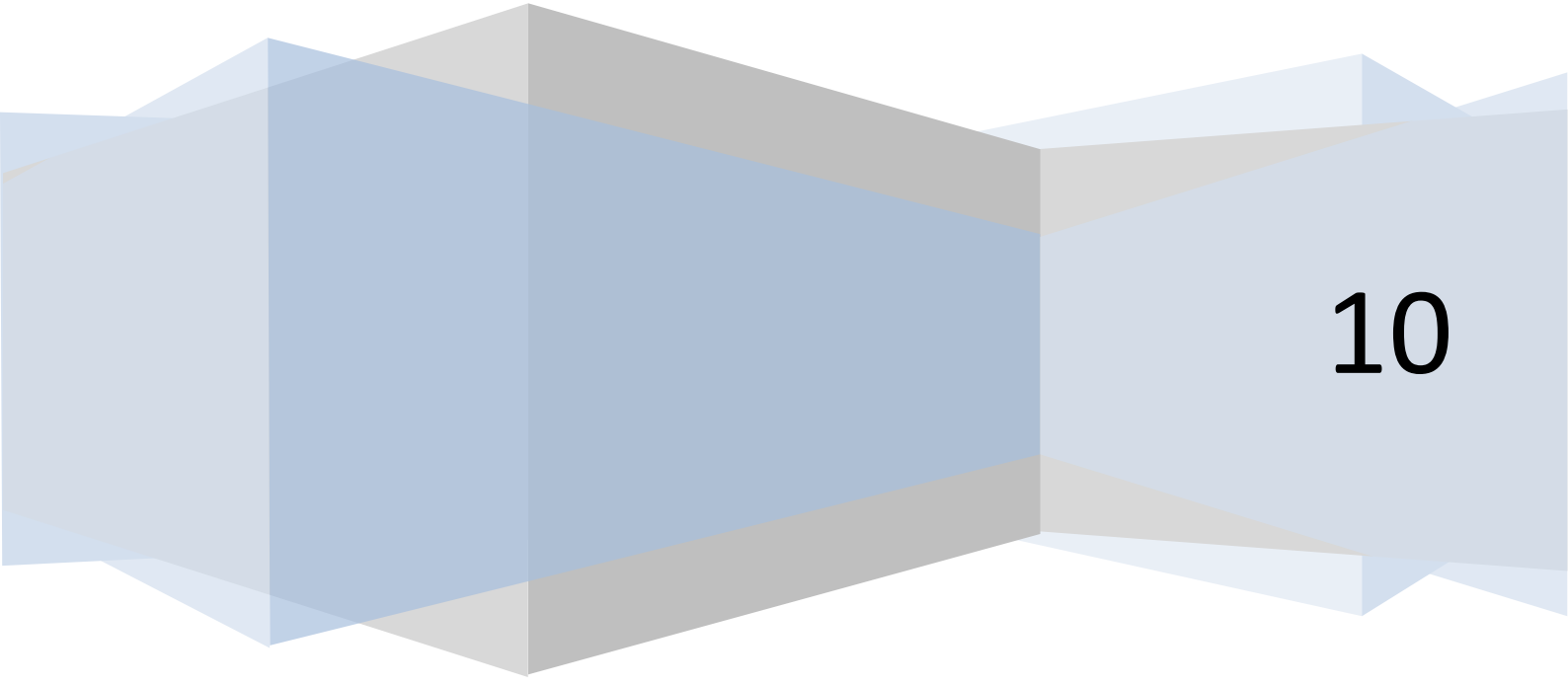
LAB2

Jerarquía de Memoria

Laboratorio de Arquitectura e Ingeniería de Computadores

Valentin Puente



10

Revisión 1.1: 02-Marzo-2010

1 INTRODUCCIÓN Y OBJETIVOS

El objetivo fundamental de esta práctica es poner en práctica, usando *Simics* los conocimientos de jerarquía de memoria vistos en clase. Para ello, emplearemos el modulo simulador de cache (g-cache) que nos permitirá obtener medidas precisas de rendimiento de la jerarquía de memoria y hacer modificaciones atendiendo al comportamiento.

El laboratorio tiene dos partes claramente 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 HERRAMIENTAS Y CONVENCIONES

En esta práctica se supone que has completado satisfactoriamente la práctica anterior. Los benchmarks y configuración de la maquina simulada que vamos a emplear son los mismos que en la práctica anterior por lo que podrás reutilizar checkpoints creados allí.

Para poder usar g-cache deberemos introducir un nuevo modo de ejecución de Simics: el modo ***stall***. Para más información de este modo consulta el Capítulo 18 de la Guía de Usuario de Simics.

1.2 PUNTOS CALIFICABLES

Deberás entregar en formato ***pdf*** 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

En su núcleo, Simics es un simulador de ISA, no un simulador de rendimiento. Es decir facilita la interface al software pero como se llevan a efecto no tienen por qué fidedignas con la realidad de la implementación (si lo fueran, si que tendríamos un simulador de rendimiento). Aunque Simics presenta un interface al SO y programas que se están ejecutando que hace parecer el computador **target** como si fuese un computador normal, muchas de las implementaciones de las funcionalidades ofrecidas distan mucho de la realidad. Por ejemplo, en el modo de ejecución normal (el que hemos empleado hasta ahora) el tiempo de acceso a memoria es ideal, es decir, requiere siempre un solo ciclo.

Sin embargo, entre otros, es posible conectar un módulo que modele adecuadamente el comportamiento de la jerarquía de memoria. Este módulo es el simulador cache **g-cache**. Con

este simulador conectado dentro de la jerarquía de memoria podemos determinar la efectividad de la jerarquía de memoria al ejecutar cualquier benchmark dentro del **target.** El módulo es completamente configurable y podemos alterar multitud de parámetros de la jerarquía simulada. Internamente **Simics** utiliza **Python**[1] como lenguaje de control. Para modificar los parámetros de **g-cache** usaremos sentencias Python (que empiezan por **@**).

Uno de los parámetros de **g-cache** es la latencia de una petición a memoria cuando pasa a través de cada uno lo diferentes niveles de la jerarquía de memoria. Ejecutando Simics con **–stall**, el usuario puede especificar a través de este módulo cuanto se retrasará la ejecución de la instrucción, teniendo en cuenta si los datos deseados se encuentran en memoria o en alguno de los niveles de cache. Simultáneamente, el módulo es capaz de monitorizar la actividad en los accesos a memoria y reportar estadísticas detalladas de su uso. Notar que todas estas operaciones han de ejecutarse por el simulador lo que hace que ejecutar cada instrucción del **target** sea mucho más costoso y por tanto más lenta la simulación.

Un aspecto más de detalle es que cuando las caches son conectadas a la simulación (típicamente después de un checkpoint) están vacías. Consecuentemente, el comportamiento no será parecido a lo que ocurriría en el sistema real ya que se producirán multitud de fallos compulsorios que son artificiales. Para obtener medidas de rendimiento correctas, debemos calentar las caches. Debemos ejecutar la aplicación durante unos cuantos millones ciclos hasta que las caches hayan alcanzado un estado más realista. En este punto reiniciaremos las estadísticas y estaremos en disposición de empezar la medida “verdadera”. Simularemos otros cuantos millones de ciclos y las estadísticas resultantes se aproximaran a lo que ocurriría en un sistema real ejecutando la aplicación que estamos evaluando.

Entonces, la metodología a seguir en esta práctica es como sigue:

1. Ejecutar Simics en modo rápido (sin –stall) y copiar las aplicaciones en el sistema simulado y preparar los benchmarks para su ejecución.
2. Sacar un Checkpoint
3. Reiniciar Simics en modo lento (con –stall)
4. Cargar las caches con el script .Simics que se facilita en el material de la práctica
5. Calentar las caches, ejecutando el benchmark
6. Pausar activamente o mediante un punto de ruptura.
7. Reiniciar las estadísticas de la cache
8. Continuar la ejecución del benchmark
9. Extraer las estadísticas útiles

[1] Se trata de un lenguaje interpretado, muy eficiente y con funcionalidades de programación orientada a objeto. Es muy común en muchos otros contextos, como desarrollo web, procesado de información, etc. Más información del lenguaje en www.python.org.

2.2 EXTRAYENDO ESTADÍSTICAS DE UNA CACHE SIMPLE

Descomprime los ficheros contenidos en el fichero benchmarks-1.tar y hazlos disponibles en la maquina simulada. Son exactamente los mismos ficheros que en los empleados en la práctica anterior. Copia todos los ficheros *.simics en un directorio de tu workspace. Para esta práctica usaremos de nuevo **targets/sunfire/bagel-common.simics.** Puedes emplear un checkpoint ya creado en la práctica anterior, para evitarnos el proceso de arranque. Creamos un checkpoint y nos salimos de Simics. Una vez tengamos los 3 benchmarks dentro del **target**, creamos un chekpoint nos salimos de Simics y seguimos el siguiente proceso para cada uno:

Arrancamos Simics sobre el checkpoint previo

```
host$ ./simics -stall -c <configuration name>
```

Ejecutamos la siguiente secuencia para asegurar el correcto funcionamiento de las caches y los benchmarks:

```
simics> magic-break-enable
simics> istc-disable
simics> dstc-disable
```

Cargamos el módulo correspondiente al simulador de cache. En este caso:

```
simics> run-command-file add-1cache-bagel-2.2.simics
```

Puedes ver la configuración elegido con:

```
simics> cache.info
```

Ejecuta uno de los benchmarks, con:

```
target# ./bzip2_sparc -z input.jpg
```

o,

```
target# ./mcf_sparc input.in
```

o,

```
target# ./soplex_sparc input.mps
```

Rápidamente alcanzaran un punto de ruptura. A partir de ahí ejecutamos 100.000.000 instrucciones[2].

```
simics> c 100_000_000
```

Reiniciamos las estadísticas de la cache, y analizamos las siguientes 1000_000 instrucciones con:

```
simics> cache.reset-statistics

simics> c 1_000_000
```

Muestra el comportamiento de la jerarquía de memoria con

```
simics> cache.statistics
```

Repetir con todos los benchmarks. Puedes finalizar el que está en ejecución y relanzar otro. Acuérdate de limpiar las estadísticas.

Para cada aplicación, anota el hit-ratio de todos los tipos de accesos a memoria (***fetch***, ***loads***, ***stores***). ¿Qué benchmark tiene el mejor comportamiento en cache? ¿Cuál es el peor?

2.3 DETERMINACIÓN DEL TAMAÑO DEL WORKING-SET O CONJUNTO DE TRABAJO

La tarea en esta sección es determinar cuál es tamaño del conjunto de trabajo de cada uno de los benchmarks analizados previamente. Esto lo podemos hacer variando el tamaño de la cache simulada hasta el valor adecuado. Podemos variar el parámetro correspondiente en el ***.simics** de configuración. Debe notarse que una vez la cache ha sido configurada a un tamaño determinado, cualquier modificación posterior desde la consola es ignorada por Simics. Si quieres hacerlo debes reiniciar Simics o crear un nuevo módulo de cache y reconectarlo después de la reconfiguración con el comando:

```
simics> @conf.phys_mem.timing_model = conf.<nombre_cache >
```

2.4 TAMAÑO MÍNIMO DE LA CACHE DE INSTRUCCIÓNES Y DATOS

Para esta sección emplearas caches de datos e instrucciones divididas. La configuración que vamos a emplear es la disponible en **add-2cache-bagel-2.4.simics**. Examina el fichero y date cuenta que hay una cache de datos (***dc***) y otra de instrucciones (***ic***) (¡Sólo tienen capacidad para

[2] Esto tardará unos 2-3 minutos. Más si usas la maquina virtual. Paciencia!

un bloque!). Además, aparece un nuevo módulo: el **id-spliter**. Este es el encargado de encaminar instrucciones y datos a la cache pertinente. Ejecuta el benchmark con esta configuración de cache. ¿Qué sugiere acerca de la localidad espacial y temporal de datos e instrucciones?

Modifica este fichero para que el tamaño de bloque sea 4 veces más grande. ¿Cómo modifica esto el rendimiento? ¿Qué sugiere acerca de la localidad espacial y temporal de datos e instrucciones?

2.5 OBTENER ESTADÍSTICAS DE UNA JERARQUÍA DE CACHE

En esta sección emplearemos una jerarquía de memoria más complicada en la simulación con el objeto de realizar un estudio de la cache más realista. Esta nueva jerarquía esta descrita en **add-2cache-bagel-2.5.simics.** La jerarquía posee dos niveles de cache con el segundo (**l2c**) unificado en datos en instrucciones y el primero separado en datos (**dc**) e instrucciones (**ic**). Notar que aparecen nuevos módulos: un **transaction-staller** que simula el retraso introducido por acceder a memoria. Esta configuración es similar a la presentada en la página 200 del manual de usuario de Simics. Al no ser una arquitectura x86 no necesitamos los ***splitters*** antes de L1 de datos e instrucciones.

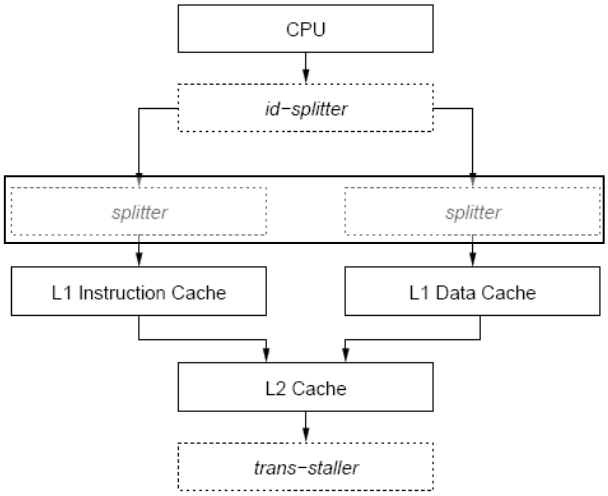


Ilustración 1 Jerarquía de Memoria Sección 2.5

Notar que los accesos desde la memoria ahora son dirigidos al **id-spliter**, dado que se encuentra en la cima de la jerarquía. Debes asegurarte de que cuando conectas las caches reconfiguradas en la jerarquía, la temporización de cada módulo desde el **id-spliter** hasta el **transaction-staller** está correctamente configurada.

Las tareas a realizar en esta sección son:

- Obtén los hit-ratios de L1 y L2 para los tres benchmarks en este sistema. ¿Cuál es la efectividad de L2 resolviendo los fallos que llegan desde el nivel L1?
- Modifica la cache L2 (l2c) para que ahora sea sólo doble de grande que cada una de las caches de L1. Elige un benchmark, y determina como las estadísticas de L2 se modifican. Haz lo mismo pero ampliando el tamaño de L1 al doble. ¿Qué conclusión puedes extraer de estos resultados?
- Calcular el tiempo promedio de acceso a memoria de cada una de las configuraciones probadas en el punto anterior.
- Tomando como base la configuración descrita inicialmente en **add-2cache-bagel-2.5.simics** suponer que el tiempo de ciclo para Bagel es 0.25ns, que hay 1.3 referencias a memoria por instrucción, y que una cache de primer nivel perfecta reportaría un CPI de 1.5. Posteriormente, asumiendo que un hit en la cache de datos establece el camino crítico del pipeline, cambiar el tamaño de la cache implica que el tiempo de ciclo del sistema debe ser modificado. Para una cache de datos de 16KB, el tiempo de ciclo debe ser incrementado en un 10%, y aumentarla a 32KB hace que el tiempo del procesador crezca un 20%. Si únicamente pudieras llevar a cabo una de las dos mejoras a cabo (sin tocar nada mas de la jerarquía de memoria, incluyendo I$ o L2) ¿Cuál es la mejora que crees que tiene más sentido? ¿Cuánto es el speedup de la mejor opción?

3 PARTE ABIERTA

3.1 AJUSTAR CÓDIGO PARA QUE ENCAJE EN LA JERARQUÍA DE MEMORIA

La multiplicación de matrices es una tarea común en una multitud de programas de índole científico o representación gráfica. A menudo, el rendimiento de las partes más críticas de estas aplicaciones se encuentra dominado por la ejecución de esta operación común. El tamaño de las matrices empleadas en estos cálculos puede ser bastante grande, por lo que el comportamiento de la jerarquía de memoria tiene un impacto significativo en su rendimiento.

Para resolver el problema de multiplicar eficientemente grandes matrices, muchos programas utilizan el algoritmo de multiplicación de matrices en bloques o *blocked matrix multiply* [1]. El algoritmo se basa en dividir la operación de multiplicación en iteraciones en las que se multiplica una sub-matriz de la matriz grande mucho más pequeña. Ajustando el tamaño de esta sub-matriz, el usuario puede controlar cuanto es el tamaño de datos empleados en cada uno de las iteraciones del algoritmo

Dado una implementación de este algoritmo, denominada *GEneralized blocked Matrix Multiply* o *GEMM* (**gemm-3.1.c**) y una jerarquía de memoria (**add-2cache-bagel-3.1.simics**), debes buscar un tamaño de bloque óptimo para la jerarquía propuesta. No puedes modificar ninguno de los parámetros de la cache o **#defines** del código C facilitado. Limítate a trabajar dentro del **main()** y el tamaño de bloque.

Recuerda que para compilar el código lo debes hacer desde el **target.** Además, el compilador dentro de Bagel solo acepta ANSI C. Debes hacer que **/net/aic/simics-3.0.31/src/include/simics/magic-instruction.h** esté disponible en el directorio **/usr/include/simics/** del **target**

Documenta los resultados obtenidos y justifica la elección adecuadamente.

3.2 AJUSTAR LA JERARQUÍA DE MEMORIA PARA UN BENCHMARK CRÍTICO

En esta sección realizaremos la tarea opuesta a lo hecho en la sección 3.1. En lugar de modificar el programa, vamos a modificar la cache. Una ventaja clara de trabajar con un simulador :) Esto podría ser análogo a crear una jerarquía de cache para una arquitectura de propósito específico. Usa **gemm-3.2.c** (puedes mirar el código, pero no debes modificarlo). Modifica tanto como desees **add-2cache-bagel-3.2.simics** incluso añadiendo nuevas caches. Recuerda que para compilar el nuevo código debes hacerlo desde el target.

Para la nueva jerarquía de memoria propuesta deberás especificar retrasos realistas. CACTI [2] puede facilitarte esos datos, además de potencia y área. Emplea un solo banco y tecnología de 0.07um. Las características más interesantes que reporta la herramienta están arriba de la columna central. Se consciente de que crear una cache enorme con un tiempo de acceso muy grande, o requerimientos energéticos elevados y demasiado área puede ser fuertemente contraproducente.

3.3 USA TU PROPIO CÓDIGO

Repite los estudios realizados en 3.1 o 3.2 pero emplea tu propio código. Quizás el código de una práctica de otra asignatura te puede servir. Reporta la parte de código o jerarquía de memoria que has modificado y razonando las decisiones de diseño tomadas.

La configuración por defecto del **target** acepta código C o C++ (que compilen con versiones 2.9 de **gcc** y **g++**). Código escrito en **Java** requiere modificar el **target** para soportarlo (instalar runtime y compilador). Se puede hacer pero requiere esfuerzo adicional y conocimientos de administración de Linux.

3.4 ESTUDIA EL EFECTO QUE TIENEN OTROS PARÁMETROS DE LA JERARQUÍA DE CACHE

Completa el análisis realizado, estudiando el efecto que tienen otros parámetros de la cache en el rendimiento del sistema para los benchmarks empleados. Consulta el capítulo 18 de la guía de usuario de Simics para conocer cuáles son los parámetros modificables y sus alternativas.

Algunas sugerencias para este estudio son:

- ¿Cuáles son las diferencias y efectos en el tráfico a memoria debido a las diferentes políticas de escritura?

- ¿Cuál es el efecto de las diferentes políticas de reemplazo en el rendimiento de la jerarquía de memoria?
- ¿Cuál es el efecto de acceder a los tags de la cache con la dirección virtual o dirección física?
- etc…

4 REFERENCIAS

[1]. Apartado 5.2 del H&P Edición 4ª.
[2]. CATI disponible en http://quid.hpl.hp.com:9081/cacti/