DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Escuelas de Ciencias Informáticas #32

Arquitecturas Avanzadas de Cómputo

Profesor Invitado, Javier Navaridas Palma

Trabajo final: Simulador de Caché L1

Mascitti, Julio Augusto - 954/11 - mascittija@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Introducción

El siguiente trabajo está enfocado en simular y experimentar los accesos a memoria que ocurren en una computadora al realizar lecturas y escrituras sobre la misma, proceso fundamental para el funcionamiento de cualquier arquitectura que sigue el modelo Von-Neumann.

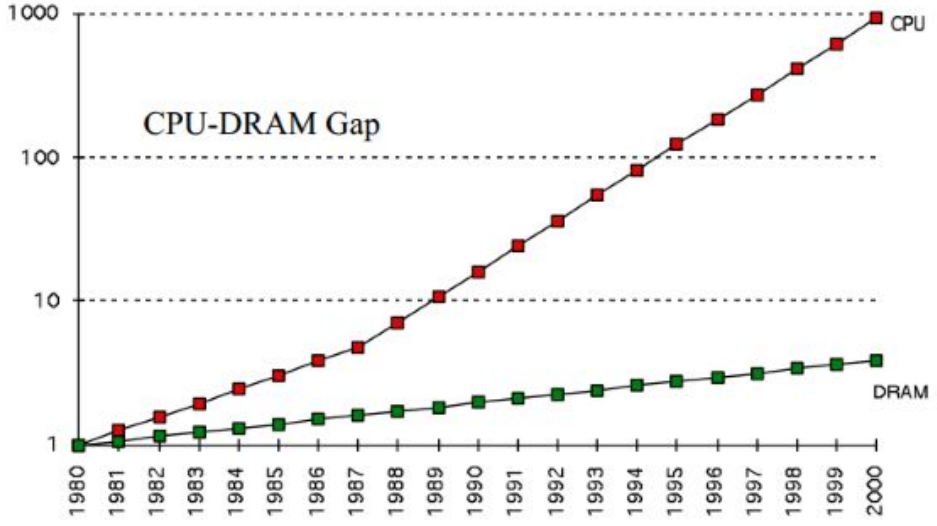
A lo largo de la carrera y en el curso pudimos, ver un panorama sobre el funcionamiento de las memorias. En todas estas oportunidades, fue meramente teórico y por ese motivo me pareció oportuno experimentar y realizar algo un poco más empírico.

Con respecto a la parte teórica, este informe pretende tener un resumen de los temas necesarios para entender a grandes rasgos, las ideas que subyacen detrás de lo que se está intentando evaluar, y no algo muy específico que entre en detalles y pierda el foco del trabajo en definiciones o implementaciones. A pedido del profesor, tampoco se incluirán los códigos de implementación necesarias para la simulación y los experimentos en el informe.

Se decidió simplificar varios aspectos. Como el de realizar una memoria caché L1 para un procesador single core. De esta manera, no es necesario pensar, diseñar, implementar o probar temas complejos, relacionados a las problemáticas de distintos niveles de caché o conflictos relacionados con la ejecución de varios procesadores simultáneamente.

Por lo tanto, se implementará una memoria principal, la cual será accedida por distintas rutinas para probar la performance de dichos accesos y luego se introducirá en el medio de la comunicación del procesador con la memoria principal, una memoria caché L1 y su respectivo controlador, para intentar mejorar la performance de la ejecución de dichas rutinas.

- Processor vs Memory Performance



1980: no cache in microprocessor;
1995 2-level cache

La utilización de cachés vino a solucionar la disparidad que se dió en la evolución de la velocidad de los procesadores y de las memorias.
Cuando los accesos a memoria, comenzaron a ser un factor determinante en la performance de los cómputos realizados, y el aumento de la velocidad de acceso no pudo seguir el ritmo de la velocidad de cómputo de los procesadores, fue necesario y revolucionario, la implementación y utilización de memorias caché.

Una memoria caché, es una memoria de acceso rápido, pero de tamaño reducido, para que el procesador pueda comunicarse con esta, y no tenga que pagar el costo de ir a buscar los datos a la memoria principal, la cual es más lenta. De esta manera, es necesario que los datos que vaya a utilizar el procesador, se encuentren en esta pequeña memoria, a la hora de requerirlos para procesar. De esto mismo, se intenta encarga el controlador de la memoria caché.

Para tal fin, las memorias caché explotan fuertemente los conceptos de vecindad espacial y vecindad temporal.

Principio de vecindad temporal
Una dirección de memoria que está siendo referenciada actualmente, es altamente probable que vuelva a ser referenciada en el futuro inmediato.
Por ejemplo cuando en un loop se utiliza una variable que es incrementada y es evaluada en cada paso para ver si se continúa dentro del mismo. Tener acceso rápidamente a este dato aumenta notablemente la performance de dicho loop.

Principio de vecindad espacial
Si se está accediendo a una dirección de memoria determinada, es altamente probable que se acceda a las direcciones de memoria vecinas en el futuro inmediato.
Por ejemplo, cuando se lee una instrucción a ejecutar, la probabilidad de que la próxima instrucción a ejecutar esté dentro de su vecindad es bastante alta. Con lo cual, tener de antemano el resto de las instrucciones, consigue aumentar la velocidad de acceso a las mismas.

En el diseño de la memoria caché se deben considerar varios factores que influyen directamente en el rendimiento de la memoria y por lo tanto en su objetivo de aumentar la velocidad de respuesta. Estos factores son las políticas de ubicación, extracción, reemplazo y escritura.
En nuestro modelo simplificado, solo nos encargamos de que sean variables las políticas de reemplazo y escritura. Utilizando una política de ubicación asociativa y una política de extracción por demanda de forma fija, sin realizar pruebas o experimentos para analizar sus comportamientos.

Política de reemplazo
Determina cuál bloque de la memoria caché debe abandonarla, cuando no queda espacio disponible para un bloque entrante.

La siguientes son las políticas que implementaremos y probaremos a lo largo del desarrollo y los experimentos.

- **Aleatoria**: El bloque a ser reemplazado se determina de forma aleatoria.
- ***FIFO***: Se usa el algoritmo *First In First Out* (primero en entrar primero en salir) para determinar qué bloque debe abandonar la caché. Este algoritmo generalmente es poco eficiente.
- **Usado menos recientemente** (***LRU***): Sustituye el bloque que hace más tiempo que no se ha utilizado en la caché.
- **Usado con menor frecuencia** (***LFU***): Sustituye el bloque que ha experimentado menos referencias recientemente.

Política de escritura
Determinan el instante en que se actualiza la información en memoria principal.
Las siguientes son las que utilizaremos a lo largo del trabajo.

- **Write Through:** Cada vez que se realiza una escritura, se escribe tanto en la memoria principal, como en la memoria caché.
- **Copy Back:** Cada escritura se realiza en la memoria caché y esta se ve impactada en la memoria principal solo cuando es desalojado el bloque de memoria que lo contiene de la caché.

Una de las métricas más importantes para poder mensurar el rendimiento de las memorias caché, es la del Hit Rate.
Un HIT se da cuando el procesador hace referencia a una dirección de memoria que se encuentra almacenada en la memoria caché, y de esta manera, no tuvo que pagar el costo de ir hasta la memoria principal a buscar el dato. De lo contrario, se considera que la operación realizada es un MISS.
El Hit Rate es la cantidad de HIT's sobre la cantidad totales de peticiones de direcciones de memoria, o sea, HITS + MISS.
De esta manera, se pretende que el Hit Rate de la memoria caché, sea lo más elevado posible. Lo cual implica, que se minimizaron los costos de ir a buscar información hasta la memoria principal, cada HIT fue una lectura óptima.

Las memorias caché tienen un tamaño reducido, pues el costo de las mismas, tanto monetariamente, como el consumo energético que generan, es elevado.
En el caso ideal se podría utilizar la tecnología de la memoria caché para hacer una memoria principal. Pero como esto no es factible, o por lo menos, no para producción en masa, se decide tener una pequeña porción de memoria, con tecnología de muy rápido acceso y a través de la controladora caché, conseguir muy buenos rendimientos del sistema en general, sin tener que recurrir a reemplazar toda la memoria por una de estas tecnologías.
Por lo tanto es común tener computadoras con memorias RAM de varios GBytes y cachés de algunos KBytes o MBytes, con rendimientos más que aceptables.

Descripción del trabajo

Para la implementación tanto del simulador, como de los experimentos, se decidió utilizar un lenguaje de alto nivel como python, con algoritmos y estructuras bastantes simples.

La memoria principal tiene direccionamiento a byte y es un arreglo en donde está contenida toda la información.

En la memoria caché se decidió tener 8 bytes por línea y se implementó con un diccionario, el cual como clave tiene la base de las direcciones que contiene. De esta manera, la memoria principal es dividida en bloques de 8 bytes y cada línea de caché tiene alguno de estos bloques en determinado momento.

Luego, para cada una de las políticas de desalojo, se decidió mantener estructuras complementarias para poder calcular el bloque de código a reemplazar cada vez que sea necesario.

- **FIFO:** Una variable que indica cual es la próxima línea a reemplazar comenzando en 0, incrementándose en cada reemplazo y volviendo a 0 luego de llegar a reemplazar la última línea de la caché.
- **RANDOM:** En cada reemplazo, se utiliza un número random para seleccionar alguna de las líneas de la caché a ser removida.
- **LRU:** Un arreglo que para cada línea de la caché, mantiene el timestamp de la última vez que fue utilizada esa línea y a la hora de reemplazar se busca la más “vieja”.
- **LFU:** Un arreglo que mantiene la cantidad de veces que fue utilizada esa línea de caché recientemente y en el momento de decidir cuál reemplazar, se busca la de menor valor. Para que una línea que fue referenciada una gran cantidad de veces no quede eternamente en la caché, se dispuso una reducción de este valor periódicamente, para que eventualmente, ese número, represente la línea menos veces utilizada “recientemente”.

Para las políticas de reemplazo, como se plantea en las definiciones, se escribe siempre en la memoria principal, pagando el costo de tener que acceder a la misma, o se realizan los cambios en la memoria caché, obteniendo una coherencia eventual, al momento de desalojar el bloque y que el mismo quede impactado en la memoria principal.

Para simular el costo de los acceso tanto a la memoria caché, como a la memoria principal, se decidió hacer configurable el tiempo que demora una lectura de memoria caché y luego un factor de delay, que representa cuanto más caro es ir a buscar un dato a la memoria principal.
Por ejemplo si una lectura de la memoria caché se configura en 1 segundo y configuro un factor de delay de 10 veces, el costo de las lecturas de la memoria principal es de 10 segundos.

Por último, para calcular el HIT RATE, cada vez que un acceso se realiza a un dato que está en la memoria caché se incrementa una variable que cuenta los HITS, y de lo contrario se incrementa una variable que cuenta los MISS. De esta manera al finalizar la rutina tengo los datos necesarios para calcular el Hit Rate correspondiente.

Repositorio Git Hub

Experimentos realizados

La primer idea de los experimentos, era cuantificar la diferencia de performance en las lecturas, intentando tener un valor real de las mismas. Pero al llevarlo a la práctica, me dí cuenta que no tiene sentido intentar medir exactamente los tiempos de la simulación, porque al fin y al cabo no es hardware específico, si no es un simulador montado en un lenguaje de alto nivel y no es posible distinguir entre los cómputos y las lecturas propiamente dichas.
Por lo cual, las medidas de performance, dan una idea de órdenes de magnitud de la diferencia, pero no se puede pretender precisión, pues para esto habría que auditar el hardware y no la simulación.

Para entrar más en detalle tuve que empíricamente testear valores, para que los tiempos configurados de delay al simular las distintas lecturas, sean significativos en los tiempos de cómputos y no se pierdan dentro de la capacidad de procesamiento. Pude ver que si configuraba el simulador con tiempos de delay muy pequeños, estos eran despreciables a la hora de computar las rutinas y no podía percibirse la diferencia entre las lecturas sobre memoria caché y la memoria principal.

La configuración final para los tiempos de lecturas es de 0.01 segundos a la memoria caché y 0.1 a la memoria principal. Me pareció razonable utilizar 1 orden de magnitud más lentas las lecturas de memoria principal, que las lecturas de la memoria caché y en los resultados era perceptible tal diferencia.

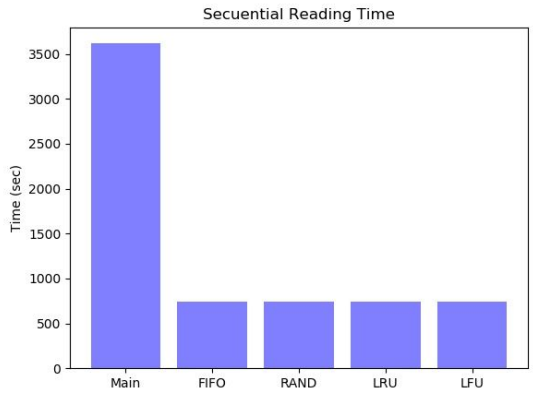
Con respecto a cuantificar los HitRates de la memoria caché, ver cómo las distintas políticas de desalojo eran capaces de modificar el rendimiento en la performance de las lecturas. A mayor cantidad de HITS, mayores lecturas acertadas, mayor HitRate y mayor performance en el desarrollo total del cómputo.

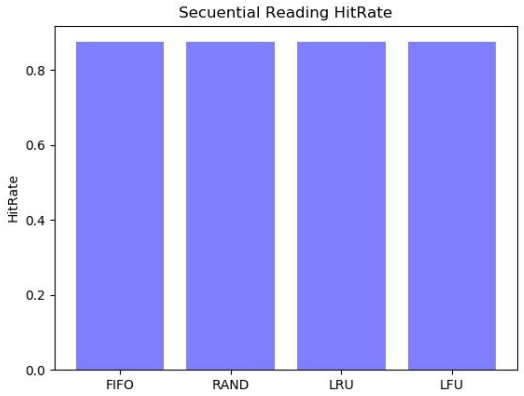
Se plantearon varios experimentos para poder ver distintos aspectos del funcionamiento de las memorias caché y sus eventuales configuraciones, que fueron surgiendo a lo largo del desarrollo. Se trató de probar algunos conceptos básicos como las lecturas y escrituras y surgieron nuevos experimentos a lo largo de la ejecución de los mismos.

Para los experimentos, se utilizó una memoria principal con 2**22 direcciones de memoria posible. Y una memoria caché con 2**4 líneas y en cada línea de la memoria caché se pueden alojar bloques de 8 KB.

Lecturas secuenciales

Una primer idea, para poder ver el beneficio de la utilización de memorias caché, es la de simplemente realizar una lectura de valores secuenciales de la memoria. Cada bloque de memoria que se lea, va a realizar 1 MISS y 7 HITS. Con lo cual la performance de rutinas que utilizan una estructura de accesos a la memoria de este estilo, tienen que ver mejorada su performance drásticamente.



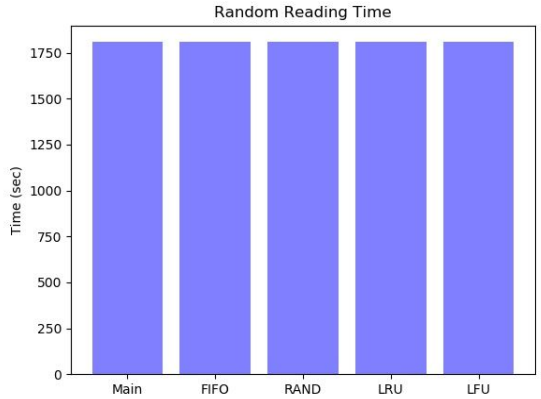


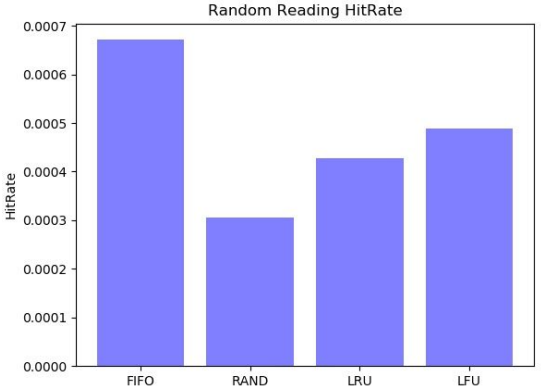
Se puede apreciar fácilmente, cómo sin importar la política de desalojo, las lecturas realizadas con una memoria caché, bajan considerablemente el tiempo de procesamiento de esta rutina. Alrededor de 5 veces más rápido.

Con respecto al HitRate, podemos ver que no importa cual sea la política de desalojo, para este tipo de comportamiento todas están alrededor del 87%. Esto se da pues no importa de qué forma desaloje los bloques, cada bloque lo leo por completo y luego nunca más vuelvo a utilizarlo.

Lecturas Aleatorias

En contraste con el experimento anterior, si las lecturas de disco no respetan ni la vecindad temporal, ni la vecindad espacial y no se puede predecir qué datos va a necesitar el procesador para agilizar sus cómputos, se puede pensar, que tener una caché no aportaría mucho a mejorar la performance de las rutinas.



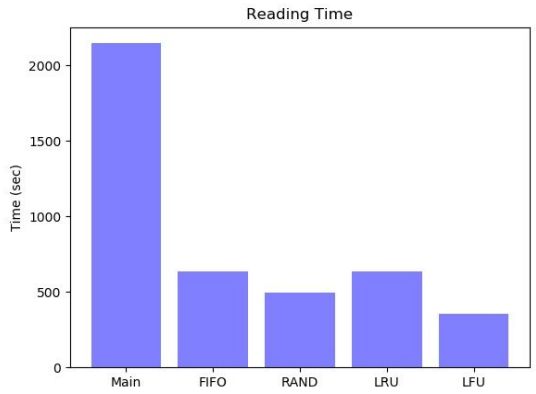


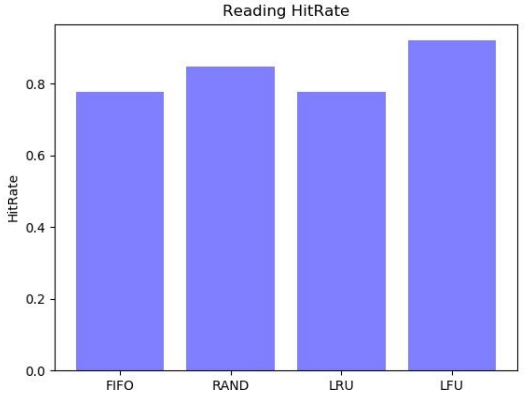
Podemos ver como los tiempos de procesamiento son idénticos, sin importar que los accesos sean directos a memoria principal, o pasen por la memoria caché.

Con respecto al HitRate, se puede ver que es prácticamente 0, con lo cual por más que se tenga una memoria caché, cada una de las peticiones tuvo que ir a buscar los datos a la memoria principal. Llo cual es peor porque hay estructuras y servicios corriendo para poder utilizar la memoria caché que no consiguieron ningún beneficio y sólo aportaron un overhead en el procesamiento de la rutina.

Lecturas para testear políticas de desalojo

Hagamos una rutina simple que pueda utiliza un poco los beneficios de tener almacenadas parte de la memoria en caché.
Para esto se van a leer varias veces las mismas 10 direcciones de memorias y otras 6 van a ser lecturas de direcciones aleatorias. Todo esto dentro de un loop para poder simular un tipo de lectura que no sea ni completamente secuencial ni completamente aleatoria.





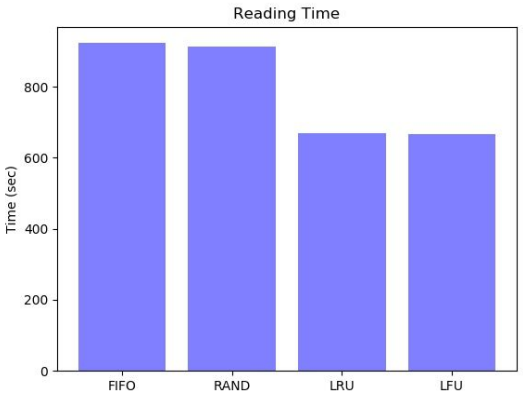
Podemos observar como dependiendo la política de reemplazo, tanto los tiempos de las rutinas como el Hit Rate de cada una de las memorias varían.

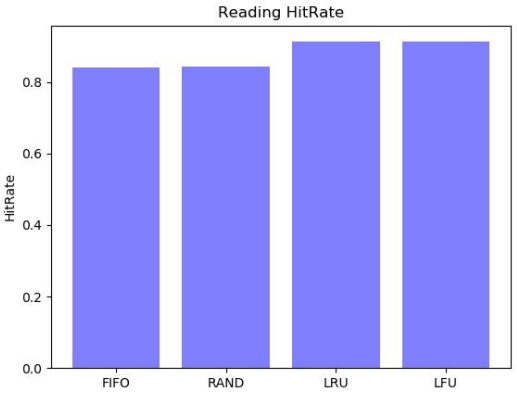
Se puede ver como hay una correlación directa entre el Hit Rate de una memoria caché y su tiempo de procesamiento de la rutina. Esta relación es inversamente proporcional. A mayor Hit Rate, menor tiempo de procesamiento de la rutina.

Lecturas donde LRU es mejor que FIFO

No se puede percibir del todo en los gráficos, pero en la rutina anterior la memoria caché con política de desalojo FIFO, se comportó exactamente igual que aquella que tenía una política de desalojo LRU. Pudimos ver que dado ciertas condiciones, la política de desalojo First In First Out se comporta exactamente igual que Last Recently Used. En este caso la lectura temporal de los datos, coincide con el orden de llegada. Por lo tanto el orden relativo de las lecturas de los bloques de memoria coinciden en las dos rutinas y a la hora de desalojar un bloque eligen exactamente el mismo.

Hagamos una rutina que pueda mostrar que hay diferencias entre estas dos políticas de desalojo. La lectura temporal, no coincide con la del orden de llegada. Por lo tanto cambiamos el orden temporal de las lecturas.

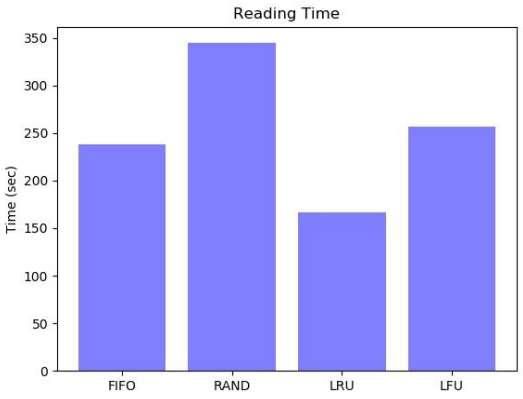


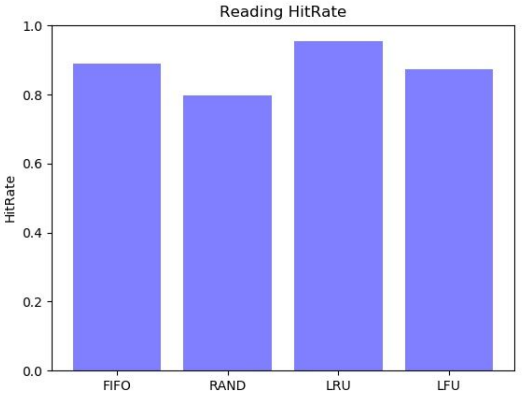


Vemos de esta manera como una caché con política de reemplazo LRU aumenta su eficiencia con respecto a la política de desalojo FIFO.
Es entendible, pues LRU utiliza un poco más de información a la hora de decidir qué bloque reemplazar de la memoria caché. Por otr parte, la política de desalojo con un algoritmo FIFO, no tiene ningún tipo de consideración sobre el comportamiento de las lecturas para tomar una decisión al respecto.

Lecturas donde LRU es mejor que LFU

Hasta ahora pudimos ver como las cachés con política de desalojo LFU, tienen uno de los mejores rendimientos, sin siquiera intentar maximizarlo. Lo cual puede inducir a pensar que en el común de los casos, es una buena política de reemplazo de bloques en la memorias caché.
Veamos si podemos conseguir una rutina, en la cual es más eficiente tener una política de desalojo LRU sobre una LFU.
Para esto vamos a tener bloques que se lean menos veces pero a lo largo de toda la rutina, y otras que se lean más veces en un ciclo pero luego nunca más.





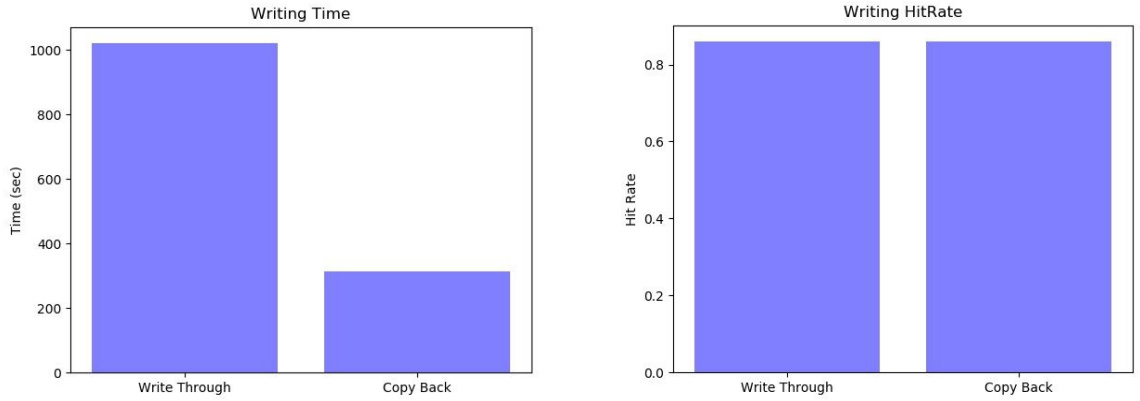
Se aprecia, cómo de esta manera, aquella caché que tiene una política de desalojo LRU es superior a la cual tiene LFU como política de reemplazo. Más aún, podemos ver como para

esta rutina una memoria con política de desalojo FIFO es un poco mejor que la que utiliza LFU.

Escrituras

Las escrituras en memoria, difieren según las políticas de escritura. Se realizan inmediatamente para mantener la coherencia en todo momento, o se acumulan los cambios y solo se ven impactados a la hora de desalojar el bloque de la memoria caché.

Hagamos una rutina que escriba posiciones de memoria y comparemos la performance teniendo una política de escritura Write Through o Copy Back.



Podemos ver como el HitRate es completamente igual, lo que indica que las direcciones que pretendían escribir estaban en la memoria caché, pero el tiempo de procesamiento es considerablemente superior cuando tengo una política de escritura Write Through ya que por cada escritura tengo que ir hasta la memoria, lo cual con Copy Back solo sucede cuando desalojo lá página.

Conclusiones

Se puede percibir los beneficios de tener una caché que intermedie entre la memoria y el procesador muy fácilmente. La vecindad temporal y espacial es inherente a la programación y sus codificaciones en la memoria. Con lo cual, poder explotar eso, es una herramienta fundamental para aumentar la performance de los cómputos, aumentando la velocidad de las lecturas.

Con respecto a la simulación, es posible complejizar el simulador todo lo que se quiera. Desde agregar más niveles de caché, como poder simular distintos procesadores que corran rutinas en simultáneo y accedan o no a la misma memoria caché. Todo esto trae aparejado sus problemas particulares, y tienen mucho más relevancias todas las políticas que intervienen en el funcionamiento de una memoria caché.

Otro aspecto fundamental, también es la simulación de las rutinas. Hoy en día hay caches para instrucciones, separadas de las de datos. Esto también cambia el panorama, pues no es lo mismo el funcionamiento de códigos que van leyendo instrucciones en su mayoría secuencial, dada la forma natural de programar algoritmos, que aquella caché que lee datos de la memoria.

Fue complicado intentar programar rutinas que le den algo de realismo al acceso de los datos en memoria, pues esto depende de un montón de factores. Desde la forma de programar, hasta las decisiones tomadas por el compilador. Con lo cual es otro área en la que se puede entrar más en profundidad y analizar con más detalles si se desea.

Por último, ninguna de las políticas elegidas para realizar los experimentos, puede ser tomada como “la mejor”. Dado que esto depende del problema en particular.
Para el caso general es evidente como LRU y LFU son superiores a FIFO y RANDOM. Pero estas, necesitan más procesamiento, que en software no es muy complicado, pero a la hora de llevarlo a hardware tiene sus desventajas.
Por lo tanto siempre depende del problema a atacar y el contexto, para poder decidir si alguna política sirve mejor que otra.

Reflexiones del curso

No se como se planifica y desarrolla un curso corto con avanzado en esta temática. Pero percibí que las primeras clases se centraron en dar un panorama básico de varios temas sin entrar en profundidad, ni que aporten un panorama amplio sobre los tópicos.
Entiendo que el cronograma fue escueto en tiempos. Pero por tal motivo, siento que hubo temas que no tendrían que entrar en el curso, si no, asumir que son conocimientos básicos para poder desarrollar el curso avanzado.
Los temas de las primeras dos clases son comunes a cualquier curso de Organización del computador de cualquier instituto que de una carrera informática. Por lo tanto, asumiendo un mínimo de conocimiento sobre el aera, esos dos días podrían haber sido aprovechados para entrar más en detalle sobre algún tópico expuesto, o utilizarlos para introducir algún otro tema a la currícula del curso.

Se nota sobradamente los conocimientos en el área de Javier. Tanto sea en temas en los cuales estuvo relacionado en su carrera, sea por investigaciones o desarrollos, como en aquellos de los cuales no son su área de conocimiento, pero pudo responder preguntas de todas maneras.

Me pareció muy interesante poder hablar de nuevas tecnologías que hay en el mercado y poder ver un panorama de para donde se está moviendo las ciencias de la computación en relación al hardware y sus problemáticas.

Hubiese sido interesante poder ver un poco más en profundidad la experiencia personal de Javier a lo largo su carrera, viendo un poco más cuáles fueron los temas investigados, que problemas puntuales se encontró, cómo los resolvieron y todo lo que sea relevante a las experiencias reales en la industria y la investigación que vivió, y le dan un carácter más tangible a los temas, que la exposición de alguno en particular yendo a la bibliografía o datos de público conocimiento.