papel que la planificación puede tener a la hora de mitigar los efectos negativos del uso de TM.

1.1.1. Estructura de la memoria

Tras esta introducción discutimos los objetivos y presentamos los conceptos sobre TM y procesadores heterogéneos necesarios para enteder el resto de la memoria. También se realiza un estudio del arte en el que se discuten los trabajos previos. Por último, encontramos un breve resumen sobre las tecnologías empleadas. En el capítulo 2 se realiza un estudio de ejecución concurrente de las aplicaciones de STAMP sobre la placa ODROID. El objetivo es comprobar si las aplicaciones se pueden ejecutar concurrentemente en nuestro sistema y, en caso afirmativo, si su rendimiento se ve afectado por la compartición de recursos. En el capítulo 3, exponemos el diseño del planificador ScHeTM. En el capítulo 4, se realiza un análisis de los resultados obtenidos. En el capítulo 5, cerramos esta memoria con las conclusiones de los experimentos, del trabajo de fin de grado y una propuesta de trabajo futuro.

1.2. Objetivos

En este trabajo se presenta ScHeTM, un planificador de aplicaciones para procesadores multi-núcleo heterogéneos que considera características de TM. Inspirado en otros planificadores para procesadores heterogéneos previos, ScHeTM trata de optimizar tanto el consumo energético, como los valores de rendimiento en tiempo de cómputo y el número de transacciones abortadas. En las pruebas realizadas con en el dispositivo ODROID-XU3, el cual incorpora un procesador big.LITTLE, se han planificado un conjunto de aplicaciones del benchmack suite STAMP. La implementación de TM utilizada ha sido la libreria de TM por software TinySTM. En primer lugar, se ha implementado un planificador básico, el cual permite lanzar un conjunto de tareas sobre un procesador heterogéneo de forma ávida. A partir de este planificador se ha desarrollado ScHeTM. ScHeTM analiza, en primer lugar, las características de las aplicaciones a ejecutar. A continuación ScHeTM asigna cada aplicación a un clúster basándose en el análisis realizado. ScHeTM se ha diseñado de forma parametrizada, de modo que se puede dar un peso diferente a las características de tiempo de ejecución, consumo energético y número de transacciones abortadas. De esta manera podemos configurar el planificador para tratar de minimizar cada una de estas características. El resultado de nuestros experimentos es que el tiempo de ejecución puede reducirse en un 40 % y el consumo de energía tiene una mejora de un 15 % respecto a un planificador ávido. Además el número de transacciones abortadas se reduce.

1.3. Antecedentes

1.3.1. Arquitectura big.LITTLE de ARM

El hardware en el que vamos a basar el trabajo, ODROID-XU3 [2] está formado por un procesador Samsung Exynos 5422. Este procesador esta basado en la arquitectura heterogénea big.LITTLE de primera generación de ARM. Esta arquitectura está formada por unos núcleos de alto rendimiento y otros de bajo consumo. Según los datos proporcionados por ARM [1] esta arquitectura nos proporciona un ahorro de hasta un 75 % en el gasto energético cuando se ejecuten escenarios de bajo rendimiento y consigue aumentar el rendimiento un 40 % en aplicaciones multi-hilo. El clúster big está formado por cua-

tro núcleos de la familia Cortex-A15 que comparten 2 Mbytes de cache. Los núcleos que forman el clúster little pertenecen a la familia Cortex-A7 y comportanten 512 Kbytes de memoria cache. El hardware de la ODROID-XU3 incorpora monitores INA231 de corriente utilizados para medir el consumo de energía. El sistema incluye una GPU MALI-T628 y 2 Gbytes de memoria DDR3 como muestra la Figura 1.3. Sobre este dispositivo se encuentra instalado un sistema operativo Linux odroid 3.10.59+.

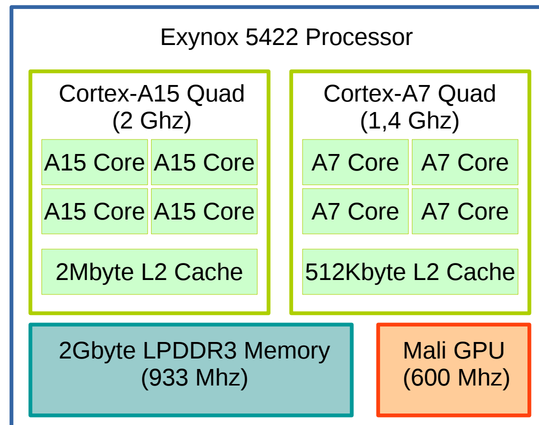

Figura 1.3: Diagrama del procesador Exynos 5422 disponible en la plataforma ODROID-XU3.

1.3.2. Planificación en procesadores heterogéneos

Los procesadores basados en big.LITTLE soportan por defecto 3 tipos de planificación distintas[1]. Una de estas planificaciones es *clustered switching* la cual opera con un solo clúster al mismo tiempo. El sistema operativo se encarga de decidir que clúster usar dependiendo de la carga de trabajo que el procesador tenga, pero no puede combinar ambos clústers. Otro tipo de planificación es *in-kernel switcher*. Esta agrupa los núcleos big y little en pares, uno de cada. En función de la carga de trabajo, el planificador decide si el par (que funciona como un único núcleo virtual) trabaja en modo bajo consumo para ahorrar de energía, o en modo alto rendimiento para reducir el tiempo de cómputo. En este tipo de planificación se pueden tener núcleos virtuales trabajando en modo little y otros trabajando en modo big, cosa que no encontramos en el modo clustered switching. El modo *heterogeneus multi-processing* (conocido tambien como *global task scheduling*) es el utilizado en este trabajo. Este modo nos permite gestionar los núcleos de ambos

[1]Disponible en `https://en.wikipedia.org/wiki/ARM_big.LITTLE`

clústers de forma independiente. Por tanto, podemos tener activos núcleos de diferente tipo al mismo tiempo.

1.4. Estudio del arte

Existen dos tipos de planificación en los procesadores heterogéneos además de los expuestos anteriormente: *thread-to-core* y *thread-to-cluster*. La planificación thread-to-core se centra en asignar una aplicación a los distintos núcleos de un procesador. Este tipo de planificación tambien la podemos encontrar en los procesadores homogéneos. Para realizar este cometido, se debe analizar el rendimiento de las aplicaciones cuando se ejecutan utilizando distinto número de núcleos. El análisis de rendimiento puede realizarse usando contadores software o hardware. Estas mediciones se pueden centrar en los fallos de cache, ciclos ociosos, fallos en la predicción de salto o en el aumento del rendimiento de las aplicaciones. La planificación thread-to-cluster, maneja la asignación de aplicaciones a un clústers específicos (esto es, a qué clúster asignar la aplicación). En trabajos previos se proponen algoritmos que usan el clúster big en las aplicaciones con más carga de cómputo [8], una planificación basada en el rendimiento histórico de las aplicaciones [9] o tener como base la igualdad del progreso de las aplicaciones [19]. En este trabajo desarrollamos un planificador thread-to-cluster con la idea de que se pueda desarrollar un planificador thread-to-core en un futuro. La propuesta de diseño de este trabajo está basado en el desarrollo de Libutti *et al.* [18], donde se presenta un planificador que trabaja a nivel thread-to-cluster. Para la planificación thread-to-core se proporciona una serie de restricciones al SO. Este planificador funciona en 2 fases separadas.

La primera fase se basa en el cálculo de las llamadas *stake functions*. El cálculo de estas funciones depende del tiempo de uso de la CPU y la sensitividad a las interferencias en la jerarquía de memoria que pueden sufrir al ejecutarse varias aplicaciones de forma concurrente. Estas funciones se calculan para cada aplicación y para cada clúster (es decir, para N clústers y M aplicaciones, hay que calcular $N \times M$ stake functions).

En la segunda fase, se realiza una planificación con los datos obtenidos por las stake functions, con el objetivo de minimizar el tiempo de cómputo y disminuir la contención de recursos cuando se ejecutan varias aplicaciones en el mismo clúster. Primero se realiza una planificación thread-to-clúster, donde las aplicaciones con mayor contención de memoria

tienen prioridad. Estas son asignadas al clúster big en caso de que esté disponible. El clúster little se emplea en el caso de que el clúster big esté ocupado o para aplicaciones con menor contención de memoria. Después de que se le asigne una aplicación a un clúster se produce la planificación thread-to-core. Para llevar a cabo esta planificación se evalúan las stake functions y el estado del clúster en cuestión. Con estos resultados se intenta reservar los recursos necesarios para la ejecución de la aplicación, y si es posible hacerlo de forma aislada. Si no se dispone de los recursos necesarios en el clúster, o este está ocupado, la planificación pasa a estar a cargo del SO. En caso contrario, se reservan los núcleos necesarios y se procede a la planificación.

En la literatura encontramos diversos trabajos que relaccionan el consumo de energía y TM. Gaona *et al.* [14] analiza el consumo energético de dos TM por hardware. Proponen serializar las transacciones que se abortan para minimizar el consumo energético. Moreshet *et al.* [22] y Ferri*et al.* [13] realizan análisis energéticos al TM por hardware introducido en [16]. Este estudio se ha realizado utilizando simuladores. Uno de los simuladores es *Virutech Simics* [20], el cual simula una arquitectura tipo SPARC. Otro de los simuladores es MPARM *simulation framework* [4] el cual simula una aquitectura ARMv7. Los resultados obtenidos muestran una mejora en el consumo energético si se comparan con el uso de cerrojos en exclusión mútua. Baldassin *et al.* [7, 6] caracterizan la librería TM por software TL2 [10] sobre procesadores homogéneos de bajo consumo basados en una arquitectura ARMv7. Usan simuladores y proponen reducir el consumo de energía usando una solución basada en la variación de voltaje y el escalonado de la frecuencia del procesador. Sanyal *et al.* [25] proponen técnicas de *clock-gating* con el objetivo de reducir el consumo de energía en TM hardware y mejorar su rendimiento. Villegas *et al.* [26] han realizado un estudio de las aplicaciones del benchmark suite STAMP y el TM por software TinySTM en un procesador heterogéneo de la familia big.LITTLE. Los resultados obtenidos son que las aplicaciones rinden de manera distinta en ambos clústers. Este es el primer trabajo que analiza el consumo energético de TM sobre hardware real (sin usar simuladores).

1.5. Tecnologías utilizadas

1.5.1. TinySTM

TinySTM[2] es una librería TM por software basada en una implementación de *Lazy Snapshot Algorith* (LSA) [11]. Esta librería se basa en *timestamps* y trabaja a nivel de palabra de memoria. TinySTM tiene varios modelos para la gestión de transacciones dependiendo del momento de adquisición de cerrojos y de cuando la memoria principal se actualiza. Estos modelos son: *write-back*, *write-through*, *commit-time locking* y *encounter-time locking*. El modo write-back guarda los cambios especulativos de memoria en un buffer hasta que, en el final de la transacción, se hacen definitivos si no existe conflicto. El modelo write-through, son los valores nuevos los que se guardan en memoria, mientras que los antiguos se guardan en un *log* por si tienen que ser restaurados en caso de conflicto. El modo commit-time utiliza cerrojos para proteger las posiciones de memoria que estan siendo actualizadas al final de la transacción. Por último, el modo encounter-time locking usa los cerrojos en los accesos a memoria en lugar de al final de la transacción. En los experimentos que se presentan aquí se ha utilizado la configuración por defecto de TinySTM, la cual usa las opciones write-back y encounter-time-locking.

1.5.2. STAMP

Stanford Transactional Applications for Multi-Processing [21] (STAMP) es un conjunto de aplicaciones muy popular, usado para la evaluación de los distintos sistemas TM. STAMP se compone por ocho aplicaciones de diferentes dominios, treinta configuraciones y varios conjuntos de datos de entradas. Para facilitar la portabilidad de estas aplicaciones, se han escrito en lenguaje C. A continuación se resumen las aplicaciones incluidas en STAMP:

- Bayes: implementa un algoritmo de aprendizaje de la estructura de una red Bayesiana [27] . Bayes utiliza la mayor parte del tiempo largas transacciones, que contienen grandes conjuntos de lectura y escritura.

- Genome: crea un genoma original combinando un gran número de secuencias de ADN. El algoritmo se compone de dos fases. En la primera fase, se crea el conjunto

[2]Disponible en `http://tmware.org/tinystm`

de secuencias únicas. En la segunda, cada hilo de ejecución intenta eliminar una secuencia del conjunto de las no combinadas y añadirlas a su partición de secuencias combinadas.

- Intruder: implementa un sistema de detección de intrusos en una red (*Network Intrusion Detection System*(NIDS)). STAMP proporciona una implementación que emula el diseño de 5 NIDS descritos en [15]. Los paquetes se procesan en paralelo en tres fases: captura, reconstrucción y detección. Las fases de captura y reconstrucción se implementan con transacciones.

- Kmeans: implementa el algoritmo de agrupación *K-Means* que clasifica objetos de un espacio de N dimensiones en K clústers. La implementación proporcionada por STAMP está tomada de [23].

- Labyrinth: implementa una variación del algoritmo de Lee [17]. La estructura base de Labyrinth es una cuadrícula de 3 dimensiones la cual representa un laberinto. Cada uno de los hilos escoge dos puntos, uno inicial y otro final, que se deben conectar utilizando los puntos adyacentes dentro de la cuadrícula. Las transacciones son utilizadas para calcular el siguiente punto del camino adyacente y añadirlo a la cuadrícula principal. Los conflictos ocurren cuandos dos o más caminos se cruzan.

- Ssca2: (*Scalable Synthetic Compact Applications 2*) [5] está compuesta por 4 *kernels*. STAMP utiliza el kernel 1 ya que es el más adecuado para TM. Este kernel genera una estructura de datos de tipo grafo. Para ello, utiliza arrays de adyacencia y auxiliares. Los hilos van añadiendo nodos al grafo en paralelo utilizando las transacciones para proteger los accesos al array compartido de adyacencia.

- Vacation: implementa un sistema *online* de transacciones similar a *SPECjbb2000* [3] pero simulando un sistema de reservas de viajes. En la ejecución de este algoritmo varios clientes acceden a la base de datos para realizar o cancelar reservas.

- Yada: (*Yet Another Delaunay Application*) implementa el algoritmo de Ruppert para la mejora de redes Delaunay [24]. Se usa como estructura de datos un grafo que almacena una serie de triángulos que se modifica dentro de la transacción.

Como se estudia en [26], no todas las aplicaciones de STAMP trabajan correctamente o tienen resultados similares entre ejecuciones. Por tanto, del conjunto de aplicaciones del

que partimos se descartan Bayes, Yada y Genome. Para poner a prueba las características de TM, cada una de las aplicaciones tiene una longitud de transacción, tamaño de los conjuntos de lectura y escritura, tiempo empleado en las transacciones y tasa de aborto de transacciones diferentes. Las características de cada aplicación se recogen en la Tabla 1.1.

Aplicación	Longitud de transacción	Conjuntos de lectura y escritura	Tiempo en transación	Número de transacciones abortadas
Intruder	Corta	Medios	Media	Alta
Kmeans	Corta	Pequeños	Bajo	Alta
Labyrinth	Larga	Grandes	Alto	Baja
Ssca2	Corta	Pequeños	Bajo	Baja
Vacation	Media	Medios	Alto	Media

Tabla 1.1: Características de las aplicaciones STAMP.

Las aplicaciones de STAMP también incluyen un conjunto de configuraciones recomendadas. Kmeans y Vacation disponen además de entradas de alta y baja contención reconocibles por el sufijo *-high* o *-low*. El tamaño de los conjuntos de datos viene indicado por los sufijos + y ++. Es este último el utilizado en este trabajo puesto que es el que proporciona una mayor carga de cómputo.

1.5.3. Lenguajes de script

Para este trabajo se han utilizado dos lenguajes de script: Bash y Python. El lenguaje Bash (Bourne Again Shell) es una shell de Unix y un lenguaje de comandos, escrito por Brian Fox para el proyecto GNU a final de los años 80. Surge como un reemplazo del Bourne Shell siendo capaz de ejecutar todas sus instrucciones sin ninguna modificación. Bash permite, además, otras operaciones como pueden ser: el cálculo de enteros, uso de expresiones regulares, redirecciones de entrada y salida, y caracteres especiales.

El otro lenguaje de scripts utilizado en este trabajo es Python. Python es un lenguaje interpretado multiplataforma. Guido van Rossum creó este lenguaje a finales de los años 80 y es el encargado de llevar la administración de Python Software Foundation. Este lenguaje da la posibilidad a los programadores de programar de distintas maneras: programación orientada a objetos, programación imperativa y programación funcional. Una de las caraterísticas principales de Python es su lenguaje de alto nivel, proporcionando una sintaxis que genera códigos faciles de leer. A día de hoy se encuentran varias versiones

de Python, la versión 2 y la versión 3. Aún se sigue trabajando con ambas versiones ya que entre ellas no son compatibles. Su uso más común es implementar scripts para servidores que funcionan de forma continua. Existen varias librerías de Python. Concretamente, en este trabajo se va a utilizar *matplolib* para producir figuras y gráficas de una forma sencilla.

1.5.4. Latex

Latex es un sistema de edición de textos normalmente utilizado en la creación de artículos, libros técnicos y tesis. Latex se basa en instrucciones. Para crear un documento en Latex primero se deben escribir mediante un editor de texto las ordenes y el contenido que se quiere mostrar en el documento. A continuación, el documento se compila y se obtiene la salida correspondiente. En nuestro caso, hemos utilizado el compilador pdflatex para generar el archivo PDF de esta memoria.

Capítulo 2

Estudio de ejecución concurrente

2.1. Caracterización de las aplicaciones

Como inicio del trabajo, partimos de estudios anteriores donde se han caracterizado las aplicaciones de STAMP en un dispositivo ODROID [26]. Las conclusiones obtenidas en este trabajo muestran la existencia de aplicaciones que ofrecen un mejor rendimiento en el clúster little y otras que rinden mejor en el clúster big. A pesar de que el rendimiento está repartido entre los 2 clústers, el clúster little siempre proporciona una mejor eficiencia energética. Esta caracterización se ha realizado siempre de forma aislada, ejecutando cada aplicación en un único clúster sin la intromisión de otras aplicaciones en el sistema.

En este trabajo, se pretende planificar de forma paralela una aplicación en cada clúster. Nos parece interesante saber si las conclusiones anteriores se pueden extenderse a una ejecución concurrente de aplicaciones. Esto se ha llevado a cabo replicando los experimentos del trabajo previo. En este caso, mientras se analiza el rendimiento y consumo energético de una aplicación en uno de los clúster, se ejecuta otra aplicación en el otro clúster. El objetivo de este experimento es comprobar los posibles cambios en el rendimiento o consumo energético de las 5 aplicaciones en una ejecución concurrente. Para ello, analizamos el comportamiento de una aplicación en un clúster mientras que en el otro ejecutamos otras aplicaciones. Para cada aplicación hemos realizado 10 ejecuciones y se muestra la media.

En la Figura 2.1 mostramos las 25 posibles combinaciones de aplicaciones (esto es, 5 aplicaciones ejecutándose en cada clúster). Podemos verificar que el comportamiento de una aplicación ejecutándose en un clúster dado no varía cuando ejecutamos otra aplicación

en el otro clúster. Por ejemplo la aplicación Intruder siempre presenta un menor gasto energético en el clúster little, independientemente de la aplicación que se ejecuta en el clúster big. Analicemos ahora el comportamiento de las aplicaciones una a una.

	Big	Little	Big	Little	Big	Little	Big	Little	Big	Little
	Intruder	Intruder	Labyrinth	Intruder	Ssca2	Intruder	Kmeans	Intruder	Vacation	Intruder
Energy (J)	797.185	565.129	302.378	251.236	229.543	300.361	232.502	245.210	1.442.691	507.865
Execution Time (s)	149.034	103.866	34.440	88.041	45.797	98.911	44.015	91.447	286.036	96.388

	Big	Little	Big	Little	Big	Little	Big	Little	Big	Little
	Intruder	Labyrinth	Labyrinth	Labyrinth	Ssca2	Labyrinth	Kmeans	Labyrinth	Vacation	Labyrinth
Ernergy (J)	716.357	508.941	318.409	430.945	205.659	337.351	235.428	338.836	1.420.148	500.214
Execution Time (s)	129.334	91.931	37.206	93.506	36.440	93.607	39.905	91.186	273.491	92.269

	Big	Little	Big	Little	Big	Little	Big	Little	Big	Little
	Intruder	Ssca2	Labyrinth	Ssca2	Ssca2	Ssca2	Kmeans	Ssca2	Vacation	Ssca2
Energy (J)	709.685	205.806	309.020	49.420	48.833		198.852	49.558	1.392.187	185.632
Execution Time (s)	138.216	38.640	35.182	33.037	33.112		37.818	32.832	277.890	34.800

	Big	Little	Big	Little	Big	Little	Big	Little	Big	Little
	Intruder	Kmeans	Labyrinth	Kmeans	Ssca2	kmeans	Kmeans	Kmeans	Vacation	Kmeans
Energy (J)	665.825	125.665	305.241	217.435	189.844	109.934	225.548	151.156	1.362.514	11.732
Execution Time (s)	131.265	26.303	34.273	23.717	38.317	26.141	41.839	27.319	274.111	24.280

	Big	Little	Big	Little	Big	Little	Big	Little	Big	Little
	Intruder	Vacation	Labyrinth	Vacation	Ssca2	Vacation	Kmeans	Vacation	Vacation	Vacation
Energy (J)	756.368	765.751	314.515	345.581	200.475	353.582	220.579	316.341	1.502.275	866.092
Execution Time (s)	143.373	162.821	35.875	140.136	39.145	146.297	41.302	142.865	292.294	164.285

Figura 2.1: Resultados de ejecutar las 25 posibles combinaciones de aplicaciones sobre ambos clústers.

Intruder. La aplicación Intruder tiene un mejor rendimiento y menor consumo energético en clúster little. Esto lo deducimos por que en la primera columna (en la que Intruder siempre se ejecuta en el clúster big) los valores son mayores que los correspondientes en las columnas little de la primera fila. Por tanto, independientemente de la aplicación ejecutandose en el otro clúster, siempre es mejor planificarla sobre el clúster little.

Labyrinth. El caso de Labyrinth es diferente. El tiempo de cómputo y el consumo energético siempre son menores en el clúster big. Esto se debe a que Labyrinth requiere mayor potencia de cómputo y se beneficia de la cache de mayor tamaño presente en el clúster big (si observamos la Tabla 1.1, Labyrinth es la aplicación con mayores requisitos). Por tanto, se recomienda planificar esta aplicación en el clúster big.

Ssca2. En el caso de Ssca2 los tiempos de cómputo son similares (en promedio, la diferencia entre ambos clústers es de aproximadamente el 10 %). Sin embargo, el consumo energético es siempre menor en el clúster little. En la casilla central de la tabla observamos que no hay datos de la ejecución Ssca2 en el clúster little. Esto se debe a que el sistema operativo cancela la ejecución de esta aplicación en el clúster little mientras se está ejecuntado otra instancia en el clúster big. Este suceso se encuentra bajo investigación. En los siguientes experimentos (esto es, cuando se ponga a prueba el planificador) se consideran únicamente aquellos en los que Ssca2 no se planifica concurrentemente en los 2 clústers.

Kmeans. La aplicación Kmeans tiene mejor consumo energético y rendimiento en el clúster big. Esta aplicación se comporta de manera similar a Intruder.

Vacation. Para la aplicación Vacation el uso del clúster little le proporciona una mejora energética significativa respecto al clúster big. Esto se debe a que se mantiene en ejecución durante más tiempo en el clúster big.

El mismo estudio se ha repetido para medir el número de transacciones abortadas. Habitualmente las aplicaciones Intruder y Kmeans son las que tienen un mayor número de transacciones abortadas (varios millones). Le sigue Vacation con varios miles de transacciones abortadas. Labyrinth y Ssca2 sólo presentan decenas de transacciones abortadas.

La Tabla 2.1 presenta un resumen de las características mencionadas anteriormente. En ella se indica el clúster en el que cada aplicación presenta mejores resultados para cada característica. Se ha considerado que presentan un comportamiento similar si las variaciones son menores al 10 %.

Aplicación	Tiempo de cómputo	Consumo energético	Transacciones abortadas
Intruder	little	little	similar
Kmeans	little	little	little
Labyrinth	big	little	similar
Ssca2	similar	little	big
Vacation	little	little	similar

Tabla 2.1: Resumen de rendimiento de las 5 aplicaciones STAMP sobre un procesador big.LITTLE, indicando por cada aplicación en que clúster genera mejor rendimiento

De este estudio podemos concluir varias cosas. En primer lugar las aplicaciones presentan un rendimiento estable mientras se ejecutan en un clúster aunque otra aplicación

se ejecute en otro clúster. En segundo lugar, comprobamos que el clúster little siempre proporciona un menor consumo energético. Por último, tanto el tiempo de cómputo como el número de transacciones abortadas es distinto para cada aplicación y cada clúster.

Capítulo 3

Planificador

3.1. Planificador ScHeTM

En este trabajo se ha diseñado ScHeTM, un planificador de aplicaciones TM que se ejecutan en un procesador multi-núcleo heterogéneo. ScHeTM realiza una planificación thread-to-clúster. En principio, se asume que cada aplicación va a utilizar un clúster al completo por lo que no se realiza la planificación thread-to-core. En las siguientes subsecciones se expone el diseño de dicho planificador.

3.1.1. Visión general

La metodología de planificación se compone de 2 fases. En una primera fase, evaluación, las aplicaciones son analizadas calculando una función de idoneidad $I_c(T, E, A)$ donde c se refiere al clúster (little o big) y los parámetros T, E, A medidas del tiempo, consumo de energía y transacciones abortadas, respectivamente. Una vez caracterizadas las aplicaciones comienza la segunda fase. En esta segunda fase las aplicaciones se planifican adecuadamente usando el análisis anterior.

La función de idoneidad de cada aplicación tiene que calcularse tanto para el clúster little ($I_{little}(T, E, A)$) como para el big ($I_{big}(T, E, A)$). Esta función está acotada entre 0 y 1 según la aplicación se adapte a cada clúster. 0 indica un mal rendimiento de la aplicación en dicho clúster, y debería evitarse, mientras que 1 indica que la aplicación funciona de forma óptima en dicho clúster. Esto se explica en la siguiente sección con más detalle. Este cálculo debe realizarse para cada una de las aplicaciones que vamos a planificar, como se muestra en la Figura 3.1. Para cálcular estas funciones, debemos usar los datos obtenidos

a la hora de instrumentar las aplicaciones como se ha visto en el capítulo anterior.

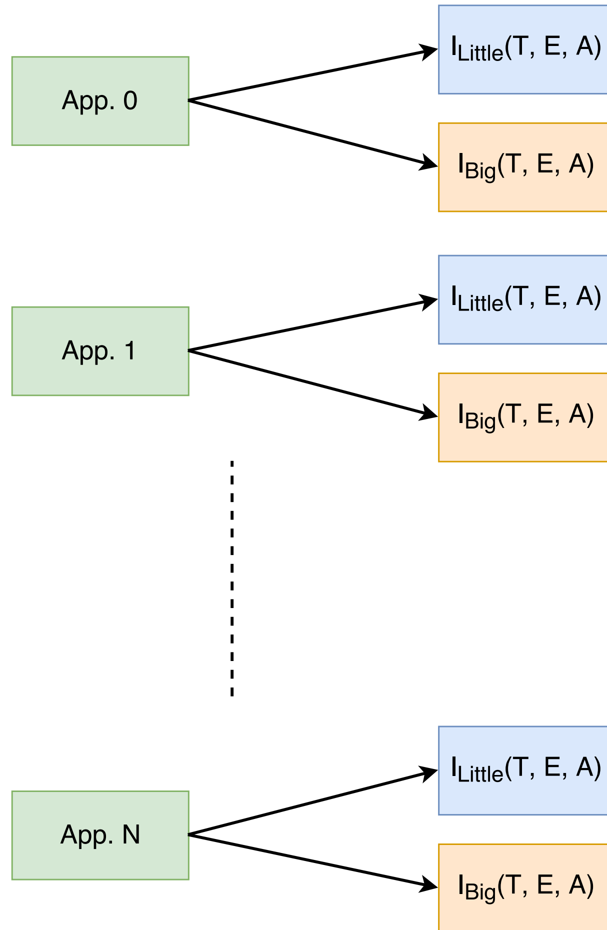

Figura 3.1: Fase de evaluación de las aplicaciones

Teniendo ya las funciones de idoneidad calculadas, el planificador está listo para recibir las aplicaciones, las cuales son proporcionadas por una cola. Con esto comienza la segunda fase. En esta fase, las aplicaciones van llegando a los clústers, los cuales se pueden encontrar en 2 estados distintos: o esperando una aplicación (si estaba previamente ocioso) o ejecutando una aplicación que se le ha asignado previamente. Esta segunda fase se repite mientras sigan existiendo aplicaciones pendientes en la cola y exista un clúster ocioso. Este proceso se explica con más detalle a continuación.

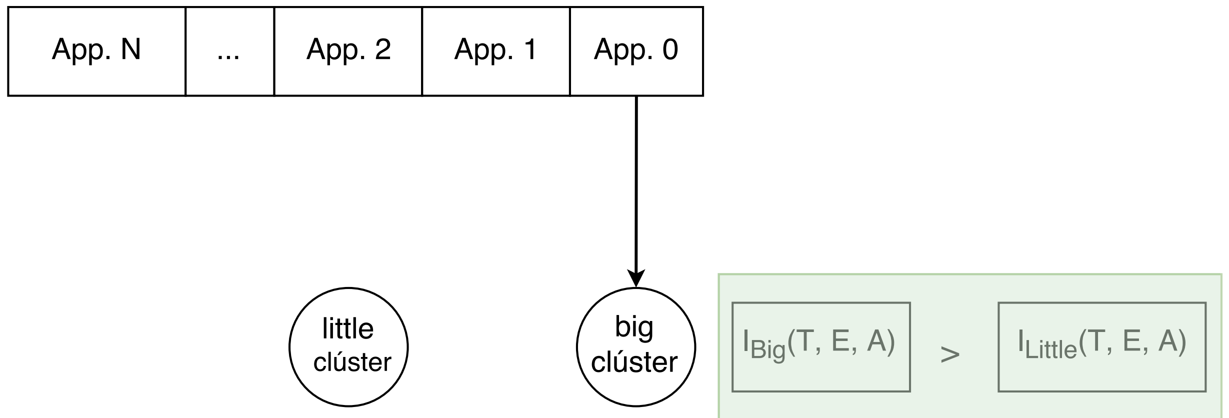

Figura 3.2: Búsqueda de aplicación y aceptación en el clúster big

En el ejemplo de la Figura 3.2 podemos ver como, en un principio, ambos clústers están ociosos y el clúster big escoge la primera aplicación de la cola disponible (aplicación 0). En este momento el clúster big compara su función de idoneidad para dicha aplicación contra la del clúster little ($I_{big}(T, E, A) > I_{little}(T, E, A)$).

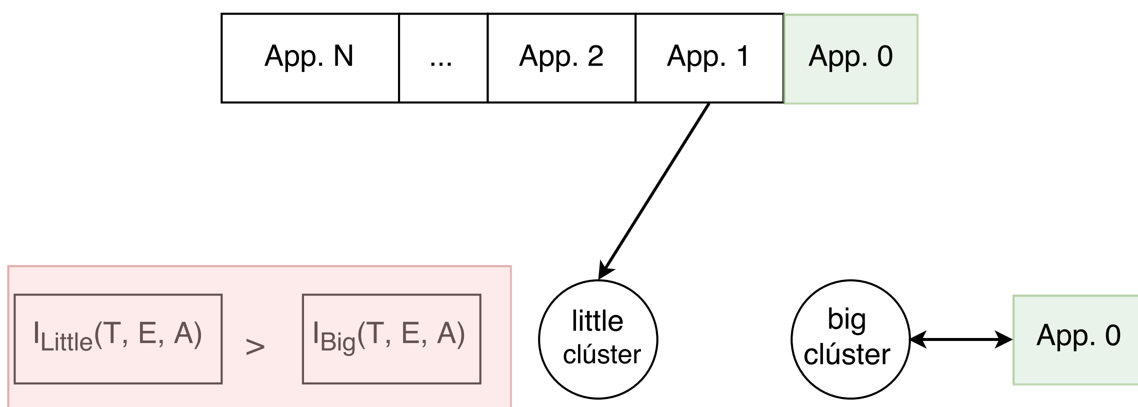

Figura 3.3: Búsqueda de aplicación y rechazo en el clúster little

Esta comparación resulta ser verdadera y el clúster big acepta la aplicación. Ahora, en la Figura 3.3 el clúster big está ocupado y el clúster little busca una aplicación para ejecutar. En este caso se obtiene de la cola la aplicación 1. El siguiente paso es comparar

las funciones de idoneidad de dicha aplicación para el clúster little ($I_{little}(T, E, A) > I_{big}(T, E, A)$), lo cual en este caso da falso y el clúster little rechaza ejecutarla.

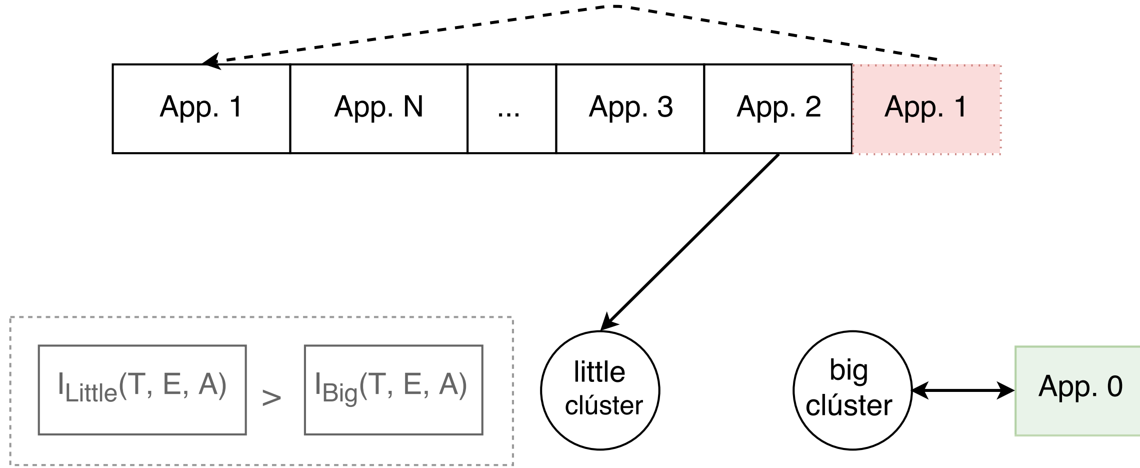

Figura 3.4: Búsqueda de aplicación y aceptación en el clúster little

El último caso, la aplicación 1 (antes rechazada) vuelve al final de la cola como se puede observar en la Figura 3.4. Entonces, el clúster little recibe de la cola la siguiente aplicación, repitiendo la misma evaluación. Si la aplicación sigue sin ser idónea, entonces el proceso se repite hasta seleccionar una. Para evitar que uno de los clúster quede ocioso demasiado tiempo, si se rechaza una de las aplicaciones, las condiciones se relajan para así poder aceptar una de las siguientes. Esto se consigue incrementando la función de idoneidad del clúster ocioso para que pueda acercar sus valores al del otro clúster. Este proceso se explica con más detalle más adelante.

3.1.2. Función de Idoneidad

En el paso 1 de la planificación debe calcularse la función de idoneidad $I_c(T, E, A)$ de cada aplicación en cada clúster. La función tiene la forma $I_c(T, E, A) = tF_c(T) + eF_c(E) + aFc(A)$, donde $Fc(N)$ indica cómo de idóneo es el clúster c para la característica N de la aplicación, obteniendo un valor entre 0 y 1. Los 3 valores $F_c(T)$, $F_c(E)$, $Fc(A)$ están parametrizados por las contantes t, e y a, respectivamente. Debe cumplirse también que $0 \leq t \leq 1$, $0 \leq e \leq 1$, $0 \leq a \leq 1$ y $t + e + a = 1$. Usando estas 3 constantes podemos indicarle a nuestro planificador ScHeTM que pondere de forma distinta a cada característica

de la aplicación. Inicialmente consideramos que cada una de las características tiene el mismo valor $t = e = a = 1/3$, lo que nos da una ponderación equilibrada entre las 3 funciones que estudiamos. Más adelante se estudian casos concretos donde una característica sobresale del resto dandole mayor peso al parámetro que la pondera.

El cáclulo de las funciones $tF_c(T), eF_c(E), aFc(A)$ es diferente para cada clúster. Sin embargo, para cada característica T, E o A se calcula de la misma forma, por lo que las llamamos de forma genérica $F_{little}(N)$ y $F_{big}(N)$ (donde N es T, E, o A).

Característica	Valor del clúster little	Valor del clúster big
Tiempo de ejecución	little: 20 s.	big: 10 s.
Consumo de energía	little: 50 J.	big: 100 J.
Transacciones abortadas	little: 500	big: 200

Tabla 3.1: Valores propuestos para el ejemplo

Como ejemplo, mostramos el cálculo de $F_{little}(N)$. El cálculo de $F_{big}(N)$ se realiza de forma similar. Una vez instrumentadas y evaluadas las aplicaciones en cada clúster, se calcula el valor k. Para la función $F_{little}(N)$ el valor de k se calcula como $k = N(little)/N(big)$ siendo $N(little)$ y $N(big)$ el valor de la característica N en los clúster little y big respectivamente. Tomando como ejemplo los datos de la Tabla 3.1, el valor k para el clúster little y la característica T (tiempo) es 2. El valor de k indica la eficiencia que tiene cada clúster en función de la característica estudiada para la aplicación actual. Un valor k menor que 1 nos indica que el clúster little rinde mejor que el clúster big para dicha aplicación y, cuanto más se acerca a 0, mejor es el rendimiento de esta. Un valor de 1 indica que la aplicación rinde de igual manera en cualquiera de los dos clúster y un valor superior a 1 indica que el clúster big tiene mejor rendimiento que el little. Estos valores mayores que 1 son ignorados y sustituidos por 1. De esta forma, calculando $F_{little}(N) = 1-$k$= 1 - N(little)/N(big)$ obtenemos 0 si el clúster little no es el adecuado para esta aplicación y 1 si se ejecuta de forma óptima. El calculo de F_{big} es similar y siguiendo el mismo razonamiento, Por lo tanto $F_{big}(N) = 1 - N(big)/N(little)$. En la Tabla 3.2 calculamos F_{little} y F_{big} para todas las características con los datos correspondientes a la Tabla 3.1.

De esta forma, la suma de los resultados obtenidos por las 3 funciones correspondientes a $F_c(T)$, $F_c(E)$ y $F_c(A)$ esta acotada entre 0 y 3. Utilizando las constantes t, c, a definidas anteriormente $a = t = e = 1/3$, $I_c(T, E, A) = tF_c(T) + eF_c(E) + aF_c(A)$ nos proporciona un valor entre 0 y 1. En la Tabla 3.3 se puede observar el cálculo de $I_c(T, E, A)$ para la

$F_c(N)$	$F_{little}(N) = 1 - N(little)/N(big)$	$F_{big}(N) = 1 - N(big)/N(little)$
$F_c(T)$	$F_{little}(T) = 1 - 1 = 0$	$F_{big}(T) = 1 - 0,5 = 0,5$
$F_c(E)$	$F_{little}(E) = 1 - 0,5 = 0,5$	$F_{big}(E) = 1 - 1 = 0$
$F_c(A)$	$F_{little}(A) = 1 - 0,4 = 0{,}6$	$F_{big}(A) = 1 - 1 = 0$

Tabla 3.2: Cálculo de la función $F_c(N)$ para los valores propuestos en 3.1

aplicación anterior. En este caso, ScHeTM considera que el clúster little es más adecuado para ejecutarla ya que, en 2 de los 3 parámetros, dicho clúster proporciona mejores resultados.

$I_{little}(T, E, A)$	$I_{big}(T, E, A)$
$I_{little}(T, E, A) = 0,37$	$I_{big}(T, E, A) = 0,17$

Tabla 3.3: Valores de la función de idoneidad

3.1.3. Planificación

Una vez tenemos terminada la primera fase y, por tanto, se han calculado todos los valores de $I_c(T, E, A)$ para cada aplicación en cada uno de los clúster, ScHeTM está listo para recibir aplicaciones y comenzar a planificarlas. En primera instancia, los dos clúster están disponibles para recibir una aplicación ya que ambos están ociosos. Así pues, cada uno de ellos toma una aplicación distinta de la cola de aplicaciones. El clúster big realiza la operación $I_{big}(T, E, A) > I_{little}(T, E, A)$. Si esta comparación es favorable el clúster big (es decir, es verdadera) la aplicación empieza su ejecución en dicho clúster. En caso contrario, cuando la condición sea falsa, la aplicación vuelve al final de la cola de aplicaciones (para ser planificada en el futuro) y el clúster empieza el mismo análisis pero con la siguiente aplicación disponible. De esta forma lo que se pretende es que cada clúster acepte las aplicaciones que mejor se adapten a él y rechace aquellas que, previsiblemente, se ejecutan de forma más adecuada en el otro clúster.

3.1.4. Reparto de carga

Uno de los problemas que surgen con el tipo de algoritmo que diseñamos en nuestro planificador es el rechazo sistemático de aplicaciones en uno de los clúster, si este considera que el otro clúster podría ejecutar las aplicaciones de forma más eficiente. En el peor de

los casos, puede que no ejecute ninguna de las aplicaciones y delegue la ejecución de las aplicaciones en el otro clúster.

Como solución a este problema se introduce la constante B (bound) como límite de aplicaciones rechazadas por un mismo clúster de forma consecutiva. En el caso de que un clúster rechace B aplicaciones, el planificador ejecuta la siguiente aplicación asignada sin importar la función de idoneidad entre los clúster. Esto soluciona el problema de que un clúster pueda quedar ocioso indefinidamente. Para hacer esto de forma gradual, al valor $I_c(T,E,A)$ se le suma el valor $r * (1 - I_c(T,E,A))/B$ a la hora de la comparación $I_{big}(T,E,A) > I_{little}(T,E,A)$. El valor r indica el número de reintentos: inicialmente (y cada vez que una aplicación es plinificada en dicho clúster) $r = 0$. Cada vez que una aplicación es rechazada $r = r + 1$. Al realizar la operación $I_c(T,E,A) + r * (1 - I_c(T,E,A))/B$ estamos incrementando el valor de $I_c(T,E,A)$ proporcionalmente, hasta alcanzar el valor máximo de 1 en el reintento B. Notar que, si todas las aplicaciones se ejecutan de forma óptima en el mismo clúster, el reparto de tareas es el mismo que en una planificación aleatoria. En el clúster menos eficiente sólo se planifican aplicaciones porque se ha alcanzado el límite B y esto puede no ser una solución óptima. Durante la evaluación hemos podido reproducir este fenómeno y discutimos sus efectos y posibles soluciones.

Capítulo 4

Evaluación

En este capítulo se discute la metodología experimental seguida de la evaluación, presentando el entorno de trabajo, métricas utilizadas y la aplicaciones usadas.

4.1. Metodología experimental

4.1.1. Implementación y automatización de las pruebas

En este trabajo se han creado una serie de scripts tanto en Bash como en Python. Estos scripts tienen el objetivo de automatizar nuestro trabajo. Los scripts usados en Bash son *test2.sh*, *parseTest.sh* y *eraseEnergy.sh*. También se usan scripts en Python para la generación de gráficas y la visualización de los resultados de forma númerica. Estos scripts son *parseTests2.py*, *generate2++.py*, *planificador.py* y *planificadorScHeTM.py*. La Figura 4.1 muestra el árbol de directorios sobre el que trabaja los scripts, lanzan los experimentos y generan los resultados.

Para explicar la implementación de nuestro planificador vamos a empezar explicando los scripts desde el más básico, hasta el más complejo.

tests2.sh. El primer script creado es tests2.sh el cual es el encargado de ejecutar las pruebas en cada aplicación. Este está replicado en el directorio de cada aplicación que vamos a usar en el experimento, y está diseñado para recibir 3 argumentos: Número de pruebas, contención en los parámetros de entrada y el clúster al que va a asignarse. Una llamada a este script tiene la siguiente forma `./tests2.sh <N> <C> <A>`

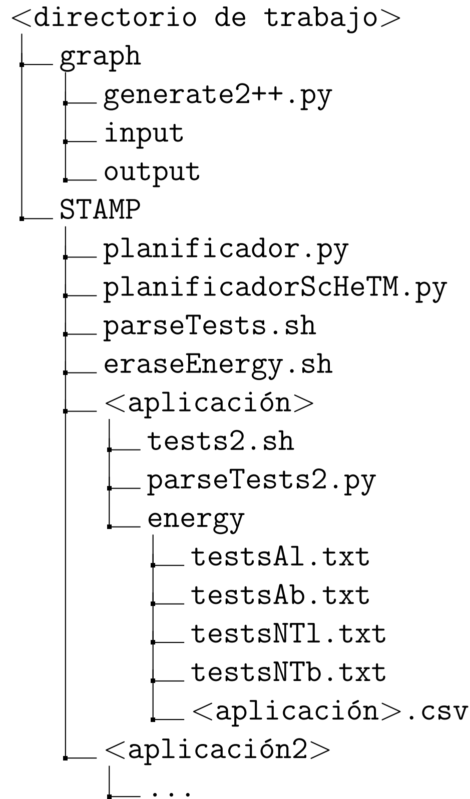

Figura 4.1: Árbol de directorios

- El argumento <N> indica el (N)úmero pruebas que se van a realizar de forma seguida. Lo normal en nuestro experimento es realizar solo una prueba por llamada.
- El argumento <C> indica la (C)ontención en los parámetros de entrada. Los valores posibles son *1* para seleccionar el parámetro ++ en STAMP o *0* para el parámetro +. En nuestras pruebas siempre usamos la máxima contención.
- El argumento <A> indica la (A)finidad de los hilos de ejecución a los núcleos. Este parámetro tiene como entrada *l*, dirigiendo la aplicación a todos los núcleos del clúster little o *b* cuando se quiere utilizar el clúster big.

Los resultados son guardados en el subdirectorio ./energy/ el cual se encuentra dentro de cada aplicación. Este directorio contiene 4 ficheros distintos: *testsNTl*, *testsNTb*, *testsAl* y *testsAb*. En el archivo *testsNTl* se almacenan los datos correspondientes a el consumo energético y de rendimiento cuando se ha ejecutando la aplicación en el clúster little, mientras que *testsNTb* nos ofrece los datos del clúster big. Los ficheros *testsAl* y *testsAb* guardan los datos de las transacciones abortadas en el clúster little y el clúster big, respectivamente. Los datos de tiempo y consumo energético se obtienen directamente de las aplicaciones. Para obtener los datos de las transacciones abortadas, antes de iniciar el

experimento, debe declararse la siguiente variable de entorno: export STM_STATS="1". Por ejemplo, para ejecutar las pruebas de Kmeans, se debe entrar en su directorio y utilizar el comando ./tests2.sh 1 1 1, este comando nos genera los datos de rendimiento, consumo de energía y transacciones abortadas en el clúster little. Dicha ejecución crea los ficheros *testsNTl* y *testsAl*. Si volvemos a ejecutar el mismo comando, los datos de esta nueva ejecución se concatenan en los archivos de datos ya existentes.

parsetests2.py. Para tratar los datos resultantes se utiliza parsetests2.py el cual está en cada uno de los subdirectorios de las aplicaciones. Este script de Python se encarga de leer los campos claves de los documentos creados por tests2.sh, obteniendo sólo los datos relevantes para el experimento y guardándolos en un archivo con el nombre de la aplicación, la contención y la extensión .csv. Por ejemplo si ejecutamos parsetests2.py en la aplicación Kmeans, en el subdirectorio ./energy se crea el archivo Kmeans++.csv.

eraseEnergy.sh. Este script se encuentra en el directorio STAMP. Su función es crear un entorno adecuado para realizar el experimento, borrando los archivos anteriores y generando nuevos ficheros vacíos.

parsetest.sh. Este script es el encargado de hacer las llamadas al script ./parsetests2.py de cada una de las aplicaciones.

planificador.py. Antes de implementar ScHeTM hemos implementado un planificador ávido. El objetivo es tener un planificador base con el que comparar nuestros resultados. Este planificador se ha implementado en el script planificador.py. En primera instancia el planificador ávido prepara el terreno de pruebas ejecutando el script ./eraseEnergy mencionado antes. En el siguiente paso crea una cola con las aplicaciones a ejecutar. Esta cola se implementa con la clase Queue() de Python. Esta clase en Python es Thread safe, lo cual nos permite que en ningún momento tengamos que usar mecanismos de exclusión mutua para evitar que ambos clústers accedan a la vez a la cola de aplicaciones. Una vez creada la cola se crean 2 threads, uno para el core little y otro para el core big. Estos hilos desencolan una aplicación y la ejecutan usando el script tests2.sh en el directorio correspondiente a esta aplicación. Una vez la cola esta vacía, se lanza la llamada a parseTest.sh el cual reúne todos los datos de toda la ejecución. De esta forma, cada clúster ejecuta la

primera aplicación que recibe de la cola tratando de estar ocioso el menor tiempo posible.

planificadorScHeTM.py. El planificador ScHeTM se encuentra implementado en el script planificadorScHeTM.py. Este script solicita un parámetro para indicar los valores de t, e, a que se desean utilizar. La cola de aplicaciones se implementa de forma similar al planificador ávido. De la misma forma se utilizan los scripts eraseEnergy.sh, tests2 .sh y parseTest.sh para la ejecución de las aplicaciones. El único cambio entre este script y planificador .py es que a la hora de escoger la aplicación a ejecutar usamos nuestra función de idoneidad explicada en el capítulo de planificación. Los valores necesarios para las funciones de idoneidad están almacenados en una matriz de datos incluida en este script. Este script también se encarga de evaluar dichas funciones, gestionar la cola y seleccionar el clúster idóneo para la ejecución.

generate2++.py. La ultima parte de nuestra ejecución se completa con el script generate2 ++.py, situado en el directorio graph. Es el encargado de leer todos los ficheros csv generados y crear las gráficas a partir de ellos. En este directorio existen 2 subdirectorios. En el directorio input hay que colocar los ficheros csv generados por los planificadores tras la ejecución. El resultado de la llamada a gnerate2++.py se guarda en el directorio output. Esta salida son una serie de gráficos en formato PDF.

4.1.2. Configuración del hardware

La evaluación de ScHeTM se ha llevado a cabo utilizando el dispositivo ODROID-XU3, que incorpora un procesador multi-núcleo heterogéneo de la familia ARM big.LITTLE. La Tabla 4.1 resume las características de este dispositivo.

Característica	Descripción
CPU	Samsung Exynos-5422: Cortex-A15 y Cortex-A7 big.little
Memoria principal	2 Gbyte LPDDR3 RAM 933MHz
GPU	Mali-T628 MP6
Almacenamiento	32GB Sandisk iNAND Extreme
Medición de energía	Sensores separados para el clúster big, el clúster little, GPU y memoria
S.O	Linux odroid 3.10.59+

Tabla 4.1: Sistema ODROID-XU3 utilizado durante la evaluación

4.1.3. Configuración de los experimentos

Para la evaluación de nuestro planificador y todas sus variantes, se crea una cola de 25 aplicaciones, la cual se compone de las 5 aplicaciones de STAMP previamente analizadas, repitiéndose de forma aleatoria. Durante el estudio previo a la evaluación de los experimentos, se ha observado que sólo la aplicación Labyrinth muestra un mejor rendimiento en el clúster big. Por lo tanto hemos decidido darle un un mayor peso a la probabilidad de que aparezca en nuestro sistema. En total, 12 de las 25 aplicaciones de la cola son instancias diferentes de la aplicación Labytinth. El resto de aplicaciones son elegidas al azar entre Ssca2, Kmeans, Intruder y Vacation. Para cada una de las aplicaciones, STAMP proporciona diferentes parámetros de entrada que hacen variar la carga de trabajo. Se le proporciona a las aplicaciones la configuración ++ tal, que es la que proporciona una mayor carga al sistema. Cada experimento realizado se ha ejecutado 10 veces. Los resultados aquí mostrados son el promedio de dichas ejecuciones. Los valores analizados en las pruebas son: el tiempo de cómputo (en segundos), el consumo de energía (en Julios), y el número de transacciones abortadas (en millones). El tiempo de cómputo total es equivalente al tiempo del clúster que más ha tardado de los 2 en terminar el experimento. Lo mismo ocurre cuando se realiza la medición de energía. El valor total del consumo de energía viene dado por el clúster que mayor consumo tiene. No obstante, el número de transacciones abortadas se determina sumando el número de transacciones fallidas en cada clúster.

4.2. Resultados experimentales

4.2.1. Planificador ávido

Como punto de partida, utilizamos un planificador thread-to-clúster que opera de forma ávida. La ejecución empieza con ambos clústers en reposo a la espera de una aplicación STAMP. Estos escogen, en orden de llegada, la primera que encuentran en la cola proporcionada y la ejecutan. Cuando uno de estos clúster termina la ejecución de esta aplicación, el planificador proporciona al clúster la primera aplicación que se encuentre en la cola, independientemente del rendimiento esperado. De esta forma, cada clúster se encuentra trabajando el mayor tiempo posible. Sin embargo, esta planificación

no garantiza un buen rendimiento de los núcleos tanto en términos de tiempo como de energía, o que la carga de trabajo esté equilibrada. Nuestro objetivo es comprobar que ScHeTM es capaz de mejorar al planificador ávido, mejorando el tiempo de computo requerido para ejecutar las aplicaciones así como verificar la reducción de energía y el número de transacciones abortadas.

En la Figura 4.2 podemos observar los resultados del planificador ávido. En este caso, tanto el valor del consumo de energía como el número de transacciones abortadas por el sistema (columna Total SoC) vienen determinadas por el clúster big. A primera vista parece que el clúster little no ha abortado ninguna transacción. Sin embargo, sí lo ha hecho, sólo que a menor escala (cientos de transacciones abortadas, frente a millones abortadas por el clúster big). El tiempo de cómputo, como se observa, esta determinado por el clúster little. En total 12 de las 25 aplicaciones han sido planificadas en el clúster big, mientras que las 13 restantes han sido ejecutadas en el clúster little. Señalar que aplicaciones como Labyrinth, que es más eficiente respecto al tiempo de computo en el clúster big, ha sido planificada varias veces en el clúster little. Por el contrario, aplicaciones como Kmeans e Intruder, según el análisis previo, tienen mejor rendimiento en el clúster little. No obstante, estas aplicaciones se han ejecutado exclusivamente en el clúster big. De hecho, estas dos aplicaciones son las que producen más transacciones abortadas en el sistema. Las aplicaciones Vacation y SSCA2, las cuales también son más eficientes en el clúster little, han sido planificadas de forma equilibrada por ambos clústers.

Figura 4.2: Evaluación de un planificador ávido, donde la primera aplicación disponible es la que se ejecuta, independientemente del rendimento esperado.

4.2.2. Planificador ScHeTM

El planificador ScHeTM está diseñado para realizar una distribución más inteligente de las aplicaciones STAMP que vamos a usar. Como previamente hemos visto, estas aplicaciones han sido instrumentadas y clasificadas en función de su rendimiento, consumo energético y número de transacciones abortadas. Los valores t, e y a de las funciones de

idoneidad se han establecido en 1/3 ($t = e = a = 0,33$) para proporcionar una planificación equilibrada entre los dos clústers. Tanto en esta configuración equilibrada como en las siguientes evaluadas en esta sección, se ha establecido el número límite de reintentos B en 5. Recordemos que este número de reintentos existe para que uno de los clústers no quede ocioso indefinidamente. En la Figura 4.3 podemos observar el rendimiento de las aplicaciones cuando se utiliza ScHeTM. En este caso, ScHeTM ha planificado un total de 17 aplicaciones sobre el clúster big, mientras que en el clúster little se han planificado 8 de ellas. La aplicación Labyrinth, la cual presenta un mayor rendimiento en el clúster big, ha sido planificada en este clúster en las 12 ocasiones en las que se ejecuta, creando mejor eficiencia en comparación con el planificador ávido. El resto de aplicaciones ofrecen mejor rendimiento en el clúster little ya que es el más adecuado para aumentar el rendimiento, por lo que este intenta ejecutar todas las aplicaciones que pueda. Sin embargo, algunas de ellas son planificadas en el clúster big para que este no esté ocioso. Por estos motivos, utilizando ScHeTM, el tiempo de cómputo se reduce en torno a un 20 % respecto al planificador ávido. El número de transacciones abortadas también obtiene una mejora utilizando ScHeTM, logrando un 33 % de mejora respecto al planificador ávido. El motivo de esta mejora es debido a que muchas de las aplicaciones que producen la mayoria de las transacciones abortadas (Intruder y Kmeans) son planificadas por ScHeTM en el clúster little y en este se producen menos abortos de transacciones que en el clúster big. Por contra, dado que el clúster big tiene mayor gasto energético y ScHeTM planifica la mayoría de las aplicaciones en este clúster, el consumo de energía aumenta un 7 % con respecto a la planificación ávido.

Figura 4.3: Evaluación de ScHeTM ponderando el tiempo de cómputo al 33 %, el consumo de energía al 33 % y el número de transacciones abortadas al 33 %.

4.2.3. Parametrización de ScHeTM: Tiempo de computo

Tal y como se explica en la sección del planificador, ScHeTM realiza un estudio de las características de un conjunto de aplicaciones STAMP (midiendo tiempo de cómputo,

consumo de energía y número de transacciones abortadas) para determinar si una aplicación es apta para ser ejecutada en un determinado clúster o no. Esto se realiza utilizando las funciones de idoneidad anteriormente descritas. Esas funciones pueden ponderarse para dar un mayor peso a una de las características de las aplicaciones antes mencionadas, utilizando para ello las constantes t, e, y a.

La Figura 4.4 muestra la ejecución de las aplicaciones en las que el tiempo de cómputo tiene un 70 % del peso a la hora de calcular la función de idoneidad ($t = 0,7$), mientras que el consumo de energía y el número de transacciones abortadas tienen un 15 % del peso cada una ($e = a = 0,15$).

Figura 4.4: Evaluación de ScHeTM ponderando el tiempo de cómputo al 70 %, el consumo de energía al 15 % y el número de transacciones abortadas al 15 %.

En este caso, como podemos observar, el tiempo de cómputo se reduce aun más y el trabajo de ambos clústers está más equilibrado. Aunque los clústers big y little ejecutan el mismo número de aplicaciones que en la versión original de ScHeTM (17 y 8, respectivamente), el trabajo está distribuido de forma diferente. Una de las mejoras viene gracias a la aplicación Vacation, la cual tiene mejor tiempo de ejecución en el clúster little, y es ejecutada en mayor medida por este clúster. Además, el movimiento de Vacation a este clúster también ha resultado en la reducción en el consumo de energía. En cambio, otra aplicación (Ssca2), que tiene un rendimiento similar en ambos clústers, es planificada con mayor frecuencia en el clúster big consiguiendo un mejor equilibrio en la carga de trabajo. En cuanto a las transacciones abortadas, este planificador nos deja en un punto intermedio entre el planificador ávido y la versión original de ScHeTM. En esta ocasión, la mayoría de las transacciones abortadas no recaen solo sobre uno de los clúster, sino que se reparte de forma más equitativa. Como resultado de los cambios introducidos en la planificación, esta versión de ScHeTM logra mejorar al planificador ávido en un 40 % en cuanto a tiempo de cómputo y reducir un consumo de energía en torno al 15 %.

4.2.4. Parametrización de ScHeTM: Transacciones abortadas

Otro de los parámetros que podemos ponderar es el número de transacciones abortadas. En este caso, hemos asignado un peso del 70 % a dicho parámetro ($a = 0,7$) y un 15 % tanto al tiempo de cómputo como al consumo de energía ($t = e = 15\,\%$). La Figura 4.5 muestra los resultados de la ejecución con esta parametrización.

En este caso no vemos una diferencia significativa con los resultados obtenidos en la versión original de ScHeTM. La explicación de estos resultados se debe a que el número de transacciones abortadas tiene una variación menor (porcentualmente) al cambiar de clúster en comparación con los otros parámetros. Por ejemplo, la aplicación Intruder varía en menos de un 10 % el número de transacciones abortadas al cambiar del clúster big al clúster little, pero su eficiencia energética es 5 veces mayor en el clúster little. Casos similares ocurren en la mayoría de las aplicaciones de STAMP que hemos seleccionado, y aunque queramos que el planificador reduzca de manera más eficiente las transacciones abortadas aumentando el parámetro a, no es suficiente como para contrarrestar las grandes diferencias que puedan existir en las otras características. Por otra parte las aplicaciones como Intruder y Kmeans se han ejecutado íntegramente en el clúster little. Como comentamos anteriormente estas aplicaciones son las que aportan la mayoría de las transacciones abortadas (contándose por millones). Estas aplicaciones ya se ejecutaban en el clúster little en ScHeTM con lo cual la disminución de las transacciones abortadas en este punto es nula. Por lo tanto, esta planificación se comporta de manera muy similar a ScHeTM proporcionando unos resultados parecidos.

Figura 4.5: Evaluación de ScHeTM ponderando el tiempo de cómputo al 15 %, el consumo de energía al 70 % y el número de transacciones abortadas al 70 %.

4.2.5. Parametrización de ScHeTM: Consumo energético

De la misma forma, en la Figura 4.6, se representan los resultados obtenidos al modificar nuestro planificador ScHeTM para proporcionar más peso al consumo energético 70 % ($e = 0,7$) y así intentar reducirlo. Los parámetros de tiempo de cómputo y transacciones

abortadas, las reducimos hasta un 15 % ($t = a = 0,15$). Podemos comprobar que esta parametrización tiene un efecto negativo en el parámetro a optimizar, aumentando el consumo de energía en un 6 % respecto a la versión original de ScHeTM en lugar de reducirlo como se esperaría al realizar esta operación. El motivo es que todas las aplicaciones tienen mejor rendimiento energético en el clúster little, así que este acepta todas las aplicaciones. El clúster big, de forma opuesta en primera instancia rechaza todas las aplicaciones, llegando en muchos casos al límite de rechazo $B = 5$. En este caso, el clúster big se ve obligado a aceptar las aplicaciones tras 5 reintentos sin importar la eficiencia de estas. Como resultado comprobamos que el comportamiento de ScHeTM enfocado al consumo de energía tiene un comportamiento parecido al planificador ávido, cuando únicamente uno de los clústers es óptimo para ejecutar las aplicaciones. Esto puede mejorarse teniendo en cuenta no sólo el número de reintentos para evitar que un clúster esté ocioso, sino otros parámetros como la similitud (o no) de las características de la aplicación cuando se efectúa en diferentes clústers.

Figura 4.6: Evaluación de ScHeTM ponderando el tiempo de cómputo al 15 %, el consumo de energía al 70 % y el número de transacciones abortadas al 15 %.

Capítulo 5

Conclusiones

5.1. Conclusiones de la planificación

El este proyecto presentamos ScHeTM, un planificador thread-to-clúster para CPUs multi-núcleo heterogéneas enfocado a aplicaciones multi-hilo que utilizan TM como soporte para exclusión mútua. En estas pruebas, y dependiendo de su configuración, ScHeTM consigue reducir un 40 % el tiempo de cómputo y un 15 % la energía necesaria para ejecutar un conjunto de aplicaciones del benchmark suite STAMP en comparación con un planificador ávido. Además, el planificador desarrollado también logra reducir el número de transacciones que abortan ya que planifica las aplicaciones en el clúster más adecuado para su ejecución. Otra de las ventajas de ScHeTM es la capacidad de adaptación de su parametrización y permite al usuario decidir una política de planificación para reducir el tiempo de cómputo, optimizar el uso de energía de la CPU, o minimizar el número de transacciones abortadas. No obstante, dado que ScHeTM trata de evitar que alguno de los clúster esté ocioso y todas las aplicaciones son energéticamente más eficientes en el otro, no se consiguen mejoras en el consumo de energía cuando se trata de hacerlo. El mecanismo utilizado para evitar que algún clúster esté ocioso hace que ScHeTM se comporte como un planificador ávido cuando ninguna aplicación puede aprovechar las características de dicho clúster.

El diseño y uso de ScHeTM como planificador de aplicaciones proporciona algunos puntos que merecen la pena ser discutidos. Otro modo de utilizar ScHeTM es para planificar aplicaciones que no utilicen memoria transaccional junto con aplicaciones de memoria transaccional. En el primer tipo de aplicaciones puede ser incluida el modelo considerado

que el ratio de transacciones abortadas en el clúster little respecto al big es 1 (esto es, que en ambos clúster el número de transacciones abortadas es el mismo). ScHeTM puede adaptarse a otro tipo de aplicaciones, como aquellas que usan cerrojos. Puesto que la obtención de un cerrojo por parte de un hilo tiene un impacto negativo en el tiempo de cómputo y consumo de energía, esta información puede utilizarse para sustituir al parámetro A (número de transacciones abortadas) en las funciones de idoneidad calculadas por ScHeTM.

Otro punto discutible es el estudio inicial de las aplicaciones. Para el diseño de ScHeTM, se ha partido de un estudio previo *offline* de las aplicaciones del benchmark STAMP observando tanto sus resultados a la hora de ejecutarse de forma paralela como la existencia de conflictos a la hora de ejecutar aplicaciones en sendos clúster a la vez. Si alguna aplicación nueva es añadida al sistema, o no se dispone de la información previa necesaria, ScHeTM podrá comportarse como un planificador ávido. Mientras las características de las aplicaciones son desconocidas, ScHeTM considerará ambos clústers igual de válidos para la ejecución de las aplicaciones. Conforme dicha aplicación se planifica en los distintos clúster, el planificador ScHeTM puede ir recopilando información de la ejecución de esta aplicación. Una vez se ha ejecutado en ambos clústers, ScHeTM tiene información necesaria para seleccionar el clúster idóneo para el resto de ejecuciones.

En estos experimentos hemos visto 3 escenarios posibles, cada uno con sus implicaciones. En primer lugar, si la característica a optimizar está bien diferenciada en ambos clústers para la mayoría de las aplicaciones (como el tiempo de cómputo en nuestro caso), entonces ScHeTM es capaz de ejecutar las aplicaciones de forma eficiente. En el caso de que la característica que queremos optimizar se comporte de forma similar en ambos clústers se comporta o bien como un planificador ávido, o bien se deja influenciar por los valores de otras características. Por último si la característica que queremos optimizar siempre presenta mejor rendimiento en uno de los clúster, la ScHeTM se comporta como un planificador ávido: el clúster más eficiente siempre acepta las aplicaciones proporcionadas por la cola, mientras que el otro rechazara todas estas, hasta que sea obligado para que no se quede en un estado ocioso.

5.2. Conclusiones del TFG

En este trabajo se han completado los objetivos planteados inicialmente. Se analizaron las aplicaciones del benchmark STAMP para así poder trabajar con ellas. Se ha propuesto un planificador ScHeTM el cual, según la configuración de parámetros que se le introduzcan, obtiene mejor rendimiento que un planificador ávido según los datos analizados.

Para la realización de este trabajo de fin de grado se han tenido que adquirir conocimientos tanto de Python como de Bash. Estos conocimientos nos han servido tanto para la realización de los experimentos como para generar gráficas. Además, se han adquirido conocimientos de la arquitectura big.LITTLE de ARM y sobre el funcionamiento de memoria transaccional. Se ha realizado también un estudio del arte previo sobre eficiencia energética de los procesadores. Finalmente cabe destacar el aprendizaje del lenguaje Latex para realizar esta memoria.

5.3. Trabajo futuro

Como trabajo futuro proponemos la introducción de mejoras en el mecanismo utilizado para evitar que los clúster permanezcan ociosos. Por ejemplo, en lugar de basarse únicamente en el número de reintentos, también puede incluirse información sobre la idoneidad de la aplicación. Otra manera de mejorar este proceso, es seleccionar la aplicación que mejor se comporte dentro de una ventana de aplicaciones, escogiendo así entre la mejor de un subgrupo, antes que haya que planificar de una forma aleatoria al llegar al máximo de reintentos. Además, planteamos mejorar la evaluación incluyendo aplicaciones tanto secuenciales como otras paralelas que no utilicen memoria transaccional. El objetivo de este experimento es comprobar como se adapta ScHeTM a distintas situaciones con más variedad de aplicaciones en ejecución. Por último, pretendemos estudiar la inclusión de un planificador thread-to-core que permita seleccionar el número de hilos de ejecución más adecuado para cada aplicación, además de planificar varias aplicaciones en un mismo clúster.

Bibliografía

[1] ARM big.LITTLE technology. `http://www.arm.com/products/processors/technologies/biglittleprocessing.php`. Accedido: 21-6-2017.

[2] ODROID — Hardkernel. `http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127`. Accedido: 21-6-2017.

[3] Standard Performance Evaluation Corporation, *SPECjbb2000 Benchmark*. `http://www.spec.org/jbb2000`, 2000.

[4] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini. An integrated open framework for heterogeneous MPSoC design space exploration. In *Proceedings of the Design Automation Test in Europe Conference*, volume 1, pages 1–6, March 2006.

[5] David A. Bader and Kamesh Madduri. *High Performance Computing – HiPC 2005: 12th International Conference, Goa, India, December 18-21, 2005. Proceedings*, chapter Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors, pages 465–476. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[6] A. Baldassin, J. P. L. de Carvalho, L. A. G. Garcia, and R. Azevedo. Energy-performance tradeoffs in software transactional memory. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 147–154, Oct 2012.

[7] A. Baldassin, F. Klein, G. Araujo, R. Azevedo, and P. Centoducatte. Characterizing the energy consumption of software transactional memory. *IEEE Computer Architecture Letters*, 8(2):56–59, Feb 2009.

[8] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 29–40, New York, NY, USA, 2006. ACM.

[9] Quan Chen and Minyi Guo. Adaptive workload-aware task scheduling for single-isa asymmetric multicore architectures. *ACM Trans. Archit. Code Optim.*, 11(1):8:1–8:25, February 2014.

[10] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Distributed Computing*, volume 4167, pages 194–208. Springer, 2006.

[11] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Trans. on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[12] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 237–246, 2008.

[13] Cesare Ferri, Amber Viescas, Tali Moreshet, Iris Bahar, and Maurice Herlihy. Energy implications of transactional memory for embedded architectures. *Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM'08)*, 2008.

[14] Epifanio Gaona-Ramírez, Rubén Titos-Gil, Juan Fernández, and Manuel E Acacio. Characterizing energy consumption in hardware transactional memory systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 9–16. IEEE, 2010.

[15] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. *Information Security Applications: 5th International Workshop, WISA 2004, Jeju Island, Korea, August 23-25, 2004, Revised Selected Papers*, chapter Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading, pages 188–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[16] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93)*, pages 289–300, 1993.

[17] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, Sept 1961.

[18] Simone Libutti, Giuseppe Massari, and William Fornaciari. Co-scheduling tasks on multi-core heterogeneous systems: An energy-aware perspective. *IET Computers & Digital Techniques*, 10:77–84(7), March 2016.

[19] L. Lugini, V. Petrucci, and D. Mossé. Online thread assignment for heterogeneous multicore systems. In *2012 41st International Conference on Parallel Processing Workshops*, pages 538–544, Sept 2012.

[20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb 2002.

[21] Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46, Sept 2008.

[22] Tali Moreshet, R Iris Bahar, and Maurice Herlihy. Energy-aware microprocessor synchronization: Transactional memory vs. locks. *Fourth Annual Boston-Area Architecture Workshop*, page 21, 2006.

[23] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *2006 IEEE International Symposium on Workload Characterization*, pages 182–188, Oct 2006.

[24] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548 – 585, 1995.

[25] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero. Clock gate on abort: Towards energy-efficient hardware transactional memory. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, May 2009.

[26] Emilio Villegas, Alejandro Villegas, Angeles Navarro, Rafael Asenjo, Yash Ukidave, and Oscar Plata. Energy efficiency of software transactional memory in a heterogeneous architecture. In *8th Workshop on the Theory of Transactional Memory (WTTM 2016 co-located with PODC 2016)*, 2016.

[27] Wikipedia. Red bayesiana — wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Red_bayesiana&oldid=90838441, 2016. Accedido: 21-6-2017.