Tarea Programada
Estructuras de Computadoras Digitales 2

Marco Antonio Montero Chavarría
Carné:A94000
Francisco Molina Castro
Carné:B14194

2 de Mayo ,2016

Abstract

El siguiente reporte contiene resultados obtenidos a partir del código del proyecto open source "cachesimulator". El mismo fue creado para el curso *IE-0521 Estructuras de Computadoras Digitales 2 de la Universidad de Costa Rica*.
Cabe destacar que el código fue realizado en lenguaje de programación *C++*, por lo que se cuenta con un Makefile para compilar el mismo y 2 archivos de ejemplo con el formato de las instrucciones de acceso a memoria para comprobar el funcionamiento.
El código del proyecto se encuentra disponible en la dirección:
"https://github.com/franmolinaca/cachesimulator/commits/master/src".

1 Introducción

La caché es una memoria que se sitúa entre la unidad central de procesamiento (CPU) y la memoria de acceso aleatorio (RAM) para acelerar el intercambio de datos, con el fin de minimizar el tiempo de ejecución de las diferentes tareas de la CPU. Esto se justifica con el manejo los mismos basándose en los principios de localidad espacial y temporal.
El principio de localidad temporal es qué tan probable se necesitará un dato particular en el tiempo, mientras que el principio de localidad espacial se basa en qué tan probable es que se requieran datos dentro de un mismo bloque. De esta forma, las cache buscan disminuir los tiempos de acceso al guardar datos que se utilizan muchas veces en un determinado tiempo y que tienen mas probabilidad de ser utilizados al pertenecer al mismo bloque de un dato previamente utilizado.
Dados estos dos parámetros, existen gran cantidad de memorias cache, de diferente tamaño de memoria, tamaño de bloque y otras variables como los diferentes tipos de funcionamiento para el acceso de los datos. Para el análisis se trabajó con dos tipos de funcionamiento, Mapeo Directo y Set Asociativo. A partir de esta división se hicieron otras 2 subdivisiones para probar con diferentes tamaños de cache y distintos tamaños de bloque.
El código creado toma información de un archivo de texto y a partir de esta, emula el comportamiento de una memoria cache. Verificando los accesos de memoria y se observó el miss rate a partir de tres modelos de cache distintos y se analizaron los datos obtenidos. Se utilizaron, un modelo mapeado directo y dos modelos asociativos, uno de 2 vías y el segundo de 4 vías.

2 Capítulo I

2.1 Explicación de alto nivel

El código realizado parte de la base teórica vista en el curso y toma 4 parámetros como entrada:

- El archivo con los accesos a memoria.
- El tipo de asociatividad (Es un entero. 1 para directamente mapeada y 2 en adelante para número de vias)
- El tamaño de la cache a emular.
- El tamaño del bloque.

Una vez obtenidos estos parámetros se calculan valores pertinentes para la simulación. Uno de ellos es el número de bloques en la cache, este se calcula a partir de dividir el tamaño de la cache y el tamaño de bloque. La cantidad de sets se obtiene a partir de dividir el número de bloques entre la asociatividad. Y finalmente la cantidad de bloques por set se define como la división del número de bloques entre la cantidad de sets. En el caso de la cache directamente mapeada, la cantidad de bloques sería igual a la cantidad de sets, por lo que este último cálculo sería igual a 1.

Para definir la estructura de datos de la cache en C++ se utilizó un *Vector* de la librería *STD*, con el fin de accesar a cada tag de manera independiente, ya que además se cuenta desde un inicio con tamaños definidos.

Para la simulación, primero se debe tomar en cuenta el funcionamiento de los dos modos de operación de la cache en el proyecto.En el tipo de cache de mapeo directo cada bloque tiene un lugar definido en la cache por lo tanto para ubicar un dato dentro del cache sólo se realiza la operación.
$Address\%BlocksInCache$
Donde Address es la dirección del bloque y BlocksInCache es el número de bloques en la cache. Entonces para leer o escribir un dato de la cache se toma la dirección y se busca a partir del tag si el dato en la posición devuelta por el módulo es el mismo que se quiere accesar, en caso contrario, se busca el dato en el siguiente nivel de memoria y se sustituye en la cache para un futuro acceso, siguiendo el principio de localidad temporal.

El modo asociativo funciona levemente distinto, en este caso la cache esta dividida en sets, por lo que con la misma operacion módulo, lo primero que es mapeado es el set y luego, dependiendo de la política de reemplazo, se elige el bloque dentro de ese set. Para obtener el set se realiza la misma operación que en el mapeo directo $Address\%SetsInCache$, pero esta vez SetsInCache es el número de sets en la cache.

Para el programa realizado, en ambos casos el valor del módulo se obtienen de la operación $Address\%SetsInCache$, el detalle radica en que para el mapeo directo la cantidad de sets se define igual a la cantidad de bloques.

Una vez definido la división dependiendo de la asociatividad, el siguiente nivel de subdivision es por bloques dentro de los sets. En el caso de la directamente mapeada, solamente encontramos un bloque por set, por lo que este es el que se reemplaza siempre, en el caso de las asociativas, se aplico la política de reemplazo de FIFO(First In First Out), de forma que el primer bloque en haber entrado es el que será reemplazado de primero cuando el set se llene y se ocupe trasladar un bloque nuevo.

En otras palabras, para memoria directamente mapeada se pregunta por el tag y se busca el mismo segun el index dado, en el caso de las asociativas, se encuentra el set y busca el mismo iterando sobre los bloques dentro del set, si no se encuentra, se escribe sobre el dato siguiendo la política FIFO.**First Input First Output**. Básicamente cada set en el caso de las cache asociativas, funcionan como una pila de papeles en la que se introducen los datos que provienen del archivo de memoria. Es decir el primer dato de entrada será el primer dato de salida en caso de estar la cache llena.

Por último, para verificar si el dato que se quiere accesar es el que está en cache, lo primero que se realiza es verificar el bit de validez, esto con *C++* se realiza verificando que la posición del vector es diferente de cero. Si la posición es cero, se produce un Miss obligatorio, y se guarda en el bloque respectivo el tag de la dirección y no la dirección misma, esto para mayor facilidad ya que al ser simulación no es de interés saber la posición exacta del dato, solo si efectivamente estaba en cache o no. Si la posición no es cero, se compara el tag del bloque que se quiere acceder, con el que estaba guardado en cache, si son iguales, es un hit, significando que el dato estaba en memoria, de lo contrario es un miss y se procede a revisar el resto del set para ver si se encuentra en alguno de los otros bloques del set o si del todo no esta, en cuyo caso se provoca un Miss y se procede a traer el nuevo bloque con la política de reemplazo FIFO.

En caso de que el lector gusté observar el código el mismo se encuentra en la dirección: *"https://github.com/franmolinaca/cachesimulator/commits/master/src"* como se indica en el abstract del presente trabajo.

2.2 Resultados

Siguiendo las indicación del enunciado de la tarea, se realizarón pruebas para los tres tipos de asociatividad con cuatro diferentes tamaños de memoria y cuatro diferentes tamaños de bloque para cada tamaño de memoria. Es decir se tienen 16 pruebas distintas por tipo de asociatividad, para un total de 48 pruebas.

2.2.1 Tablas

Ver tabla 1.

Table 1: Tabla resultados

Associativity	Mem Size	Block Size	Misses	Count	Miss Rate
1	1024	4	28766743	49642128	0.57948247
1	1024	16	30726598	49642128	0.61896214
1	1024	64	36702805	49642128	0.73934794
1	1024	128	42153376	49642128	0.84914523
1	65536	4	17290757	49642128	0.34830813
1	65536	128	18876248	49642128	0.38024655
1	65536	2048	27778691	49642128	0.55957897
1	65536	16384	41583511	49642128	0.83766576
1	1048576	4	403459	49642128	0.00812735
1	1048576	128	14873793	49642128	0.29962037
1	1048576	2048	13094696	49642128	0.26378192
1	1048576	16384	3496368	49642128	0.07043147
1	2097152	4	380200	49642128	0.00765882
1	2097152	128	9020950	49642128	0.18171965
1	2097152	2048	6962586	49642128	0.14025559
1	2097152	16384	10050	49642128	0.00020245
2	1024	4	15215717	49642128	0.30650815
2	1024	16	15845310	49642128	0.31919079
2	1024	64	20184792	49642128	0.4066061
2	1024	128	24096741	49642128	0.48540911
2	65536	4	5908661	49642128	0.11902514
2	65536	128	5348278	49642128	0.10773668
2	65536	2048	13149332	49642128	0.26488252
2	65536	16384	26275899	49642128	0.52930646
2	1048576	4	253481	49642128	0.00510617
2	1048576	128	3481200	49642128	0.07012592
2	1048576	2048	1850	49642128	3.7267E-05
2	1048576	16384	9879	49642128	0.000199
2	2097152	4	301276	49642128	0.00606896
2	2097152	128	868782	49642128	0.0175009
2	2097152	2048	1709	49642128	3.4426E-05
2	2097152	16384	5160	49642128	0.00010394
4	1024	4	12453295	49642128	0.25086143
4	1024	16	13619357	49642128	0.27435079
4	1024	64	15380917	49642128	0.30983597
4	1024	128	19980187	49642128	0.4024845
4	65536	4	4167683	49642128	0.08395456
4	65536	128	2642162	49642128	0.05322419
4	65536	2048	6415214	49642128	0.12922923
4	65536	16384	16433377	49642128	0.33103692
4	1048576	4	263293	49642128	0.00530382
4	1048576	128	810713	49642128	0.01633115
4	1048576	2048	1204	49642128	2.4254E-05
4	1048576	16384	5093	49642128	0.00010259
4	2097152	4	246086	49642128	0.0049572
4	2097152	128	129891	49642128	0.00261655
4	2097152	2048	1381	49642128	2.7819E-05
4	2097152	16384	2716	49642128	5.4712E-05

2.2.2 Gráficos

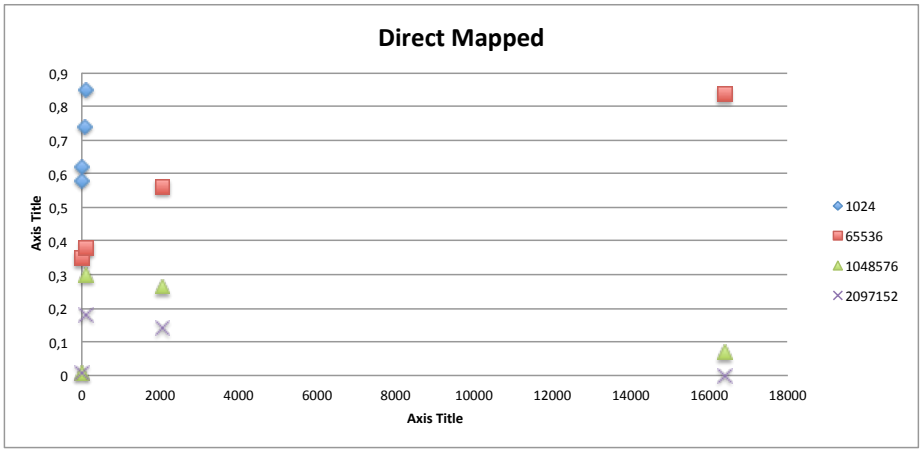

Figure 1: Direct mapped cache.

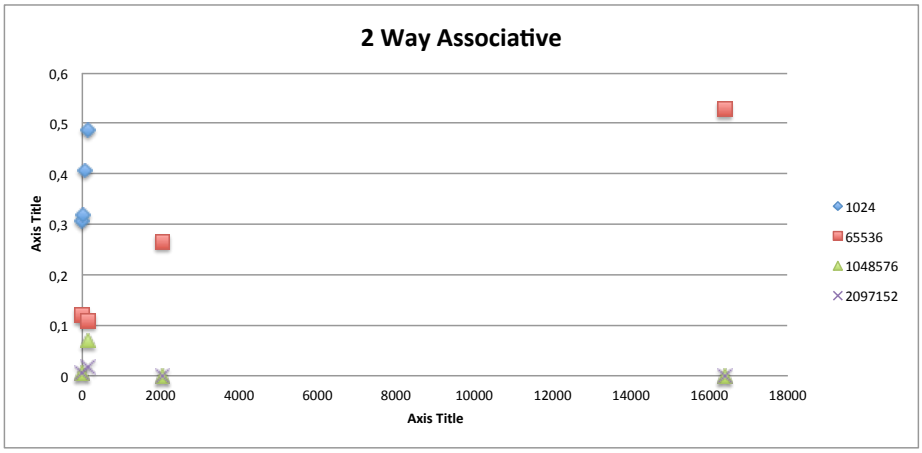

Figure 2: 2 Way associative cache.

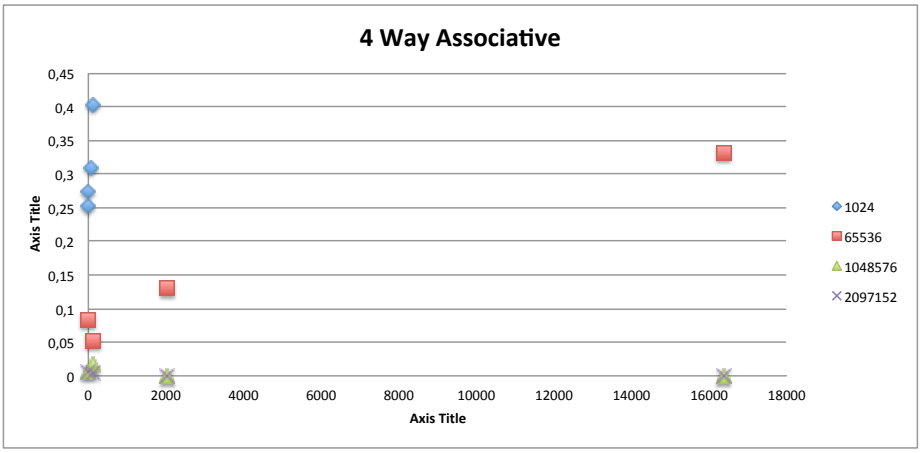

Figure 3: 4 Way associative cache.

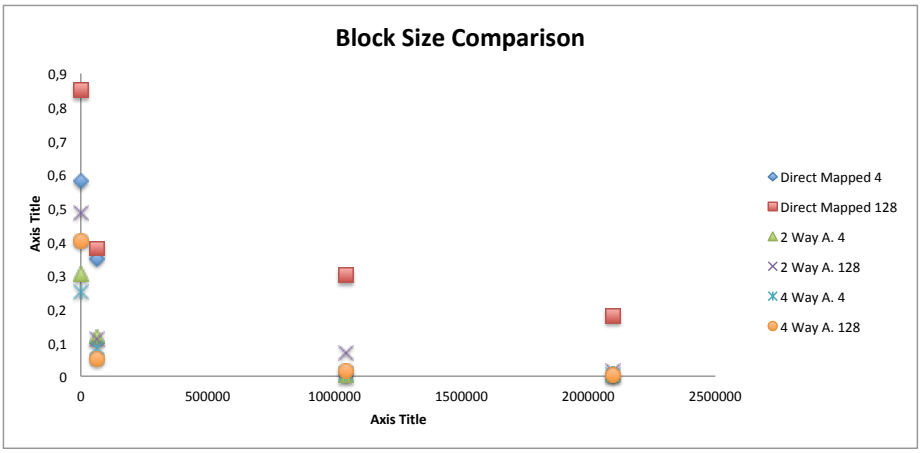

Figure 4: Comparación por tamaño de bloque.

2.3 Análisis Resultados

Se realizaron 4 comparaciones para concluir los diferentes desempeños de los tipos de caches analizadas. Tres gráficas para comparar el desempeño de caches con la misma asociatividad, variando el tamaño y la cantidad de blqoues y una gráfica para comparar el desempeño de estas mismas caches pero con la misma cantidad de bloques, variando el tamaño y la asociatividad.

En el caso de la tabla 2.2.2 se nota para la cache directamente mapeada una mejora significativa con el aumento de tamaño de la cache de unos cuantos kB a una cache del orden de los MB. Se nota además una gran mejora con el aumento radical del tamaño del bloque(de 4B a 16kB), sin embargo un decremento en el desempeño con el incremento leve, como pasar de bloques de 4B a 64B.

Para los gráficos 2.2.2 y 2.2.2 no cabe duda que conforme aumenta el tamaño de la cache, disminuye el miss rate y aumenta el desempeño, dándose un aumento

muy significativo entre los tamaños de 1kB a 64kB, de 64kB a 1MB pero un aumento leve o incluso un decremento leve de 1MB a 2MB. Con el desempeño hay que tomar en cuenta asímismo el precio de las memorias, ya que las cache por su velocidad son sumamente caras, por ello se plantea el costo-beneficio de duplicar la memoria, para el caso de la ultima comparación, a cambio de una mejora muy muy leve en el miss rate.

Por último, la gráfica 2.2.2 muestra tambien claramente una mejora en el miss rate proporcional a un aumento en la asociatividad y el tamaño de la cache, siendo mucho mejor el miss rate de las cache 4 Way associative que el de las directamente mapeadas. Cabe tambien notar que el porcentaje de aumento es mucho mayor con el paso de directamente mapeada a 2 way associative que con el paso de 2 way associative a 4 way associative. Lo que lleva a pensar que el implementar un poco mas de lógica en hardware para lograr una asociatividad, impacta de gran forma sobre el desempeño de la cache, pero que su mejora deja de ser significativa despues de cierto nivel de asociatividad.

3 Conclusión

Aparte de amalgamar el tema de jerarquía de memorias, visto durante los cursos *Estructuras de computadoras I y II* en lo relativo a memorias caches, la tarea programada brinda un acercamiento mas real a la forma de funcionamiento de estas memorias y a una parte del proceso de diseño de procesadores que pocas veces se practica en los demás cursos ya que se tiene libre albedrío a la hora de elegir el tamaño de las memorias y los bloques a probar, así como el lenguaje de programación a utilizar.

Se logró mostrar las diferencias en desempeño basándose en el miss rate de una memoria, depediendo del tamaño de bloque, tamaño de cache y los diferentes tipos de asociatividad utilizada.

Finalmente se aproxima a grandes rasgos a uno de los campos más relevantes en el mundo de arquitectura de computadoras, el diseño de la memoria cache, ya que esta sección es la que permite transacciones más rápidas de información entre el procesador y memorias más lentas como la RAM y el Disco Duro, lo cuál tiene un impacto sumamente importante en la eficiencia del procesador como tal.

4 Bibliografía

Material del Curso Estructuras de Computadoras Digitales II, José Daniel Hernández, 2016