ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
Grado en Ingeniería de Computadores

Análisis energético de Memoria Transaccional Software en procesadores de bajo consumo

Energy consumption analysis of Software Transactional Memory on low power processors

Realizado por
Emilio Villegas Fernández
Tutorizado por
Rafael Asenjo Plaza
y
Alejandro Villegas Fernández
Departamento
Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Julio de 2016

Fecha defensa:
El Secretario del Tribunal

Resumen: Tradicionalmente, en programas multi-hilo, los mecanismos de exclusión mútua se implementan mediante el uso de cerrojos, que garantizan que únicamente uno de los hilos accede a la sección de código en la que se manipulan dichos datos. La Memoria Transaccional (TM) es una alternativa a los cerrojos enfocada a obtener un mejor rendimiento y proporcionar mayor facilidad de programación. TM puede implementarse por software o hardware, siendo las alternativas software más convenientes en términos de flexibilidad y portabilidad. Trabajos recientes han analizado y propuesto soluciones de TM en las que el consumo energético es un factor a tener en cuenta. Buena parte de estos trabajos se realizan sobre simuladores de hardware o sobre procesadores orientados a la computación de altas prestaciones; los estudios sobre hardware físico orientado al bajo consumo no han sido explorados aún. Encontrar soluciones TM software energéticamente eficientes en procesadores actuales de bajo consumo, como pueden ser los incorporados en dispositivos móviles y empotrados, es un campo de investigación abierto.
Este proyecto realiza el análisis energético de una librería TM software existente en el mercado sobre un dispositivo de bajo consumo basado en procesadores ARM. El principal objetivo es proporcionar métricas de rendimiento y energía sobre el comportamiento energético de dicha librería en el procesador mencionado. Un objetivo adicional es la instrumentación de benchmarks de prueba, lo cual proporciona una herramienta indispensable para realizar futuras investigaciones en el área.

Palabras claves: Eficiencia Energética, Memoria Transaccional. Procesadores Multi-núcleo

Abstract: Traditionally, in multi-threaded programs, mutual exclusion mechanisms are implemented using locks. Locks guarantee that only one thread accesses the section of code that manipulates shared data. Transactional Memory (TM) is an alternative to locks that intends to obtain better performance and ease programming. TM can be implemented in software or hardware. Software implementations are more convenient in terms of portability and flexibility. Recent research propose TM solutions where energy consumption is a key factor to take into account. Most of the research is done using simulators or high-performance processors, but not on low-power processors. The design of energy-efficient TM solutions for low-power processors (such as the ones present in mobile devices) is an open research field.
This project analyzes a software TM library running on a device that features a low-power ARM processor using a set of well-known TM applications. The main goal is to provide energy and performance metrics of the library and the applications on such processor. An additional goal is to provide the instrumentation of the applications, which can be used for future research projects.

Keywords: Energy efficiency, Transactional Memory, Multi-core Processors.

Índice general

Capítulo 1

Introducción

A lo largo de este primer capítulo de la memoria se explican las motivaciones que nos han llevado a realizar este trabajo de fin de grado, los objetivos del trabajo, el estado del arte, las tecnologías que se utilizan y la forma en la que se estructura la memoria.

1.1. Motivación

Actualmente podemos encontrar dispositivos móviles y empotrados en una gran variedad de entornos. Estos dispositivos tienen una serie de restricciones. En ocasiones dependen de una fuente de alimentación integrada, por ejemplo una batería, lo que hace que el consumo deba ser el menor posible para prolongar su duración. Otra restricción es la temperatura. En casos como los teléfonos móviles, la temperatura debe ser baja para un uso cómodo por parte del usuario. Además de cumplir con los casos anteriores, el dispositivo debe presentar un rendimiento aceptable para las tareas que se le encarguen.

Fabricantes de procesadores como ARM se están centrando en la creación de procesadores de bajo consumo energéticamente eficientes. Mediante la creación de la arquitectura multi-núcleo big.LITTLE [1] logran mantener un equilibrio en el consumo de energía, temperatura y rendimiento. Esta arquitectura incorpora dos conjuntos de núcleos: unos más potentes orientados al rendimiento y otros núcleos con menor capacidad de cálculo pero energéticamente más eficientes. Llamaremos a estos conjuntos de núcleos *clusters*, concretamente *cluster big* y *cluster little*. Los núcleos de ambos clusters comparten la misma ISA y el acceso a la misma memoria principal, esto permite la ejecución de una aplicación en cualquiera de los clusters.

Para aprovechar las capacidades de estos procesadores, las aplicaciones deben ser programadas con un diseño multi-hilo. Esto puede suponer un reto para los programadores al tener que sincronizar los hilos de ejecución para acceder a datos compartidos. La porción de código que accede a estos datos, conocida como sección crítica, debe garantizar la exclusión mutua en su acceso por parte de los hilos de ejecución. La exclusión mutua se implementa de forma tradicional utilizando cerrojos. Un cerrojo de grano grueso ga-

rantiza que ningún hilo de ejecución entre en la sección crítica si algún hilo se encuentra ejecutándola. Este tipo de cerrojos tiene un gran impacto en el rendimiento en el caso de que se necesite acceder a la sección crítica frecuentemente aunque se acceda a posiciones de memoria diferentes. Los cerrojos de grano fino obtienen, en estos casos, un mejor rendimiento. Su uso consiste en proteger individualmente las posiciones de memoria de forma que se pueda acceder de forma paralela a la sección crítica únicamente si van a modificarse datos disjuntos. Esto supone un mayor esfuerzo de programación, además de realizar la difícil tarea de comprobar que es correcta la implementación y que no existen *deadlocks* ni *livelocks*.

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

(a) Cerrojo de grano grueso

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

(b) Cerrojo de grano fino

Figura 1.1: Ejemplos de código de cerrojos de grano grueso y grano fino. Suponemos que GlobalLock, GlobalArray y Locks son variables globales y compartidas.

Como vemos en la Figura 1.1a, un cerrojo de grano grueso es suficiente para proteger toda la estructura de datos, pero se produce un acceso serializado a la sección crítica. En la Figura 1.1b, se utiliza un cerrojo por posición de memoria a proteger (cerrojos de grano fino). Si dentro de la función se requieren modificar varias posiciones de memoria, la solución de grano grueso sería válida, pero ineficiente. En el caso de cerrojos de grano fino se podrían tratar de obtener todos los cerrojos necesarios, pero se requeriría un mayor esfuerzo de programación para evitar deadlocks. Por ejemplo, el hilo de ejecución 0 puede tratar de obtener los cerrojos que protegen a las direcciones de memoria A y B, en ese orden. Al mismo tiempo, el hilo de ejecución 1 trata de obtener los cerrojos de las posiciones B y A, también en ese orden. Una posibilidad que el hilo 0 obtenga el cerrojo de la posición A y que, al mismo tiempo, el hilo 1 obtenga el cerrojo de la posición B. A continuación, el hilo 0 se queda esperando a que el cerrojo de la posición B esté libre, pero esto no ocurrirá puesto que ha sido tomado por el hilo 1. Del mismo modo, el hilo 1 quedará en espera a que el cerrojo de la posición A esté libre, pero esto no ocurrirá porque ha sido tomado por el hilo 0. De esta forma, ningún hilo puede progresar y se produce un deadlock.

El uso de Memoria Transaccional (TM) [23] se propone como una alternativa optimista a los cerrojos para implementar secciones críticas. La idea de las transacciones es la de proporcionar al programador una interfaz similar a la de cerrojos de grano grueso, pero con un rendimiento similar al de los cerrojos de grano fino. Esto se consigue permitiendo la ejecución paralela de transacciones y registrando los accesos a memoria. En el caso de

que dos o más transacciones accedan a memoria y creen conflicto, sólo una de ellas puede continuar. Las demás deben deshacer los cambios especulativos en memoria y reiniciar su ejecución. La implementación de una TM debe asegurar la corrección, debe implementar mecanismos que eviten deadlocks y debe garantizarse que cualquier transacción puede (eventualmente) terminar.

Existen varias propuestas de soluciones TM en hardware [22] que están siendo implementadas actualmente por algunos procesadores multi-núcleo [26]. Además, se han propuesto numerosas soluciones TM software implementadas en forma de librería [12, 13, 15, 16, 17]. Otro tipo de soluciones TM son las híbridas, que combinan las características de las soluciones TM hardware y software [14].

```
void Insert(Object elem, Int pos)
{
  TM_BEGIN();
  //Sección crítica
  GlobalArray[pos] = elem;
  TM_COMMIT();
}
```

Figura 1.2: Función Insert implementada utilizando TM

En la Figura 1.2 se puede ver que la implementación de la función utilizando interfaz TM es muy similar a la de los cerrojos de grano grueso vista en la Figura 1.1a. El uso de TM hace que pueda obtener un rendimiento similar a la implementacion de cerrojo de grano fino y, además, aporta una implementación libre de deadlocks.

Uno de los objetivos importantes para los próximos años es la implementación de TM en procesadores multi-núcleo heterogéneos de bajo consumo. Actualmente existen análisis y propuestas TM que se centran en el consumo de energía de los procesadores sobre los que se ejecutan [20, 19, 33, 8, 7, 25, 37]. Estas propuestas se realizan sobre procesadores homogéneos, sin tener en cuenta a los procesadores multi-núcleo heterogéneos. Además, están realizadas sobre simuladores en los que se estima el consumo energético.

1.2. Objetivos

En este trabajo se analiza el consumo energético de una librería TM ejecutada sobre una plataforma que cuenta con un procesador multi-núcleo heterogéneo con arquitectura big.LITTLE. En primer lugar se realiza un estado del arte sobre diferentes implementaciones TM, ya sean software o hardware. A continuación, adaptamos la librería *software* TinySTM [16, 17] y las diferentes aplicaciones de STAMP *benchmark suite* [31] a la arquitectura ARM big.LITTLE presente en el dispositivo ODROID-XU3 [2]. Para realizar el análisis de energía se implementa una librería con la que instrumentar las aplicaciones de STAMP. Esta librería obtiene los datos de energía directamente de los sensores disponibles

en el dispositivo. Posteriormente, con los datos obtenidos en las diferentes aplicaciones, se realizará un análisis del rendimiento de ambos clusters por separado y una comparativa entre ambos.

1.3. Estado del arte

Recientemente se han realizado análisis del consumo energético de TM sobre distintos tipos de procesadores. Gaona *et al.* [20] analizan el consumo energético de dos TM por hardware. Uno de los sistemas TM hardware analizado es *LogTM* [32] en el que los nuevos valores se almacenan directamente en memoria y los valores anteriores se almacenan en un *undo-log*. Esto es, antes de hacer una escritura, el hardware guarda automáticamente el valor antiguo del bloque cache para posteriormente poder deshacer el cambio, en caso de que sea necesario. De esta manera logra que los *commits* sean rápidos, mientras que los *aborts* son más lentos. Para la detección de conflictos se aprovecha al máximo el protocolo de coherencia, observando las solicitudes y las invalidaciones para bloques de transacciones de escritura y lectura.

Otro sistema TM hardware analizado es *Scalable TCC* [11]. En esta implementación, las escrituras de transacciones se hacen en un *buffer* aparte, que almacena los valores nuevos y que sólo se escribirán en la memoria cuando se haga un commit. Estas pruebas se han realizado en el simulador *Wisconsin GEMS toolset* [30] junto con *Virutech Simics* [29] en los que simulaban una arquitectura tipo SPARC. Proponen la serialización dinámica de transacciones en hardware, con el objetivo de reducir el consumo de energía [19]. Esto se logra tratando de minimizar el trabajo especulativo cuando en las transacciones encuentran conflictos y deben abortar.

Moreshet *et al.* [33] y Ferri *et al.* [18] realizan análisis energéticos del TM por hardware introducido en [23]. Estos análisis se han realizado en simuladores. Concretamente, han utilizado *Virutech Simics* simulando una arquitectura tipo SPARC. Otro simulador utilizado es MPARM *simulation framework* [5] simulando una arquitectura ARMv7. Los resultados muestran una mejora en el consumo energético comparado con el uso de soluciones de exclusión mutua utilizando cerrojos.

Baldassim *et al.* [8, 7] analizan la librería de TM software TL2 [13] sobre procesadores homogéneos de bajo consumo basados en la arquitectura ARMv7 mediante el uso de simuladores. Proponen una solución basada en la variación dinámica del voltaje y el escalado de la frecuencia del procesador con el objetivo de reducir el consumo de energía.

Klein *et al.* [25] analizan diferentes variaciones de la librería STM TL2. Proponen varias soluciones basadas en una estrategia de memoria *scratch-pad* [9] con el fin de reducir el consumo. El análisis se realiza en simuladores, utilizando una arquitectura ARMv7.

Sanyal *et al.* [37] proponen el uso de técnicas de *clock-gating* para reducir el consumo de energía en TM hardware y mejorar su rendimiento. Utilizan una versión modificada del simulador *M5 full-system* [10], para simular la arquitectura Alpha 21264 con soporte

para Scalable TCC.

En el estudio previo, no se han encontrado análisis sobre procesadores con núcleos heterogéneos. El dispositivo ODROID-XU3 posee una arquitectura heterogénea big.LITTLE de ARM pensada para adaptarse a todo tipo de escenarios. Incorpora núcleos de alto rendimiento y otros de bajo consumo. Así, permite ahorrar hasta un 75 % de energía en escenarios de bajos requerimientos de cómputo y incrementar hasta un 40 % el rendimiento en aplicaciones multi-hilo.

El procesador que incorpora es un Samsung Exynos 5422. El procesador tiene 4 núcleos Cortex-A7 en el cluster little con 512Kbytes de cache L2 y 4 núcleos Cortex-A15 en el cluster big con 2Mbytes de cache L2. Incorpora una GPU MALI-T628 y 2 Gbytes de memoria DDR3. El sistema operativo instalado es Linux odroid 3.10.59+.

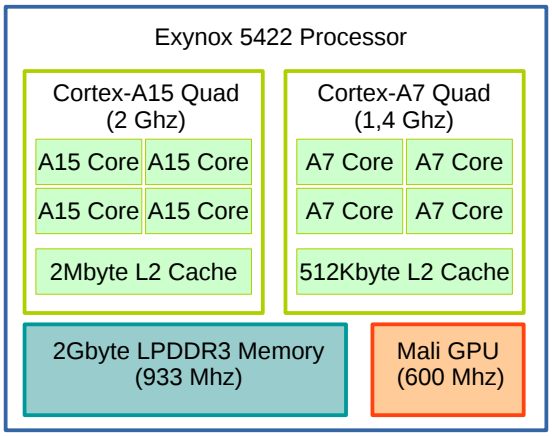

Figura 1.3: Diagrama del procesador Exynos 5422 disponible en la plataforma ODROID-XU3.

1.4. Tecnologías utilizadas

1.4.1. STAMP

Stanford Transactional Applications for Multi-Processing (STAMP) es un conjunto de aplicaciones muy popular para la evaluación de distintos sistemas de TM. Incluye ocho aplicaciones de diferentes dominios, treinta configuraciones y varios conjuntos de datos de entrada. Para facilitar la portabilidad de estas aplicaciones, se han escrito en lenguaje C. Las llamadas a las funciones de TM se realizan mediante macros de C, que hacen que sean fácilmente portable a otros sistemas. A continuación se muestra la lista de aplicaciones incluidas en STAMP:

- bayes: implementa un algoritmo de aprendizaje de la estructura de una red Bayesiana [38]. La implementación mediante transacciones es más simple que utilizando

locks. Esta aplicación utiliza casi todo su tiempo de ejecución en largas transacciones, que tienen conjuntos grandes de lectura y escritura.

- genome: toma un gran número de secuencias ADN y las combina para formar el genoma original. El algoritmo tiene dos fases. En la primera fase, se crea un conjunto de secuencias únicas. En la segunda fase, cada hilo intenta eliminar una secuencia del conjunto de secuencias no combinadas y añadirlas a su partición de secuencias combinadas.

- intruder: utiliza un sistema de detección de intrusos en una red (*Network Intrusion Detection System*(NIDS)). La implementación proporcionada en STAMP emula el diseño de 5 NIDS descrito en [21]. Los paquetes son procesados en paralelo en tres fases: captura, reconstrucción y detección. En la implementación con transacciones estas se incluyen en las fases de captura y reconstrucción.

- kmeans: utiliza el algoritmo de agrupación *K-Means* que agrupa objetos de un espacio de N dimensiones en K clusters. La implementación proporcionada por STAMP está tomada de [34].

- labyrinth: implementa una variación del algoritmo de Lee [28]. La estructura de datos principal es una cuadrícula de 3 dimensiones que representa un laberinto. Cada hilo escoge un punto inicial y otro final que deben conectar por caminos, utilizando puntos adyacentes de la cuadrícula. Las transacciones se utilizan para calcular el camino y añadirlo a la cuadrícula principal. Los conflictos ocurren cuando dos caminos calculados se cruzan.

- ssca2: (*Scalable Synthetic Compact Applications 2*) [6] está compuesta por 4 *kernels*. En STAMP se utiliza el kernel 1 al ser adecuado para TM. Este kernel construye una estructura de datos en forma de grafo utilizando arrays de adyacencia y otros arrays auxiliares. En la versión transaccional los hilos añaden nodos al grafo en paralelo. Utiliza transacciones para proteger el acceso al array de adyacencia.

- vacation: implementa un sistema *online* de transacciones similar a SPECjbb2000 [4] pero simulando un sistema de reservas de viajes. Durante la ejecución varios clientes realizan acciones que interactúan con la base de datos.

- yada: (*Yet Another Delaunay Application*) implementa el algoritmo de Ruppert para la mejora de redes Delaunay [36]. La estructura de datos utilizada es un grafo que almacena todas las redes de triángulos, un conjunto que almacena los límites de las redes y una cola de tareas que almacena los triángulos que deben ser mejorados. El acceso a la cola de trabajo utiliza transacciones.

Como el objetivo es estresar las diferentes características del TM, cada aplicación tiene unas características de longitud de transacción, tamaño de los conjuntos de lectura

y escritura y tiempo empleado en las 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
bayes	Larga	Grandes	Alto
genome	Media	Medios	Alto
intruder	Corta	Medios	Media
kmeans	Corta	Pequeños	Bajo
labyrinth	Larga	Grandes	Alto
ssca2	Corta	Pequeños	Bajo
vacation	Media	Medios	Alto
yada	Larga	Grandes	Alto

Tabla 1.1: Características de las aplicaciones STAMP.

STAMP también incluye, para cada aplicación, un conjunto de configuraciones recomendadas. Kmeans y vacation disponen además de entradas de baja y alta contención, reconocibles por el sufijo *-high* o *-low*. El tamaño de los conjuntos indicados viene indicado por los sufijos + y ++.

1.4.2. TinySTM

TinySTM[1] [16, 17] es una librería de TM por software. Es una implementación del algoritmo *Lazy Snapshot Algorithm* (LSA) [16] que trabaja a nivel de palabra de memoria.

TinySTM incorpora varios modos para la gestión de transacciones. Estos modos varían en el momento de adquisición de los cerrojos y cuando la memoria principal se actualiza. Los modos son los siguientes:

- El modo *write-back* en el que los cambios especulativos en memoria se almacenan en un buffer hasta el final de la transaccion y en ese momento, si no existen conflictos, se hacen definitivos.
- El modo *write-through* en el que los cambios especulativos se guardan directamente en memoria y los valores antiguos en un *log*, en caso de conflicto los datos antiguos deben restaurarse.
- El modo *commit-time locking* en el que se utilizan cerrojos durante el final de la transacción para poder proteger las posiciones que están siendo actualizadas.
- El modo *encounter-time locking* en el que se utilizan cerrojos en cada acceso a memoria en lugar de al final de la transacción.

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

TinySTM tiene opciones de compilación distintas para utilizar los 4 modos anteriores. Por defecto utiliza la opción de compilación WRITE_BACK_ETL que utiliza los modos write-back con encounter-time locking.

1.4.3. Lenguajes de script

El primer lenguaje de scripts utilizado en este trabajo es Python [3]. Python es un lenguaje interpretado multiplataforma. Fue creado a finales de los 80 por Guido van Rossum y administrado por Python Software Foundation. Permite a los programadores adoptar varias formas de programación: programación orientada a objetos, programación imperativa y programación funcional. Es un lenguaje de alto nivel con una sintaxis que permite tener códigos fáciles de leer. Actualmente existen varias versiones de Python, la versión 2 y versión 3. Se siguen trabajando sobre ambas versiones ya que la versión 3 no es compatible con la 2 y surgió para modificar elementos del diseño de Python. En la actualidad su uso se extiende desde pequeños scripts a servidores que trabajan de forma ininterrumpida. Existen diversas librerías para Python, concretamente matplotlib [24] es utilizada para producir figuras como pueden ser gráficas de una forma sencilla.

Otro lenguaje utilizado es Bash (Bourne Again SHell). Bash es una *Shell* de Unix y un lenguaje de comandos, escrito por Brian Fox para el proyecto GNU. Bash surge como reemplazo del Bourne Shell. Es capaz de ejecutar todas sus instrucciones sin ninguna modificación. Bash permite además el cálculo de enteros, redirecciones de entrada y salida, uso de expresiones regulares y caracteres especiales.

1.4.4. C y C++

El lenguaje C fue desarrollado en un principio por Dennis M. Ritchie. Es un lenguaje imperativo, diseñado para ser compilado. La primera estandarización del lenguaje fue en ANSI (ANSI C) y posteriormente como estándar ISO. Con esto, si los programas creados siguen el estándar, serán portables entre plataformas y arquitecturas. Es un lenguaje débilmente tipado. Dispone de estructuras de lenguajes de alto nivel pero a la vez permite el control a muy bajo nivel. Permite el acceso a memoria de bajo nivel mediante punteros.

El lenguaje C++ fue diseñado a mediados de los 80 por Bjarne Stroustrup. Su creador quería incluir en el lenguaje C mecanismos para la manipulación de objetos. Por lo tanto es un lenguaje de programación multiparadigma. C++ permite redefinir operadores y crear nuevos tipos que se comporten como tipos fundamentales.

1.4.5. LaTeX

LaTeXes un sistema de composición de textos. Suele ser utilizado en la creación de artículos, tesis y libros técnicos. LaTeXpermite al autor del documento centrarse únicamente en el contenido de este, haciendo que no tenga que preocuparse por detalles del

formato. En la elaboración de un documento existen dos etapas: en la primera mediante un editor de texto, se escriben las ordenes y el contenido que se quiera escribir. En la segunda, hay que procesar este archivo para mostrar la salida correspondiente.

1.5. Estructura de la memoria

En este primer capítulo, introducción, se han tratado los puntos de motivación y objetivos del trabajo. Además, se realiza el estudio del estado del arte donde se analiza el trabajo relacionado y los antecedentes. Finalmente, en el apartado de tecnologías utilizadas, se explican los diferentes lenguajes utilizados, librería de TM TinySTM y los benchmarks de STAMP.

En el capítulo 2, desarrollo, se trata el punto de partida de la librería de energía, pasando por toda la adaptación hasta tener una librería portable con la que instrumentar el código. A continuación, se trata el proceso de instrumentación de STAMP utilizando la librería, explicando la compilación tanto de la librería, TinySTM y STAMP. Además, se llevan a cabo pruebas para comprobar el funcionamiento de la instrumentación realizada.

En el capítulo 3, evaluación, se explica la metodología experimental, el proceso que se ha seguido para realizar los diferentes experimentos. Además, se presentan los resultados obtenidos en los experimentos.

En el capítulo 4, cerramos esta memoria con las conclusiones de los experimentos, del trabajo de fin de grado y el trabajo futuro.

Capítulo 2

Desarrollo

2.1. Librería de energía

2.1.1. Punto de partida

Como punto de partida tenemos la librería *Energy Meter v1.0*, desarrollada en el Departamento de Arquitectura de Computadores en un trabajo previo. La librería proporciona acceso a los datos obtenidos en los sensores INA231[1] del dispositivo. El lenguaje de programación en el que se ha desarrollado es C. La librería incluye la definición de dos tipos de *struct*. El primero de ellos, *struct energy_sample*, almacena los datos del medidor, así como la energía consumida total. En el segundo, *struct em_t*, se almacenan las mediciones instantáneas de energía. Es importante saber que el medidor recoge muestras periódicamente. Si el tiempo de muestreo es mayor que el tiempo de ejecución del código instrumentado, la medición será incorrecta. El tiempo mínimo para el muestreo, que además es el valor por defecto de la librería, es de 50 milisegundos.

Para instrumentar una aplicación mediante el uso de esta librería, se deben utilizar las siguientes funciones:

- struct energy_sample * energy_meter_init(int sample_rate, int debug), donde *sample_rate* es el periodo de tiempo entre muestras y la opción *debug* hace que se muestre información adicional durante la ejecución. Esta función inicializa el medidor de energía.

- void energy_meter_start(struct energy_sample *sample) lanza el hilo que se encarga de muestrear la energía consumida y, desde este momento, empieza a contar el tiempo de muestreo.

- void energy_meter_stop(struct energy_sample *sample) detiene la ejecución del hilo que muestrea y es en este momento cuando se guardan los datos totales muestreados en sample.

[1] http://www.ti.com/lit/ds/symlink/ina231.pdf

- void energy_meter_printf(struct energy_sample *sample1, FILE * fout) se utiliza para, una vez detenido el muestreador, escribir en un fichero o en pantalla la información del cluster little, cluster big, memoria, GPU y el tiempo de ejecución.

- void energy_meter_destroy(struct energy_sample *sample) se encarga de liberar correctamente la memoria utilizada. Una vez completado todo el proceso de muestreo debe llamarse al final de la aplicación.

- void energy_meter_read(struct energy_sample *sample, struct em_t * read) toma una muestra de la energía consumida en todo el chip en el mismo instante de la llamada.

- void energy_meter_diff(struct energy_sample *sample, struct em_t * start_diff) toma una muestra de la energía consumida y calcula la diferencia con la que recibe como parámetro, y sustituye el valor de la misma. Es útil para medir la energía entre dos puntos del código.

- void energy_meter_read_printf(struct em_t * read_or_diff , FILE *fout) muestra la información de la energía consumida en el chip en las muestras proporcionadas como parámetro.

Además de las funciones detalladas, la librería contiene varias macros en C para sumar dos muestras de energía y para inicializar los valores de las muestras de energía a cero. En todas las funciones anteriores, el parámetro de entrada *struct energy_sample * sample* se refiere a la estructura que contiene los datos del medidor correctamente inicializados. En la Figura 2.1 se muestra un diagrama de las funciones aportadas por la librería.

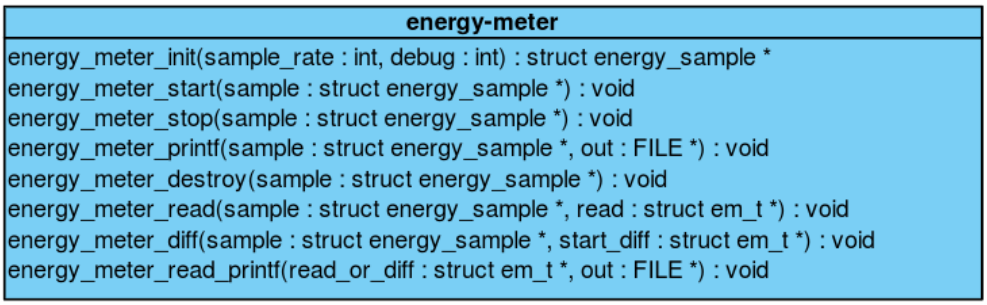

Figura 2.1: Diagrama de la librería Energy Meter v1.0

2.1.2. Interfaz genérica C++

Pensando en la adaptación de la librería de energía, escrita en C, se decide realizar una implementación utilizando una interfaz genérica mediante clases de C++. Aunque las aplicaciones a instrumentar (STAMP) están escritas en C, el objetivo es proporcionar una interfaz C++ para dar soporte a aplicaciones escritas en este lenguaje y facilitar el

trabajo futuro. Además, la creación de esta interfaz permite que, una vez instrumentadas las aplicaciones, sólo haya que cambiar la implementación de la interfaz para que la instrumentación siga funcionando. Por ejemplo, para utilizar un procesador Intel, habría que crear una implementación de esta interfaz que acceda a los contadores de energía proporcionados por este fabricante, pero no habría que modificar el código instrumentado. No es obligatoria la implementación de todos estos métodos, pero debe observarse cuales son las utilizadas en el código ya instrumentado para su correcto funcionamiento. La interfaz define los siguientes métodos:

- IEnergy(): constructor de la clase, normalmente está vacío. Se proporciona un método para crear instancias de la clase.
- static IEnergy * create(): este método se debe implementar para devolver una instancia de la clase que implementa la interfaz. Al ser estático no es necesario que tengamos el objeto creado para realizar la llamada. Su acceso es a través de la interfaz y no a través de la clase implementada. Con esto se hace que el código instrumentado no se tenga que modificar cuando cambiemos de librería.
- void energy_initialize (): este método se debe implementar para inicializar las estructuras y variables de nuestra librería.
- void energy_start (): este método se debe implementar para que el muestreador comience a funcionar.
- void energy_stop(): este método se debe implementar para detener el muestreador.
- void energy_destroy(): este método se debe implementar para liberar memoria al final de la ejecución.
- void energy_printf(FILE * out): este método se debe implementar para escribir en out la información del muestreador.
- void energy_sample_start(): este método se debe implementar para comenzar una medición parcial, por ejemplo para medir el consumo antes y después de una llamada a una función.
- void energy_sample_stop(int acc): este método se debe implementar para detener la medición parcial, debe llamarse siempre después de energy_sample_start(). El parámetro acc se puede utilizar para acumular la diferencia de consumo en la variable dando el valor 1, o por el contrario, para sustituir el valor dando el valor 0.
  Si se desea medir el valor de varios trozos de código y obtener un total, se deben acumular todos los consumos leídos (ver Figura 2.2a). Si se desea solamente medir un trozo, se debe sustituir el valor de la variable anterior (ver Figura 2.2b)
- void energy_sample_clear(): este método se debe implementar para inicializar en cualquier momento el valor del medidor parcial.
- double energy_sample_getCPU(): este método se debe implementar para obtener en cualquier momento la energía consumida por la CPU de una medición parcial.
- double energy_sample_getGPU(): este método se debe implementar para obtener en cualquier momento la energía consumida por la GPU de una medición parcial.

- double energy_sample_getMEM(): este método se debe implementar para obtener en cualquier momento la energía consumida por la memoria de una medición parcial.
- double energy_sample_getTIME(): este método se debe implementar para obtener en cualquier momento el tiempo de ejecución de una medición parcial.

```
// Creación de la instancia
IEnergy *e = IEnergy::create();
// Inicialización
e->energy_initialize();
// Lanzar el muestreador
e->energy_start();
// Aquí comienza primera zona
// de código a muestrear
e->energy_sample_start();
/*
 * Sección de código a muestrear
 */
// Terminamos la primera zona
// de muestreo y sustituye
// el valor anterior
e->energy_sample_stop(0);
/*
 * Código que no se quiere muestrear
 */
// Aquí comienza segunda zona
// de código a muestrear
e->energy_sample_start();
/*
 * Sección de código a muestrear
 */
// Terminamos la segunda zona
// de muestreo y acumulamos
// este nuevo consumo al ya
// leido anteriormente
e->energy_sample_stop(1);
// Se muestra el consumo acumulado
e->energy_sample_printf(stdout);
// Se para el muestreador
e->energy_stop();
// Se libera memoria
e->energy_destroy();
```

(a) Medición del consumo de varios trozos de código de forma acumulativa.

```
// Creación de la instancia
IEnergy *e = IEnergy::create();
// Inicialización
e->energy_initialize();
// Lanzar el muestreador
e->energy_start();
// Aquí comienza primera zona
// de código a muestrear
e->energy_sample_start();
/*
 * Sección de código a muestrear
 */
// Terminamos la primera zona
// de muestreo y sustituye
// el valor anterior
e->energy_sample_stop(0);
// Se muestra el consumo de este
   trozo
e->energy_sample_printf(stdout);
/*
 * Código que no se quiere muestrear
 */
// Aquí comienza segunda zona
// de código a muestrear
e->energy_sample_start();
/*
 * Sección de código a muestrear
 */
// Terminamos la segunda zona
// de muestreo
e->energy_sample_stop(0);
// Se muestra el consumo de esta
   zona
e->energy_sample_printf(stdout);
// Se para el muestreador
e->energy_stop();
// Se libera memoria
e->energy_destroy();
```

(b) Medición del consumo de varios trozos de código sin acumular.

Figura 2.2: Ejemplo del uso de energy_sample_start y energy_sample_stop para instrumentar una o varias secciones de código.

Los ejemplos de la Figura 2.2 muestran varios códigos. En uno de ellos (Figura 2.2a), se miden varias secciones de código diferentes y se muestra la suma el consumo en el chip

en ambas secciones. En el otro (Figura 2.2b), se miden por separado dos secciones, y al final de cada sección se muestra el consumo en el chip. Es importante observar el uso del parámetro acc del método energy_sample_stop en ambos casos.

A continuación en la Figura 2.3 se presenta el diagrama de la interfaz IEnergy.

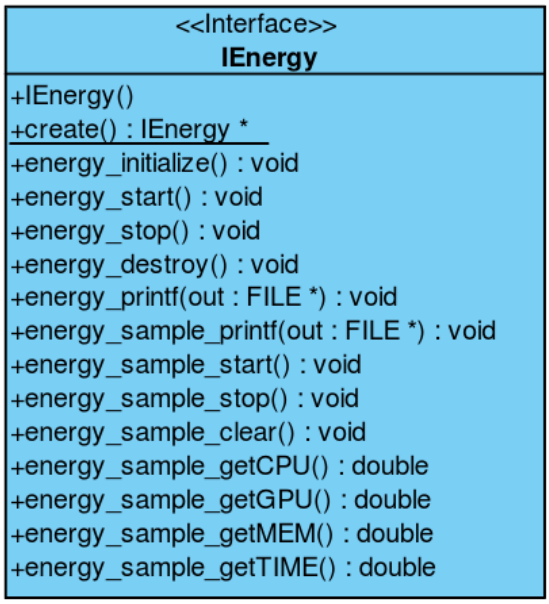

Figura 2.3: Diagrama de la interfaz genérica en C++.

2.1.3. Implementación de la interfaz IEnergy

Una vez diseñada la interfaz, se define una implementación de esta. La implementación la realiza la clase *COdroidEnergy* utilizando las funciones y tipos de datos proporcionados en la librería Energy Meter v1.0. Además, implementa las mediciones acumulativas tanto en energía como en tiempo ya que la librería original no proporciona este soporte. Para ello se declaran variables para cada dato proporcionado, es decir, energía consumida en A7, A15, memoria y GPU, y tiempo parcial. El siguiente listado describe los detalles de implementación de cada método.

- El constructor de la clase se ha decidido dejar vacío, ya que se utiliza un método para la creación e inicialización de instancias.
- El método estático create devuelve una instancia de la clase COdroidEnergy. A este método se accede a través de la interfaz IEnergy en el código instrumentado. Al ser estático no es necesario tener el objeto creado para realizar la llamada.
- El método energy_initialize inicializa el medidor de energía y las variables para las mediciones parciales, utilizando llamadas a energy_meter_init y la macro init_em proporcionadas por Energy Meter v1.0, y a la función energy_sample_clear.

- Los métodos energy_start, energy_stop, energy_destroy y energy_printf llaman a las funciones equivalentes proporcionadas por la librería Energy Meter v1.0.
- El método energy_sample_start llama a la función de la librería que lee la energía consumida hasta ese mismo instante. Además, guarda el instante de tiempo en el que se llama a la función.
- El método energy_sample_stop recibe un parámetro acc. Mediante llamadas a la librería, calcula la diferencia de energía consumida y el tiempo de ejecución desde la última vez que se llamó al método energy_sample_start.
  En función del parámetro acc este valor sustituye al anterior o se acumula, y su significado es el mismo que el definido en la interfaz.
- El método energy_sample_clear inicializa los valores sobre los que guardamos datos de mediciones parciales.
- El método energy_sample_printf muestra los valores de las mediciones parciales realizadas. Recibe como parámetro el flujo de salida deseado.
- Los métodos energy_sample_getCPU, energy_sample_getGPU, energy_sample_getMEM y energy_sample_getTIME devuelven los datos de las mediciones parciales de CPU, GPU, Memoria y el contador de tiempo, respectivamente.

En la Figura 2.4 se muestra el diagrama de la clase COdroidEnergy. Implementa la interfaz IEnergy utilizando las funciones proporcionadas por la librería Energy Meter v1.0.

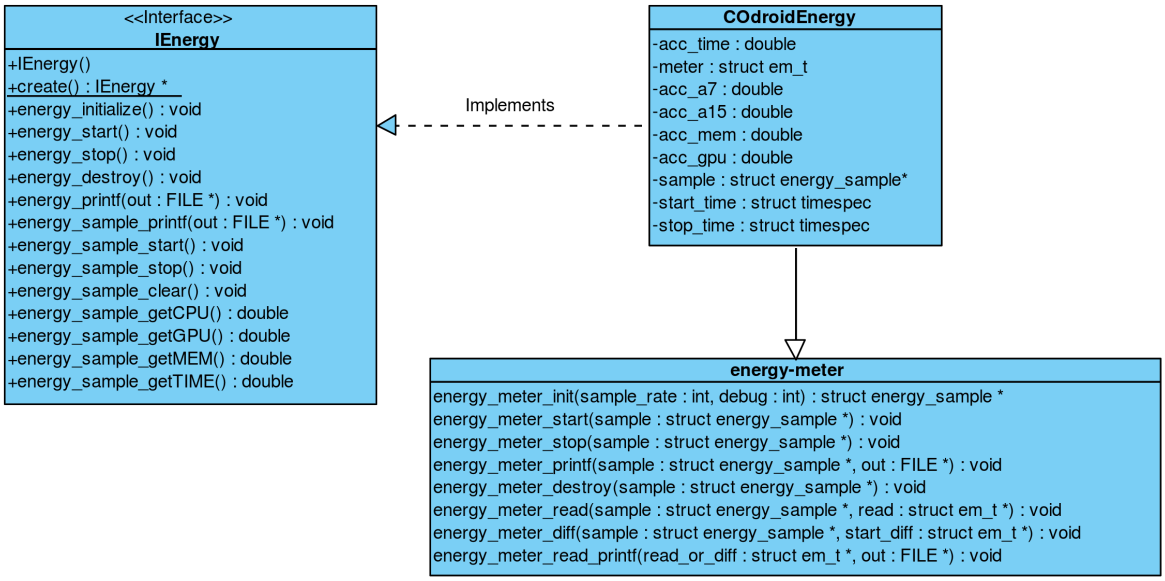

Figura 2.4: Diagrama de la clase COdroidEnergy.

2.1.4. Adaptador de la librería C++ a C

Una vez definida una implementación de la interfaz IEnergy, podemos utilizarla para instrumentar aplicaciones escritas en C++. Durante el desarrollo de este trabajo se ha

decidido instrumentar el código del conjunto de aplicaciones STAMP, que en este caso están escritas en C. En lugar de utilizar la implementación C original de la librería de energía (esto es, Energy Meter v1.0) se ha decidido crear una serie de funciones en C que, siguiendo el patrón de diseño adaptador, permitan utilizar una implementación C++ de la interfaz IEnergy. El motivo es poner en práctica la filosofía de diseño de la interfaz IEnergy: debe servir para, de forma genérica, instrumentar código para realizar mediciones de energía. Las implementaciones de esta interfaz deben estar escritas en C++, mientras que las aplicaciones que deseen utilizarla deben estar también escritas en C++, o bien, implementar un adaptador adecuado.

En la implementación de cada una de estas funciones se hace uso del tipo *Energy*, este tipo está definido como un puntero a *void*. De esta forma podemos transformar el tipo Energy mediante un *casting* en la implementación de la interfaz IEnergy. Al tener acceso a la interfaz, se puede llamar a los métodos implementados anteriormente.

- void energy_init(const Energy *e)
- void energy_start(const Energy *e)
- void energy_stop(const Energy *e)
- void energy_destroy(const Energy *e)
- void energy_printf(const Energy *e, FILE * out)
- void energy_sample_start(const Energy *e)
- void energy_sample_stop(const Energy *e, int acc)
- void energy_sample_printf(const Energy *e, FILE * out)
- void energy_sample_clear(const Energy *e)
- double energy_sample_getCPU(const Energy *e)
- double energy_sample_getGPU(const Energy *e)
- double energy_sample_getMEM(const Energy *e)
- double energy_sample_getTIME(const Energy *e)

Para la compilación de la librería de energía se ha realizado un *script* en Bash que facilita el trabajo. La librería externa debe colocarse en el directorio *lib*, dentro de un subdirectorio con el nombre de la librería externa. Por ejemplo, en este caso la librería se coloca en /lib/energy−meter/. La librería externa debe disponer de un fichero Makefile que haga posible su compilación mediante el comando make. Para compilar la librería de energía se debe ejecutar el comando `./compile.sh <directorio−librería−externa>`, por ejemplo `./compile.sh energy−meter`. Además, para facilitar la compilación de STAMP, se ha creado un fichero Defines.energy.mk. En este fichero se encuentran rutas y dependencias necesarias para que STAMP compile con la librería de energía. Para ello hay que incluir en el Makefile de cada una de las aplicaciones STAMP a compilar una línea que referencia a este fichero: include ../../ energy/Defines.Energy.mk.

La instrumentación de un código escrito en C mediante el adaptador se puede ver en la Figura 2.5. El código muestra el consumo de energía y el tiempo de ejecución de una

sección de código. Además, muestra al final de la ejecución el consumo total del programa que ha realizado el medidor.

```
#include "AdaptEnergy.h"

main(int argc, char **argv)
{
  // Declaración de la variable de energía
  Energy * e;
  // Creación de la instancia
  e = energy_create();
  // Inicialización del medidor
  energy_init(e);
  // Inicia el medidor de energía
  energy_start(e);
  // Inicio la medición de una sección de código
  energy_sample_start(e);
  /*
   * Sección de código a muestrear
   */
  // Paro la medición de la sección de código
  // Como solo se mide esta parte, el parámetro
  // acc es 0 y sustituye el valor parcial anterior
  energy_sample_stop(e, 0);
  // Muestro los datos del consumo de la sección
  energy_sample_printf(e,sdtdout);
  // Para el medidor de energía
  energy_stop(e);
  // Muestra los datos de todo el medidor
  energy_printf(e,stdout);
  // Libera memoria
  energy_destroy(e);

}
```

Figura 2.5: Medición del consumo de un código utilizando el adaptador.

La Figura 2.6 muestra el diagrama del adaptador completo. AdaptEnergy hace uso de la clase COdroidEnergy a través de la interfaz IEnergy y presenta una implementación de todos los métodos de la interfaz.

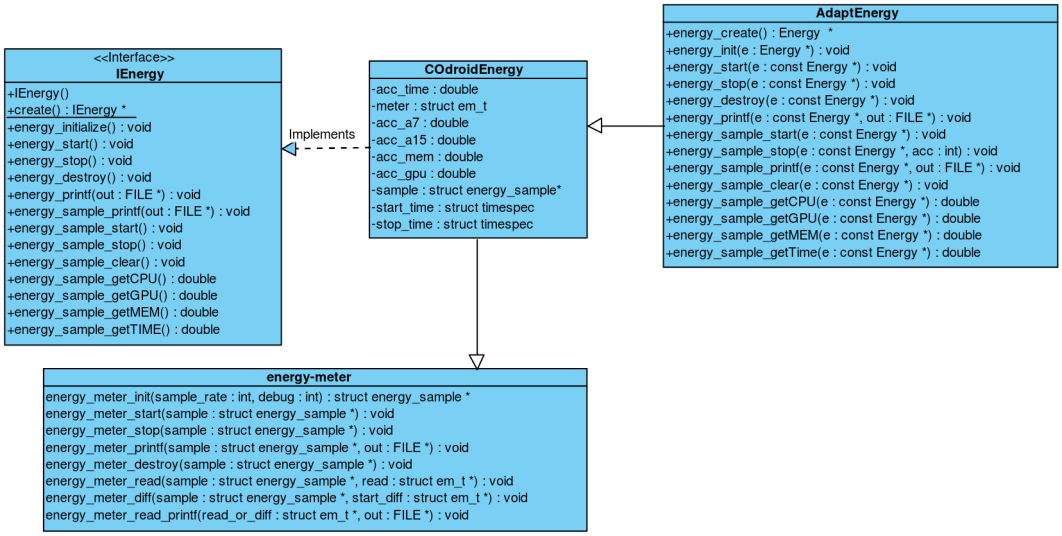

Figura 2.6: Diagrama del adaptador de C++ a C.

2.2. Instrumentación de la librería TinySTM y de las aplicaciones STAMP

2.2.1. Compilación de TinySTM y STAMP en ARM

Para utilizar las aplicaciones STAMP junto con la librería TinySTM, en primer lugar se debe compilar TinySTM. Una vez se han obtenido los ficheros objeto de la librería, se deben enlazar a las aplicaciones STAMP y compilarlas adecuadamente.

TinySTM, tal y como se obtiene desde `http://tmware.org/tinystm`, está preparado para compilarse en diversas arquitecturas, pero no es inmediatamente utilizable en arquitecturas ARM. Concretamente, utiliza una librería para operaciones atómicas llamada atomic_ops que no está disponible para arquitecturas ARM. Esta librería está situada en el subdirectorio ./src/atomic_ops dentro del directorio de TinySTM. En este directorio encontramos el fichero atomic_ops.h en el que, utilizando la directiva "if defined()", se incluyen los ficheros necesarios para cada tipo de arquitectura y, como se ha comentado, la arquitectura ARM no está disponible. Sin embargo, es posible enlazar una implementación externa de la librería. TinySTM es compilable utilizando el comando make. Entre los ficheros de configuración utilizados para construir los objetos se encuentra Makefile.common. Este fichero nos permite especificar la ruta en la que se encuentra la librería atomic_ops.h que, por defecto, apunta al directorio mencionado anteriormente. La ruta se puede modificar cambiando el directorio al que se apunta en la variable LIBAO_HOME. El compilador presente en el dispositivo ODROID ya incluye una implementación de atomic_ops.h para arquitecturas ARM, disponible en el directorio /usr/include/atomic_ops. Por tanto, únicamente es necesario cambiar la ruta de la localización atomic_ops.h a dicho directorio y compilar

TinySTM.

Para una mejor organización se han creado varios directorios con copias de STAMP. En cada uno de ellos se encuentran las diferentes implementaciones de STAMP que se utilizan. El directorio stamp se encuentra la implementación de STAMP lista para compilar con TinySTM. En el directorio stamp−sec se encuentra la implementación de STAMP para compilar utilizando el código secuencial. En el directorio stamp−func se encuentra la implementación de STAMP para compilar utilizando TinySTM pero el código se encuentra instrumentado para sólo medir las llamadas a las funciones de TM. En cada uno de estos directorios existe un subdirectorio ./common que contiene los diferentes archivos Makefile que serán utilizados para compilar. En Makefile.common.mk hay que especificar la ruta en la que se encuentra la implementación STM utilizada, en este caso TinySTM. En cada aplicación de STAMP encontramos otro fichero Makefile.stm que será llamado para compilar la aplicación. En este fichero se debe incluir la ruta del fichero Defines.Energy.mk de la librería de energía.

Para automatizar el proceso de compilación de las ocho aplicaciones de STAMP, se dispone de un script en Bash que va entrando en cada una de las aplicaciones y compilándolas una a una. Este script es común tanto para stamp, stamp−seq y stamp−func. En el caso de stamp y stamp−func se utiliza TinySTM por lo que la llamada se realiza mediante `./compile−all.sh tiny`. En caso de stamp−seq la compilación se hace para ejecución secuencial con `./compile−all.sh seq`.

Mediante este proceso, hemos conseguido compilar tanto TinySTM como todas las aplicaciones STAMP en esta arquitectura. No obstante, el funcionamiento de alguna de ellas no ha sido correcto, tal y como se detalla en las pruebas iniciales que hemos realizado (sección 2.2.2).

2.2.2. Pruebas iniciales

En primer lugar, se han realizado pruebas de STAMP utilizando TinySTM. Como se ha comentado en la sección 2.2.1 compila correctamente, pero su funcionamiento no está asegurado puesto que no ha sido probado en arquitecturas ARM con anterioridad. Hemos diseñado dos pruebas para evaluar el funcionamiento de STAMP con TinySTM. La primera prueba se hace utilizando la configuración con menor carga proporcionada por STAMP, para descartar en primer lugar aplicaciones que no funcionen correctamente. Como se pretende evaluar la ejecución en distintos hilos, se lanzan una serie de ejecuciones con 1, 2, 4 y 8 hilos simultáneos. No se consideran número de hilos que no sean potencias de 2 (esto es, 3, 5, 6 y 7) porque las aplicaciones STAMP introducen esta restricción. Tampoco se han considerado más de 8 hilos puesto que disponemos de 8 núcleos físicos. Para lanzar esto experimentos se han creado scripts en Bash. Como resultado se obtiene que las aplicaciones funcionan correctamente y los resultados de salida son correctos, a excepción de genome que al ejecutar con más de un hilo provoca un fallo de segmentación. La segunda prueba se realiza con la configuración que proporciona la mayor carga de

trabajo (sufijo ++). En este caso genome sigue teniendo el mismo comportamiento erróneo. Además, bayes y yada comienzan a dar problemas por consumo excesivo de memoria, haciendo que el *Out-of-Memory Manager* del sistema operativo no permita la ejecución en cierto punto.

En segundo lugar, se hacen las pruebas iniciales con STAMP utilizando la implementación secuencial en lugar de TinySTM. Como se comenta en la sección 2.2.1, la compilación es correcta. La metodología seguida es igual que con la primera evaluación con TinySTM, realizando dos pruebas pero, al ser una ejecución secuencial, sólo se lanza 1 hilo de ejecución. Las primera prueba se lanzan con la menor carga de trabajo. La ejecución es correcta en la mayoría de los casos, a excepción de kmeans que provoca un fallo de segmentación. La segunda prueba solo varía de la primera en que utiliza la mayor carga (parámetro ++). En este caso ocurre lo mismo que utilizando TinySTM: las aplicaciones bayes y yada dan problemas por consumo excesivo de memoria.

Además, se observa que tanto en TinySTM como en la ejecución secuencial, la aplicación bayes presenta una mucha irregularidad en los tiempos medidos para varias ejecuciones utilizando los mismos parámetros. Este problema se ha documentado anteriormente en otro estudio relacionado sobre STAMP [35], y debido a ello, la aplicación bayes se descarta frecuentemente de los estudios de TM.

2.2.3. Instrumentación de las aplicaciones

Las aplicaciones de STAMP vienen por defecto instrumentadas con contadores de tiempo. Estos contadores de tiempo están situados en las zonas donde se necesita medir el uso de transacciones en la aplicación. Como el objetivo del trabajo es medir consumo energético, se instrumentan las aplicaciones con la librería de energía siguiendo el modelo explicando en la sección 2.1.4. Por lo tanto se añaden la instrumentación de energía junto a la de tiempo ya existente en STAMP. Tras analizar el código de las diferentes aplicaciones se observa que las estructuras son muy similares. Esto facilita la incorporación de la nueva instrumentación. De forma general, el código de las aplicaciones STAMP sigue el modelo presentado en la Figura 2.7.

Primero, una sección de inicialización donde se crean todas las estructuras que van a utilizarse. Le sigue la sección que realiza todo el trabajo paralelo utilizando transacciones, y es donde se introducen las nuevas funciones de energía. La siguiente sección es de verificación y finalmente, la última sección, libera la memoria utilizada.

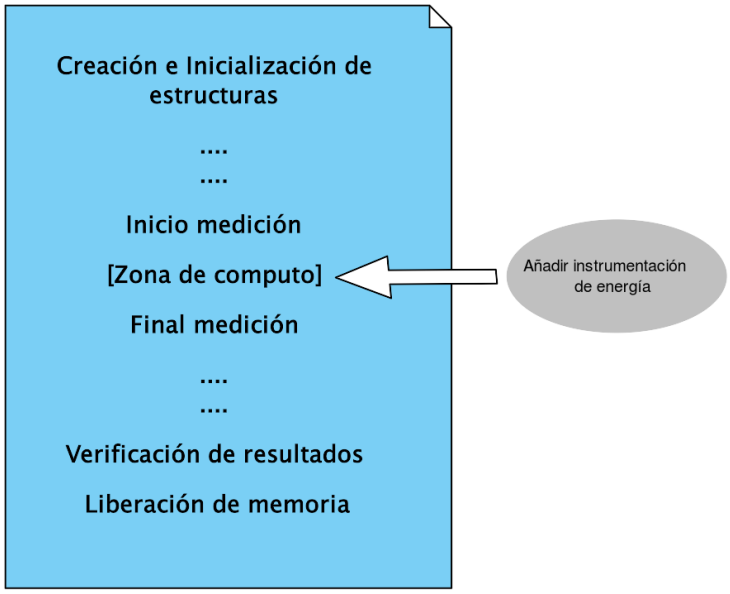

Figura 2.7: Estructura genérica de las aplicaciones STAMP

2.2.4. Instrumentación de las funciones

Para la instrumentación de las funciones de TM se tienen varias opciones. La primera consiste instrumentar la librería de TinySTM añadiendo los contadores de tiempo y de energía en el interior de cada función (ver Figura 2.8b). La segunda opción es instrumentar el código de STAMP y añadir las funciones de medición antes y después de la llamada a cada función (ver Figura 2.8a).

```
void test()
{
  //...
  //...
  energy_sample_start(e_meter);
  TM_BEGIN();
  energy_sample_stop(e_meter, 0);
  //...
  //...
}
```

(a) Instrumentación de una función en las aplicaciones STAMP

```
void TM_BEGIN()
{
  energy_sample_start(e_meter);
  // ...
  // Código de la función
  // ...
  energy_sample_stop(e_meter, 0);

}
```

(b) Instrumentación de una función en la librería TinySTM

Figura 2.8: Ejemplos de código de instrumentación de funciones en STAMP y TinySTM, la variable e_meter se supone como una variable de energía correctamente inicializada.

Debido al funcionamiento de nuestra librería de energía, el método más sencillo es el presentado en la Figura 2.8a. Esta implementación además hace el código más portable para probar otras librerías TM.

Capítulo 3

Evaluación

En este capítulo se presenta la metodología experimental seguida en la evaluación, presentando las aplicaciones sobre las que se trabajan, los clusters en los que se ejecutan las aplicaciones, las métricas utilizadas y el tratamiento de los resultados obtenidos.

3.1. Metodología experimental

3.1.1. Aplicaciones

En la sección 2.2.2 se presentan las pruebas iniciales realizadas. Estas pruebas sirven para poder seleccionar aquellas aplicaciones que utilizamos durante la evaluación.

Para la evaluación de las pruebas secuenciales se han escogido las aplicaciones vistas anteriormente que no presentan problemas en su ejecución con la configuración con el sufijo ++ . Del mismo modo se ha decidido utilizar para las pruebas con TinySTM las que no presentan problemas de ejecución con la configuración ++. Se ha escogido la configuración ++ porque es la que introduce mayor carga de trabajo en el dispositivo. La Tabla 3.1 contiene el listado de las aplicaciones utilizadas para la evaluación.

Aplicación	Pruebas con TinySTM	Pruebas en secuencial
bayes	No	No
genome	No	No
intruder	Si	Si
kmeans	Si	No
labyrinth	Si	Si
ssca2	Si	Si
vacation	Si	Si
yada	No	No

Tabla 3.1: Aplicaciones seleccionadas para la evaluación.

Se han seleccionado intruder, kmeans, labyrinth, ssca2 y vacation para los experimentos

con TinySTM ya que han funcionado sin problemas. Además, estas aplicaciones poseen diferentes características en cuanto a tamaño de los datos que trata, longitud de transacciones, etc... (ver Tabla 1.1). Para evaluar la ejecución secuencial se han evaluado aquellas aplicaciones que han funcionado sin problemas. La aplicación bayes ha sido excluida porque, como se ha comentado anteriormente, presenta diferencias notables en los tiempos de ejecución entre distintas ejecuciones.

3.1.2. Procesadores y métricas

Primero se ejecutan las aplicaciones dejando que el planificador del sistema operativo decida dónde planificar los hilos de ejecución. Por defecto, el planificador coloca las aplicaciones que utilizan de 1 a 4 hilos en el cluster big y es a partir de este número cuando empieza a utilizar ambos clusters. Mediante el comando taskset[1] seleccionamos los núcleos sobre los que se ejecutan los hilos de las aplicaciones. Para utilizar este comando adecuadamente, tenemos que averiguar el identificador de cada uno de los núcleos de cada cluster. En nuestro caso, el sistema operativo ha asignado los índices 0, 1, 2 y 3 para el cluster little (A7) y los índices 4, 5, 6 y 7 para el cluster big (A15). Hay que tener en cuenta que aunque no tengamos ejecutando la aplicación en ciertos núcleos, estos generan un consumo residual.

Durante cada uno de los experimentos se examinan 3 métricas. En primer lugar, el tiempo de ejecución de la sección de código que utiliza transacciones. En segundo lugar, la energía medida en Julios, que es la medida proporcionada por la librería de energía. Finalmente, se calcula el EDP [27] (*Energy-Delay Product*) con los valores de tiempo y energía medidos. El cálculo a realizar es la multiplicación de la energía total consumida en la ejecución de la aplicación por el tiempo empleado (esto es, EDP = Energía × Tiempo). El EDP se utiliza como métrica de la eficiencia energética: a menor valor de EDP significa una mayor eficiencia. En cada experimento se llevan a cabo 10 ejecuciones y con los datos obtenidos se realiza un promedio de esas 10 ejecuciones. Durante los experimentos, en las mismas condiciones, los valores medidos son consistentes, es decir no hay una variación significativa en dos ejecuciones diferentes de una misma aplicación.

3.1.3. Automatización de las pruebas

Para realizar las pruebas se han realizado varios scripts en Bash que automatizan el proceso. Los scripts son tests.sh, energy_tests_high_e.sh, energy_tests_low_e.sh, energy_tests_high.sh y energy_tests_low.sh. Igualmente, para la visualización de los resultados de forma numérica y gráfica se han creado scripts en Python llamados parseTests.py y generate++.py. La Figura 3.1 muestra el árbol de directorios sobre el que trabajan estos scripts para generar lanzar los experimentos y generar los resultados.

[1] http://linux.die.net/man/1/taskset

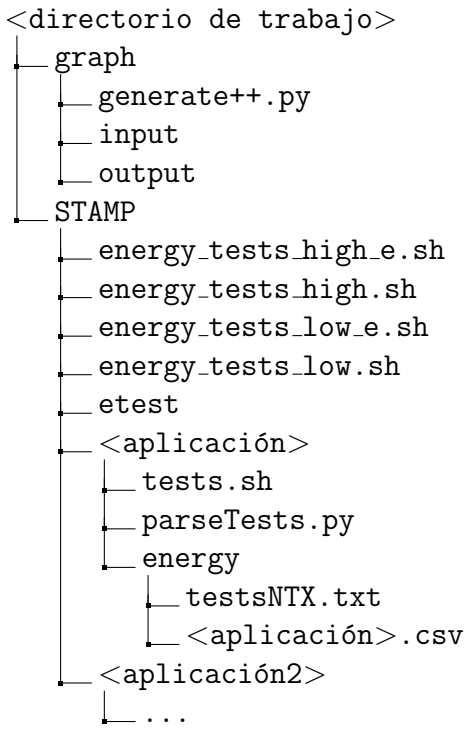

Figura 3.1: Árbol de directorios creado para la evaluación

El script tests.sh sirve para ejecutar las pruebas para una aplicación concreta. Este script se encuentra replicado en el directorio de cada aplicación y está diseñado para recibir 4 argumentos. Una llamada a este script tiene la forma `./tests.sh <N> <P> <C> <A>`

- El argumento <N> indica el (N)úmero de hilos a utilizar, pudiendo recibir los valores 1, 2, 4 u 8.
- El argumento <P> indica el número de (P)ruebas a realizar. Los resultados de las P pruebas se almacenan en el mismo fichero de salida uno tras otro.
- El argumento <C> indica la (C)ontención en los parámetros de entrada. Los valores posibles son *high* para seleccionar el parámetro ++ en STAMP o *low* para el parámetro +.
- El argumento <A> indica la (A)finidad de los hilos de ejecución a los núcleos. Es una cadena de la forma 0,1,2,3,4,5,6,7 en la que se indican los índices de los núcleos que queremos utilizar.

Los resultados de las pruebas se escriben en el subdirectorio ./energy/ con el nombre de fichero testsNTX.txt siendo X el número de hilos utilizados. Por ejemplo, para ejecutar las pruebas de intruder, se debe entrar a su directorio y utilizar el comando `./tests.sh 4 10 high 0,1,2,3`. Este comando lanza intruder utilizando 4 hilos, realiza un total de 10 pruebas, utiliza la configuración ++ proporcionada por STAMP y selecciona los núcleos 0, 1, 2 y 3 correspondientes al cluster little (A7). Los resultados de esta prueba se escriben en el fichero ./energy/testsNT4.txt, que contiene los datos de 10 pruebas. Si se ejecu-

ta de nuevo el mismo comando (`./tests.sh 4 10 high 0,1,2,3`), el fichero ./energy/testsNT4.txt no se reemplaza, sino que los resultados de las nuevas pruebas se añaden al final de dicho fichero.

Los scripts energy_tests_high_e.sh, energy_tests_low_e.sh, energy_tests_high.sh y energy_tests_low.sh lanzan una batería de pruebas en las que se ejecutan todas las aplicaciones, con distinto número de hilos. Los scripts energy_tests_high_e.sh, energy_tests_low_e.sh lanzan 1, 2 y 4 hilos de ejecución, mientras que energy_tests_high.sh y energy_tests_low.sh lanzan 1, 2, 4 y 8 hilos de ejecución. Los indicadores low y high sirven para distinguir la contención en la entrada de cada prueba. Un ejemplo de llamada a estos scripts es `./energy_tests_high_e.sh <P> <A>`, donde:

- El argumento <P> indica el número de (P)ruebas a realizar en cada aplicación.
- El argumento <A> indica la (A)finidad de los hilos de ejecución a los núcleos. Es una cadena de la forma 0,1,2,3,4,5,6,7 en la que se indican los índices de los núcleos que queremos utilizar.

Los 4 scripts descritos anteriormente tienen los mismos argumentos de entrada. Estos scripts llaman al script tests.sh que se encuentra en el directorio de cada aplicación. Todos estos scripts hacen una llamada a otro, parseTests.py, que se explica a continuación, para generar ficheros csv con los resultados de los experimentos. Los ficheros csv generados se copian al directorio ./etest/. Por ejemplo, para lanzar 10 pruebas con 1, 2 y 4 hilos utilizando la opción ++ y utilizando el cluster big, se debe ejecutar el siguiente comando: `./energy_tests_high_e.sh 10 4,5,6,7` . Para hacer la misma prueba pero con 1, 2, 4 y 8 hilos, utilizando ambos clusters se debe ejecutar el siguiente comando: `./energy_tests_high.sh 10 0,1,2,3,4,5,6,7` . Señalar que estos 4 scripts borran los resultados anteriores que el script tests.sh pudiese haber generado (esto es, se eliminan los ficheros testsNTX.txt descritos anteriormente).

Para el tratamiento de los resultados se han realizado 2 scripts en Python. El primer script convierte los ficheros de salida al formato csv, mientras que el segundo lee estos ficheros para representar los resultados de forma resumida en formato texto y gráficamente. A continuación se detalla el funcionamiento de estos scripts.

El primer script está situado en el directorio de cada aplicación y se llama de la forma (`./parseTests.py <F> <H>`). Este script lee los ficheros testsNTX.txt, generado por las aplicaciones, y lo convierte a formato csv.

- El argumento <F> indica el nombre del (F)ichero de salida.
- El argumento <H> indica con qué número de (H)ilos se ha realizado el experimentos. Debe tener el valor 3 si el experimento se ha realizado con 1, 2 y 4 hilos, o bien 4 si se ha realizado con 1, 2, 4 y 8 hilos.

Por ejemplo, situados en el directorio de la aplicacion intruder la ejecución del comando `./parseTests.py salida 3` genera el fichero ./energy/salida.csv con los datos de los experimentos almacenados en ./energy/testsNT1.txt, ./energy/testsNT2.txt y ./energy/testsNT4.txt.

El script generate++.py, situado en el directorio ./graph/ es el encargado de leer todos los csv generados y crear las gráficas a partir de ellos. En este directorio existe un árbol de subdirectorios en los que hay que copiar los ficheros de entrada que se encuentran en ./etest/. Este árbol de subdirectorios contiene un directorio input en el que se deben colocar los ficheros csv de entrada y un directorio output en el que generate++.py genera los ficheros de texto (en formato txt) y gráficos de salida (en formato pdf).

3.2. Resultados experimentales

3.2.1. Evaluación secuencial

En primer lugar, se ejecutan las aplicaciones seleccionadas utilizando la forma secuencial (esto es, sin utilizar TinySTM). Esta ejecución se realiza con un solo hilo y, por tanto, sólo existen dos experimentos: ejecutar en el cluster little o ejecutar en el cluster big.

En la tabla 3.2 se muestra el promedio de los resultados obtenidos al ejecutar las 4 aplicaciones que evaluamos en secuencial, utilizando el cluster big.

Aplicación	$T.E._{A15}$	$C.E._{A15}$	EDP_{A15}
intruder	42.1697s	80.1944J	3381.7769
labyrinth	97.9596s	273.6930J	26810.8741
ssca2	19.2961s	35.1619J	678.4915
vacation	44.3644s	84.6508J	3755.4857

Tabla 3.2: Tiempo de ejecución en segundos ($T.E.$), consumo de energía en Julios ($C.E.$) y Energy-Delay Product (EDP) obtenidos en los experimentos secuenciales en el cluster big (A15).

En la tabla 3.3 se muestra el promedio de los resultados obtenidos al ejecutar las 4 aplicaciones que evaluamos en secuencial, utilizando el cluster little.

Aplicación	$T.E._{A7}$	$C.E._{A7}$	EDP_{A7}
intruder	63.5022s	71.2247J	4522.9262
labyrinth	268.6831s	314.9532J	84622.6043
ssca2	30.5778s	34.4065J	1052.0772
vacation	65.1864s	73.1984J	4771.5504

Tabla 3.3: Tiempo de ejecución en segundos ($T.E.$), consumo de energía en Julios ($C.E.$) y Energy-Delay Product (EDP) obtenidos en los experimentos secuenciales en el cluster little (A7).

Observando ambas tablas comprobamos que los tiempos de ejecución obtenidos en el cluster big son menores, puesto que tienen más potencia de cómputo. En cuanto a

energía consumida por la aplicación, el cluster little resulta más eficiente para todas las aplicaciones salvo labyrinth. El motivo es que el cluster big ejecuta mucho más rápido esta aplicación y consigue un ahorro de energía a pesar del mayor gasto de potencia. Observando el EDP, observamos que en todas aplicaciones es mayor en el cluster little. Esto significa que el ahorro de energía que obtenemos al utilizar este cluster no compensa que los tiempos de cómputo sean mayores con respecto al cluster big.

3.2.2. Evaluación TinySTM

Para la evaluación de TinySTM se realizan 3 pruebas. En la primera, se lanzan las 5 aplicaciones seleccionadas utilizando los núcleos little (A7). En la segunda, se lanzan utilizando los núcleos big (A15). Ambas pruebas se realizan con 1, 2 y 4 hilos. Finalmente, se lanzan pruebas dejando que el planificador decida: con 1, 2, 4 hilos selecciona los núcleos big y con 8 hilos ambos clusters.

Una vez realizadas las pruebas se analizan los resultados obtenidos en el cluster little, cluster big y ambos a la vez. Además, se realiza una comparativa entre ambos clusters. La Tabla 3.4 contiene los resultados numéricos obtenidos de la evaluación utilizando 4 hilos. Estos resultados se completan y se comentan de forma gráfica en las siguientes subsecciones.

Aplic.	$T.E._{A7}$	$T.E._{A15}$	$C.E._{A7}$	$C.E._{A15}$	EDP_{A7}	EDP_{A15}
intruder	87.0s	127.3s	120.9J	635.9J	10517.8018	80977.3776
kmeans	22.3s	37.5s	31.9J	187.7J	714.0321	7044.2739
labyrinth	91.4s	34.5s	136.9J	295.9J	14993.9341	10228.2794
ssca2	32.6s	35.6s	45.9J	171.8J	1501.6922	6120.4337
vacation	138.3s	271.7s	188.2J	1317.0J	26042.5344	357935.0756

Tabla 3.4: Tiempo de ejecución en segundos ($T.E.$), consumo de energía en Julios ($C.E.$) y Energy-Delay Product (EDP) de las aplicaciones sobre los distintos clusters utilizando 4 hilos.

3.2.3. Análisis del cluster little

La Figura 3.2 muestra los resultados de tiempos de ejecución del cluster little. En cuanto a tiempo de cómputo, TinySTM muestra una buena escalabilidad en los núcleos A7. En todos los experimentos la reducción en tiempo es aproximadamente del 50 % utilizando 2 hilos y del 70 % utilizando 4 hilos (según la media geométrica), respecto al uso de 1 hilo.

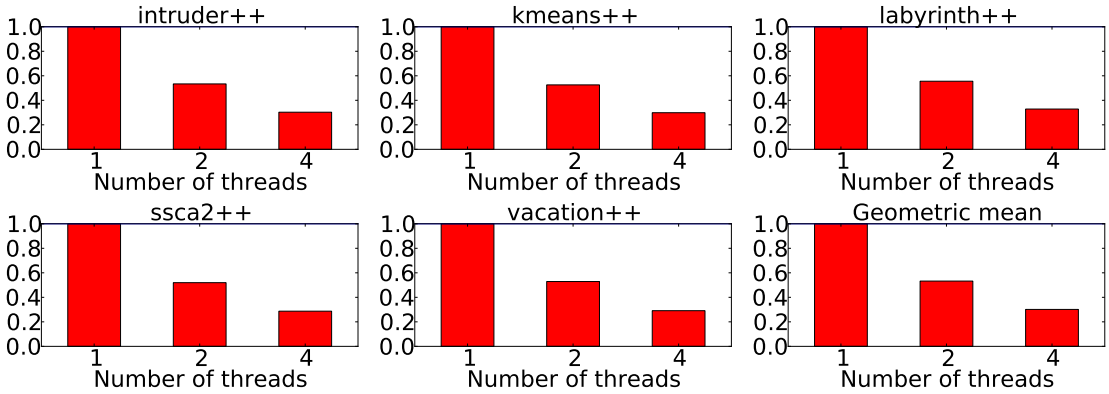

Figura 3.2: Tiempo de ejecución normalizado a 1 hilo en cluster little.

En la Figura 3.3 se muestra el consumo de energía de las aplicaciones en el cluster little. Igual que la anterior, esta presenta una reducción de aproximadamente 50 % al utilizar 2 hilos y 70 % al usar 4 hilos respecto al consumo de 1 hilo. Los procesadores A15 no se están utilizando lo que resulta en un consumo de energía reducido.

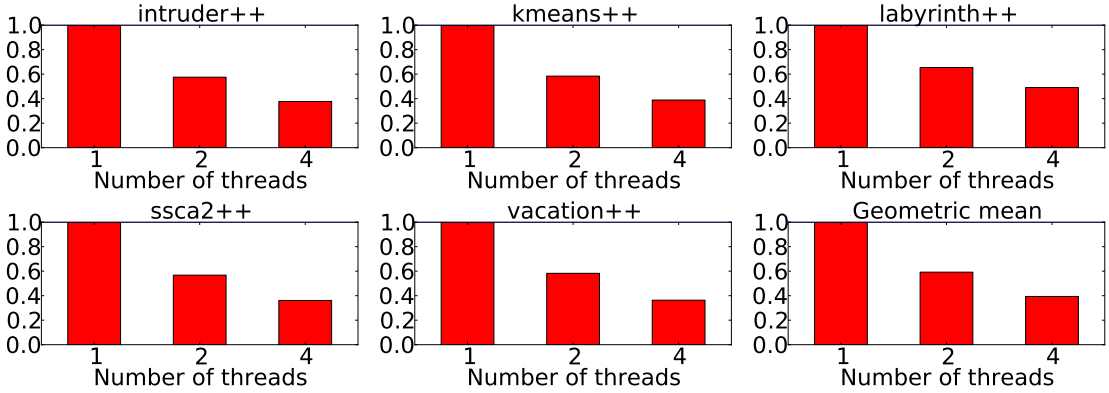

Figura 3.3: Consumo de energía normalizado a 1 hilo en cluster little.

En la Figura 3.4 se muestra el Energy-Delay Product. Se ve una reducción del 70 % aproximadamente al utilizar 2 hilos y del 90 % al utilizar 4 hilos respecto a 1.

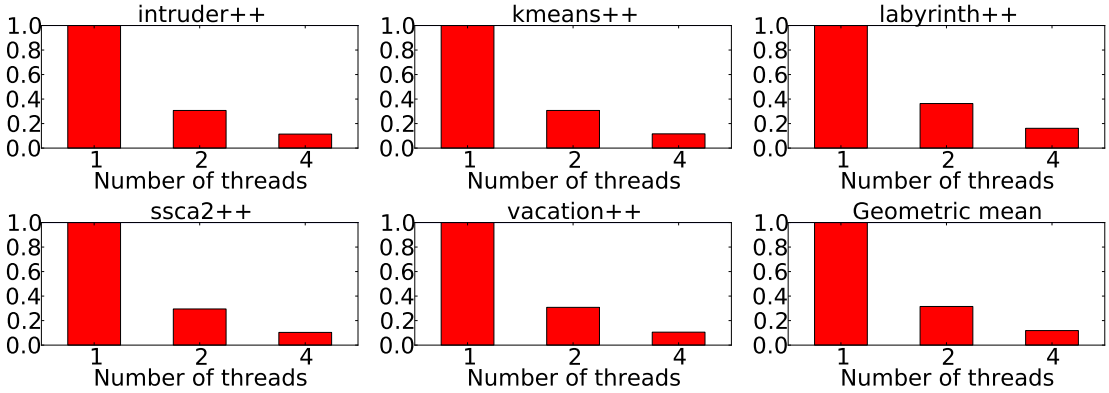

Figura 3.4: EDP normalizado a 1 hilo en cluster little.

En conclusión, para las aplicaciones evaluadas y utilizando TinySTM en el cluster little, es beneficioso aumentar el número de hilos de ejecución para mejorar tanto en tiempo de ejecución como la eficiencia energética.

3.2.4. Análisis del cluster big

La Figura 3.5 muestra los tiempos de ejecución normalizados en el cluster big. En este caso la media geométrica muestra que existe una reducción de aproximadamente 60 % utilizando 2 hilos y del 54 % utilizando 4 hilos, respecto al uso de 1 hilo. Por tanto, aunque existe cierta escalabilidad en las aplicaciones, no es tan pronunciada como la que se produce en el cluster little.

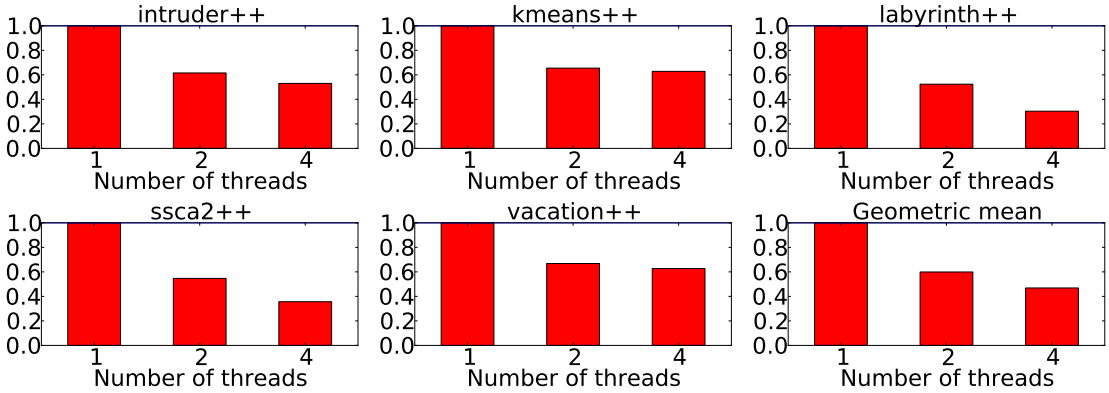

Figura 3.5: Tiempo de ejecución normalizado a 1 hilo en cluster big.

La Figura 3.6 muestra los consumos de energía normalizados en el cluster big. Este cluster no es tan eficiente en términos de energía como el cluster little. Se observa que aplicaciones como intruder, kmeans y vacation muestran un incremento en el consumo energético cuando utilizan los 4 núcleos. Observando la media geométrica, por lo general aumenta el consumo de energía en las 5 aplicaciones al utilizar 4 hilos.

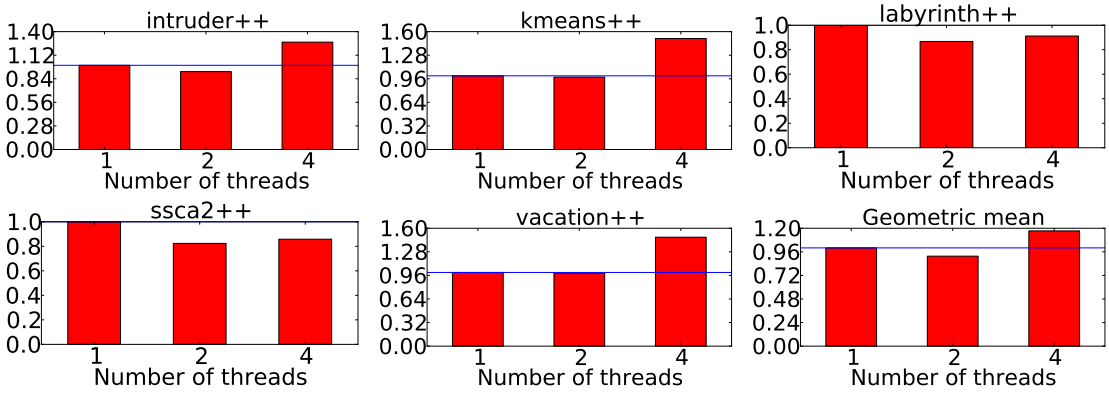

Figura 3.6: Consumo de energía normalizado a 1 hilo en cluster big.

La Figura 3.7 muestra el EDP. Como resultado del elevado consumo de energía, el

EDP no resulta óptimo en todas las aplicaciones utilizan 4 hilos. Las aplicaciones intruder, kmeans y vacation encuentran un EDP más eficiente utilizando únicamente 2 hilos.

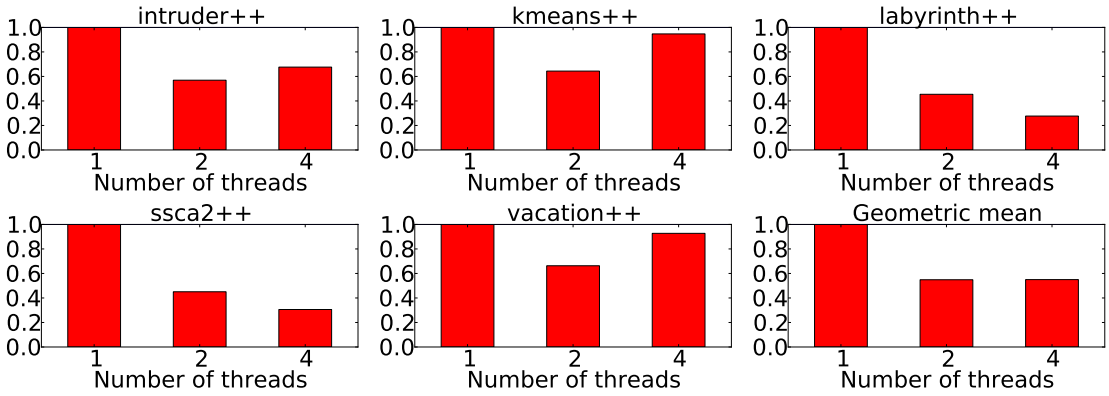

Figura 3.7: EDP normalizado a 1 hilo en cluster big.

La media geométrica muestra que, en cuanto a tiempos de cómputo, es preferible utilizar 4 hilos para la mayoría de aplicaciones. Sin embargo, utilizar 4 hilos resulta en un consumo de energía superior en comparación con el uso de 2 hilos. Estas diferencias se compensan en el EDP. Apenas existe diferencia en el promedio del EDP comparando la ejecución con 2 y 4 hilos.

3.2.5. Análisis de ambos clusters

En los datos mostrados a continuación se utilizan hasta 8 hilos con el objetivo de utilizar ambos clusters. Antes de ejecutar las aplicaciones, observamos el comportamiento del sistema operativo en cuanto a planificación de hilos. Durante las pruebas hasta 4 hilos, el sistema operativo planifica utilizando el cluster big, y es cuando introducimos los 8 hilos cuando comienza a planificar con ambos clusters.

La Figura 3.8 muestra, hasta 4 hilos de ejecución, un rendimiento similar a la mostrada en la Figura 3.6. Es cuando se incrementan los hilos hasta 8 cuando se observa que los tiempos de ejecución mejoran. Como se ha comentado anteriormente, la planificación de los 4 primeros hilos se realiza sobre el cluster big, y no se aprecian diferencias en los tiempos de cómputo. Al incrementar el número de hilos hasta 8 se comienza a utilizar el cluster little, lo cual reduce los tiempos de cómputo.

La Figura 3.9 muestra, igual que la anterior, un rendimiento similar al del cluster big mientras se utilizan hasta 4 hilos. Se observa que existe una mejora en el consumo de energía de todas las aplicaciones excepto en kmeans dónde empeora ligeramente respecto al uso de 1 y 2 hilos.

La Figura 3.10 refleja el EDP, calculado respecto a las 2 gráficas anteriores. Como se ha comentado anteriormente, siempre existe una mejora utilizando 8 hilos en cuanto a tiempos de ejecución. En la gráfica de consumo de energía kmeans empeora utilizando 8

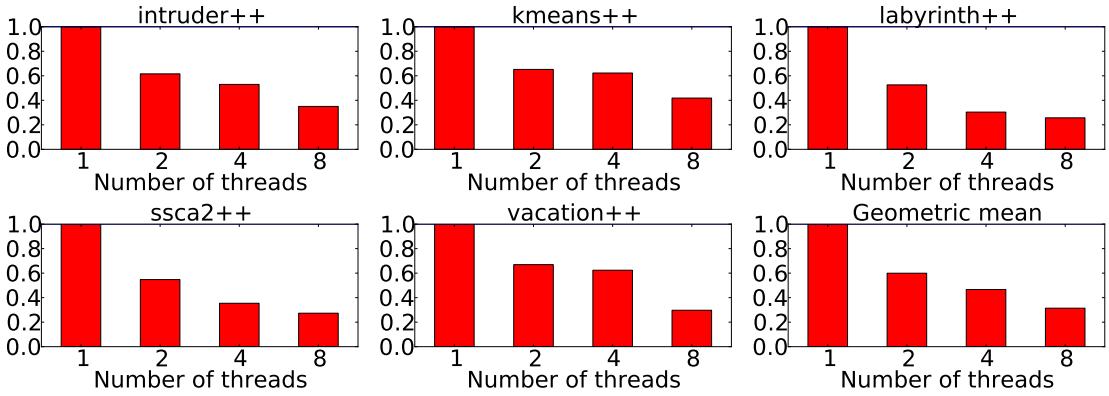

Figura 3.8: Tiempo de ejecución normalizado a 1 hilo utilizando ambos clusters.

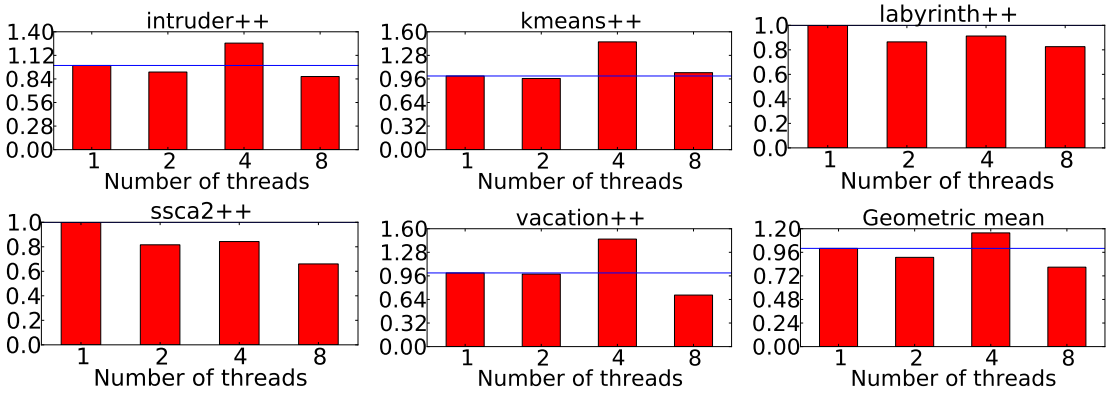

Figura 3.9: Consumo de energía normalizado a 1 hilo utilizando ambos clusters.

hilos, cuando los demás mejoran. Este empeoramiento del consumo de energía no tiene impacto en el EDP de kmeans. El EDP mejora utilizando ambos clusters en todos los casos.

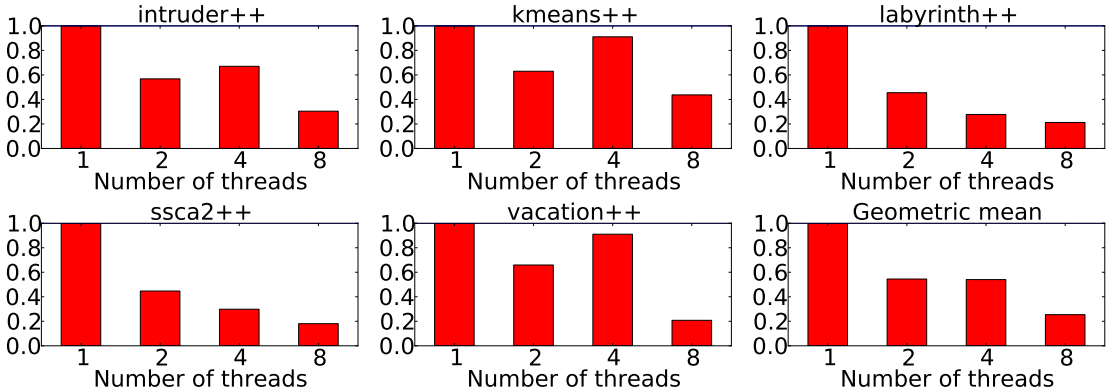

Figura 3.10: EDP normalizado a 1 hilo utilizando ambos clusters.

3.2.6. Comparativa entre ambos clusters

En las siguientes Figuras se muestra una comparativa entre los clusters little y big. Las métricas se representan utilizando el cociente $A7/A15$: valores mayor que 1 significan que

hay mejor rendimiento en el cluster big y valores menores que 1 hay un mejor rendimiento en el cluster little.

En la Figura 3.11 se muestra el resultado de la operación TiempoEj.$_{A7}$/TiempoEj.$_{A15}$. Utilizando un hilo de ejecución obtiene mejor rendimiento el cluster big gracias a su potencia de cálculo. Conforme aumenta el número de hilos, en rendimiento empieza a cambiar a favor del cluster little. La aplicación labyrinth es una excepción, se observa que en todos los casos, respecto al tiempo, tiene un mejor rendimiento el cluster big dado que requiere mayor potencia de cálculo.

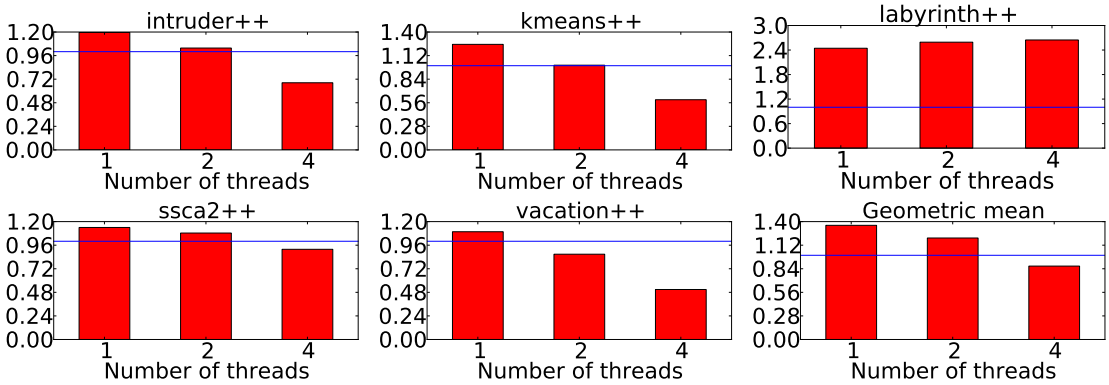

Figura 3.11: TiempoEjecución$_{A7}$/TiempoEjecución$_{A15}$

En la Figura 3.12 se muestra el resultado de la operación ConsumoEn.$_{A7}$/ConsumoEn.$_{A15}$. En todos los casos excepto en labyrinth con 1 hilo de ejecución, el cluster little produce mejores resultados que el cluster big. Esta aplicación en concreto se beneficia de la mayor potencia de cálculo del cluster big, por lo que logra terminar antes y reducir su consumo energético.

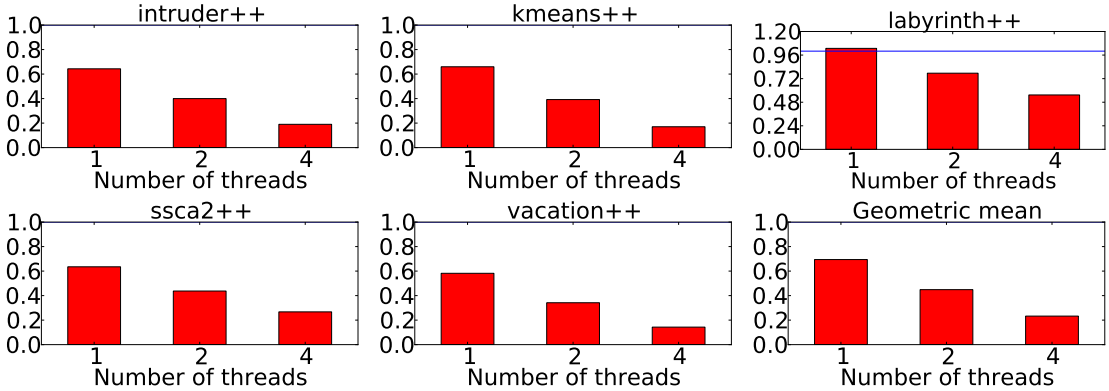

Figura 3.12: ConsumoEnergía$_{A7}$/ConsumoEnergía$_{A15}$

En la Figura 3.13 se realiza la operación EDP$_{A7}$/EDP$_{A15}$. Se comprueba una mayor eficiencia en el cluster little excepto en labyrinth, donde se observa que el cluster big es más apropiado. Dado que de las aplicaciones utilizadas labyrinth es la única que presenta transacciones largas y modificar una gran cantidad de datos (ver Tabla 1.1), necesita

una gran potencia de cálculo. Los núcleos Cortex-A15 son capaces de proporcionar esta potencia y amortizar el mayor consumo de energía.

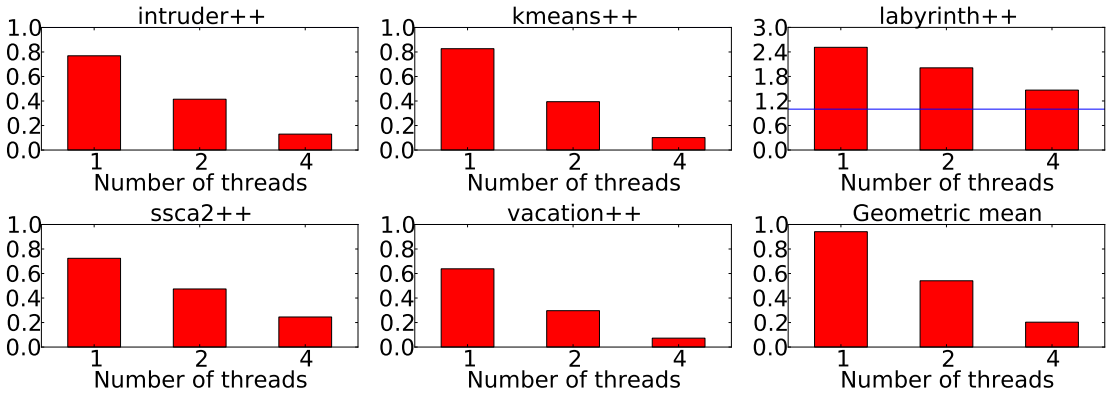

Figura 3.13: EDP_{A7}/EDP_{A15}

3.2.7. Análisis del consumo de las funciones

Con el objetivo de evaluar las funciones TM proporcionadas por TinySTM, se ha instrumentado una de las aplicaciones. Una de las restricciones de la librería Energy Meter v1.0 es que no nos proporciona mediciones fiables para tiempos inferiores a 50 milisegundos. Por tanto, no obtenemos una medición correcta de aquellas funciones que se ejecuten en un tiempo inferior. Para realizar esta medición se ha instrumentado la aplicación labyrinth, puesto que es la que tiene más carga computacional y hace un uso más exhaustivo de las funciones de TinySTM.

Las Tablas 3.5 y 3.6 muestran el número de llamadas, la energía consumida acumulada, el promedio del consumo por llamada a cada función, el tiempo de ejecución acumulado y el promedio de tiempo por llamada a cada función de TinySTM utilizada por la aplicación labyrinth. Observamos que los tiempos promedio por llamada de cada función siempre son muy pequeños (siempre inferiores a 1 ms), por lo que las mediciones obtenidas no son fiables. No obstante, se han incluido en esta memoria puesto que la instrumentación realizada sí será de utilidad cuando se utilice en dispositivos con una mayor resolución en las medidas de consumo energético.

Los tiempos de ejecución presentados sí ofrecen una información correcta, dado que los contadores de tiempo dependen del reloj del sistema y no de la resolución de los sensores de energía. Lo que se observa en la sección 3.2.6, se refleja en los datos de las tablas 3.5 y 3.6 utilizando un hilo de ejecución. En promedio por llamada, las funciones más costosas en tiempo son TM_THREAD_ENTER y TMGRID_ADDPATH. La primera de las funciones se dedica a lanzar los hilos de ejecución en TM y la segunda es la que lleva a cabo la mayor parte del trabajo en el código de la aplicación. El cluster big presenta una reducción de aproximadamente la mitad en el tiempo de ejecución respecto al cluster little.

Función	Número de llamadas	Energía consumida (acumulada)	Tiempo de ejecución (acumulado)	Energía media por llamada	Tiempo medio por llamada
TMQUEUE_POP	512	0.26974J	0.22937s	0.00053J	0.00045s
TMGRID_ADDPATH	512	0.45996J	0.39139s	0.0009J	0.00076s
TM_BEGIN	1026	0.54123J	0.46013s	0.00053J	0.00045s
TMLIST_INSERT	1	0.00059J	0.00046s	0.00059J	0.00046s
TM_QUEUE_ISEMPTY	513	0.26971J	0.22936s	0.00053J	0.00045s
TM_LOCAL_WRITE	512	0.27007J	0.22957s	0.00053J	0.00045s
TM_THREAD_ENTER	1	0.00051J	0.0008s	0.00051J	0.0008s
TM_THREAD_EXIT	1	0.00063J	0.00055s	0.00063J	0.00055s
TM_END	1026	0.58024J	0.48454s	0.00057J	0.00047s

Tabla 3.5: Energía consumida y tiempos de ejecución de las funciones TM en labyrinth utilizando el cluster little.

Función	Número de llamadas	Energía consumida (acumulada)	Tiempo de ejecución (acumulado)	Energía media por llamada	Tiempo medio por llamada
TMQUEUE_POP	512	0.54467J	0.19125s	0.00106J	0.00037s
TMGRID_ADDPATH	512	0.87171J	0.30514s	0.0017J	0.0006s
TM_BEGIN	1026	1.09094J	0.38342s	0.00106J	0.00037s
TMLIST_INSERT	1	0.00109J	0.00039s	0.00109J	0.00039s
TM_QUEUE_ISEMPTY	513	0.54458J	0.19132s	0.00106J	0.00037s
TM_LOCAL_WRITE	512	0.54398J	0.19083s	0.00106J	0.00037s
TM_THREAD_ENTER	1	0.00041J	0.00058s	0.00041J	0.00058s
TM_THREAD_EXIT	1	0.00121J	0.00043s	0.00121J	0.00043s
TM_END	1026	1.15188J	0.39859s	0.00112J	0.00039s

Tabla 3.6: Energía consumida y tiempos de ejecución de las funciones TM en labyrinth utilizando el cluster big.

Capítulo 4

Conclusiones

4.1. Conclusiones de los experimentos

Este trabajo de fin de grado presenta el análisis de una librería y un conjunto de aplicaciones de TM sobre una plataforma con un procesador heterogéneo con arquitectura big.LITTLE.

En la evaluación se muestra que la escalabilidad que presenta la librería, en términos de rendimiento y energía, es mejor sobre el cluster little. Existe una excepción con la aplicación labyrinth que muestra un mejor rendimiento y una utilización más eficiente de la energía en el cluster big. En todos los casos, cuando se utiliza simultáneamente el cluster big y little con 8 hilos de ejecución, resulta en una mejoría en el rendimiento y eficiencia energética.

En el caso del análisis de las funciones de TinySTM utilizando la aplicación labyrinth, no se ha podido llegar a una conclusión sobre la eficiencia energética dado que la resolución de los sensores es menor que el tiempo de ejecución de las funciones. En términos de rendimiento, el tiempo de ejecución de las funciones TM resulta menor en el cluster big, gracias a su potencia.

4.2. Conclusiones del trabajo de fin de grado

En este trabajo se han cumplido los objetivos planteados inicialmente. Se ha logrado instrumentar el benchmark STAMP y se ha realizado el análisis sobre la arquitectura big.LITTLE.

Para la realización de este trabajo de fin de grado se han tenido que adquirir conocimientos sobre la arquitectura big.LITTLE y sobre el funcionamiento de la memoria transaccional. Además, para conocer el área de trabajo, se ha tenido que realizar un estudio del estado del arte sobre eficiencia energética de procesadores.

En la implementación de la librería, se ha dedicado tiempo al aprendizaje del uso de clases con C++, la adaptación de código de C++ con clases a C para hacer el adaptador

portable entre los diferentes lenguajes y el uso del patrón de diseño adaptador.

Para la instrumentación de STAMP, se ha estudiado el funcionamiento de las distintas aplicaciones de STAMP, y aprendido el uso de Makefile para realizar la compilación e incorporar la librería de energía.

Tanto para la realización de los experimentos como para la representación de la información se ha aprendido Bash y Python junto con la librería matplotlib para la realización de gráficas.

Finalmente, la memoria ha sido realizada en LaTeX.

4.3. Trabajo futuro

Como trabajo futuro se propone utilizar los datos e infraestructura creados en este trabajo de fin de grado para realizar mejoras en las aplicaciones y sistemas de TM existentes. Por ejemplo, la instrumentación de las aplicaciones puede utilizarse para crear un planificador que, de forma automática, asigne a cada aplicación el cluster en el que resulta más eficiente. Otra propuesta de trabajo futuro es evaluar las aplicaciones TM ya instrumentadas en otras arquitecturas. Por ejemplo, se puede realizar una implementación de la interfaz IEnergy para equipos basados en arquitecturas x86. De esta forma, recompilando las aplicaciones sin realizar ningún cambio en la instrumentación, podemos evaluar el comportamiento en otro tipo de arquitecturas y establecer una comparativa.

Bibliografía

[1] ARM big.LITTLE technology. `http://www.arm.com/products/processors/technologies/biglittleprocessing.php`. Accessed: 2016-04-28.

[2] ODROID — Hardkernel. `http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127`. Accessed: 2016-04-28.

[3] Website Python.org. `http://www.python.org`. Accessed: 2016-05-25.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, pages 73–78, 2002.

[10] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July 2006.

[11] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 97–108, Feb 2007.

[12] Luke Dalessandro, Michael F Spear, and Michael L Scott. NOrec: Streamlining STM by Abolishing Ownership Records - PPoPP '10. pages 67–77, 2010.

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

[14] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *23rd Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'14)*, pages 3–14, 2014.

[15] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *ACM Sigplan Notices*, volume 44, pages 155–165. ACM, 2009.

[16] 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.

[17] 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.

[18] 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.

[19] E. Gaona, R. Titos-Gil, M. E. Acacio, and J. Fernández. Dynamic serialization: Improving energy consumption in eager-eager hardware transactional memory systems. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 221–228, Feb 2012.

[20] 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.

[21] 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.

[22] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd*. Morgan & Claypool Publishers, USA, 2010.

[23] 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.

[24] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[25] F. Klein, A. Baldassin, G. Araujo, P. Centoducatte, and R. Azevedo. On the energy-efficiency of software transactional memory. In *Proceedings of the 22Nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, SBCCI '09, pages 33:1–33:6, New York, NY, USA, 2009. ACM.

[26] Nasser A. Kurd, Muntaquim Chowdhury, Edward Burton, Thomas P. Thomas, Christopher Mozak, Brent Boswell, Praveen Mosalikanti, Mark Neidengard, Anant Deval, Ashish Khanna, Nasirul Chowdhury, Ravi Rajwar, Timothy M. Wilson, and Rajesh Kumar. Haswell: A family of IA 22 nm processors. *J. Solid-State Circuits*, 50(1):49–58, 2015.

[27] James H Laros III, Kevin Pedretti, Suzanne M Kelly, Wei Shu, Kurt Ferreira, John Vandyke, and Courtenay Vaughan. Energy delay product. *Energy-Efficient High Performance Computing*, pages 51–55, 2013.

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

[29] 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.

[30] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[31] 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.

[32] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, David A Wood, et al. Logtm: log-based transactional memory. In *HPCA*, volume 6, pages 254–265, 2006.

[33] 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.

[34] 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.

[35] Wenjia Ruan, Yujie Liu, and Michael Spear. STAMP Need Not Be Considered Harmful. pages 1–7, 2014.

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

[37] 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.

[38] Wikipedia. Red bayesiana — wikipedia, la enciclopedia libre. `https://es.wikipedia.org/w/index.php?title=Red_bayesiana&oldid=90838441`, 2016. [Internet; descargado 3-junio-2016].