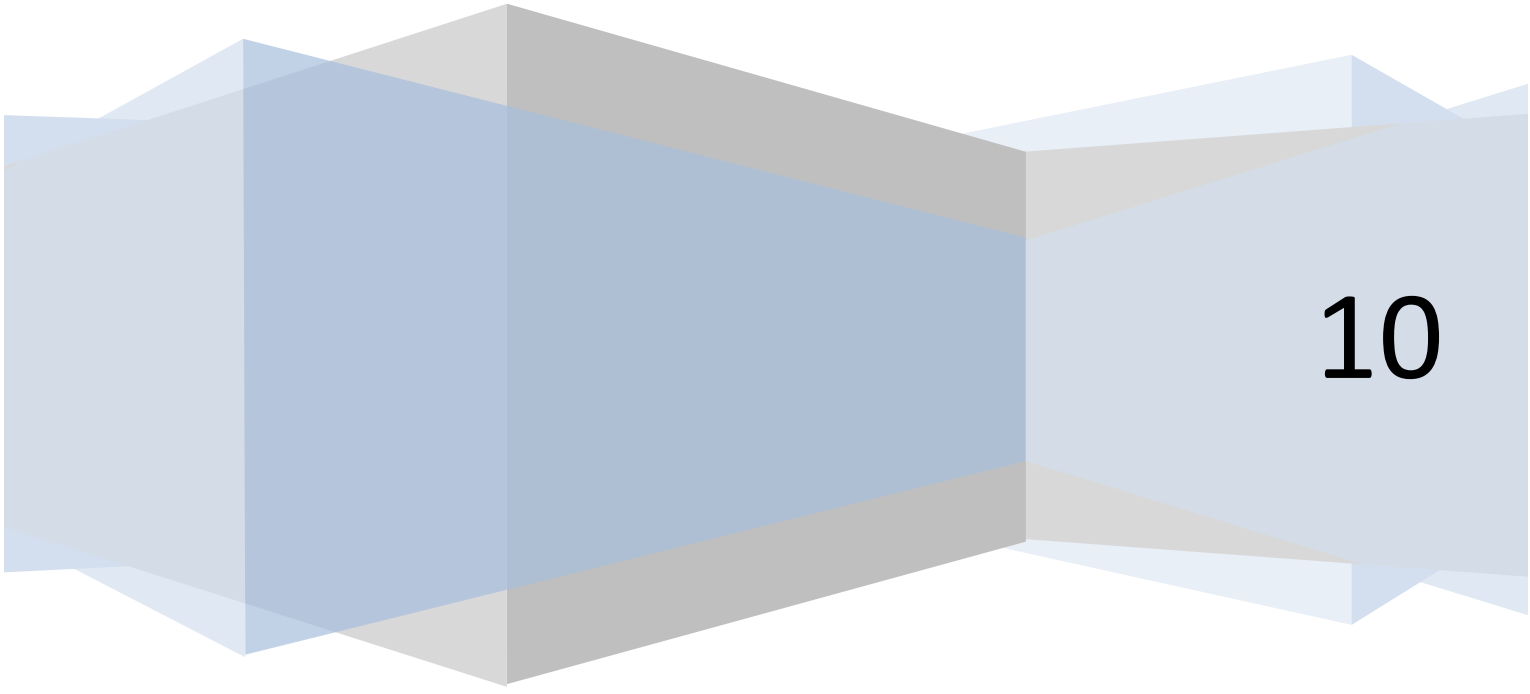
LAB3

# Paralelismo a nivel de Instrucción

## Laboratorio de Arquitectura e Ingeniería de Computadores

Valentin Puente




Revision 1.1: 20/05/2010

# 1 INTRODUCCIÓN Y OBJETIVOS

El objetivo fundamental de esta práctica es poner en práctica, usando *Simics* los conocimientos de paralelismo a nivel de instrucción vistos a lo largo de diversos temas en clase. Para ello, emplearemos la *MicroArchitectural Interface* o MAI de Simics.

Como en el resto de prácticas, la práctica tiene dos partes diferenciadas. Una parte guiada y una parte abierta. Todo el mundo tiene que realizar obligatoriamente la parte guiada y la nota será asignada en función de la corrección y presentación de los resultados hasta un máximo de 6 puntos. La parte abierta te permitirá realizar un trabajo mucho más creativo y en función de las tareas desarrolladas y los argumentos que las apoyen, podrás obtener hasta 4 puntos adicionales. Las partes guiadas han de ser desarrolladas de forma individual.

## 1.1 INTRODUCCIÓN A LA MAI DE SIMICS

Simics supone, por defecto, que cualquier instrucción se ejecuta en un solo ciclo de reloj. Esto permite simulaciones rápidas para evaluar aspectos relacionados con el software/firmware del sistema, el cuál es uno de los escenarios de uso más habituales de Simics. Como hemos visto en el laboratorio anterior, Simics puede ser extendido con módulos detallados de la jerarquía de memoria. Estos módulos permiten tener en cuenta de forma precisa su temporización en la evaluación de rendimiento, a expensas de incrementar el tiempo de simulación.

En este laboratorio iremos un poco más allá, empleando modelos que permitan contemplar el retraso preciso en la ejecución de cada instrucción en función del tipo de micro arquitectura empleado por el procesador. El modelo puede ser configurado para contemplar la temporización de las instrucciones ejecutadas como si fueran un procesador en orden o fuera de orden, tanto con ejecución superescalar o segmentada. La MAI sirve para que el modelo micro-arquitectural interaccione con Simics.

Los modelos, micro-arquitecturalmente detallados pueden ser escritos tanto en C como en el lenguaje de modelado de dispositivos de Simics (SDML). Para este laboratorio, en lugar de codificar nuestros propios modelos, vamos a usar un modelo ya facilitado por Simics. En concreto emplearemos la extensión MAI del procesador Sun UltraSPARC III (Bagel). Esta extensión permite ejecución fuera de orden, especulación en saltos e incluye una jerarquía de memoria.

Cuando estemos trabajando con modelos detallados, lógicamente aparecen un montón de nuevas variables para ajustar su comportamiento. Podemos configurar la anchura del pipeline (número de instrucciones que podemos hacer el *fech*, ejecutar, *retirar* y *commit* en un único ciclo), el tamaño del ROB, características del predictor de saltos, parámetros de la jerarquía de memoria relacionados con la ejecución, fuera de orden, etc… Iremos introduciendo estas variables a medida que las necesitemos en cada una de las partes de la práctica.

En este modo de ejecución, Simics tienen en cuenta todas las dependencias posibles, ya sean de control, registros o memoria. Para hacerlo construye un árbol de instrucciones. Este árbol de instrucciones a todos los efectos juega el papel de ROB en un procesador convencional. El contenido del árbol puede ser examinado con el comando `print-instruction-queue`. La cola de loads/stores también es simulada.

El módulo empleado en este laboratorio también incorpora especulación de saltos, aunque el predictor de dirección es ligeramente diferente a lo visto en clase. En este caso lo que hace es colocar en el árbol de instrucciones, los dos caminos posibles del salto. Las instrucciones son especulativamente ejecutadas hasta que el salto precedente es resuelto y entonces solo aquellas que se encuentran en el camino correcto son commiteadas. Otros procesadores como el MIPS R10000 empleaba esta aproximación. Aunque el rendimiento alcanzado de esta política es similar al alcanzado por las técnicas vistas en clase, hoy en día no es usual su uso por las implicaciones energéticas.

La ejecución de las instrucciones esta divido en 6 etapas (init, fetch, decode, execute, retire, commit) y cada instrucción sólo puede avanzar de una etapa a la siguiente cuando todas las dependencias implicadas han sido satisfechas. De este modo las instrucciones pueden ser ejecutadas fuera de orden, pero aplicando de forma implícita el retraso asociado a la microarquitectura subyacente y su posición relativa el flujo de instrucciones ejecutado.

Un punto que es necesario hacer hincapié es que `steps` y `cycles` ya no son necesariamente equivalentes. Avanzar la simulación un ciclo puede implicar que múltiples o ningún paso tengan lugar. Similarmente, avanzar la simulación una instrucción puede pausar la simulación en el medio de un ciclo. Por esta razón, es recomendable usar `cycles` y después contar el número de `steps` que han tenido lugar**.**

Puedes consular la guía de usuario de la MAI incluida en la documentación de apoyo de la práctica.

### 1.2 PUNTOS CALIFICABLES

Deberás entregar en formato ***pdf***[1] en través de la tarea abierta en WebCT un documento que responda a los siguientes apartados de la práctica:

## 2 PARTE DIRIGIDA

### 2.1 METODOLOGÍA GENERAL

[1] Cualquier documento que no se adecue a este formato (rar, doc, docx, etc…) no será evaluado

Solo debes estudiar el comportamiento del sistema desde un punto de vista arquitecturalmente preciso cuando estés seguro que la aplicación se encuentra en una zona representativa. Para ser lo más eficiente posible en esta tarea, debes hacer un uso adecuado de los tres modos de simulación de *Simics*. Solo debes simular en modos altamente detallados cuando sea preciso obtener datos precisos de lo que está ocurriendo en el sistema.

La metodología a seguir en este laboratorio está compuesta por los siguientes puntos:

i) Iniciar Simics en modo rápido, pero usa un tipo de máquina que sea micro arquitecturalmente extensible (MA)
ii) Monta el sistema de ficheros del **host** y copia en el **target** los ficheros necesarios
iii) Haz un checkpoint
iv) Reinicia Simics en modo *stall* y ejecuta el benchmark hasta haber calentado las caches
v) Haz un checkpoint
vi) Reinicia Simics en modo MA y recolecta los datos necesarios

Durante este proceso iterativo, Simics puede devolver errores dependiendo del modo que estés empleando o desde que checkpoint estés arrancando. Siempre y cuando tu simulación finalice con éxito, obvia los mensajes de error que reporta.

## 2.2 SETUP

Inicia Simics en modo **–fast** usando `targets/sunfire/bagel-ooo-common.simics`. Asegúrate de copiar en ese punto todos los ficheros facilitados dentro del target. Para ahorrarte tiempo en secciones posteriores crea un checkpoint. Posiblemente sea buena idea hacer algo similar en las *magic* de cada uno de los benchmarks a comprobar. Desafortunadamente los *checkpoints* creados en prácticas anteriores no sirven, ya que para habilitar **MAI** los hacemos incompatibles. Como ves, todo es mucho **más lento**. Procura ser ordenado con la creación y ordenación de *checkpoints* para no repetir trabajo innecesariamente.

Recuerda habilitar los puntos de ruptura *magic*. Es posible que al arrancar desde el checkpoint salga un error relativo a '*last_cache*'. Ignóralo. Después, para cada benchmark:

Arranca Simics y suponiendo que ya tienes los binarios de los benchmarks copiados en el disco (bzip_sparc, mcf_sparc, soplex_sparc), haz lo siguiente:

```
host$ ./simics -c exec_loaded_with_ma.conf

simics> magic-break-disable

simics> c

target# ./<benchmark>_sparc input.<jpeg/in/mps>
```

Si la aplicación no ha finalizado después de unos 2 minutos de ejecución podemos interrumpirla en el target (con ^C). En este punto ya hemos completado la carga en memoria del ejecutable y las librerías o código compartido requerido para su ejecución. De esta manera, la ejecución posterior corresponderá en su mayoría al código de la aplicación. Durante la primera ejecución de una aplicación puede existir una porción significativa de código de sistema cuando el tamaño del ejecutable es significativo o se requiere la carga de librerías dinámicas. Por esta razón, *soplex* presenta fuertes variaciones entre la primera y segunda ejecución. Es posible que sea necesario volverá copiar desde el host algunos ficheros de entrada para re-ejecutar la aplicación.

```
simics> magic-break-enable

target# ./<benchmark>_sparc input.<jpeg/in/mps>
```

En este punto, grabar un checkpoint por benchmark con.

```
simics> write-configuration <benchmark>
```

~~Configura las caches del sistema, intentando que la jerarquía de memoria sea lo más parecida posible a la un Pentium 4. Busca la información donde consideres oportuno. Facilita el fichero de configuración desarrollado en la memoria de la práctica. Puedes emplear alguno de los ficheros de la práctica anterior como punto de partida, con la salvedad de que el modulo de memoria ahora se llama **g-cache-ooo.** En el punto 5.4 del manual de la MAI nos detalla que parámetros del modulo g-cache no son empleables aquí.~~

A continuación pasamos a calentar las caches con una simulación de 100 millones de ciclos y crearemos un checkpoint al final.

## 2.3 CPI DE UN SEGMENTADO USANDO LA MAI

En esta sección estableceremos cual es el paralelismo a nivel de instrucción de cada uno de los benchmarks. Para lograrlo, ejecutaremos cada benchmark en una maquina simulada que posee un único procesador segmentado. Calcularemos con precisión cuál es el CPI de cada benchmark. Inicia simics en modo **–ma** cada uno de los *checkpoints* generados previamente con:

```
host$ ./simics –ma –c <benchmark>_warmed_caches
```

Parametriza la configuración del pipeline para ajustarlo a un segmentado sencillo. Para lograrlo fijaremos a 1 el número de *executes* y *commits* por ciclo. Modifica además el miss-penalty de las cache datos (un solo nivel), fijandolo a 10 ciclos. La cache de instrucciones se supone ideal.

```
simics> ma_cpu0->execute_per_cycle = 1
```

```
simics> ma_cpu0->commits_per_cycle = 1

simics> staller_cpu0->stall_time = 10
```

Anota la información relativa a la cuenta de instrucciones previa al comienzo de la ejecución (ya no disponemos de tracer). El comando retornará información relativa a la cuenta de instrucciones (step) y ciclos (cycles) del target. No te olvides de resetear también las estadísticas de la cache antes de realizar la medida.

```
simics> cpu0.print-time

simics> cache_cpu0.reset-statistics
```

Ejecuta al menos 10 millones de ciclos y contabiliza el número de instrucciones que se ejecutan en este tiempo. Recuerda que ahora instrucciones y ciclos no son equivalentes, por lo que deberás utilizar el comando rc. Para contabilizar el número de instrucciones deberás ejecutar de nuevo el comando *print-time*, anotando el nuevo valor de *steps*.

```
simics> rc 10_000_000
```

¿Cuál es el benchmark con mejor CPI y cuál el que obtiene el peor?

¿Qué proporción de la perdida de rendimiento está originada en la efectividad de la jerarquía de memoria? ¿Y en los riesgos de datos o control? ¿Cómo se modifican estas proporciones si la memoria pasa a estar a 100 ciclos?

## 2.4 EFECTO DE LA ESPECULACIÓN EN LOS SALTOS

El modelo de MAI que hemos usado previamente especula en los saltos sí no que para el pipeline cuando encuentra un salto. Simics proporciona otro modelo más complejo que permite especular en los saltos. Este es el target `bagel-ma-common.simics` ([1] pág. 30). ¿Puedes establecer para cada uno de los benchmarks anteriores cuanto beneficio reporta la especulación llevada a cabo por `bagel-ma-common.simics`? Es suficiente con que hagas esta evaluación para un solo benchmark. Procura que sea de los que menos entrada salida tenga.

Para hacerlo debes proceder igual que el caso anterior, teniendo especial cuidado en añadir el comando siguiente antes de simular:

```
simics> cpu0->reorder_buffer_size = 4

simics> ma_cpu0->retires_per_cycle = 1
```

## 2.5 ARQUITECTURA SUPERESCALAR CON PLANIFICACIÓN DINÁMICA

En esta sección examinaremos cual es el efecto de ampliar el ancho de issue del procesador para el target `bagel-ma-common.simics` con uno sólo de los benchmarks (el mismo que el empleado en el punto 2.4). Para lograrlo deberás elegir todos los parámetros siguientes:

```
simics> ma_cpu0->fetches_per_cycle = <width>
simics> ma_cpu0->execute_per_cycle = <width>
simics> ma_cpu0->retires_per_cycle = <width>
simics> ma_cpu0->commits_per_cycle = <width>
simics> cpu0->reorder_buffer_size = 32
```

Cambia de forma coordinada el tamaño de *width* a {2,4,8,16} manteniendo el tamaño del *reorder* buffer constante a 32. Repetir las mediadas con un *reorder* buffer de 64. ¿Cómo crees que interacciona el *reorder* buffer con la anchura del pipeline? ¿Qué factores crees que están limitando la efectividad de incrementar el ancho del pipeline?

## 2.6 EFECTO DE LA JERARQUÍA DE MEMORIA EN LA EFICIENCIA OOO: EL MEMORY WALL

En esta sección analizaremos como afecta la jerarquía de memoria en la máquina con ejecución fuera de orden. Para llevar a cabo la tarea variaras el tiempo de acceso de la cache y la memoria principal. El tiempo de acceso a la cache es modelado con los parámetros `penalty_read` y `penalty_write`, del objeto `cache_cpu0` (Por defecto valen 1). El tiempo de acceso a la memoria se modela con el parámetro `stall_time` del objeto `staller_cpu0`.

Usando el mismo benchamark que en la sección precedente y una arquitectura con wide-issue 4 y reorder buffer de 32 entradas, determina el IPC para los siguientes combinaciones de parámetros en (lectura cache, escritura cache, memoria) :

**(1, 1, 10), (2, 2, 10), (5, 5, 10), (1, 1, 20), (1, 1, 50)**

¿Qué impacto tiene el incremento en el tiempo de acceso a la memoria en el rendimiento del computador? ¿Hasta qué punto logra la ejecución fuera de orden paliar el creciente incremento en el tiempo de acceso a la jerarquía de memoria?

# 3 PARTE ABIERTA

## 3.1 CONCURSO DE PREDICCIÓN DE SALTOS!

Realmente, en la práctica anterior hemos estado empleando ejemplos de uso de la MAI de simics. En esta sección vamos a usar el target `bagel-ma-common.simics` **y** modificarle. Para llevar a cabo esta tarea debes importar en tu ***workspace*** el módulo ***sample-micro-arch***. En ***modules/sample-micro-arch/sample-micro-arch.c*** tienes la descripción de la arquitectura. En principio el código es complejo. Nos centraremos exclusivamente en la predicción de saltos. Deberíamos mejorar la función original ***bpredict*** y añadir un DIRP con su BHT.

Suponer que se trata de un superescalar de 4 vías con un ROB de 64 entradas. Los parámetros de la jerarquía de memoria serán los de defecto. Propón el mejor predictor de saltos que se te ocurra teniendo en cuenta que el espacio disponible para implementarlo es de 32KB. Puedes consultar libros u otras fuentes de información que consideres oportuna, pero no copies código encontrado en internet (ojo, Google existe!). Compararemos la efectividad de todas las propuestas en una presentación pública en el laboratorio. La nota será proporcional a la efectividad de la idea presentada.

Opcionalmente esta parte se puede llevar a cabo con la infraestructura descrita en [2]. Sólo requiere un compilador de C++ para poder ser empleada.

# 4 REFERENCIAS

[1].- Simics Micro-Architectural Interface, User guide.

[2].- Contest Branch prediction. http://cava.cs.utsa.edu/camino/cbp2/cbp2-infrastructure-v2/doc/index.html