Facultad de Informática

Adrián García García

Instrumentación del runtime de OpenMP

25 de junio de 2017

Índice

1. Introducción

Este trabajo tiene como objetivo realizar una instrumentación del runtime de OpenMP para permitir la recolección de métricas de rendimiento en los programas que hacen uso de esta API de paralelización. La conferencia relacionada con este trabajo estaba orientada más hacia el paradigma de computación basado en tareas y la paralización de diferentes programas que se ejecutan en procesadores simétricos convencionales. Sin embargo, en el ámbito de investigación en el que participo junto al profesor Juan Carlos Sáez Alcaide hemos observado que la gran mayoría de benchmarks multihilo de suites comerciales (SPEC, PARSEC, Minebench, etc.) hacen uso de directivas de paralelización convencionales como *#pragma parallel for*. Con el objetivo de evaluar el rendimiento de estos benchmarks, he realizado una instrumentación del runtime de OpenMP y más adelante presentaré un caso de estudio para exponer diferentes retos que debe afrontar su runtime en sistemas multicore asimétricos (AMP).

El modelo de paralelización basado en tareas divide el trabajo en unidades de computación denominadas tareas. El runtime contiene un pool de tareas y un pool de worker threads que ejecutan esas tareas en paralelo. Por otra parte, el modelo de paralelización tradicional de bucles basado en hilos difiere en la forma de gestionar los hilos. En el paradigma basado en tareas, el pool de hilos ejecuta múltiples tareas a lo largo de su tiempo de vida, mientras que en el paradigma de paralelización tradicional los hilos finalizan su ejecución cuando termina la carga de trabajo. Este último modelo se beneficia de la ejecución cargas de trabajo más grandes para evitar la sobrecarga que supone la inicialización de hilos.

La parelización basada en directivas de compilación que reparten el trabajo de los bucles realiza una división del espacio de iteraciones en fragmentos de igual tamaño para cada hilo. No obstante, esta aproximación trae significativos problemas de rendimiento y justicia en los sistemas AMP, ya que sus cores funcionan a diferente frecuencia y pueden presentar características microarquitectónicas diversas (diferentes tamaños de cache, ejecución en orden o fuera de orden, etc.). Estas diferencias provocan que un hilo que se ejecuta en un core big generalmente termine antes su asignación de trabajo. Cuando sucede esto, estos hilos se suelen bloquear en una barrera de memoria a través de una espera activa con el objetivo de sincronizarse con el resto de hilos que se ejecutaban en cores small. Considero que esta situación representa un desperdicio de recursos debido a dos motivos principales:

- Los sistemas operativos de propósito general no son capaces de identificar a los hilos que realizan una espera activa y, por tanto, su planificador no puede intercambiar estos hilos por otros que sean capaces de hacer un uso efectivo de los cores.
- En el contexto de monitorización del rendimiento y recogida de métricas para la toma de decisiones de planificación, las técnicas de espera activa como mecanismo de sincronización generan unos valores de rendimiento engañosos. Esto se debe a que los hilos que usan estas técnicas producen unos valores altos de IPC, cuando en realidad se encuentran ejecutando instrucciones inútiles para el progreso de la aplicación. Estas medidas pueden llevar a tomar decisiones erróneas al planificador o incluso a realizar cargos adicionales en plataformas de Cloud Computing en las que se factura a los usuarios en base al tiempo de uso de CPU y a diferentes métricas recogidas en tiempo de ejecución.

Con el objetivo de recoger información de diferentes métricas de rendimiento he elegido utilizar PMCTrack [2], se trata de una herramienta de monitorización del rendimiento a través de contadores hardware para Linux. Esta herramienta ha sido diseñada especialmente para ayudar a los desarrolladores del kernel en la implementación de algoritmos de planificación que utilicen los datos de los contadores de monitorización del rendimiento (Performance Monitoring Counters - PMCs) para realizar optimizaciones en tiempo de ejecución. A pesar de ser una herramienta orientada al sistema operativo, PMCTrack también está equipada con una serie de componentes en espacio de usuario (bibliotecas, entorno gráfico y de línea de comandos) que permiten obtener información de monitorización de aplicaciones en ejecución. En este trabajo se utilizará la biblioteca de PMCTrack para instrumentar el código del runtime de OpenMP y poder recoger diferentes métricas de los bucles paralelizados con este framework. Se puede encontrar un manual de usuario junto a una guía de instalación en la página web oficial del proyecto [4]. A continuación se muestra un comando básico cómo ejemplo de uso de PMCTrack, este comando proporciona al usuario el número de instrucciones retiradas y los ciclos de procesador por segundo. Para indicar los conjuntos de eventos hardware a monitorizar, es preciso utilizar la opción -c. Como se muestra en el ejemplo, la herramienta de línea de comandos permite especificar los eventos hardware a monitorizar usando mnemotécnicos. Al final de la línea, se especifica el comando para ejecutar la aplicación que deseamos monitorizar (por ejemplo, ./galgel00).

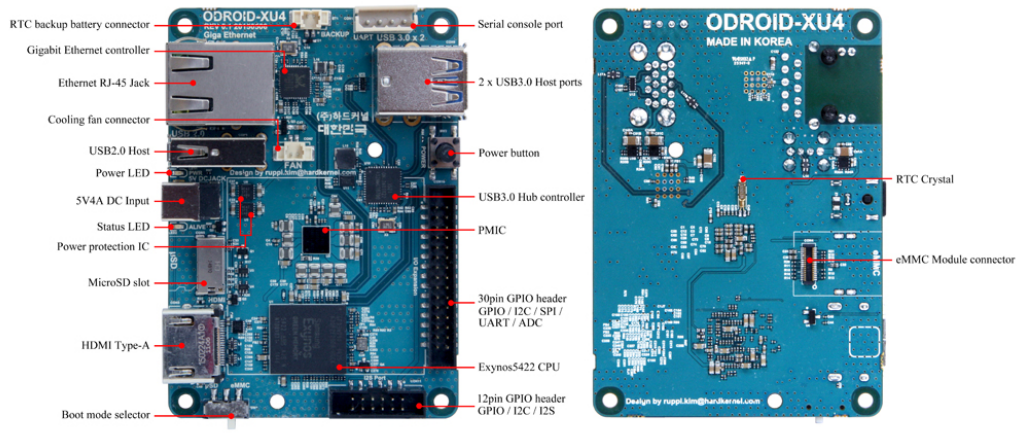

Figura 2: Placa ODROID-XU4

```
$ pmctrack -c instr,cycles ./galgel00
[Event-to-counter mappings]
pmc1=instr
pmc2=cycles
[Event counts]
nsample     pid       event           pmc1           pmc2
      1   27204        tick     2247040326     1723742839
      2   27204        tick     2423705061     1957082929
      3   27204        tick     2466664612     1944385684
      4   27204        tick     2280669757     1964700454
      5   27204        tick     2595158266     1983839065
      6   27204        tick     2462262543     1980869030
      7   27204        tick     2474564037     1942807991
      ...
```

Para la realización de este trabajo se ha empleado la placa de desarrollo Odroid XU4 [5], el principal motivo de su elección es que incorpora un procesador multicore asimétrico y, además, ya contaba con experiencia trabajando con ella en el Trabajo de Fin de Grado y anteriores investigaciones. Esta placa está provista de un SoC(*System-On-Chip*) Exynos 5422 de Samsung fabricado en 28nm, que integra un procesador ARM big.LITTLE de ocho núcleos. Estos se dividen en dos clusters, cuatro cores Cortex A15 (2GHz) con ejecución fuera de orden y 2MB de último nivel de cache (L2) y otros cuatro cores Cortex A7 (1.4GHz) con ejecución en orden y 512KB de LLC (L2). Ambos comparten espacio de direcciones para acceder a una memoria principal DDR3 de 2GB que funciona a 750 MHz. Para mostrar la configuración de la placa con mayor detalle, la figura 3 muestra el diagrama de bloques de la placa Odroid XU4.

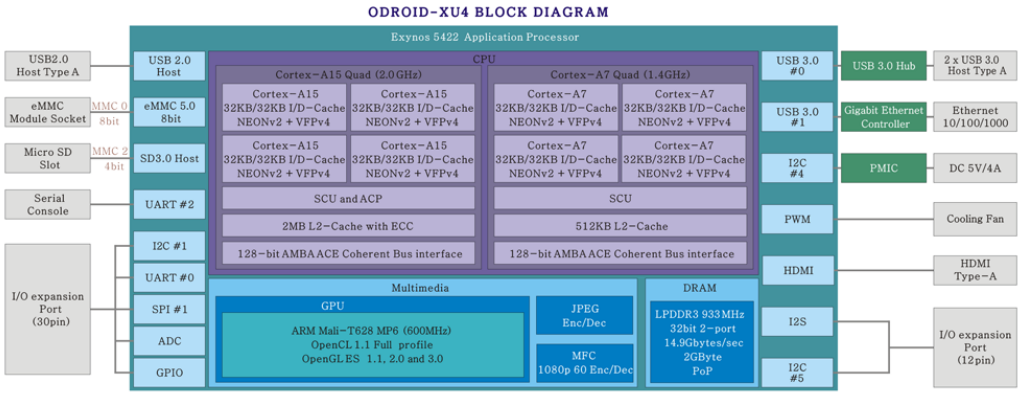

Figura 3: Diagrama de bloques de la placa Odroid XU4

El resto del trabajo está estructurado de la siguiente forma, en la Sección 2 se describe de forma detallada el entorno experimental y los pasos seguidos para su construcción. En primer lugar, se explica cómo usar un compilador cruzado y por qué es necesario en la Sección 2.1. A continuación, en la Sección 2.2 se describe el proceso seguido para instrumentar el runtime de OpenMP y en qué partes del código fue necesario introducir llamadas a la librería de PMCTrack para recoger métricas de la paralelización de bucles en tiempo de ejecución. Más adelante, en la Sección 2.3 se expone la forma de activar la instrumentación para ejecutar un benchmark y medir el rendimiento de su ejecución con diferentes métricas. Por último, en la Sección 3 se presentan los resultados obtenidos al instrumentar la ejecución de benchmarks paralelos en la placa Odroid XU4 y en la Sección 4 se exponen las conclusiones que he obtenido durante la realización de este trabajo.

2. Entorno experimental

En esta sección expondré los pasos seguidos para construir el entorno experimental usado en este trabajo. Además, introduciré diferentes conceptos relacionados con el runtime de OpenMP. Una buena parte de la información sobre el funcionamiento del runtime y de cómo se puede instrumentar proviene de un Trabajo de Fin de Master de la Universidad de Aalto (Finlandia), llamado Ïnstrumentation of OpenMP task scheduling"de Marko Rasa [1]. Este trabajo me resultó muy útil como investigación previa para ampliar mis

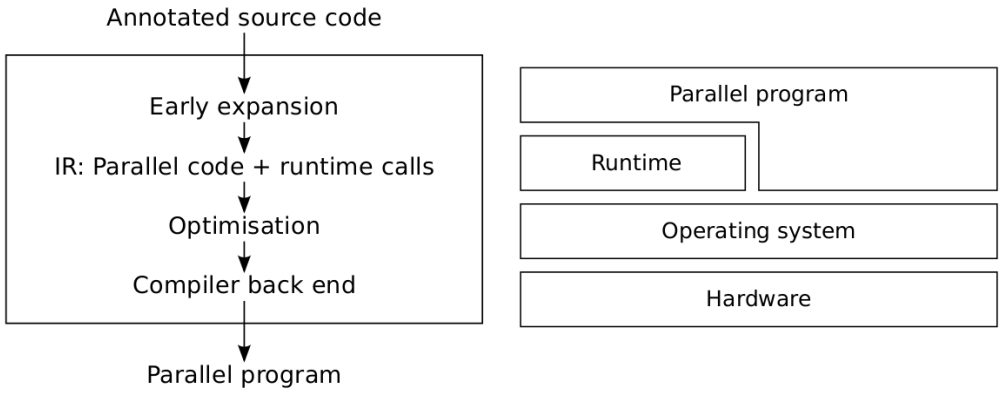

Figura 4: Proceso de compilación para un programa paralelo (izquierda) y software stack de la plataforma (derecha).

conocimientos sobre OpenMP y su funcionamiento a bajo nivel. Mantiene una aproximación similar, aunque intenta instrumentar la ejecución de programas paralelizados con tareas lo que no representa exactamente el objetivo de este trabajo.

El runtime system de OpenMP provee a los programas de usuario las herramientas para paralelizar su ejecución actuando de interfaz entre el compilador y el sistema operativo, este último expone diferentes recursos como la gestión de memoria dinámica e hilos al runtime. Que a su vez se encarga de proveer una abstracción de estos recursos para que el compilador pueda generar los binarios de diferentes programas, que usan estos recursos mediante llamadas a la librería del runtime system. En la Figura 4 se muestran dos esquema, a la derecha se explica la arquitectura explicada anteriormente y a la izquierda se muestra un esquema del proceso de compilación típico de un programa paralelo. Como se explica en [6], el proceso comienza con el código fuente anotado con directivas de compilación. Estas anotaciones son expandidas a llamadas al runtime y tras pasar una fase de optimización el backend del compilador genera el ejecutable final.

El primer problema que debía afrontar era ser capaz de modificar y compilar GCC junto al runtime de OpenMP con el código necesario para instrumentar su ejecución para la arquitectura de 32 bits de ARM de la placa Odroid. Sin embargo, las placas de este tipo tienen una memoria insuficiente para hacer una compilación de GCC junto a sus librerías adicionales, entre ellas la que necesitamos modificar (libgomp). A continuación explicaré el

procedimiento seguido.

2.1. Construcción de un compilador cruzado

El proceso de compilación parte de una máquina de 64 bits con un compilador GCC instalado, que usaremos para construir un compilador cruzado a partir de las fuentes modificadas de libgomp y GCC. La guía seguida para construir un compilador cruzado se puede encontrar en [7]. Explicaré a grande rasgos los pasos seguidos para su construcción:

- En primer lugar se deben generar los compiladores GCC y G++ para la arquitectura x86.
- Tras esto, se usarán estos compiladores para generar los headers de la librería estándar de C y los archivos de inicialización (stdio.h, stdlib.h, pthread.h, libc.so, etc.).
- Usando los componentes ya generados de Glibc, se continuará con la compilación de la librería de soporte al compilador (libgcc.a y libgcc.so).
- En este punto, ya es posible compilar la librería estándar de C (libc.a y libc.so).
- Finalmente, compilaremos la librería estándar de C++ (libstdc++.a y libstdc++.so).

Una vez completados estos pasos, podremos usar nuestro compilador específico para la arquitectura de 32 bits de ARM para generar los binarios modificados de la librería de OpenMP junto a su runtime system (libgomp).

Durante todo este proceso utilicé la máquina etna07 cómo host para realizar los diferentes pasos del proceso, para hacerlo más sencillo utilicé un túnel SSH y la herramienta sshfs para montar un sistema de ficheros en mi propia máquina que reflejase los cambios al código directamente en etna07 y poder compilar los ficheros allí usando sus 32 cores.

2.2. Instrumentación del runtime de OpenMP

A continuación se explicará el procedimiento seguido para instrumentar el runtime de OpenMP usando la librería de PMCTrack. Con el fin de aprender

a usar esta librería, se consultaron los ejemplos disponibles de forma pública junto a las fuentes del proyecto en el repositorio oficial de PMCTrack en GitHub [3].

El objetivo de la instrumentación es escribir los valores de diferentes eventos hardware a un fichero que representen la ejecución de los diferentes bucles de un programa paralelizado con OpenMP. Los pasos seguidos para modificar el código del runtime system son los siguientes:

- Copiar en el directorio libgomp/ los ficheros necesarios para enlazar con el modulo del kernel de PMCTrack: pmctrack.h, core.c y pmu_info.c.
- Incluir en el fichero libgomp.h y team.c los archivos copiados anteriormente.
- Añadir en la estructura gomp_thread de libgomp.h un descriptor de fichero para escribir los resultados de la monitorización y un descriptor de PMCTrack usado para interactuar con la librería (configurar, iniciar y parar los contadores hardware).
- El archivo team.c se encarga de gestionar las tareas de inicialización y gestión de un equipo de hilos que paralelizan la ejecución de un bucle. Utilicé la función *gomp_thread_start* para inicializar el descriptor de PMCTrack y configurar los eventos hardware que se desean monitorizar.
- El archivo loop.c contiene el código que comienza la parelización de los bucles. En concreto, usé la función *gomp_loop_init* para colocar la llamada a PMCTrack que pone los contadores en marcha.
- Por último, usé la función *free_work_share* del archivo team.c para colocar la llamada a la función que para los contadores y escribir la información al fichero de log.

2.3. Recolección de métricas

Para activar y configurar la utilización del código que instrumenta la ejecución de programas paralelos se utilizan una serie de variables de entorno:

- GCC_ROOT: Permite especificar el directorio raíz en el que se encuentran los binarios de GCC y libgomp modificados y compilados específicamente para la arquitectura ARM de 32 bits.

- GOMP_PMCTRACK: Si se encuentra esta variable de entorno, se activa la monitorización de bucles.
- GOMP_PMCTRACK_EVENTS: Esta variable permite indicar un string que especifique los eventos de contadores hardware a monitorizar, es posible utilizar tanto el código hexadecimal como mnemotécnicos.
- GOMP_PMCTRACK_FILE: Permite especificar la ruta a un fichero en el que se volcará la información de los contadores hardware.
- GOMP_PMCTRACK_NR_LOOPS: Si se define esta variable de entorno se establece un límite en el número de bucles a monitorizar, tras obtener información de los mismos se acabará con la ejecución del programa.

Usando las anteriores variables de configuración, se muestra a continuación un comando para monitorizar el número de instrucciones retiradas, ciclos de procesador completados y fallos de caché de último nivel para los bucles de un programa (*./example*) de la siguiente forma:

```
benchlocal@etna07:~/GCC_OMP/test$ GCC_ROOT=~/GCC_OMP/gcc-git/instdir
    GOMP_PMCTRACK=1 GOMP_PMCTRACK_EVENTS=instr,cycles,llc_misses
    GOMP_PMCTRACK_FILE=${PWD}/a.txt GOMP_PMCTRACK_NR_LOOPS
    OMP_NUM_THREADS=1 ./example
Number of threads = 1
Thread 0: 100000 iterations
Thread 0: 100000 iterations
Thread 0: 100000 iterations

benchlocal@etna07:~/GCC_OMP/test$ cat a.txt
[Event-to-counter mappings]
pmc0=instr
pmc1=cycles
pmc2=llc_misses
[Event counts]
nsample    pid      event          pmc0          pmc1          pmc2
      1  15715       self       2502869       2066839             0
[Event-to-counter mappings]
pmc0=instr
pmc1=cycles
pmc2=llc_misses
[Event counts]
nsample    pid      event          pmc0          pmc1          pmc2
      1  15715       self       2500340       1931197             0
[Event-to-counter mappings]
pmc0=instr
pmc1=cycles
pmc2=llc_misses
[Event counts]
nsample    pid      event          pmc0          pmc1          pmc2
      1  15715       self       2500338       1938776             0
```

3. Resultados

Los experimentos realizados en este trabajo tienen como objetivo motivar la necesidad de que el runtime system de OpenMP sea consciente de las diferencias en rendimiento que los hilos experimentan si paralelizan su ejecución en distintos tipos de cores en un sistema multicore asimétrico. Para ello, se procedió a ejecutar una serie de benchmarks tanto en cores big como small que paralelizan la ejecución de sus bucles con OpenMP mientras se medían diferentes métricas para caracterizar su rendimiento.

Para parsear los logs de pmctrack y poder generar las gráficas, he creado un script que coge todos los logs de una carpeta filtrando por un array en el que se le puede indicar el número de experimento y utiliza la librería pandas de python junto a matplotlib para crear las diferentes gráficas. Se puede encontrar el script junto a las fuentes de este trabajo en LaTeX en mi repositorio de GitHub [8].

Para medir el rendimiento he empleado la métrica Speedup Factor o SF, definido como $MIPS_{big}/MIPS_{small}$, que mide el ratio de rendimiento producido cuando un programa se ejecuta en un core big respecto a uno small. En los siguientes resultados presento dos tipos de gráfica, por una parte una que representa la evolución de los valores de SF medidos a lo largo del tiempo durante la ejecución de los diferentes bucles paralelizados con OpenMP y por otra una que presenta el porcentaje de muestras de SF que se caen en las diferentes particiones del espacio de muestras que se generan al dividirlos en rangos de 0.5.

En las Figuras 5, 6 y 7 se puede observar las medidas de Speedup Factor (SF) obtenidas durante la ejecución de 3 benchmarks paralelizados con OpenMP. Se tomaron métricas de la ejecución de 8 benchmarks en total, aunque he incluido las gráficas que representan las clases de comportamiento más dispares observados en todo el conjunto de experimentos. Por una parte existen algunos benchmarks, cómo los observados en las Figuras 5 y 6, cuyos valores de SF pueden experimentar una gran variabilidad a lo largo de su ejecución y sus bucles pueden presentar diferencias significativas en el beneficio relativo que obtienen al ejecutarse en un core big respecto a uno little. Además, los valores de SF a lo largo del tiempo de la Figura 5 muestran como el comportamiento de los benchmarks atraviesa diferentes fases claramente diferenciadas y mantiene un comportamiento que podría ser predicho si el runtime fuera capaz de darse muestrear los valores de diferentes métricas y ajustar su planificación de forma dinámica en tiempo de ejecución. Por otra

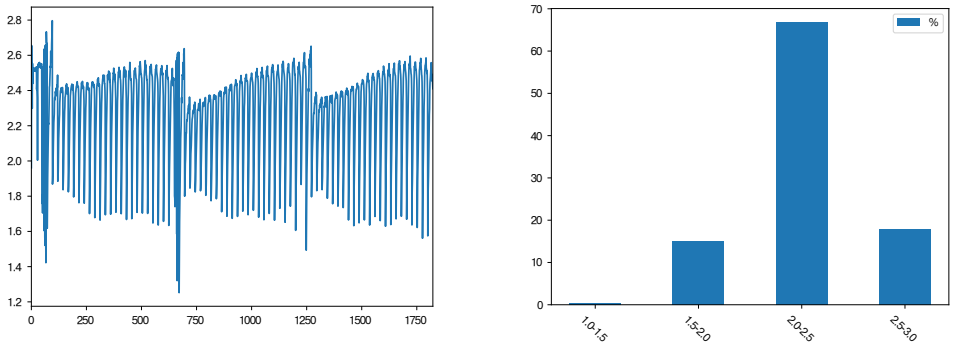

Figura 5: Variación de SF a lo largo de la ejecución (izda.) y porcentaje de muestras que pertenecen a las diferentes particiones de SF (dcha.) del benchmark rnaseq.

parte, existen otros benchmarks con un comportamiento similar al observado en la Figura 7, el 100 % de las muestras obtenidas para el benchmark blackscholes caen en la misma partición (2.5 a 3.0) y como se puede observar en la gráfica de variación a lo largo del tiempo la variabilidad es muy pequeña, presentando unos valores muy próximos a 2.6 durante toda la ejecución.

4. Conclusiones

En base a los experimentos realizados, se puede observar que el rendimiento obtenido por los programas paralelizados con OpenMP puede variar en gran medida cuando se ejecutan en cores con diferentes características. El hecho de que el runtime system ignore este hecho, genera situaciones de injusticia y problemas de rendimiento en los sistemas multicore asimétricos, especialmente cuando se ejecutan programas paralelizados con el paradigma tradicional de bucles (la mayor parte de benchmarks multihilo intensivos en CPU son así).

Considero que se podría mejorar enormemente el funcionamiento del runtime system de OpenMP si se implementasen estrategias de planificación que tuviesen en cuenta la variabilidad del Speedup Factor de cada hilo a largo de su ejecución y usase esa información para ajustar las decisiones de planificación del runtime. Por ejemplo, para benchmarks que muestran un comportamiento estable sería posible muestrear sus valores de SF durante una

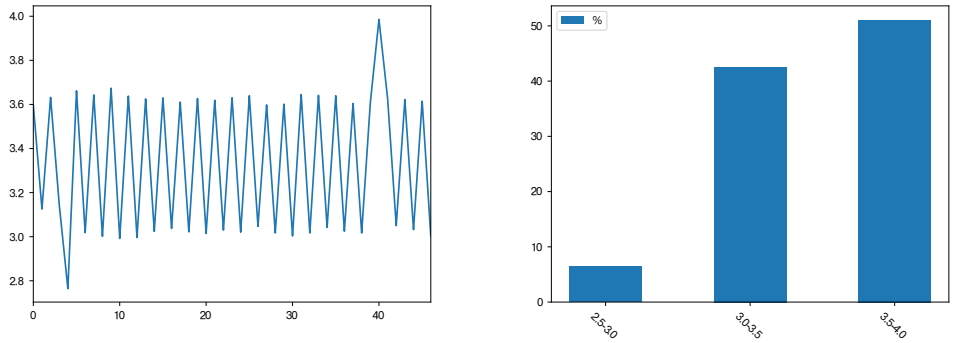

Figura 6: Variación de SF a lo largo de la ejecución (izda.) y porcentaje de muestras que pertenecen a las diferentes particiones de SF (dcha.) del benchmark semphy de Minebench.

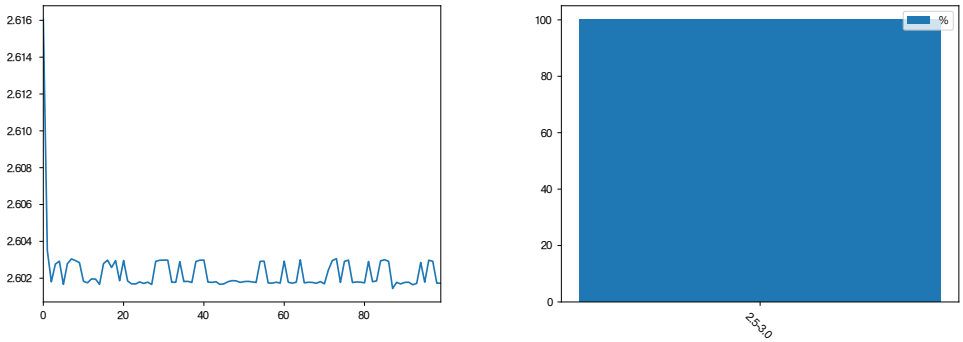

Figura 7: Variación de SF a lo largo de la ejecución (izda.) y porcentaje de muestras que pertenecen a las diferentes particiones de SF (dcha.) del benchmark blackscholes de PARSEC.

fase inicial de su ejecución y usar esos valores para ajustar la cantidad de trabajo que se le asigna a los hilos en función de si se ejecutan en un core big o uno small. De esta forma, el tiempo que tarda en ejecutar su porción de trabajo podría igualarse independientemente del tipo de core en el que se ejecutan. Estas ideas representan una avenida interesante de trabajo futuro, que posiblemente continuaré como parte de mi colaboración con Juan Carlos Saez Alcaide y que podría formar parte de mi Trabajo de Fin de Máster o, en caso de recibir la beca, de mi Tesis Doctoral.

Referencias

[1] Instrumentation of OpenMP task scheduling. Author: Marko Rasa. Master's Thesis of the Degree Programme of Computer Science and Engineering on the Aalto University, Finland.

[2] *An OS-oriented performance monitoring tool for multicore systems*. Saez, J. C., Casas, J., Serrano, A., Rodríguez-Rodríguez, R., Castro, F., Chaver, D., & Prieto-Matias, M. (2015, August). In European Conference on Parallel Processing (pp. 697-709). Springer International Publishing.

[3] *Repositorio oficial de PMCTrack*. Ejemplos de código utilizados disponibles en *pmctrack/test/test_libpmctrack/*. Enlace a git.

[4] *Manual de usuario y guía de instalación*. Página web oficial.

[5] *Odroid XU4: User Manual*. Hardkernel. Página web.

[6] Antoniu Pop and Albert Cohen. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In Proceedings of the 15th Workshop on compilers for Parallel Computers (CPC'10), Vienna, Austria, July 2010. URL.

[7] Guía para la construcción de un compilador cruzado. URL.

[8] Fuentes de este trabajo junto al script de python creado para parsear los resultados de pmctrack y generar las gráficas presentadas en este trabajo. Enlace a GitHub.