INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

PROGRAMA DE GRADUADOS DE LA DIVISION DE ELECTRONICA, COMPUTACION, INFORMACION Y COMUNICACIONES

Unidad Generadora de Direcciones Determinísticas

T E S I S

PRESENTADA COMO REQUISITO PARCIAL PARA OBTENER EL GRADO ACADEMICO DE:

MAESTRO EN CIENCIAS

En Ingeniería Electrónica

Especialidad en Sistemas Electrónicos

Ing. César Javier Rostro Martínez

MONTERREY, NUEVO LEON MAYO DEL 2002

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

División de Electrónica, Computación, Información y Comunicaciones

Programa de Graduados

Unidad Generadora de Direcciones Deterministicas

TESIS

Presentada como requisito parcial para obtener el grado académico de

Maestro en Ciencias En Ingeniería Electrónica

Especialidad en Sistemas Electrónicos

Ing. César Javier Rostro Martínez

Monterrey, N.L. Mayo de 2002

A mis Padres, por estar cerca de mi cuando mas lo necesite

Reconocimientos

A **Dios** por darme la oportunidad de vivir, de hacer posibles las "coincidenciasz darme la oportunidad de poder tener un mejor futuro.

A mi **Padre**, **Madre** y **Hermana**, por apoyarme en una empresa mas que me propuse y no sabia si lograría terminar.

A mi asesor **Dr. Alfonso Ávila Ortega**, por su apoyo en la realización de este trabajo de investigación.

Al **Dr. Ignacio Celis** por su invaluable apoyo, ayuda y buen humor, así como su paciencia para responder mis constantes preguntas.

Al **Dr. David Garza** por su ayuda, visión y conocimientos que dieron a este trabajo de investigación la seriedad y profundidad que necesitaba.

A la **Dra. Patricia Hinojosa** por su inesperada ayuda que hicieron posible la conclusión de la investigación.

Al **Dr Roque Eduardo García Baltierra** por ayudarme a tener una visión mas amplia de lo que es la vida.

A **Ing. Pablo Hennings** por su ayuda desinteresada sin la cual no se hubiera podido realizar esta tesis.

Al **Ing. Luis Toledo, Ing. Manuel Melendez, Ing. Horacio Francisco**, la raza maligna, por todos estos años de invaluable amistad, ayuda y apoyo.

A **M.C. Jose Gomez**, por todo el trabajo y buenos momentos que compartimos juntos a través de esta travesía que fue el estudio de la maestría.

CÉSAR JAVIER ROSTRO MARTÍNEZ

Instituto Tecnológico y de Estudios Superiores de Monterrey
Mayo 2002

Unidad Generadora de Direcciones Deterministicas

César Javier Rostro Martínez, M.C.
Instituto Tecnológico y de Estudios Superiores de Monterrey, 2002

Asesor de la tesis: PhD. Alfonso Avila Ortega.

Actualmente los procesadores están alcanzando velocidades cada vez mas rápidas mientras que las memorias no mejoran más que en un valor meramente significativo. El procesador cada vez mas veloz es capaz de realizar operaciones en unos cuantos picosegundos, en el caso del Pentium IV, pero las memorias DRAM mas veloces tienen velocidades de entre 80 y 400 nanosegundos, como se puede apreciar la diferencia es enorme, formándose un cuello de botella, una técnica que podría ayudar a mejor el desempeño del sistema sería el **prefetching** que consiste en traer los datos al *cache* antes de que el procesador los necesite y de esta forma evitar los *miss cache*.

En está tesis se probó un método basado en la **RPT**, **R**eference **P**rediction **T**able, mejorando significativamente el desempeño del sistema. Se probó con las Livermore loops que son 24 Kernels de cálculo científico y son un estándar entre la comunidad científica para probar las nuevas modificaciones que se han hecho a la arquitectura de algún sistema computacional. Se mostrarán el resultados de las modificaciones hechas al *cache* y se podrá apreciar la mejora en el desempeño del *cache*.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

Nuestra vida en las últimas décadas ha cambiado ha una velocidad sorprendente, al igual que la tecnología que nos rodea y esto es en parte gracias a los sistemas computacionales tan veloces que existen actualmente que permiten que nuevos artículos de uso diario se diseñen y se produzcan rápidamente ya que su fabricación es hecha ahora por medio de la computadora, que a principios de los 80's hacían unos cientos de operacionales por segundo y las actuales pueden realizar miles de millones en un segundo. Esto hace que los artículos en muchas ocasiones evolucionen tan rápido que no tenemos tiempo de disfrutarlos mas que unos cuantos meses cuando ya hay algo mas novedoso, mas rápido, mas eficiente, mas compacto.

Sin embargo, no importa que tan rápido sea el procesador de un sistema, los datos requieren de un espacio para almacenarse y desafortunadamente la tecnología de diseño de las memorias no ha tenido el mismo auge en el incremento de su velocidad. En la actualidad la velocidad de los microprocesadores supera por mucho la velocidad de respuesta de las memorias, la velocidad de los microprocesadores está entre los 1.6 y 2.2 GHz., como es el caso del Pentium 4, mientras que la velocidad de las memorias oscila entre los 80 y 400 ns[2]. La separación entre la velocidad de los procesadores y de las DRAM esta creciendo a 50 % por año y el tamaño y organización de la memoria en un chip sencillo de DRAM esta haciéndose complicado, sin embargo el tamaño crece a 60 % por año[5]. La tecnología de estas industrias ha mejorado de una forma desigual. La capacidad de las DRAM se ha cuadruplicado en promedio cada tres años desde 1976, mientras que la velocidad de los microprocesadores ha hecho lo mismo desde 1986[5].

Sin embargo, la división en los dos campos tiene sus desventajas también. La Figura Figura 1.1 muestra que mientras el desempeño de los microprocesadores ha ido mejorando a una velocidad de 60 %[5] por año, el tiempo de acceso a la DRAM ha mejorado menos de 10 %[5] por año. De aquí que los diseñadores de sistemas de computo estén enfrentado una separación en el desempeño procesador-memoria, el cual es el principal obstáculo para mejorar el rendimiento de los sistemas computacionales[5].

La jerarquía en la memoria, fue uno de los primero pasos para tratar de acortar la brecha existente entre procesador y memoria. La memoria *cache* vino a mejorar el desempeño del sistema, pero debido a que no son de gran tamaño (*8 KBytes Level One, 256 KBytes Level Two Cache*, para el Pentium IV[8]), comparada con la memoria principal, no pueden tener

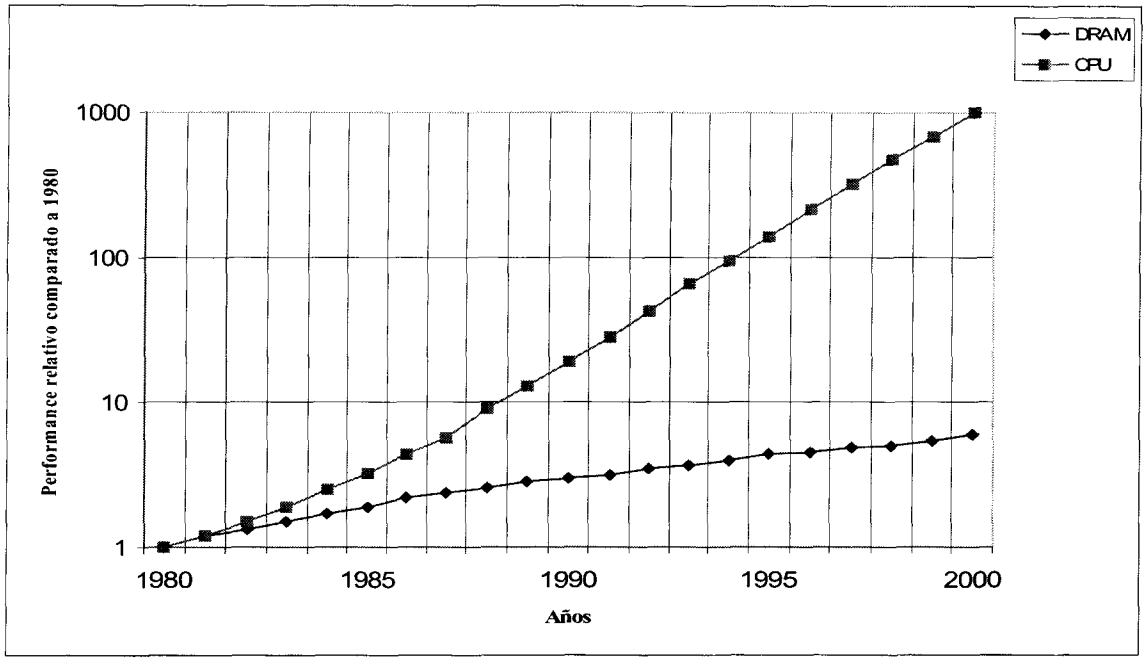


Figura 1.1: Separación de desempeño entre el procesador y la memoria

almacenados todos los datos que en un momento dado necesita el programa, en ocasiones ocurren los *miss cache*, esto es, cuando los datos no están almacenados en *cache*, por lo que se tienen que traer los datos de memoria principal, esto provoca que se pierda tiempo, y que se degrade el desempeño del sistema. Estudios muestran que hasta en un 75 %[5] del tiempo total de ejecución de un programa (de tipo de matrices y de bases de datos), el procesador consume ciclos en espera de datos de memoria [5]. El problema se concentra en que mientras los datos no estén presentes en el *cache*, el procesador estará esperando a que la memoria principal se los proporcione.

Los arquitectos de sistemas han intentado reducir esta separación introduciendo un mayor número de niveles de *cache* en la jerarquía de memoria. Desafortunadamente, estos incrementos de niveles de *cache* hacen que el retraso de la memoria se incremente todavía mas en el peor de los casos.

La memoria *cache* esta fabricada en base de memorias SRAM, este tipo de memorias son las mas rápidas que existen actualmente, pero presentan el inconveniente de que son sumamente caras, si quisiéramos manufacturar una computadora con únicamente memorias SRAM, el precio de fabricarla seria muy alto, lo que la haría incosteable, por lo que se utilizan memorias DRAM, EDORAM, SDRAM, para incrementar la memoria de las computadoras, pero presentan el inconveniente de que tienen tiempos de acceso lentos limitando el rendimiento de los sistemas computacionales

Un segundo intento por acortar la brecha entre memoria y procesador son las técnicas de *prefetching*, que consiste en precargar datos en el *cache* antes de que el procesador los necesite, con esta estrategia lograríamos que cuando el procesador necesite un dato y no lo tenga en sus registros, al momento de solicitarlo al *cache*, éste lo tenga y no detenga la ejecución del programa, mediante la inserción de instrucciones *stall*, para traerlo de memoria principal o de HD o dispositivos mas lentos.

1.1. Justificación

El cálculo de las direcciones de acceso a memoria de datos es muy importante ya que las direcciones son necesarias para realizar prefetching de un dato. Calcular la dirección del dato que el procesador necesitara para ejecutar alguna operación ya sea matemática o de otra indole y traer al *cache* el dato si es que no lo tiene, es el proceso de realizar prefetching; Actualmente el prefetching se ejecuta por medio de software y hardware. Existen esquemas de prefetching simple, por ejemplo: El segmento de código en la Figura 2.a es un ejemplo de prefetching basado en lazos.

```
sin precarga de datos:

for (i = 0; i < N; i++)                    Figura 2.a
     sum = a[i] + sum;

con precarga de datos:

for (i = 0; i < N; i+=4){                  Figura 2.b
           fetch( &a[i+4]);
           sum = a[i] + sum;
           sum = a[i+1] + sum;
           sum = a[i+2] + sum;
           sum = a[i+3] + sum;
   }
```

Este lazo suma los elementos del arreglo "a". Si asumimos un block de 4 palabras en el *cache* este segmento de código causara un *miss cache* cada cuatro iteraciones. Se puede tratar de evitar estos *misses* en el *cache*, modificando el programa como se muestra en la Figura 2.b. La precarga de los elementos del arreglo a ser utilizados en la proxima iteración del lazo es programada justo antes del cálculo para la actual iteración empiece.

Como se aprecia en el ejemplo anterior el código crece considerablemente, lo que provoca que el procesador tenga que ejecutar esas instrucciones de precarga desviando la atención

del procesador de el programa principal, gastando tiempo en estas instrucciones, como consecuencia el programa se llevará mas tiempo de ejecutar. El cálculo eficiente y oportuno de direcciones por medio de hardware, es el enfoque de esta tesis. La Unidad Generadora de Direcciones Deterministicas esta basada en la Tabla de Predicción de Referencias (**RPT**), por sus siglas en inglés. Con los algoritmos para cálculos de direcciones desarrollados, es posible desarrollar una unidad como tal, con esta unidad lograríamos reducir el tamaño del código de los programas significativamente además de que ejecutaría de una forma más rápida ya que el cálculo de direcciones se realizaría por hardware lo liberaría al procesador procesador de la ejecución de otras instrucciones del programa.

1.2. Objetivo

El objetivo principal de esta tesis es: Diseñar y evaluar una unidad para calcular direcciones de accesos a datos deterministicos, es decir aquellos datos que ya se sabe con anterioridad que se van a necesitar sin tener algún tipo de incertidumbre, un ejemplo para direcciones deterministicas son los accesos generados por lazos sencillos y anidados. Con esta unidad lo que se pretende es almacenar en el *cache* los datos que se van a necesitar cuando el programa en ejecución los requiera, de esta forma se evitaría que el procesador detuviera la ejecución del programa y no se tenga que ir a traerlos a memoria principal o disco duro, consumiendo ciclos en espera por los datos y degradando la eficiencia del sistema.

En si lo que se busca es reducir la cantidad de tiempo que el procesador consume esperando datos cuando ocurren los fallos de memoria (*miss penalty*) con una unidad de cálculo de direcciones de accesos deterministicos.

1.3. Contribución y alcance de la tesis

Simulación y medición base (baseline) de rendimiento para accesos determinísticos útiles en la evaluación futura de accesos a memoria no determiniticos es lo que desarrolla esta tesis. comprobar que es eficiente realizar la precarga de datos, *data prefetching* (de aquí en adelante se hará mención de la palabra prefetching para mantener coherencia en la tesis) por medio de hardware, como ya se ha tratado de demostrar en otros trabajos de investigación como son, por ejemplo: Steven P. Vander Weil[7], Dr Chen.S[1]. Se utilizaran las herramientas que existen actualmente de simulación para procesadores y *cache* y además *benchmarks* de uso frecuente en investigación para verificar el desempeño de los sistemas computacionales. Con esto se pretende darle una validez a este trabajo, la aportación mas importante que se deja es probar que es mas eficiente realizar prefetching por medio de hardware. Esta tesis a diferencia del trabajo realizado por el Dr Chen, se prueba con una nueva carga de trabajo y se tratara de comprobar que la precarga de datos por medio de hardware es eficiente. Esta tesis es parte fundamental de varias actividades que forman parte de una investigación mas amplia. A continuación se enumeran los pasos del trabajo completo con el propósito de

mostrar qué lugar ocupa la contribución de esta investigación en el esquema global.

1. **Desarrollar la arquitectura de una unidad de generación de direcciones de accesos deterministicos.** Que corresponde a esta tesis
2. Medir la capacidad de transferencia de datos de una memoria muti-banco en un subsistema de memoria que incluye un acelerador de accesos deterministicos de tipo arreglo.
3. Determinar la frecuencia de ocurrencia de las diferentes categorías de acceso a memoria.
4. Determinar el tamaño óptimo de la RPT (por sus siglas en ingles: *Reference Prediction Table*) para programas de aplicación que accesan memoria mediante apuntadores.
5. Estudiar la frecuencia de aplicación y eficiencia del uso de direccionamiento basado en corrimiento de bits para predicción de accesos a la memoria de datos.
6. Determinar el desempeño de un subsistema de memoria con un acelerador para accesos deterministicos, una unidad generadora de direcciones, un history buffer y el hardware para direccionamiento por corrimiento de bit.

1.4. Organización de la tesis

La tesis se compone de 6 capítulos. El capítulo 1, que es éste precisamente, contiene la introducción, la justificación y el objetivo. El capitulo 2 presenta antecedentes concentrandose en los problemas que enfrentan los actuales sistemas microprocesador-memoria y en describir algunos métodos y técnicas que ya se han desarrollado. El capitulo 3 presenta la unidad generadora de direcciones. El capitulo 4 contiene una descripción del software que se simuló, se describe como esta formado y se explica el concepto bajo el cual funciona. El capitulo 5 contiene los resultados de esta investigación, los datos arrojados de las simulación implementada con la Unidad Generadora de Direcciones y por último el capitulo 6 tiene las conclusiones a las que se llegaron después de haber analizado los resultados obtenidos durante este trabajo de investigación.

Capítulo 2

Precarga de datos y generación de direcciones

Antecedentes

En este capitulo se definen algunos de conceptos fundamentales que son necesarios para comprender mejor el contexto y propósito de esta tesis: **Mejorar el desempeño del subsistema procesador-memoria**. Se presentan algunas técnicas y estrategias que se han implementado en los sistemas computacionales, que representan un intento por lograr acortar la brecha existente entre el procesador y el sistema de memoria y mejorar el rendimiento.

Una de estas técnicas es el uso de jerarquías de memoria cache, que ha logrado mejorar el desempeño del sistema. Las memorias más caras SRAM usadas en los caches han conseguido reducir la relación velocidad de respuesta de la memoria-procesador, sin embargo es muy común en los programas de cálculo científico desperdiciar mas de la mitad del tiempo en stalls debido a peticiones a memoria[7]. La pobre utilización del *cache* en aplicaciones científicas es parcialmente un resultado de la política de fetch de la memoria en la mayoría de los *caches*. Bajo esta política, el controlador de *cache* realiza *fetch* de datos a la memoria principal solo después de que el procesador realizo un acceso al *cache* de datos y no lo encontró. El procesador detiene la ejecución debido a que no es posible continuar hasta que los datos sean proporcionados por la memoria principal [7]. Esta política siempre resultara en un *cache miss (un cold start o compulsory miss)* la primera vez que una aplicación trate de accesar un bloque de datos en particular por que solo encontrara datos accesados previamente en el *cache*

También, si el dato al que se hace referencia es parte de una operación de un arreglo grande, el *cache* desechará los datos recientemente menos usados para hacer lugar para el nuevo arreglo de datos que esta siendo trasladado al *cache*. Si los datos que fueron desplazados son requeridos nuevamente, el procesador tendrá que traerlos de nuevo al *cache*. Esta situación es conocida como *capacity miss*[7].

2.1. Retraso y ancho de banda

El desempeño del sistema de memoria se mide con respecto a dos parámetros principales: **Retraso** y **Ancho de banda**. El **Retraso** de la memoria es el tiempo que se utiliza para leer una *palabra*, entendiéndose como palabra un *byte*. Este parámetro incluye retrasos ocasionados por la decodificación de la dirección, los *latches*, verificación de paridad, detección y corrección

de errores, tiempo de acceso a memoria y tiempo de arbitraje, entre otros. El **ancho de banda** es la máxima cantidad de información que se puede transferir hacia o desde la memoria por segundo y puede expresarse en bits por segundo ó Bytes por segundo, comúnmente se conoce como Megahertz del bus principal [7].

Se han presentado algunas mejoras al sistema de memoria basándose en los dos parámetros antes mencionados. En la siguiente sección hablaremos de algunas mejoras que se han propuesto, empezando por las de hardware.

2.2. Jerarquía de memoria

El uso de jerarquías de memoria es el método que se ha utilizado tradicionalmente para reducir el tiempo de acceso de la memoria y consiste de varios niveles, cada uno de mayor capacidad que el anterior pero de menor velocidad de respuesta.

Cada uno de estos niveles ha sido sujeto de estudios y desarrollo de técnicas que permitan el uso más eficiente de la tecnología. A continuación explicamos brevemente cada uno de estos niveles.

2.2.1. Registros

Debido a que los registros trabajan a la misma velocidad que el procesador, sería ideal poder guardar todos los datos del programa en ellos y no tener que hacer accesos fuera del procesador. El uso extensivo e intensivo de los registros puede reducir significativamente la frecuencia de acceso a memoria, pero las limitaciones físicas y económicas permiten tener sólo unos cuantos, pueden variar desde 16 hasta 128, y es responsabilidad del programador y del compilador hacer el mejor uso de ellos.

2.2.2. Caches

Un *cache* es un lugar seguro para guardar cosas. En el contexto de los sistemas de memoria este término se utiliza para designar al nivel entre memoria principal y el procesador. El *cache* ha sido motivo de discusión en innumerables artículos y se han propuesto muchos cambios a su arquitectura y de hecho, ésta ha cambiado significativamente a medida que se ha incrementado la frecuencia del reloj de los procesadores. En la actualidad, el nivel *cache* puede dividirse en varios subniveles y la tecnología ha permitido que los niveles inferiores se incluyan en el chip del procesador.

El uso que se hace del *cache* tiene un impacto importante en el desempeño de la jerarquía de memoria ya que éste se degrada significativamente en el caso de una **falla de** *cache*, *miss cache*, en otras palabras, el dato que el procesador solicita no se encuentra en el *cache*. Una de las técnicas para reducir la ocurrencia de estos fallos es el *data prefetching* o precarga de datos, que consiste en cargar un bloque de datos en el *cache* antes de que sea solicitado, basándose en predicciones y análisis del comportamiento del programa.

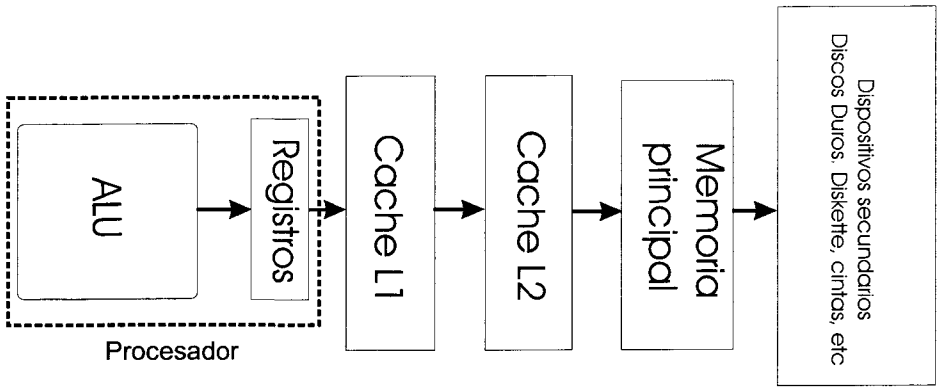


Figura 2.1: Niveles de memoria en un sistema de computo

2.2.3. Comportamiento del *cache*

Medir el desempeño de un *cache* es un parámetro muy importante para saber que tan buenos resultados aporta en una jerarquía de nivel. Uno de los parámetros a medir es el tiempo de acceso de la memoria, que se define:

Tiempo promedio de acceso a memoria = Hit time + Miss rate*Miss penalty

donde **Hit time** es el tiempo de respuesta de un hit en el *cache*, **Miss rate** es la fracción de los accesos al *cache* que resultan en miss, por ejemplo, el número de accesos que resultan en un miss divido por el número de accesos y **miss penalty** es el número de ciclos *stall* que depende de el número de misses y el costo por miss[2]. Teniendo en consideración la ecuación anterior tenemos la siguiente con la cual medimos el tiempo del CPU:

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}}\right) \times \text{Clock cycle time}$$

donde IC es la cuenta de instrucciones (*instruction count*), $CPI_{execution}$ es el tiempo que le lleva a una instrucción ejecutarse, es decir el número promedio de ciclos de reloj por instrucción. Esta ecuación nos da el desempeño del sistema tomando en cuenta las variables mas importantes

2.2.4. Memoria principal

Este nivel utiliza tecnología DRAM (*Dynamic Random Access Memory*), EDORAM (*Extended Data Dynamic Random Access Memory*) o SDRAM (*Synchronous Dynamic Random Access Memory*) cuya principal característica es la alta densidad y la alta capacidad de almacenamiento. El tamaño de las DRAM, EDORAM o SDRAM se incrementa en múltiplos de 4, aproximadamente cada 3 años[5].

El desarrollo de la capacidad de las memorias ha sido el foco de preocupación en la industria por su pobre incremento en la velocidad de respuesta en los últimos años, a diferencia de la industria de los procesadores. El lento desarrollo se atribuye también al enfoque en la producción, mientras la producción de procesadores se concentra en el alto rendimiento, la producción de memorias se concentra en la capacidad de almacenar datos. Las innovaciones hechas a los chips de DRAM y SDRAM se limitan a aquellas que pueden ser implementadas con poca inversión económica y en poco tiempo.

A continuación se presentan otras técnicas implementadas mediante hardware para acortar la brecha ya existente.

2.3. Modificaciones a los niveles de memoria

1. *Caches más grandes*: la ventaja de tener un *cache* más grande es que existe la posibilidad de que más datos puedan caber en el *cache*, ofreciendo así un mejor desempeño. Pero este desempeño puede ser engañoso ya que para conjuntos más grandes de datos, puede que no haya ninguna mejora, además de que el seguir haciendo los *caches* cada vez mas grandes presenta el inconveniente de encarecer el sistema debido al costo de las memorias *cache*, además de que *caches* mas grandes tienen tiempo de respuesta más largo. Cuando ocurre un it cache miss, este sera mas costoso en tiempo mientras mas niveles de *cache* existan porque los niveles son mas lentos conforme crecen en tamaño.

2. *Sistemas de memoria de bus más ancho*: el ancho de bus de la memoria se refiere al tamaño máximo de tamaño de bloque que puede transferirse en un acceso a memoria principal; al hacer mas ancho el bus, el bus puede incrementar el ancho de banda.

3. *Uso selectivo del cache*: si bien es cierto que la implementación del *cache* ha representado una de las soluciones principales para el problema del retraso, existen ciertos programas cuyo comportamiento no presenta características de **localidad espacial** o **localidad temporal**, donde: localidad *espacial* expresa que datos cuyas direcciones están cerca una de la otra tenderán a ser solicitadas conjuntamente, localidad *temporal* expresa que datos accesados recientemente tiene probabilidad de ser accesados nuevamente en un futuro cercano; en estos casos el desempeño se mejora si los datos se transfieren directamente de la memoria principal al procesador. De esta forma, si se puede reconocer que el acceso a memoria no tiene localidad, el tiempo del mismo puede reducirse y evitar así la eliminación de una línea de *cache* para cargarlo, misma que puede contener el siguiente acceso del procesador.

2.4. Técnicas de precarga de datos

La precarga de datos e instrucciones es una de las técnicas que se utiliza para reducir el retraso de la memoria y consiste en tratar de predecir que datos necesitará el procesador

y buscarlos antes de que sean pedidos. De esta manera, cuando el procesador los solicita ya están cargados en el *cache*. Existen dos tendencias de la técnica de precarga: la precarga basada en software o en hardware.

2.4.1. Prefetching por medio de software

Una instrucción prefetch especifica la dirección de un nuevo bloque a ser traído al *cache*. Al ejecutar la instrucción de prefetch el procesador carga anticipadamente en el *cache* el bloque que contiene la palabra requerida. Como el procesador no necesita los datos todavía, puede continuar ejecutando el programa mientras el sistema de memoria trae los datos requeridos hacia el *cache*.

La mayoría de los ajustes recaen en la juiciosa colocación de las instrucciones de *fetch* dentro de la aplicación. Elegir donde colocar una instrucción de prefetch relativa a la correspondiente instrucción load o store es conocida como *prefetch scheduling*.

En la practica, no es posible predecir exactamente cuando programar un prefetch de tal forma que los datos lleguen exactamente cuando sean requeridos por el procesador. Si el compilador programa el prefetch muy tarde, los datos no estarán en el *cache* en el momento que el procesador los necesite. Si el compilador programa el prefetch mucho antes, la instrucción de precarga podría desplazar datos del cache antes de que sean usados por el procesador para hacer espacio a los datos que el controlador traerá después. Esta situación, en la cual hay un *miss cache* que no debería haber ocurrido sin prefetching es llamada contaminación del cache (*cache pollution*).

Las instrucciones de prefetch pueden ser agregadas a mano por el programador o por el compilador durante los pasos de optimización. En cualquier caso, el prefetching es usado mas frecuentemente en lazos responsables de grandes cálculos de arreglos, tales como lazos anidados, las cuales son comunes en códigos de programas científicos, proveen excelentes oportunidades de prefetching debido a que muestran una pobre utilización del cache y frecuentemente tienen patrones predecibles de referencias a memoria.

Simple Prefetching

El código segmentado en la Figura Figura~2.2(a) es un ejemplo de un lazo sencillo. Este lazo suma elementos de un arreglo. Si asumimos un *cache* bloque de 4 palabras, este código segmentado originara un *cache miss* cada 4 iteraciones [7]. Podemos tratar de evitar estos cache misses usando los comandos de prefetch añadidos en la Figura Figura~2.2(b) El prefetch del elemento a ser usado en la siguiente iteración esta programado precisamente antes de que el cálculo de la actual iteración empiece[5].

Sin embargo, realizar *prefetching* en todas las iteraciones de este lazo es innecesario porque cada instrucción de carga de datos de hecho trae bloques de cuatro elementos del arreglo hacia el *cache*. De tal forma que *prefetching* debería ser hecho cada cuarta iteración. Una solución es evitar los comandos *fetch* con un if condicional que verifique cuando (i modulo

4) = 0 es verdadero[5].

Desenrollando el Lazo

Desenrollando el lazo por un factor de r (donde r es igual al número de palabras a ser precargadas por el *cache* block) es más efectivo que usar un explícito predicado prefetch. Como muestra la Figura Figura˜2.2(c), desarrollar el lazo implica reproducir el cuerpo del lazo r veces e incrementar la distancia de 1 a r. En el proceso, el compilador no repite los comandos de prefetch, pero hace cambios en el índice del arreglo, lo cual se usa para calcular la dirección de prefetch, de i + 1 a i + r.

Sin embargo, *cache misses* se producirán durante la primera iteración del lazo debido a que el prefetch nunca ha sido validado para esta iteración. Además, prefetches innecesarios se realizaran en la última iteración del lazo desarrollado, en el cual los comandos de prefetch tratan de accesar arreglos de datos limitados anteriores[7].

Software pipeling

Las técnicas de *software pipelining*, mostradas en la Figura Figura˜2.2(d), pueden resolver estos problemas. En esta figura, se han extraído algunos de los segmentos del código del cuerpo del lazo y colocarlos en cualquier parte del lazo original. Se ha agregado un lazo prologo consistente en instrucciones de fetch al inicio del lazo principal para precargar datos para la primera iteración. Añadimos un epilogó al final del lazo principal para ejecutar los cálculos finales sin inicializar instrucciones de prefetch innecesarias.

El código de la Figura Figura˜2.2(d) cubre todas las referencias del lazo por que un prefetch precede cada referencia. Sin embargo una optimización final podría ser necesaria para asegurarse que el compilador programa los prefetches con el tiempo suficiente. Como se muestra en los ejemplos de las Figura Figura˜2.2(a) hasta la Figura Figura˜2.2(d) asumimos que programando el prefetch de datos una iteración antes nos dará suficiente tiempo para traer el dato solicitado de la memoria principal al *cache*, sin embargo, esto podría no ser el caso de las lazos pequeños o cortos.

$$\delta = \lceil \frac{l}{s} \rceil$$

Para tales lazos, podría ser necesario inicializar los prefetches δ iteraciones antes de que el dato sea solicitado, donde δ, la *prefetch distance*, es **Aquí**; l es el *cache miss latency* promedio, calculado en ciclos de procesador y s es el número estimado de ciclos en la ejecución mas corta de un lazo, incluyendo prefetch adelantado. (un compilador puede automáticamente determinar cual es el "camino" mas corto de un lazo sin muchos problemas). Usando el operador "techo", el cual redondea el resultado al número entero mas grande superior y calculando la *prefetch distance* usando la ejecución mas corta significa que los datos serán precargados con suficiente anticipación como es necesario, lo cual aumentara la probabilidad de que los datos precargados estén en el *cache* antes de que el procesador los necesite.

```
for (i = 0; i  < N; i++)
   sum = a[i] +sum;
```

(a)

```
for (i = 0; i < N; i++) {
    fetch( &a[i+1]);          Precarga para el arreglo a
    sum = a[i] + sum;
}
```

(b)

```
for (i = 0; i < N; i+=4) {
    fetch( &a[i+4]);
    sum = a[i] + sum;         Desenrollando la loop 4 veces
    sum = a[i+1] + sum;
    sum = a[i+2] + sum;
    sum = a[i+3] + sum;
}
```

(c)

```
fetch( &sum);
fetch( &a[0]);

for (i = 0; i < N-4; i+=4) {
    fetch( &a[i+4]);
    sum = a[i] + sum;         El epilogo elimina las precargas
    sum = a[i+1] + sum;       innecesarias en la última iteración
    sum = a[i+2] + sum;
    sum = a[i+3] + sum;
}
for ( ; i < N; i++)
      sum = a[i] + sum;
```

(d)

```
fetch( &sum);
for ( i = 0; i < 12; i += 4)
      fetch( &a[i]);

for (i = 0; i < N-4; i+=4) {
    fetch( &a[i+12]);
    sum = a[i] + sum;         Incrementa la distancia
    sum = a[i+1] + sum;       de precarga a 3
    sum = a[i+2] + sum;
    sum = a[i+3] + sum;
}
for ( ; i < N; i++)
      sum = a[i] + sum;
```

(e)

Figura 2.2: Prefetching por medio de Software

En la Figura Figura~2.2(d), si consideramos un *miss latency* promedio de 100 ciclos de procesador y un tiempo mínimo de iteración de lazo de 45, δ sera de 3. La Figura Figura~2.2(e) muestra la versión final del segmento de código, alterado para tener una distancia de prefetch de tres. Ahora que ha hecho prefetching tres iteraciones por adelantado, se ha expandido el prólogo para incluir un lazo que haga *prefetch* a varios *cache blocks* para las tres primeras iteraciones del lazo principal. Además, el lazo principal ha sido recortado para detener el prefetch de datos tres iteraciones antes de que termine el cálculo. No hay cambios necesarios para el epilogó, el cual realizar las iteraciones restantes sin hacer prefetching.

Aplicando Software Prefetching

Sofisticados algoritmos-compilador basados en el concepto anterior, software pipelining, han sido desarrollados para que automáticamente agreguen el prefetching durante el paso de optimización del compilador, con grados variables de éxito. Entre los esfuerzos en hardware están los de, *one block-lookahead* (OBL)[1]; *tabla de referencia predictoria* (RPT)[1] junto con el de *Lookahead program counter* (LA-PC)[1]; *Speculative hardware data prefetching* propuesto Eickermeyer and Vassiliadis[3]; Otras técnicas están basadas en cadenas de Markov[3] y en predictores aplicando la correlación[3]. Sin que ninguno haya logrado un éxito total. Debido a que el compilador deber de tener la capacidad para asegurar patrones predecibles de acceso a memoria, el prefetching normalmente se restringe a lazos cuyos patrones tienen comportamientos de arreglos sencillos. Tales lazos son muy comunes en programas científicos pero mucho menos en aplicaciones generales.

Se han hecho intentos para establecer estrategias de prefetching por software para aplicaciones con estructuras irregulares de datos que han tenido éxito limitado. Debido a que tales aplicaciones no siguen un patrón predecible, es difícil anticipar sus accesos a memoria.

Aplicaciones mas generales también muestran mucho mas reuso de datos que las aplicaciones científicas. Este reuso frecuentemente conduce a una alta utilización del *cache*, lo cual disminuye los beneficios del software prefetching. Este reuso se basa en que los accesos a memoria no tienen un comportamiento regular y constantemente se hacen accesos a memoria.

2.4.2. Predecir misses en el cache de datos en aplicaciones no numéricas

Existe otro método de prefetching por medio de software que se utiliza para aplicaciones no científicas. Este método se aplica encontrando el *perfil* de correlación que tiene el software que se este ejecutando. Mientras que las técnicas basadas en software proveen beneficios de "ocultar"el retraso, también incurren en tiempos de ejecución mas largos, por ejemplo una programación agresiva de cargas de bloques puede hacer que se saturen los registros al incrementar el tiempo de "vida"de éstos. El prefetching controlado por software requiere instrucciones adicionales para calcular la dirección de prefetch y generar los prefetches mismos. Los trabajos anteriormente explicados se han conducido hacia resolver el problema en códigos numéricos, el siguiente método se concentra en el caso más difícil pero más importante que

es el de los casos de *misses dinámicos* en casos no numéricos[4].

Predecir misses en el cache de datos en aplicaciones no numéricas

Para superar la inhabilidad del compilador para analizar la localidad de los datos en códigos no numéricos, se hace uso de la información del perfil. Un ejemplo del tipo de información de perfil son los valores exactos de misses de todas las referencias estáticas a memoria, nos referiremos a esta propuesta como perfil resumido, ya que la relación de los misses de cada referencia a memoria es resumida a un simple valor.

Si el perfil resumido indica que todas las instrucciones de referencia a memoria (por ejemplo: aquellas que son ejecutadas con suficiente frecuencia para hacer una contribución significativa al tiempo de ejecución) tienen razones de miss cercanas a 0 % ó 100 %, entonces el aislamiento de misses dinámicos es trivial, simplemente se aplican las técnicas de tolerancia al retraso solo a las referencias estáticas que son las que tiene los misses[4]. En contraste, si la importancia de las referencias tienen razones de miss intermedios (ejemplo 50 %), entonces no tenemos suficiente información para distinguir cuales casos dinámicos, ya que esta información es perdida en el proceso de resumir las razones de miss. La actual propuesta para negociar con razones de miss intermedias es tratar todas las referencias estáticas a memoria con razones de miss encima o debajo de cierto umbral como si siempre fueran misses o hits, respectivamente. Sin embargo, esta estrategia de todo o nada fallara al momento de "ocultar"el retraso de la memoria cuando las referencias son pronosticadas a ser hits pero son en realidad misses y generaran gastos innecesarios cuando las referencias son pronosticadas a ser misses pero realmente son hits. En vez de solucionar por este sup-óptimo desempeño, es preferible predecir hits y misses dinámicos con mayor exactitud.

Perfil de correlación

Por medio de exponer el comportamiento del *cache* directamente al usuario, operaciones de información de memoria, *informing memory operations*, da la capacidad de tener nuevas clases de herramientas de perfil las cuales puede reunir información mas compleja que simplemente la pre-referencia de razones de misses, por ejemplo, los *cache* misses pueden ser correlacionados con información tal como la trayectoria de control de flujo, los resultado de referencias previas del *cache*, etc., para ayudar a predecir el comportamiento dinámico de los *cache* misses. Esta propuesta es conocida como perfil de correlación.

La Figura Figura~2.3 ilustra como la información de perfil de correlación podría ayudarnos. La instrucción load mostrada en la Figura Figura~2.3 tiene una razón total de miss de 50 %. Sin embargo, dependiendo del contexto dinámico de la carga, podría verse que tiene un comportamiento predecible. En este ejemplo, los contextos A y B tienen una alta probabilidad de producir misses, mientras que los contextos C y D no, de tal forma que se aplicarían las técnicas a los contextos A y B y no a C o D.

Los contextos dinámicos mostrados en la Figura Figura~2.3 deberían ser vistos simple-

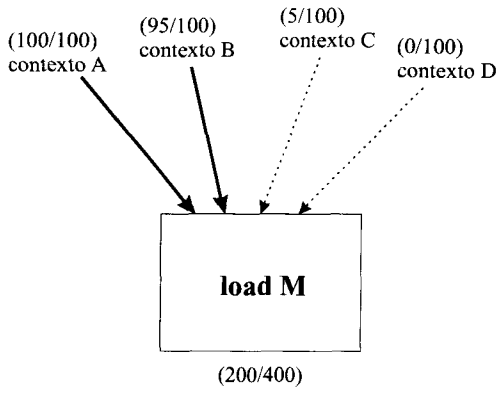


Figura 2.3: Ejemplo de correlación en los cache misses

mente como un conjunto que no traslapa casos dinámicos de la carga que pueden ser agrupados juntos por que comparten patrones de comportamiento común. Se consideran tres tipos de información que puede ser usada para distinguir estos contextos. La primera es información de *control de flujo*, por ejemplo, la secuencia de N bloques básicos numerados precediendo la carga. Los otros dos están basados en secuencias de resultados de los accesos al cache (por ejemplo: hits o misses) para referencias previas a memoria: *auto* correlación, (*self*-correlation), considera los resultados del *cache* de N previos casos dinámicos de las referencias estáticas dadas, y **correlación global**, *global correlation*, se refiere a los previas N referencias dinámicas a través del programa entero.

Correlación de control de flujo

Esta técnica de perfil correlaciona los resultados del *cache* con las recientes trayectorias de control de flujo. Para reunir esta información, la herramienta de perfil contiene los mas recientes N bloques básicos numerados en un buffer FIFO, y relaciona este patrón contra el resultado de hit/miss para una conocida referencia a memoria. Correlación de control de flujo es útil para detectar casos donde hay probabilidad de reuso de datos, *data reuse* o desplazamientos en el *cache*, *displacement cache*, que se consideraría miss pollution.

Si no hay una trayectoria que conduzca al reuso de datos, ya sea temporal o espacial, entonces la proxima referencia es probable que sea un hit en el *cache*. Considere el ejemplo mostrado en la Figura Figura 2.4(a)-(b), donde una gráfica es transversa por el procedimiento recursivo **walk**. Algunos ciclos de trayectoria (ejemplo, A→B→C→D→A o P→Q→R→S→P) resultara en el reuso temporal de p→data. En este ejemplo, correlación de control de flujo potencialmente puede detectar que si las últimas cuatro decisiones transversales conduzcan a un ciclo (ejemplo, *derecha*, *abajo*, *izquierda* y *arriba*), entonces hay una alta probabilidad que la próxima referencia p→data tenga un *cache* hit

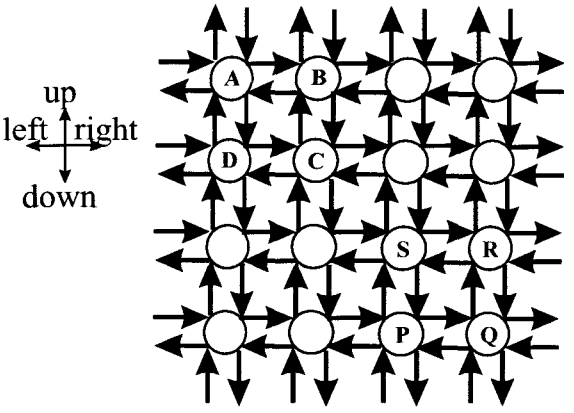
Algunas trayectorias de control de flujo podría incrementar la probabilidad de un *cache*

(a) Código con reuso de datos

```
Struct node {
  int data;
  struct node
      *left, *right,
      *up, *down;
};

void walk(node* p) {
   work(p->data);
   if (go_left(p->data))
     p = p->left;
   elsif (go_right(p->data))
     p = p->right;
   elsif (go_up(p->data))
     p = p->up;
   elsif (go_down(p->data))
     p = p->down;
   elsif p = NULL;
   if  (p = NULL) walk(p);
}
```

(b) Gráfica ejemplo



(c) código con desplazamientos en el cache

```
x = *p;
if (x > 0) {
  for (j = 0;
    j < 100000; j++)
    a[i] = foo(i);
}
y = *p;
```

Figura 2.4: Correlación de control de flujo detecta reuso y desplazamientos en el cache

miss debido al desplazamiento de una línea de datos antes de que sea reusada. Por ejemplo, si la condición "x > 0"es verdadera en la Figura Figura~2.4(c), entonces el subsecuente lazo *for* es probable que desplace *p del *cache* de primer nivel antes de que pueda ser cargado otra vez. Note que mientras las trayectorias que accesan grandes cantidades de datos son problemas obvios, el desplazamiento podría ser también debido a un conflicto de mapeo.

Auto correlación

En auto correlación, perfilar una carga L mediante correlacionando su resultados del *cache* con los previos N resultados del *cache* de L mismo. Esta aproximación es particularmente útil para detectar formas de localidad espacial las cuales no son aparentes al momento de compilar. Por ejemplo, considere el caso en la Figura Figura~2.5 donde un árbol es construido en preorden, asumiendo que llamadas consecutivas a el asignador de memoria regrese localidades continuas de memoria y que el tamaño de la línea es lo suficientemente grande para contener dos *treeNodes*. Dependiendo del orden transversal (y el para el cual el árbol es modificado después de su creación), podría experimentar localidad espacial cuando el árbol es transverso subsecuentemente. Por ejemplo, si el árbol es transverso en preorden, es de esperarse que p→data sea 50 % y el tendrá dificultad en reconocer esto como una localidad espacial, perfil auto correlacionado predecirá con mayo exactitud los resultados dinámicos del *cache* para p→data.

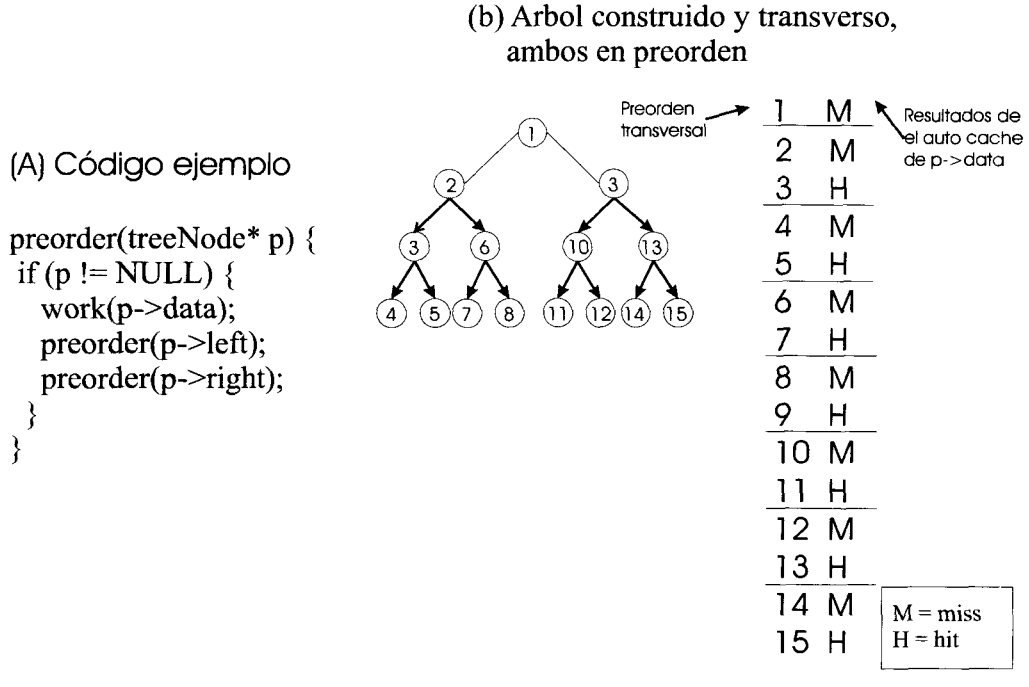


Figura 2.5: Perfil de auto correlación detecta localidad espacial para p→data

Correlación global

En contraste con auto correlación, la idea detrás de correlación global, *global correlation*, es correlacionar los resultados del *cache* de una carga de L, con los previos N resultados del *cache* pese a sus posiciones dentro del programa. Las herramientas de perfil mantienen este patrón usando un sencillo FIFO de N tamaño el cual es actualizado en donde ocurra un acceso dinámico al *cache*. Hay que hacer notar que desde los anteriores casos de L esta misma podría aparecer en este patron de historial global, correlación global podría capturar algo del mismo comportamiento como auto correlación, particularmente en lazos extremadamente pequeño.

Correlación global es particularmente útil para detectar patrones "ráfaga"de misses a través de multiples referencias. Un ejemplo de esta situación es cuando nos movemos a una nueva porción de una estructura de datos que no ha sido accesada en un periodo de tiempo largo (por ello han sido removidos del *cache*), en dado caso el hecho que el primer acceso a un dato sea un miss es una buena señal de que hay referencias asociadas o objetos vecinos que serán misses. La Figura Figura 2.6 ilustra un caso donde una tabla grande sin arreglo (demasiado grande para caber en el *cache*) es organizada como una matriz de listas entrelazadas. En este caso, se podría esperar una fuerte correlación entre si los misses de htab[i] (el apuntador principal de la lista) y si los accesos subsecuentes a curr→data (la lista de elementos) también son misses. De igual forma, si la misma entrada es accesada dos veces dentro de un intervalo corto (por ejemplo htab[10]), el hecho que el apuntador principal realice un hit es una indicador de peso de que la lista de elementos (ejemplo A→data y B→data) serán hits también.

Resumiendo, de los resultados de correlación del *cache* con el contexto en el cual las referencias ocurren, por ejemplo el flujo de control circundante o los resultados del *cache* de referencias previas, pueden potencialmente predecir el comportamiento dinámico del *cache* con mayor exactitud de la que es posible la razón de misses resumidos.

2.5. Prefetching por hardware en DSP's

Los DSP (Digital Signal Processors) implementan un módulo de direccionamiento usando hardware, separado para generación y para comparación. Para simplificar el hardware, los DSP's restringen la dirección de inicio, el valor de desplazamiento, y/o el tamaño del buffer.

El módulo de direccionamiento permite a un continuo set de localidades de memoria a ser accesado por medio de un buffer circular. El módulo de direccionamiento puede ser controlado especificando el rango de direcciones y el valor del desplazamiento. En este caso, el apuntador de dirección es calculado usando la Ecuación Ecuación 2.1, cuando el desplazamiento es positivo y usando la Ecuación Ecuación 2.2, cuando el desplazamiento es negativo.

$$ptr = \begin{cases} (ptr + disp) & \text{if} ptr + disp \leq end \\ (ptr + disp - end - 1 + begin) & \text{if} ptr + disp > end \end{cases} \quad (2.1)$$

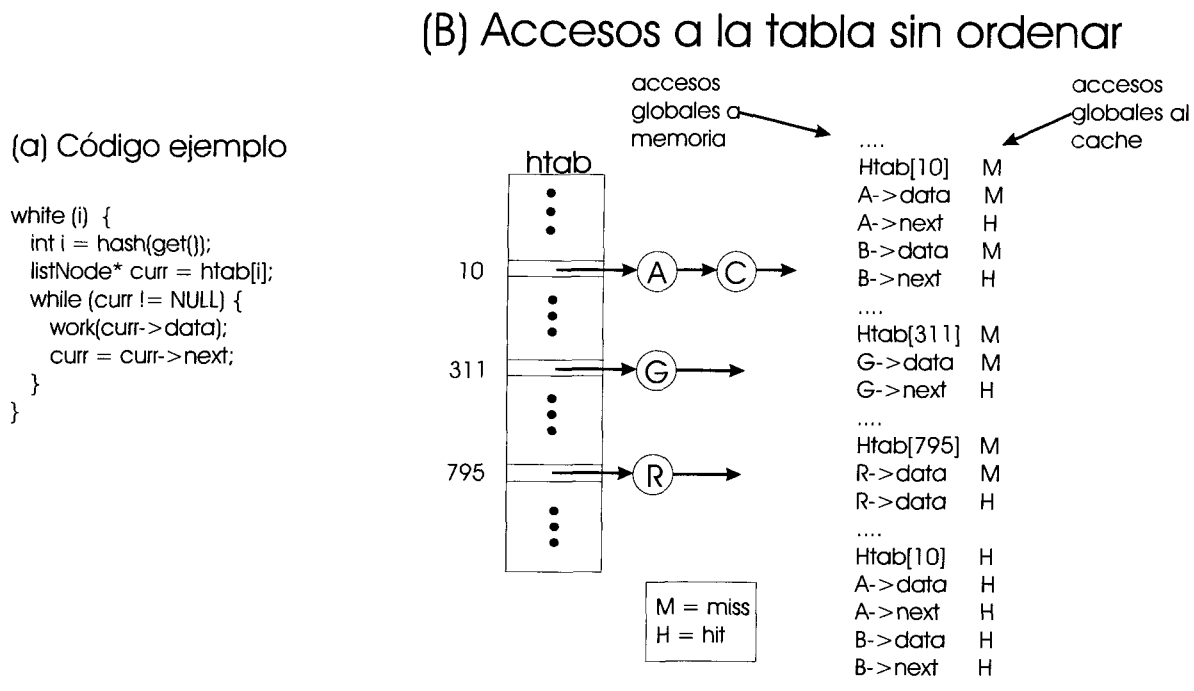


Figura 2.6: Perfil de correlación global detecta ráfagas de cache misses para curr→data

$$ptr = \begin{cases} (ptr - disp - begin + end + 1) & \text{if} ptr + disp < begin \\ (ptr + disp) & \text{if} ptr + disp \geq begin \end{cases} \quad (2.2)$$

En la Figura Figura 2.7 se muestra el típico hardware utilizado en los DSP's para calcular las direcciones; el *ptr* es el apuntador de direcciones, *begin* es la dirección más pequeña del buffer circular, *end* es la dirección más grande del buffer circular y *disp* es el valor del desplazamiento después de cada acceso del buffer circular.

Las típicas implementaciones de módulos de direccionamiento restringen ya sea la localidad asociada a la dirección de inicio o el tamaño del buffer circular o el valor del desplazamiento. Estas restricciones son usualmente apropiadas por que implementar un verdadero modulo de direccionamiento requiere hardware adicional para la generación de direcciones y comparación en el curso critico de las unidades de direccionamiento. Esto reduce la velocidad máxima de operación posible de un DSP.

La Figura Figura 2.7 muestra una arquitectura eficiente que implementa un modulo de direccionamiento de acuerdo a las ecuaciones anteriores.Los valores de *ptr, disp, begin y end* son cargados en registros que toman valores de bus YAB. El signo del valor de *disp* es usado para seleccionar entre los valores de *begin* y *end* a ser restados de la suma de los valores de *ptr* y el de *disp* en el sumador *carry-save*

2.6. Tipo de acceso secuencial

Prefetching por medio de hardware clasifica el tipo de accesos secuenciales, de esta forma se puede predecir la siguiente dirección que se solicitará. Se pueden generar diferentes tipos de direccionamientos a partir de secuencias de direcciones que se denominan *primitivas o base*. Éstas se obtuvieron después de analizar una selección de algoritmos que son sumamente utilizados en la solución de problemas de tiempo real en los DSP.

El tipo de accesos se resume en la Tabla Tabla 2.1

tabla 2.1: tabla de los cálculos mas comunes de direccionamiento primitivo

Cálculo	Secuencia de direccionamiento
Operaciones matriciales (Multiplicación, suma, transpuesta, adjunta, etc)	Secuencial, secuencial con offset
Transformada rápida de Fourier	Mezclada, secuencial, Bit-invertido
Transformada rápida de Fourier 2D	Mezclada, secuencial, Bit-invertido
Convolución	Reflejado, secuencial
Convolución 2D	Reflejado, direccionamiento matricial
Correlación	Secuencial, secuencial con offset
Correlación 2D	Secuencial con offset, direccionamiento matricial

Basado en una secuencia de direccionamiento de longitud N, si A_i es dirección i contenida en el rango, $0 \leq i \leq N - l$ la secuencia de direccionamiento base esta definido como sigue:

1. Direccionamiento secuencial: $A_0, A_1, A_2, A_3, \ldots, A_{N-1}$
2. Direccionamiento secuencial con offset(k): $A_{0+k}, A_{1+k}, A_{2+k}, \ldots, A_{N-1+k}$
3. Direccionamiento mezclado (base r, $N/r = p$): $A_0, A_p, A_{2p}, \ldots, A_1, A_{p+1}$, $A_{2p+1}, \ldots, A_2, A_{p+2}, \ldots, A_{2p+2}, \ldots, A_{N-1}$
4. Direccionamiento Bit-invertido (ej. $N = 8$): $A_0, A_4, A_2, A_6, A_1, A_5, A_3, A_7$
5. Direccionamiento reflejado: $A_0, A_{N-1}, A_1, A_{N-2}. \ldots, A_i, A_{N-i}, \ldots, A_{\frac{N}{2}-1}, A_{\frac{N}{2}}$

Esta información es muy importante puesto de que aquí se puede partir para simplificar mucho la selección de la dirección de datos a precargar, una vez sabiendo que tipo de aplicación se correrá, se puede adaptar el hardware y así cometer *miss cache* lo menos posible, lo que redituara en un mayor performance del sistema.

2.7. Prefetching secuencial en microprocesadores

Prefetching secuencial puede tomar ventaja de la localidad de referencia espacial de un programa precargando bloques de *cache* consecutivos mas pequeños, sin introducir algunos de los problemas asociados con grandes bloques de *cache*. Prefetching secuencial puede ser implementado con relativa simplicidad en hardware. La efectividad de usar bloques de *cache* de tamaño grande para la precarga de datos esta limitado por la contaminación del mismo (*cache pollution.*) Cuando el tamaño del bloque del *cache* se incrementa, así también la cantidad de datos útiles desplazados del *cache* para hacer lugar o espacio para el bloque.

En multiprocesadores compartiendo la memoria con *caches* individuales, incrementando el tamaño de los bloques de *cache*, incrementa la probabilidad de que multiples procesadores necesitaran datos del mismo bloque. Esto incrementaría los efectos de falso compartimiento (*false-sharing effects*), lo cual ocurre cuando multiples procesadores tratan de accesar diferentes palabras del mismo bloque de *cache* y al menos uno de los accesos es un *store*.

El hardware del *cache* opera solo sobre bloques de *cache* completos, así los accesos son tratados como operaciones aplicadas a un simple objeto. El tráfico coherente del *cache* es así generado para asegurarse que los cambios hechos a un bloque por una operación de *store* sea vista por todos los procesadores haciendo accesos al bloque.

2.7.1. Accesos OBL

Los esquemas de precarga secuencial mas simples son variaciones sobre los accesos *one-block-lookahead* (OBL), el cual inicializa automáticamente una precarga para el bloque b + 1, cuando el bloque b es accesado. Este acceso difiere de la simple duplicación del tamaño del bloque del *cache* debido a que los bloques almacenados, demandado y el bloque precargado son considerados artículos separados para reemplazos y coherencia en el *cache*. El uso separado, de bloques mas pequeños le dice a la computadora que no tiene que desplazar grandes cantidades de datos cada vez que los reemplace en el *cache*. Bloques mas pequeños también reducen la oportunidad de compartimientos falsos *"false sharing"*.

La implementación de OBL difiere dependiendo del tipo de acceso al bloque b inicialice la precarga de b + 1. El algoritmo *prefetch-on-miss* inicializa una precarga para el bloque b + 1, en momento en que ocurre un *cache miss* para el bloque b. Si b + 1, está en el *cache*, ningún acceso a memoria se realiza. El algoritmo *tagged prefetch* asocia un bit bandera o etiqueta a todos los bloques en el *cache*. Este bit detecta cuando un bloque es cargado o cuando se hace referencia a un bloque por primera vez. En cualquier caso, el proximo bloque de la memoria es cargado. Precarga etiquetada *Tagged prefetching* o con bandera es mas cara de implementar

debido al bit adicional en el *cache* y a la necesidad de un controlador de *cache* mas complejo. Sin embargo, precarga etiquetada elimina *misses* en un *cache* unificado (datos e instrucciones) mas del doble de veces así como el algoritmo *prefetch-on-miss.*

2.7.2. Precarga secuencial adaptiva

Una política de precarga secuencial adaptiva permite que los valores de K, donde K es conocida como el grado de prefetching, varié durante la ejecución del programa para relacionar los grados de localidad espacial del programa en un tiempo determinado. El *cache* calcula periódicamente una eficiencia de precarga (*prefetch efficiency*) para determinar las características de localidad espacial del programa actual.

La eficiencia en la precarga es una relación del numero de veces que un bloque de *cache* resultó en un hit entre el total del número de precargas. El valor de K es inicializado en uno, incrementa cuando la eficiencia en la precarga excede determinado valor superior de umbral y decrece cuando la eficiencia cae por debajo de cierto valor de umbral. Si el valor de K se reduce a 0, la precarga termina. En este punto, el hardware de precarga empieza a determinar con que frecuencia un *cache miss* ocurría mientras el bloque b-1 es traído al *cache.* El hardware empieza nuevamente a realizar precargas si la frecuencia excede el valor de umbral designado.

2.7.3. Comparando aproximaciones

Simulaciones en una memoria multiprocesador compartida encontraron que el *adaptive prefetching* reduce los *cache misses* más eficientemente que el *tagged prefetching* pero no reducía el tiempo de ejecución significativamente. La relación mas baja de *misses* de la precarga adaptiva secuencial fue parcialmente desplazada por el incremento en el trafico de la memoria y la controversia creada por las precargas adicionales innecesarias. *Tagged prefetching* es mas simple, ofrece un buen desempeño y es una opción atractiva cuando la simplicidad y costo son importantes consideraciones en el diseño.

2.7.4. Precarga con adelantos arbitrarios

Cuando los *strides* de patrones de referencia del procesador atraviesan bloques no consecutivos de memoria, el prefetching secuencial provocara precarga innecesaria y se transformara en ineficiente. Se necesitaran técnicas de prefetching mas elaboradas para tomar ventaja de los patrones de adelanto cortos y largos mientras ignora referencias que no están en el arreglo-base.

Tal técnica emplea un hardware de precarga especial que monitorea los patrones y relación que tienen con la dirección, es decir encuentra si hay alguna relación entre las direcciones que se están generando del procesador y deduce las oportunidades de precarga mediante la comparación de direcciones sucesivas usadas por las instrucciones *load* o *store.* Si el hardware de precarga encuentra que un *load* o *store* en particular esta generando un patron de memoria-dirección predecible, automáticamente generara precargas para esa instrucción.

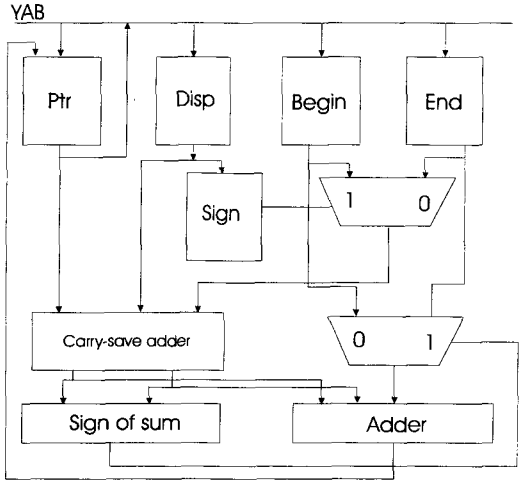


Figura 2.7: Módulo de direccionamiento de acuerdo a las ecuaciones anteriores

Capítulo 3

Unidad generadora de direcciones

3.1. Hardware creado e implementado

En la actualidad la técnica de prefetching mas aceptada es por medio de software. Software prefetching consiste en la inserción de instrucciones dentro del programa principal, como se conoce con anticipación la forma en la que se ejecutara el programa el grado de incertidumbre al momento de realizar la precarga de datos es mínimo, esto implica el desarrollo de compiladores sofisticados capaces de reconocer una secuencia en el direccionamiento y de insertar instrucciones de precarga en el lugar adecuado del programa principal para evitar el *miss pollution* sin embargo, software prefetching presentan dos grandes inconvenientes:

1. El procesador ejecutara instrucciones de precarga que consumen ciclos de reloj y que podrían ser utilizados por otras instrucciones.

2. El tamaño del código crece al insertar instrucciones de precarga. El programa ocupa mayor espacio en los dispositivos de almacenamiento.

La técnica de hardware prefetching no hacen uso de instrucciones de software prefetching, que son las instrucciones que se insertan en el programa para traer los datos con anticipación al *cache*. Precarga de datos por hardware prefetching no requiere cambio en los programas, así que no hay necesidad de intervención del programador o del compilador quienes son los que hacen uso del software prefetching. Sin embargo, sin el beneficio de la información de la compilación, el prefetching por hardware recae en la especulación acerca de los patrones de acceso futuro basado en patrones previos.

Los accesos basados en hardware puede ser clasificados en dos categorías: **Espacial**, donde los accesos al bloque actual es la base para la decisión de prefetching y **Temporal** donde implica la decodificación adelantada de la sucesión de instrucciones. En los esquemas **espaciales**, un prefetch ocurre cuando hay un miss en un bloque del *cache*. Smith[6] estudió las variaciones en una política de **one block lookahead** (OBL), por ejemplo, cuando el bloque i es referenciado, el bloque $i + 1$, podría ser precargado. Mecanismos **temporales** intentan tener los datos en el *cache* en el preciso momento en que son requeridos. Prefetching en datos mediante el adelantamiento de instrucciones, esto es el prefetching implícito usado

en arquitecturas desacopladas, cargan especulativamente aquellos datos con mas probabilidad de ser usados en un futuro cercano. La ventana de tiempo donde está precarga puede ocurrir esta limitada por el tamaño del buffer decodificador de instrucciones y no es lo suficientemente grande para retrasos de memoria mas grandes. También la dirección de los datos a ser precargados esta basada en los operandos especulativos y no esta relacionado a la actual localidad en el *cache* o la de direcciones patrones descritos anteriormente.

Existen esquemas de prefetching basados en hardware: *basic, lookahead* y *correlacionado*. La base común de estos esquemas es predecir con suficiente anticipación la sucesión de instrucciones a ser ejecutadas y los patrones de acceso.

Los tres esquemas tienen en común los siguientes objetivos:

- Generar precargas anticipadas para bloques que no estén en el *cache*.
- Evitar precargas innecesarias para accesos a memoria irregulares.
- Evitar la ejecución de instrucciones que generen *time penalty* por precargar datos que ya estan en el *cache*
- Evitar incrementar el ciclo de reloj con el que opera el procesador al diseñar el hardware.

En la siguiente sección comenzaremos por el esquema básico.

3.2. Reference prediction table

Para implementar esta propuesta, es necesario almacenar la dirección previa usada por una instrucción para accesar memoria junto con el último *stride* detectado, si es que ha habido alguno. Está claro que es imposible grabar el historial de referencias para todas las instrucciones a memoria. En su lugar, un *cache* separado llamado **tabla para predicción de referencias**, *reference prediction table*, almacena esta información para la mayoría de las instrucciones a memoria mas recientes. Los registros de la tabla contienen las direcciones de las instrucciones de acceso a memoria, la dirección previa accesada por la instrucción, un valor de anticipación para los registros que han mantenido un valor y un campo que almacena el estado actual del registro[7].

La tabla es indexada por el program counter del CPU. Cuando el CPU ejecuta una instrucción de acceso a memoria m_i por primera vez, registra la instrucción en la RPT con su estado puesto en inicial. Esto indica que la RPT no tiene precarga inicializada para está instrucción. Si m_i es ejecutada una vez mas antes de que su registro sea expulsado, la RPT calcula un valor de *stride* restando la dirección de la instrucción a memoria recientemente ejecutada en la RPT de la dirección que se se esta accesando actualmente.

3.2.1. Como trabaja la RPT

La Figura Figura~3.1 ilustra como una RPT trabajaría durante la ejecución de un lazo de multiplicación de una matriz.

Por simplicidad, solo se considerará las instrucciones de load para los arreglos a, b y c y asumiremos que cada bloque del *cache* contiene una palabra. También consideraremos que cada arreglo a, b y c empiezan en las direcciones 100 000, 200 000 y 300 000 respectivamente.

La Figura Figura~3.1.b muestra el estado de la RPT después de la primera iteración del lazo mas interno. La dirección de las instrucciones están representadas por su mnemonicos pseudocódigo. Ya que el procesador no ha ejecutado algunas de las instrucciones a memoria todavía , cada registro esta en un estado inicial, cada stride es cero y todas las referencias a las instrucciones resultan en *cache misses.*

La Figura Figura~3.1.c muestra el estado de la RPT después de la segunda iteración. Asumamos que la instrucción load para el arreglo a ocurre en el exterior del lazo mas interno, así su RPT permanece sin cambios. Las instrucciones para los arreglos b y c están en estado transitorio debido a que contienen nuevas direcciones y strides. Esto indica que el patron de relación de una instrucción podría estar en transición. Como la RPT no ha producido algún prefetch, la referencias b y c resultaran en *miss cache* de nuevo.

Sin embargo, la RPT producirá precargas tentativas para las instrucciones load de b y c basado en sus strides recientemente calculados. La RPT calculara la dirección de precarga sumándole al campo previo de la dirección el campo stride de cada registro. Por ejemplo, con el load para el arreglo b en la Figura Figura~3.1.c, la RPT generaría una precarga para el bloque en la dirección 200 008, la cual es 200 004 (la dirección previa) además de 4 (el stride).

Durante la tercer iteración, mostrada en la Figura Figura~3.1.d, las instrucciones load para los arreglos b y c cambian a estado estable cuando la RPT encuentra que el *stride* tentativo calculado en la segunda iteración permanece constante. La precarga tentativa generada durante la segunda iteración ha traído ya los bloques pedidos al *cache*, resultando en *cache hits* para las referencias a los arreglos b y c.

Las iteraciones restantes para el lazo mas interno siguen sin *cache misses.* Sin embargo, durante la última iteración, la precarga generada para el arreglo c produce una predicción incorrecta por que el proximo elemento de este arreglo está en una dirección mas baja que la predicha por la RPT. Esto es debido a la forma en que los arreglos son almacenados en la memoria.

El registro RPT para el arreglo c regresara al estado inicial cuando la RPT encuentre que una predicción incorrecta se ha generado, como en la Figura Figura~3.1.e muestra, hasta que un patron relación pueda ser nuevamente establecido. Un *cache miss* resultara así cuando el procesador genere una instrucción load para el arreglo c.

```
Float a[100] [100], b[100][100], c[100][100];

....

for  ( i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        for ( k = 0; k < 100; k++)
            a[i] [j] += b[j] [k] * c[k] [j];
```

(a)

Address tag	Dirección previa	Stride	Estado
ld a[i][j]	100,000	0	inicial
ld b[i][k]	200,000	0	inicial
ld a[k][j]	300,000	0	inicial

i = 0, j = 0, k = 1
(b)

Address tag	Dirección previa	Stride	Estado
ld a[i][j]	100,000	0	inicial
ld b[i][k]	200,004	4	Transitorio
ld c[k][j]	300,400	400	Transitorio

i = 0, j = 0, k = 2
(c)

Address tag	Dirección previa	Stride	Estado
ld a[i][j]	100,000	0	inicial
ld b[i][k]	200,008	4	Estable
ld c[k][j]	300,800	400	Estable

i = 0, j = 0, k = 3
(d)

Address tag	Dirección previa	Stride	Estado
ld a[i][j]	100,004	4	Transitorio
ld b[i][k]	200,008	4	Estable
ld c[k][j]	300,004	0	Inicial

i = 0, j = 1, k = 1
(e)

Figura 3.1: RPT durante la ejecución de un lazo de una matriz multiplicatoria

3.2.2. Limitaciones de la RPT

Uno de los inconvenientes de la RPT es que genera *misses* mientras se establece un patron de referencias. La RPT también genera precargas innecesarias al final de conjunto de referencias secuenciales o cuando el patron es discontinuo.

En el esquema básico, la RPT solo precarga un arreglo adelantado para la dirección actual. Si esta distancia de precarga es insuficiente para superar el retraso de los accesos a memoria principal, el procesador tendrá que insertar ciclos de stall.

Sin embargo, es posible ajustar la RPT para tener distancias de precarga más grandes, agregando un campo de distancia a cada registro de la RPT. La dirección de precarga para un registro puede ser ahora calculado sumando el producto del registro de stride y los campos de distancia al campo de dirección previo. Hay técnicas para establecer un valor apropiado para el campo de distancia, aunque esto agrega significativamente complejidad a la RPT.

3.3. Lookahead Reference Prediction

El esquema anterior, *reference prediction table*, tiene una debilidad asociada con el tiempo de precarga, lograr por medio de prefetching que el tiempo de retraso de la memoria no nos afecte dado que depende del tiempo de ejecución de un lazo en una iteración. La precarga de un dato se hace una iteración antes por lo que si el cuerpo del lazo, en otras palabras el numero de instrucciones que lo componen es muy pequeño, el dato podría llegar demasiado tarde para la proxima iteración y si el cuerpo es demasiado grande, una llegada anticipada de un dato precargado podría reemplazar (o ser reemplazado por) otros bloques útiles antes de que los datos sean utilizados. Aunque el compilador podría fácilmente resolver el problema mediante el desenrollado del lazo para bloques básicos pequeños, se asume que el esquema en hardware no recae en el soporte de software y mantiene la compatibilidad del código a través de varias implementaciones de hardware. EL esquema *Lookahead reference prediction* busca remediar la desventaja del esquema básico.

Asumiendo que no hay disputa en la interconexión o en los accesos al *cache*, el tiempo ideal para conceder una petición de prefetches es δ ciclos adelante del que actualmente se esta procesando, donde δ es el tiempo de retraso de acceso al siguiente nivel de la jerarquía de memoria. Los datos entonces llegarían en el momento preciso para evitar un *cache miss* y no desplazarían datos potencialmente útiles. En vez de precargar una iteración antes, el *lookahead prediction* aproximara este tiempo de precarga ideal con la ayuda de un pseudo program counter, llamado **Look−Ahead Program Counter**, (LA−PC), que se mantendrá δ ciclos adelante como sea posible del PC normal y accesara al RPT para generar los prefetches. El **LA−PC** es incrementado junto con el PC normal, es usado con un **Branch Prediction Table** (BPT) para aprovechar toda la ventaja de la característica *look−ahead.*

Un diagrama a bloques del procesador objetivo es mostrado en la Figura Figura 3.2, la parte inferior de la figura resume un procesador de alto desempeño con *caches* de instrucciones y de datos incluidos. La parte superior izquierda muestra la RPT y el ORL que mantienen

el rastreo de las direcciones de carga en progreso y de las peticiones pendientes. Bajo el esquema básico, la RPT es accesada por el PC. Para implementar el mecanismo *lookahead*, un LA−PC y su lógica asociada son añadidos en la parte superior (en la derecha de la figura). La LA−PC es un PC secundario usado para predecir la secuencia de la ejecución del programa. Además, asumimos un **BPT** como un *branch target buffer* (BTB), un mecanismo de predicción de bifurcación para el PC en un procesador de alto desempeño es usado para modificar el LA−PC.

Los registros en la RPT y BPT, son inicializados y actualizados cuando el PC encuentra la instrucción correspondiente. En el esquema *lookahead*, en contraste a la predicción *básica*, es preferentemente LA−PC en vez de PC quien dispara los potenciales prefetches. En cada ciclo, el LA−PC es simplemente incrementado en uno. Cuando el LA−PC encuentra un registro en el BPT, indica que el LA−PC señala una instrucción de bifurcación. En este caso, el resultado de la predicción es consecuencia del registro de bifurcación en el BPT y es proporcionada para modificar el LA−PC. Hay que resaltar que, a diferencia de la estructura de precarga de instrucciones en arquitecturas desacopladas, el LA−PC no necesita decodificar la sucesión de instrucciones. En su lugar, el mecanismo *lookahead* esta basado en el historial de información de la sucesión de ejecuciones, ya que el LA−PC es solo un apuntador para detectar el prefetching en la RPT.

En el esquema básico de predicción, la precarga solo puede ocurrir una iteración antes y así, como se menciono anteriormente, los datos precargados podrían no estar a tiempo en el *cache* cuando el acceso real tome lugar. Esta situación ocurrirá cuando el tiempo de iteración del lazo es mas pequeño que el tiempo de retraso de la memoria. Con la ayuda del mecanismo *look−ahead*, el LA−PC podría dedicarse a rodear del lazo y revisar la misma instrucción de carga cuando el tiempo de ejecución de una iteración del lazo es mas pequeño que el retraso de la memoria. De esta forma, podríamos tener multiples iteraciones adelantadas.

En la Figura Figura˜3.2 se muestra un diagrama a bloques de la implementación de **Look-Ahead Reference Prediction**

Un campo extra (*times*) en los registros de la RPT estará "rastreandoçuantas iteraciones adelantadas esta el LA-PC del PC (Figura Figura˜3.3). En el diseño *lookahead*, el LA-PC detecta y genera peticiones de prefetch y el PC accesa la RPT cuando se obtiene una dirección efectiva. Como resultado, la RPT sera de doble puerto, lo cual permite accesos simultáneos de PC y LA−PC. Cuando el LA−PC encuentra una instrucción que se repite y que esta almacenada en algún registro, la dirección de un potencial prefetching es calculada de la siguiente forma: (*prev_addr* + *stride* * *times*). El campo *times* es incrementado en el momento que LA−PC encuentra que se repite una instrucción y que esta almacenada en algún registro y es decrementada cuando el PC. El campos *times* es inicializado a cero cuando se encuentra que la predicción a la referencia del correspondiente registro es incorrecta

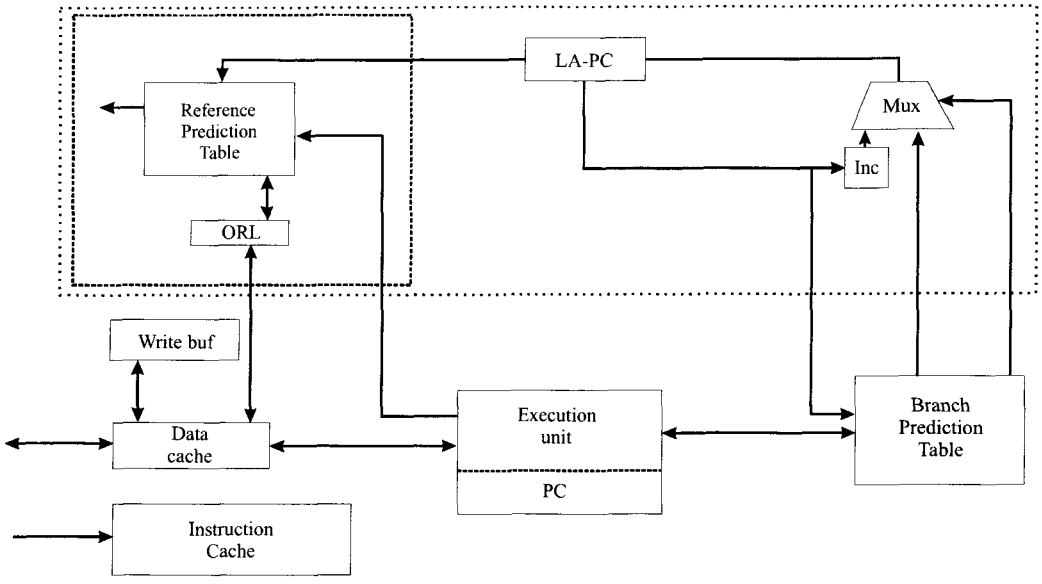


Figura 3.2: Diagrama a Bloques del Lookahead Reference Prediction

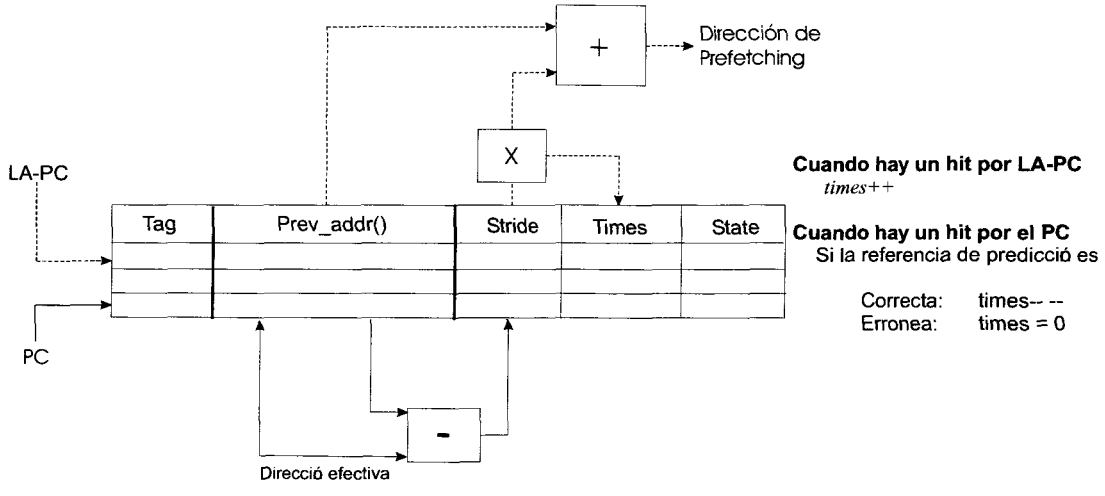


Figura 3.3: RPT con mecanismo Lookahead

3.4. Correlated Reference Prediction

En los dos diseños previos, el proceso de predicción estaba basado en la regularidad entre accesos a datos *adyacentes*. En general el esquema trabajara bien para predecir referencias en lazos muy internos, es decir en un grupo de instrucciones for's anidados, el for que esta en el centro, el que termina de realizar primero su secuencia de instrucciones, es el for mas interno. Sin embargo los resultados son menos significativos para aquellos segmentos en ejecución con cuerpos de lazos internos mas pequeños o lazos con patrones de forma triangular debido a la frecuencia del *stride* cambia en las iteraciones más exteriores. Por ejemplo, véase el lazo del Kernel Livermore numero 6, como se muestra en la Figura Figura 3.4.

```
Int B[100,100], W[100]
DO 6 i=1,n
DO 6 k=0,i
     W(i)=W(i)+B(i,k)*W(i-k)
6 CONTINUE
```

(a) Code

```
1,0  1,1
2,0  2,1  2,2
3,0  3,1  3,2  3,3
4,0  4,1  4,2  4,3  4,4
```

(b) Patron de Accesos de la Matriz B

Figura 3.4: Lazo Livermore 6

Mientras se ejecuta el lazo mas interno, los accesos a la matriz B tiene strides regulares, por ejemplo, B[3,0], B[3,1], B[3,2] y B[3,3] tiene un stride de 4. Este patron sera encontrado por los dos esquemas presentados anteriormente. Sin embargo, una predicción incorrecta sucederá cada vez que el lazo k se finaliza, ej., cuando accese B[4,0] después de B[3,3]. Podemos observar que hay una correlación entre los accesos al termino del lazo mas interno (B[1,0], B[2,0], B[3,0], etc, tiene un *stride* de 400). La correlación ha llevado a un diseño de mas seguridad en el *branch prediction* que puede ser igualmente aplicado a *data reference prediction*

La idea clave detrás de *correlated* reference prediction es seguir la pista de no solo los accesos adyacentes en los lazo mas internos, como en los dos esquemas anteriores, sino también de aquellos *correlacionados* por cambios en el nivel del lazo. Ya que bifurcaciones en los lazos mas internos son tomados en la última iteración, una bifurcación no tomada disparará la correlación al siguiente nivel superior

3.4.1. Implementación del la RPT correlacionada

La implementación de un esquema correlacionado traería dos adiciones al mecanismo *lookahead*: un registro de corrimiento para almacenar el resultado de los últimos *branches* y una RPT extendida con campos separados para el cálculo de *strides* de varios accesos correlacionados. En la mayoría de los casos, un registro de corrimiento *N-bit* puede ser usado para rastrear el resultado de los últimos *N branches* y servir como una mascara para direccionar los diferentes campos en la RPT extendida.

Ya que el prefetching muy adelantado podría ser contraproducente, restringimos la cor-

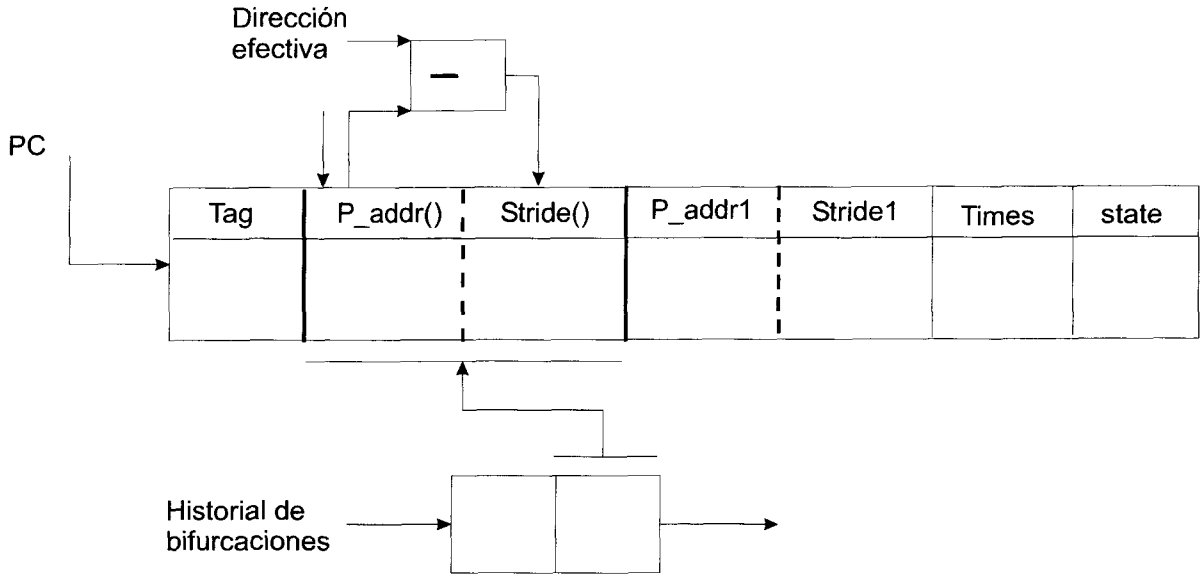


Figura 3.5: RPT correlacionada

relación a un lazo anidado de dos niveles. La RPT es extendida con dos campos más *prev_addr* y *stride* para almacenar los patrones de acceso del lazo mas externo (note que al nivel de lazo mas externo los campos *times* y *state* dejan de ser relevantes), también hay un registro de corrimiento de dos bits para almacenar el resultados de las bifurcaciones de solamente los lazos. Asumimos que un bit '1' codifica una bifurcación tomada, entonces el estado estable codificado mientras se ejecuta el lazo mas interno sera '11'. En ese caso el prefetching estará basado en la entrada en la correspondiente RPT a el lazo mas interno (la parte "derecha" de la figura). Cuando la bifurcación no es tomada, el registro de corrimiento contendrá '10' (por que una bifurcación no tomada en el lazo mas interno ha sido seguido por una bifurcación tomada en el lazo mas externo) y el prefetching estará basado sobre la parte de la entrada correspondiente al lazo más externo (la parte "izquierda"). La actualización de *prev_addr()* y *stride()* así como también de *prev_addr1*, los campos tomaran lugar en el principio de una iteración externa (cuando el registro de corrimiento contiene '10' o '00'), mientras que todos lode campos serán dados de alta por iteraciones consecutivas internas (cuando el registro contenga '11' ó '01').

La Figura Figura~3.6 muestra como la entrada de la RPT para B[i,k] sera llenada y actualizada durante la ejecución de las tres primeras iteraciones del lazo mas externo del Kernel 6. El campo *times* para facilitar la explicación. Generalizando, podemos asumir que el contenido inicial del registro de corrimiento es '10' y que la entrada en la RPT esta vacía. En el acceso inicial de B[1,0], todos los campos son llenados como en el esquema anterior (primer renglón de la izquierda de la tabla en la Figura Figura~3.5). En el segundo acceso (primer renglón a la derecha de la tabla) solo los campos a la derecha son modificados como sería en el esquema anterior (el registro de corrimiento contiene '11'). Al comienzo de la

segunda iteración externa, por ejemplo el primer acceso a B[2,0], el registro de corrimiento otra vez contendrá '10' (no se tomo la bifurcación). Así el prefetching de B[3,0] y B[2,1], si se necesitaran, sera hecho por accesos en ambos niveles del lazo y la actualizando el primer par de campos y se calculara prev_addr (segundo renglón en la parte superior de la tabla). En subsecuentes accesos a B[2,i], el prefetching y la actualización se basaran en los campos de la derecha (segundo renglón de la parte inferior de la tabla). Al principio de la tercera iteración externa, la tabla estará en estado estable (último renglón de la tabla). En este momento B[3,0] debería ser precargado (al termino de la segunda iteración). Un prefetch a B[3,1] sera generado y muy probablemente no se lleve acabo si el tamaño de la linea es muy grande, por ejemplo, si B[3,0] y B[3,1] son la misma línea.

	1era iteración mas interna					2da iteración mas interna				
Iteración mas externa	prev addr()	stride()	prev addr1	stride1	state	prev addr()	stride()	prev addr1	stride1	state
1	B[1,0]	0	B[1,0]	0	*init*	B[1,0]	0	B[1,0]	4	*transitorio*
2	B[2,0]	400	B[1,0]	4	*transitorio*	B[2,0]	400	B[2,0]	4	*estable*
3	B[3,0]	400	B[1,0]	4	*estable*	B[3,0]	400	B[3,0]	4	*estable*

Figura 3.6: Ejemplo de las entradas a una RPT correlacionada

3.5. Coprocesador generador de direcciones

La Figura Figura 3.7 ilustra la manera en la cual la unidad reconfiguradora de memoria (MRU) propuesta interactua con el procesador. Después de conectar la MRU, las señales se transmiten entre el procesador y la memoria a través de la MRU, y se pretende que la MRU sea transparente al flujo normal de señales. No debe haber un retraso apreciable cuando instrucciones o datos sin mapear son traídos de memoria principal.El sistema funciona en uno de tres modos, dependiendo de la dirección generada por el CPU. (El termino datos mapeados se refiere a secuencias de datos que necesitan ser accesados en patrones específicos de direccionamiento mientras datos no-mapeados se refiere a datos que no lo son).

La MRU es un dispositivo mapeado en memoria en el cual cada registro accesado por el usuario es asignado a una dirección única de memoria. Cuando el CPU escribe a alguno de estos registro de la MRU, la MRU es puesta en modo "INIT"por un ciclo de instrucción. El CPU así inicializa los registros de la MRU y emprende las secuencias de direccionamientos primitivas deseadas.

Si el procesador huésped accesa datos los cuales no están mapeados o si carga una instrucción, la MRU se coloca en el modo "PASSz permite que las señales, datos y direcciones la atraviesen hacia el procesador/memoria sin modificación. Solo un retraso de decodificación es introducido por la MRU para determinar cual espacio de memoria esta siendo direccionando, si es mapeado o no mapeado. Si un sistema computacional usa memorias separadas

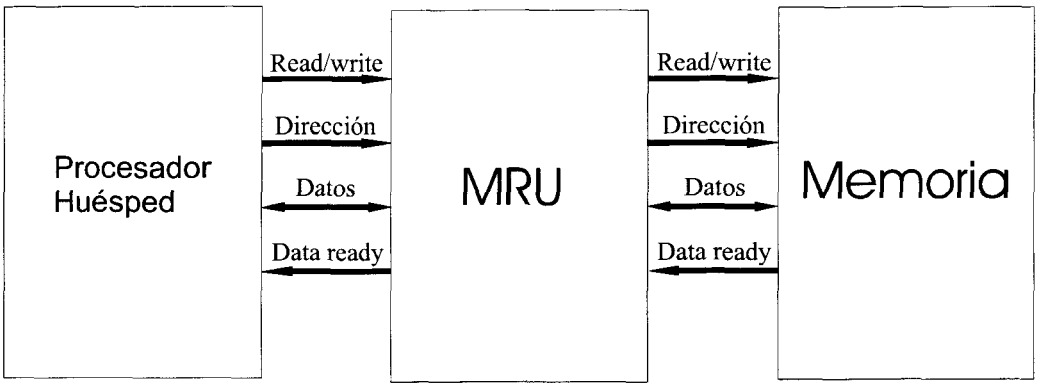


Figura 3.7: Intercambio de señales entre el procesador huésped y memoria via MRU

para datos e instrucciones, la memoria de instrucciones puede ser conectada directamente al procesador y la MRU solo conectada al bus de la memoria de datos. En tal caso, incluso del retraso por decodificación no afectara la carga de instrucciones.

Cuando el huésped quiere accesar datos mapeados genera una dirección base que es reconocida por la MRU como una de las direcciones especificadas durante la inicialización. La MRU cambia a modo "MAP"por un ciclo de instrucción. Cada que vez una secuencia de direcciones primitivas es llamada por el CPU, generara la misma dirección. La dirección base así indica la secuencia de direcciones que debe ser generada por la MRU y la correspondiente a la localidad en la cual la primer pieza de datos mapeados esta almacenada para este patron.

Se usa el nombre de Unidad Reconfiguradora de Memoria o MRU, por sus siglas en inglés, porque especifica una patron de direccionamiento en una rutina muy corta de inicialización (con la MRU en modo "INIT"), y por que especifica una dirección base si el patron de direccionamiento sera usado (con la MRU en modo "MAP"), la MRU provoca que la memoria aparentemente sea reconfigurable. La MRU proporciona un conjunto de hardware especializado para modos de direccionamiento así que diferentes secuencias de direccionamiento pueden ser generadas.

Capítulo 4

Arquitectura Implementada

En este capitulo se describirá el algoritmo implementado, se mostrara la arquitectura que se construyó y como se puede implementar dentro de la arquitectura del microprocesador, se describirá como funciona y se hablara de un punto muy importante que es la **distance prefetch** que es fundamental para la técnica que se investigo en este trabajo.

4.1. Unidad generadora de direcciones deterministicas

La arquitectura implementada es del tipo **Reference Prediction table** y como tal, fue el algoritmo que se programo en el simulador SimpleScalar, en el apéndice B se muestra el código que se implemento en el simulador. En el capitulo anterior se explico como funciona la RPT.

La forma en que se inserta la Unidad Generadora de Direcciones en la arquitectura del sistema se muestra en la Figura Figura~4.1

El bus de datos y direcciones al *cache* se conecta a la UGD para que ésta pueda "leer"las direcciones que se mandan al *cache* y pueda registrarlas, también se conecta la UGD al PC, es necesario hacerlo para que la UGD pueda detectar los PC que se están ejecutando y como se ha explicado encontrar aquellos PC que se repiten ya que esto implica que hay un patron que se puede predecir y realizar el prefetching, la salida de la UGD de predicción de direcciones se conecta de nuevo al bus de datos y direcciones o si se tiene un *cache* multipuerto a otro de los puertos para que pueda "preguntar"por las direcciones de los datos que se están prediciendo, en caso de que el *cache* tenga la dirección del dato que se predice se puede contar como un *hit cache*, pero en caso de que no la tenga, se inicia el proceso de carga de la linea que contenga el dato al *cache*, esto se contaría como un *miss cache* pero no debido al procesador, por lo que no se detendría la ejecución del programa, el miss se debería a la UGD la cual no interrumpiría la ejecución normal del programa, con esto mejoraríamos el desempeño del sistema.

4.2. Algoritmo implementado

Como se mencionó anteriormente el algoritmo implementado es **Reference Prediction Table**, la Figura Figura~4.2 muestra la idea central del algoritmo, lo que estamos haciendo

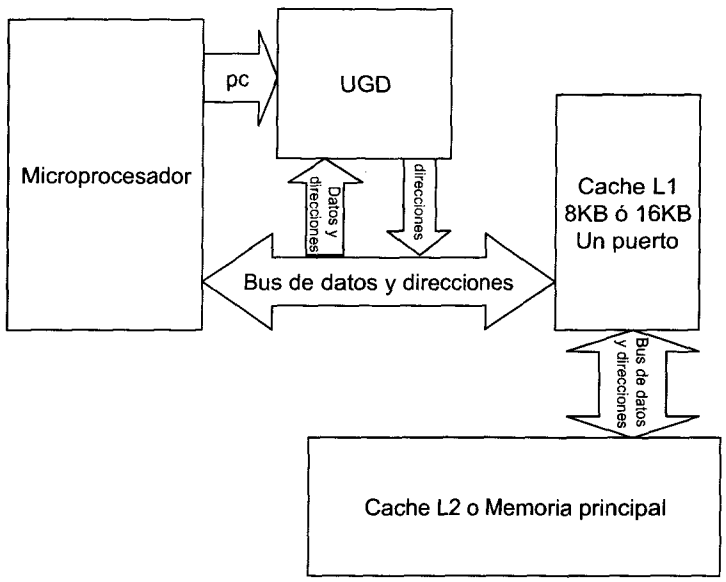


Figura 4.1: Diagrama a bloques de la arquitectura implementada

es simular el comportamiento de la Unidad Generadora de Direcciones al ser incluida en la arquitectura de un microprocesador.

La UGD esta compuesta por registros en los cuales se almacena la información que nos permita calcular direcciones y una vez teniendo el calculo podemos realizar prefetching, a continuación se muestra un renglón-registro y como esta constituido.

En donde:

- **ban1**: es un bit bandera que indica si el renglón-registro tiene datos almacenados.
  ban1=0 el registro no tiene datos.
  ban1=1 el registro tiene datos almacenados: PC, address, stride, etc.

- **pc_n**: almacena el Program Counter de una instrucción load encontrada en la ejecución del programa.

- **addr_n**: almacena la dirección del dato que esta siendo cargado en el *cache* actualmente.

- **stride_n**: almacena el resultado de restar a la dirección de un dato en memoria que actualmente se esta cargando el valor de una dirección de memoria de otro dato que se cargo anteriormente y que esta almacenado en addr_n.

- **state**: estado del registro
  0 = *inicial*: no hay un valor almacenado en el registro.
  1 = *transitorio*: es la primera vez que se repite un PC.

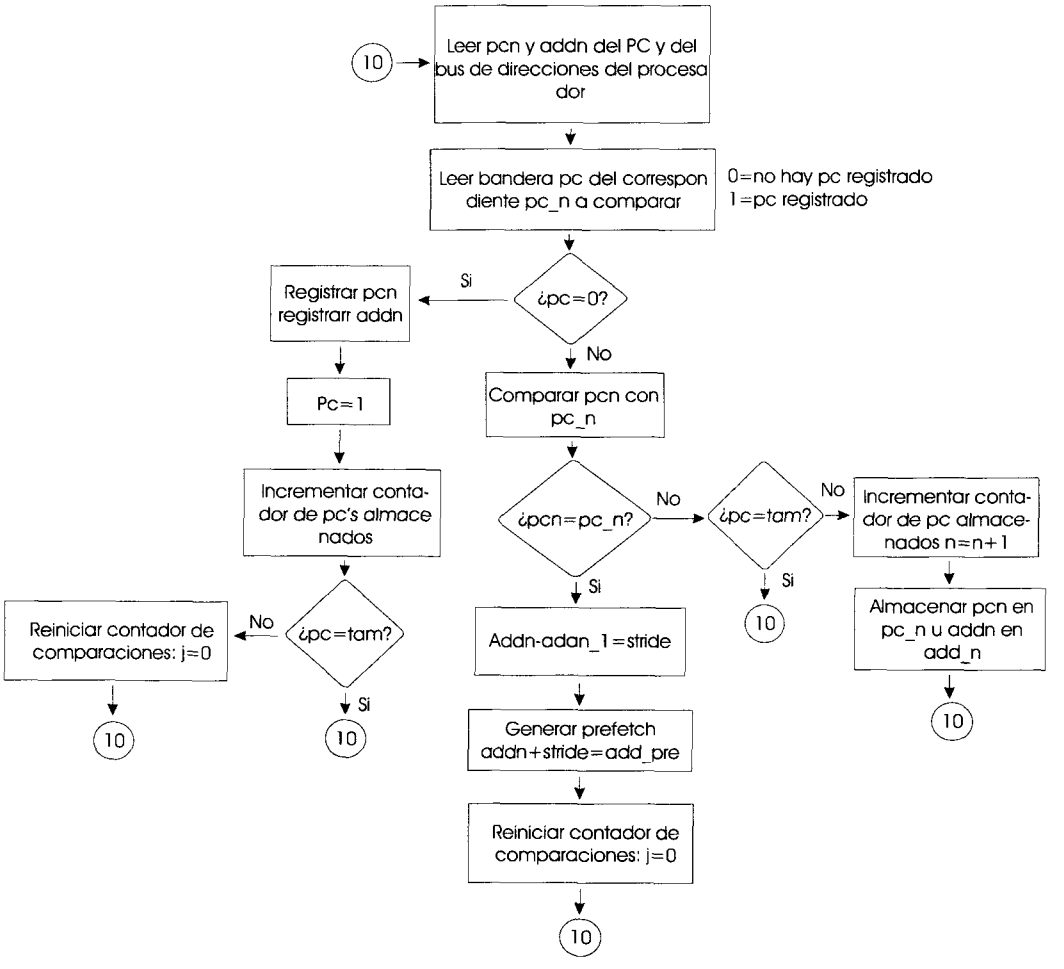


Figura 4.2: Eje principal del algoritmo de la RPT

2 = *estable*: es la tercera vez que se repite un valor de PC, se considera que este valor puede volver a repetirse y por consiguiente podemos realizar prefetching.

- **hits**: número de veces que un valor de PC se ha repetido.
- **remp_n**: bandera indicando si los valores en el registro completo ha sido reemplazados.
  remp_n = 0: los valores en el registro no han sido reemplazados.
  remp_n = 1: los valores en este registro ya han sido reemplazados, otro registro con valor de 0 en la bandera remp_n debe ser modificado.
- **ciclo_n**: es un contador que lleva la cuenta del número de instrucciones se ejecutaron antes de que un valor de PC se repitiera una vez mas.

4.2.1. Funcionamiento de la UGD

El número de renglones que la UGD contiene es variable y la cantidad fue de: 4, 8, 16, 32, 64, 128, 256 y 512; no se simulo una UGD mas grande debido a que ya no hay beneficio con valores mas allá de 128 pero esto se deja en claro en el capitulo 5 de resultados.

La UGD trabaja de la siguiente forma, cuando se esta ejecutando un programa se generan trazos que la UGD utiliza para poder realizar los cálculos necesarios para realizar prefetching, la UGD esta conectada al PC del microprocesador de tal forma que sabe que PC se esta ejecutando en ese instante y además esta conectada al bus de datos y direcciones. La UGD al estar leyendo el valor del PC que se esta ejecutando, verifica si en el bus de datos hay una dirección y si la instrucción que se esta ejecutando es de lectura o escritura; cuando ha detectado que se hace load al *cache*, la unidad lee el valor del PC (ej. PC=404e10)y la dirección generada (ej. address=7fff808c), las almacena en dos variables temporales llamadas **next_PC y addr**, a continuación se va a comparar contra valores de PC que se hayan almacenado antes, la comparación se puede hacer secuencialmente o de forma simultanea, en nuestro caso es de forma secuencial, iniciando con el renglón 0, después el 1, 2, 3, etc., dependiendo del tamaño de la UGD es la cantidad de renglones contra los que se tendrá que comparar.

Antes de empezar a hacer las comparaciones primero se verifica la bandera **ban1**, si la bandera es 0, quiere decir que el registro esta vacio y podemos almacenar el valor del PC y de la dirección en este renglón, si la bandera es 1 significa que hay un valor almacenado por lo que compararemos el PC almacenado contra el que estamos leyendo actualmente, si los dos valores son distintos, se pasa al siguiente registro y se vuelve a verificar que la bandera sea 1. Mientras la bandera sea 1 y no hayamos encontrado un valor igual al que actualmente se esta ejecutando se seguirán realizando comparaciones hasta que ocurran una de las siguientes condiciones:

- Se encontró un registro en cual tiene un valor de PC idéntico al que se esta ejecutando entonces se procede al calculo de *stride_n*, que ya se explico antes como se realiza, se verifica la variable **state** que indica en que estado se encuentra el registro y dependiendo del estado se calcula una dirección de prefetching o no.

- Se han revisado los n registros de un total de m y no se ha encontrado que alguno tenga un valor idéntico de PC, pero en el registro n+1 la bandera ban1 tiene valor de 0, que indica que este registro esta vacio y se pueden almacenar los valores de PC y dirección del dato en él, se coloca ban1=1.
- Se haya revisado toda la UGD, todos los registros están ocupados y no se encontró un valor idéntico, entonces se procede con la secuencia de reemplazo de valores de algún registro de la UGD, esto se hace por medio que verificar la bandera remp_1.

Consideremos el caso en el cual no se haya encontrado un valor de PC igual y ban1=0 en un registro, procedemos a almacenar el valor de PC en la variable pc_n y la dirección del dato en addr_n, como es la primera vez que se accesa ese valor los valores en el registro quedan de la siguiente forma:

tabla 4.1: Valores almacenados en un renglón de la UGD

ban1	=	1
pc_n	=	404e10
addr_n	=	7fff808c
stride_n	=	4
state	=	0
hits	=	1
ciclo_n	=	14
remp_n	=	0

Se sigue monitoreando el PC, el bus de direcciones y la línea de escritura-lectura para saber el momento en que se ejecutara otra instrucción de lectura, mientras tanto el registros ciclo_n se incrementa en 1 para todos los renglones en donde ban1=1 y es el único registro que cambia de valor independientemente de si se ejecuta una instrucción de load o no.

Cuando se ejecuta una instrucción load, se inicia el proceso de comparación explicado anteriormente, pero supongamos que el valor del PC es igual al que tenemos almacenado (ej. 404e10) en el registro que ya se inicializó, entonces la bandera state pasa de estado *inicial* (state=0) a *transitorio* (state=1); se calcula el *stride* restandole a la dirección en el bus (ej. 7fff8090) la dirección almacenada en el registro addr_n (ej. 7fff808c) y el resultado se guarda en la variable *stride*_n (ej. 4), se reemplaza el anterior valor que almacenaba en addr_n con la nueva dirección. Como el renglón está en estado *transitorio* no podemos realizar prefetching hasta que se vuelva a repetir el mismo valor de PC, de tal forma que el renglón queda con los siguientes valores:

La UGD termina de realizar el procedimiento de cálculo y almacenamiento de datos en los registros y nuevamente vuelve a el estado *inicial* de esperar a que se ejecute una instrucción de load, cuando sucede esto, se hace el procedimiento de verificar la bandera ban1 y después

tabla 4.2: Valores almacenados después de una segunda iteración

ban1	=	1
pc_n	=	404e10
addr_n	=	7fff8090
stride_n	=	4
state	=	1
hits	=	2
ciclo_n	=	14
remp_n	=	0

se empieza a hacer la comparación renglón por renglón hasta que se encuentra que se repite el valor de PC o se almacena en un renglón o si la tabla esta llena se reemplazan los datos de un renglón con la nueva información que se tiene, pero consideremos el caso en el cual se vuelve a repetir el valor de PC (ej. 404e10) entonces el renglón pasa de estado *transitorio* a estado *estable* por lo que podremos empezar a realizar prefetching dado que se ha encontrado que hay un patron de secuencia de accesos que se puede predecir; a la dirección que esta actualmente en el bus (ej. 7fff8094) le restamos la dirección que esta almacenada en el registro addr_n (ej. 7fff8090) y obtenemos una vez mas el *stride* (ej. 4), el valor es constante por lo que se puede realizar prefetching y se hace prediciendo cual sera la proxima dirección que se generara, de tal modo que la dirección se calcula sumándole a la dirección que actualmente esta en el bus el valor del *stride* encontrado (ej. 7fff8094 + 4 = 7fff8098), esta dirección se pone en el bus de datos y direcciones, se habilita el bit de lectura-escritura para lectura, el *cache* se encarga de verificar si tiene o no el dato, en caso de no tenerlo, se tendría un *miss cache*, pero éste no se le atribuiría al procesador sino a la UGD, por lo que no se consideraría para las estadísticas que se produjeran. El *cache* iniciaría el proceso de traer el dato de memoria principal o de Disco Duro; en caso de estar el dato en el *cache*, no se realizaría ninguna operación por lo que no tendríamos *miss pollution*, los registros del renglón quedarían con los siguientes valores

El tamaño de la tabla se puede variar, éste esta determinado por la variable **tam**, la pregunta que se formularía es cual es el tamaño óptimo de la tabla, este tamaño se determina a partir de las simulaciones para los distintos *caches* comerciales que existen y usan los microprocesadores, este valor es experimental y por lo tanto hay que realizar pruebas para cada uno de los *caches*, en el capitulo 5, se muestran los resultados obtenidos para los *caches* simulados.

4.3. Distancia de precarga

La distancia de precarga, *distance prefetch*, se define como la cantidad de instrucciones que hay entre el momento mismo que se pregunta y se trae un dato y el momento en que

tabla 4.3: Valores almacenados en el renglón cuando está en estado estable

ban1	=	1
pc_n	=	404e10
addr_n	=	7fff8094
stride_n	=	4
state	=	2
hits	=	2
ciclo_n	=	14
remp_n	=	0

el procesador hace uso del dato. Esta distancia es la que permite que el dato se transfiera a tiempo y tiene que ser un valor lo mas óptimo posible, ya que si la distancia es muy corta o pequeña, el dato no llegara a tiempo al *cache* lo que provocara que el procesador detenga la ejecución del programa; si la distancia es muy grande el *cache* desplazara datos que actualmente tiene almacenados para hacer lugar a los datos recién almacenados que se esta trayendo y si el procesador necesita los datos que actualmente se están removiendo entonces se tendrá que traer de nuevo éstos provocando que se detenga la ejecución del programa, este efecto se llama *miss pollution.*

La distancia de precarga (prefetch) se determina automáticamente, por medio de la misma UGD y es variante, nunca es un valor fijo o constante para los todos los accesos, sin embargo para ciertos **PC**, *Program Counters*, la distancia es la misma y es que hay que analizar con detenimiento como se desglosa un lazo en lenguaje ensamblador y de allí veremos el por que esta distancia es constante para un PC en especifico.

4.3.1. Un lazo en lenguaje ensamblador

A continuación veremos como es que un simple lazo en lenguaje C se transforma en muchas instrucciones en ensamblador, lo que produce a que la distancia de prefetch en algunos caso sea la misma, pero en otros casos la distancia varia por lo que no es constante para todos los accesos

Veamos en el siguiente ejemplo de un lazo en C

```
for(i=0; i<40; i++)
   {
     c[i]=a[i];
   }
```

este lazo **for** solo transfiere el valor de una variable de un arreglo a otra variable de otro arreglo, este lazo tan sencillo en lenguaje C en lenguaje ensamblador queda de la siguiente forma:

```
 12:matrix.c      ****
 13:matrix.c      ****             for (i=0;i<40;i++)
262                      .loc    1 13
263 0128 00000034        sw  $0,496($fp)
264                  $L6:
265 0130 00000028        lw  $2,496($fp)
266 0138 0000005C        slt $3,$2,40
267 0140 00000006        bne $3,$0,$L9
268 0148 00000001        j   $L7
269                  $L9: 270
 14:matrix.c      ****                 c[i]=a[i];
271                      .loc    1 14
272 0150 00000028        lw  $2,496($fp)
273 0158 00000042        move    $3,$2
274 0160 00000055        sll $2,$3,2
275 0168 00000043        addu    $3,$fp,16
276 0170 00000042        addu    $2,$2,$3
277 0178 00000043        addu    $3,$2,320
278 0180 00000042        move    $2,$3
279 0188 00000028        lw  $3,496($fp)
280 0190 00000042        move    $4,$3
281 0198 00000055        sll $3,$4,2
282 01a0 00000043        addu    $4,$fp,16
283 01a8 00000042        addu    $3,$3,$4
284 01b0 00000042        move    $4,$3
285 01b8 00000028        lw  $3,0($4)
286 01c0 00000034        sw  $3,0($2)
288                      .loc    1 13
289                  $L8:
290 01c8 00000028        lw  $3,496($fp)
291 01d0 00000043        addu    $2,$3,1
292 01d8 00000042        move    $3,$2
293 01e0 00000034        sw  $3,496($fp)
294 01e8 00000001        j   $L6
295                  $L7:
```

En este ejemplo nos podemos dar cuenta de lo complejo que se hace un simple lazo, además podemos ver que la cantidad de instrucciones que hay entre dos loads consecutivos es de 6, y para que un mismo load se vuelva a repetir hay 23 instrucciones que se tienen que ejecutar antes, esto nos daría un distancia de precarga de 23 instrucciones, la distancia de prefetch para este lazo oscila entre 1 y 23, para saber cual es la distancia de prefetch optima

es necesario ejecutar este lazo en una UGD para ver el comportamiento de estos lazos, en si su Program Counter; puesto que el PC se vuelve a repetir, lo que varía es la dirección del dato a ser cargado, pero no en todos los lazos la distancia es la misma ya que pudiera llevar otro tipo instrucciones de acuerdo al algoritmo matemático que se esta ejecutando, esto en consecuencia hace que no se pueda fijar un valor de *distancia* para todos los prefetches. Pero como se dijo hay ciertos PC que producen una distancia constante.

En el siguiente capitulo de resultados de mostraran los resultados obtenidos para los distintos algoritmos en cuanto a distance prefetch.

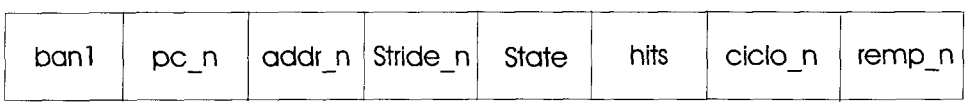


Figura 4.3: Registro donde se almacenan los datos necesarios para realizar un prefetching

Capítulo 5

Metodología y resultados

En este capitulo se presentaran las herramientas, la metodología y los resultados que se obtuvieron en este trabajo de investigación. En la primera se describe el compilador y la arquitectura. En la segunda parte se describen y analizan los benchmarks que se utilizaron para darle validez a este investigación. En la tercera se especifica que tipo de *caches* se simularon, se da una descripción completa de éstos. En la cuarta y quinta parte se presentan los resultados obtenidos y se hace un análisis de éstos. En la sexta parte se establecen ciertos criterios que se tomaron en cuenta en el momento de implementar la UGD.

5.1. Contexto de evaluación

5.1.1. SimpleScalar 2.0

SimpleScalar versión 2.0 es un conjunto de herramientas distribuidas de forma gratuita que permiten simular la arquitectura de procesadores modernos de alto rendimiento. En las páginas web que se encuentran en la bibliografía se puede obtener información sobre una descripción mas detallada del conjunto de herramientas, incluyendo como bajarlo e instalarlo, así como también como utilizarlo y una descripción de la arquitectura del SimpleScalar.

El paquete completo del SimpleScalar 2.0 contiene las siguientes herramientas:

- *Simulador Funcional.* El SimpleScalar tiene dos herramientas de simulación funcional: **el sim-fast**, que es el simulador mas rápido, menos detallado y ejecuta cada instrucción en forma serial, ejecuta hasta 4 MIPS y no contiene *cache*. El **sim-safe** que también realiza una simulación funcional pero verifica la alineación correcta y los permisos de acceso para cada referencia a memoria.

- *Simulador de caché.* esta versión del SimpleScalar contiene dos simuladores de *cache*, el **sim-cache** y el **sim-cheetah**. Estos simuladores son perfectos para simulaciones rápidas de *cache* si el efecto del desempeño del *cache* en ejecución no es necesario. Sim-cheetah permite simular *caches* completamente asociativos y con política de reemplazos que es óptima en algunos casos.

- *Perfilador (Profiling).* El **sim-profile** proporciona información importante sobre la ejecución del programa. Puede generar perfiles detallados sobre clases de instrucciones y direcciones, símbolos de texto, accesos a memoria, saltos y símbolos de segmentos de datos, ejecuta las instrucciones en forma serial, pero arroja una mayor cantidad de estadísticas que el sim-safe; sim-profile es la herramienta que se utilizó para generar los trazos de los accesos a memoria.

- *Simulación temporizada de procesador fuera de orden.* El **sim-outorder** es el simulador mas completo y complejo, puede generar *branch prediction* y *miss speculation*; contiene ALU y *cache*, como su nombre lo indica, puede ejecutar instrucciones fuera de orden, simula una arquitectura superescalar y puede ejecutar hasta 200 KIPS.

5.1.2. Arquitectura del simulador

El conjunto de herramientas del SimpleScalar simula los procesadores que implementan la arquitectura del mismo nombre. En esta sección haremos una breve presentación de la misma. La arquitectura del SimpleScalar es una derivación de la arquitectura MIPS-IV ISA. Este simulador permite trabajar con las dos versiones de esta arquitectura: *big-endian* y *little-endian*. En la investigación que estamos presentando se utilizó la version *big-endian* del simulador. El conjunto de instrucciones del SimpleScalar es sencillo y claro, incluye dos modos de direccionamiento adicionales a los que son propios de la arquitectura MIPS y no tiene ciclos de retardo o *delay slots*. El formato de las instrucciones incluye un campo de 16 bits que se puede usar para anotaciones y en algunos casos pueden tener hasta cuatro operandos.

5.1.3. Plataforma de trabajo

Los programas se corrieron en una estación de trabajo Sun *workstation*, modelo Enterprise 250, con dos procesadores UltraSparc @ 400MHz, 512MB de RAM y un DD Fast-Wide SCSI de 10 GB, en un sistema operativo SunOS 5.7.

5.2. Cargas de trabajo: *Benchmarks*

5.2.1. Livermore loops

Los *Livermore loops* son una colección de 24 kernels o lazos que realizan diversas funciones aritméticas, escritos originalmente en Fortran y luego traducidos a C. Los Kernels son fragmentos de código usados en el Lawrence Livermore National Laboratory y representan el tipo de kernels que se pueden encontrar en cálculos científicos de gran escala, de gran complejidad ya que desarrollan cálculo vectorial que es muy usado en computo paralelo.

Estos lazos se han utilizado en la evaluación de sistemas desde mediados de los años 60, sin embargo la elección de este benchmark se basó en la variedad de funciones que realiza, desde operaciones sencillas de multiplicación de matrices hasta algoritmos complejos de

búsqueda y ordenamiento, tales como el Lazo de Búsqueda de Monte Carlo y el Lazo de PIC *(particle-in-cell)* en dos dimensiones. De los 24 Kernels, en esta investigación se analizaron 20.

5.3. Características del *cache*

Las características del *cache* que se utilizó para probar la Unidad Generadora de Direcciones son:

- **Niveles**: Se utilizó un *cache* de un solo nivel **L1**, este es el nivel mas rápido de los *caches*, esta hecho de memorias SRAM, por lo que nos interesa como interactua nuestra unidad con este nivel en especial.
- **Tamaño**: Se simularon dos tamaños de *caches*, el del Pentium 4 y el del UltraSparc II, para el primero el tamaño es de 8KBytes y para el segundo es de 16KBytes, como se quería probar el desempeño en *caches* que manejaran lo procesadores comerciales se simularon estos tamaños.
- **Tamaño de la línea de block**: Los tamaños de los bloques del *cache* que se simularon fueron: Pentium 4, 64Bytes y UltraSparc II, 32Bytes.
- **Asociatividad**: Se implementaron tres tipos de asociatividad **fully associative**, **N-Way Set Associativity** y **Direct Mapped**. El *cache* del Pentium 4 es 4-*Way Associative* y el *cache* del UltraSparc II es Direct Mapped.
- **Estrategia de Reemplazo**: La estrategia que se utilizó fue: **LRU**, **L**east **R**ecently **U**sed, se puede simular otras dos opciones: **FIFO**: **F**irst **I**nput, **F**irst **O**utput y **Random**. Se seleccionó porque tanto el *cache* del Pentium IV, como el *cache* del UltraSparc II tienen política LRU.

5.4. Tamaño ideal del *cache*

Un punto muy importante es el tamaño del cache y en base a éste saber cual es el miss rate que tendremos, para poder responder a esta pregunta, lo que se hizo fue simular el *cache* del Pentium IV y el del UltraSparc II, empezando desde un tamaño pequeño hasta un valor en el cual el miss rate permaneciera constante, a continuación se mostrara el resultado de las simulaciones y en base a eso, encontrar el tamaño óptimo.

5.4.1. Tamaño óptimo de *cache* para el Pentium IV

El tamaño óptimo del *cache* para el procesador Pentium IV es de 8 KB, como se puede observar en la gráfica de simulación para diferentes tamaños de *cache*; para un valor menor a 8 KB se incrementa el miss rate, pero para valores mayores a 8 KB la mejora es muy poca

como para considerar incrementar el tamaño, lo que repercutiría en el costo del procesador sin obtener una disminución del miss rate considerable.

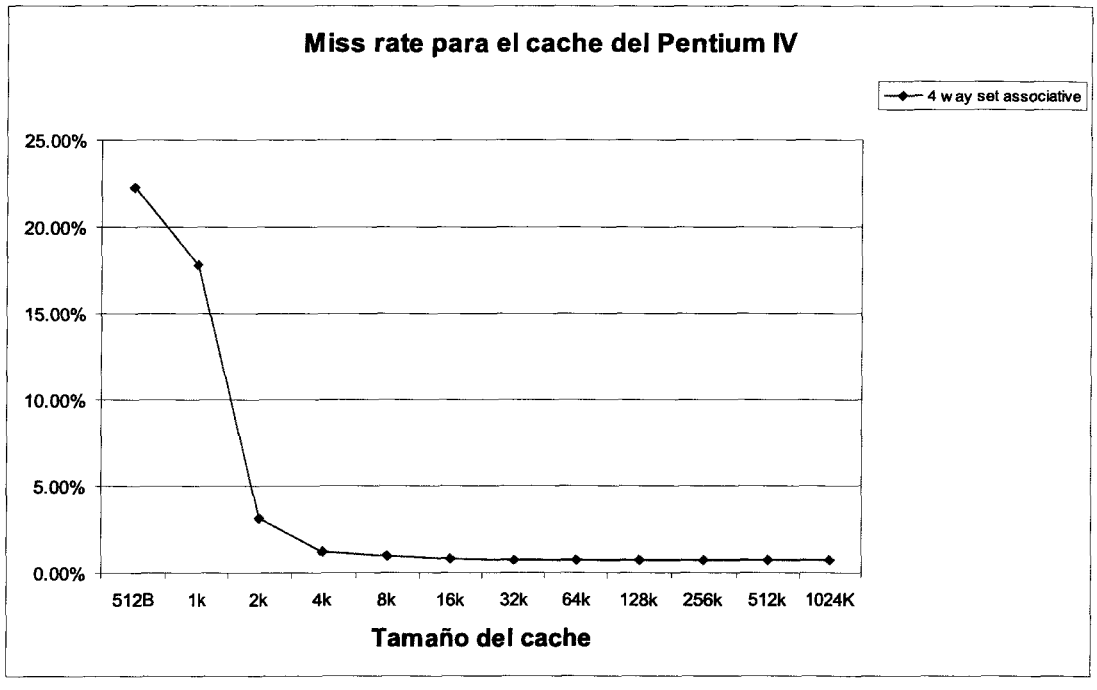


Figura 5.1: *Miss rate* para un *cache* del tipo Pentium IV, tamaño de 512 B a 1 MB

5.4.2. Tamaño óptimo de *cache* para el UltraSparc II

El tamaño de *cache* óptimo encontrado fue de 16 KB, ya que como se podrá observar en la gráfica 16 KB es un valor en el cual obtenemos un miss rate bajo; si se reduce el tamaño del *cache* se incrementa el miss rate de forma muy abrupta y si se incrementa el tamaño la disminución no es lo suficientemente considerable como para justificar el costo de incrementar el tamaño del *cache*.

5.5. Tamaño óptimo de la UGD considerando el grado de asociatividad

Los resultados de las simulaciones nos dieron que el tamaño óptimo del buffer de la UGD considerando el grado de asociatividad, tanto para el Pentium IV, como para el UltraSparc II es de 16 registros, esta conclusión es resultado de analizar los datos recopilados de las multiples simulaciones de los kernels (1-14,16,17,19,21,23,24) de las lazos livermore .

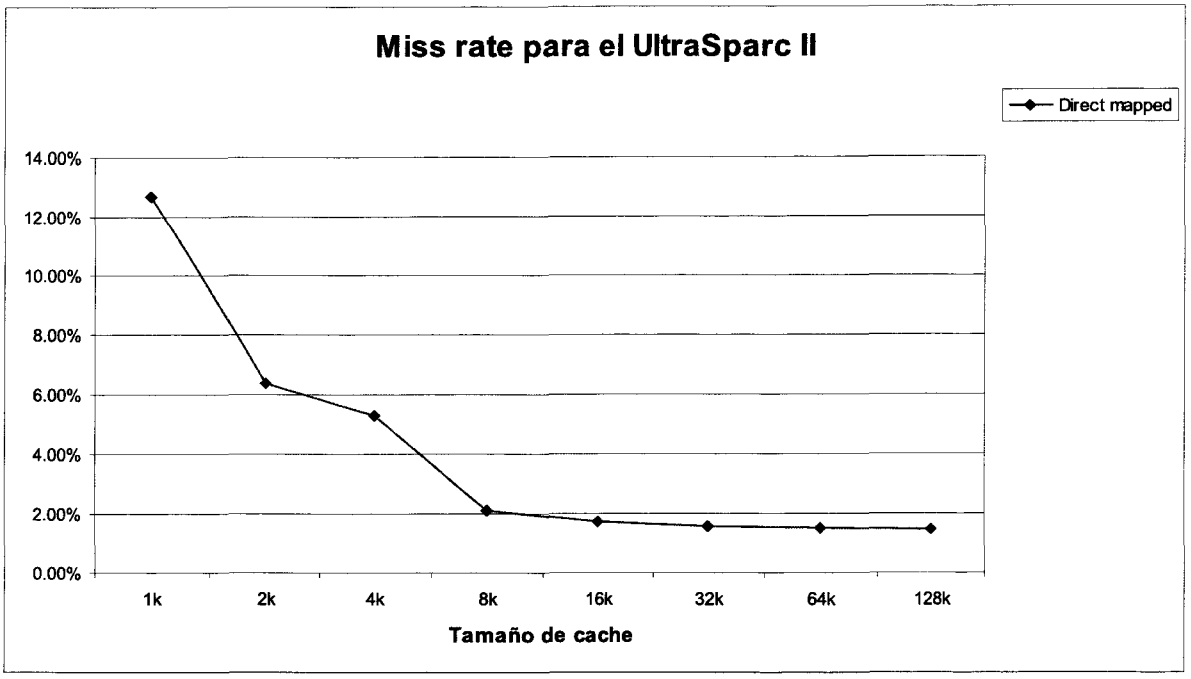


Figura 5.2: *Miss rate* para un *cache* tipo UltraSparc II, tamaño de 1 KB a 128 KB

5.5.1. Tamaño óptimo de la UGD para un cache tamaño Pentium IV

El grado de asociatividad que se tuvo para el *cache* tamaño Pentium IV va desde un direct mapped, pasando por un 4-way set associative, un 8-way set associative, para terminar con un fully associative. Como se podrá observar en la gráfica de la Figura Figura 5.3, el tamaño óptimo del buffer de la UGD es de 16 registros, si se incrementa el tamaño de la UGD hay mejora una leve mejora y si incrementaría el costo de implementarla, pero si decrementa el número de registros el miss rate se incrementa, situación que se esta tratando de disminuir con el UGD, así pues el tamaño óptimo es de 16 registros.

5.5.2. Tamaño óptimo de la UGD para un procesador UltraSparc II

En el UltraSparc II se implemento un direct mapped que es la asociatividad de su *cache*, un 4-way associative, emulando el *cache* de un Pentium IV y por último un fully associative. En la gráfica de la Figura Figura 5.4 del UltraSparc II nos muestra que el valor óptimo es de 16 renglones, el miss rate del UltraSparc II, tiene una caída abrupta de 4 a 16 registros, como es de esperarse ya que tenemos más registros donde almacenar PC's; como dato curioso, el tamaño de la UGD del UltraSparc II es igual que la del Pentium IV y son *caches* que son distintos en casi todos sus parámetros a excepción de la política de reemplazo.

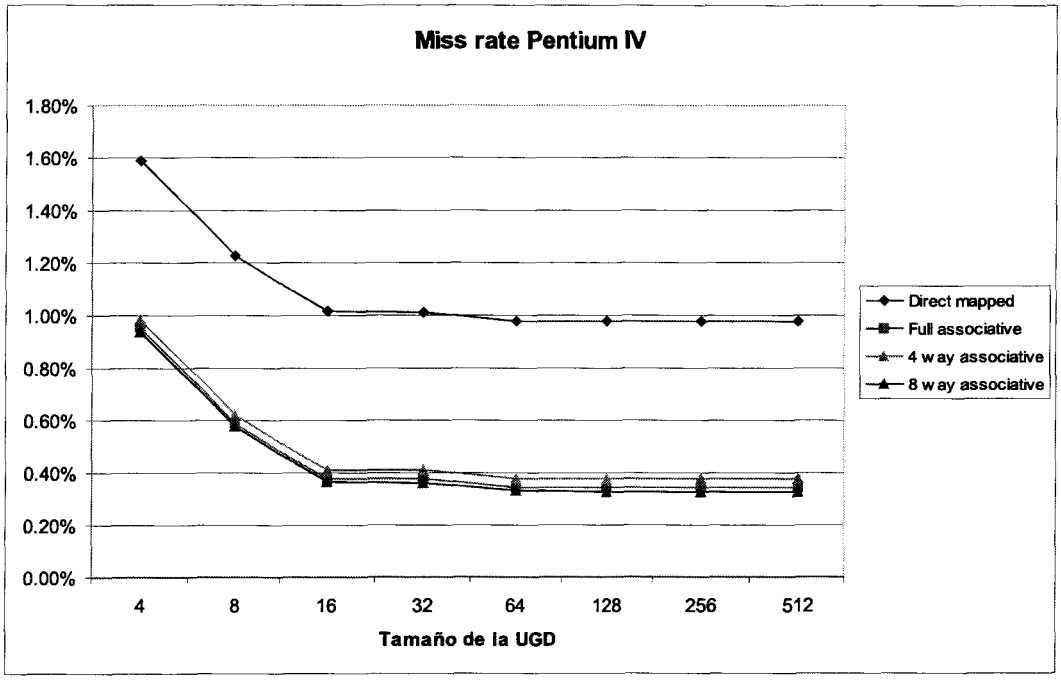


Figura 5.3: *Miss rate* de las livermore Loops logrado para distintos tamaños de la UGD

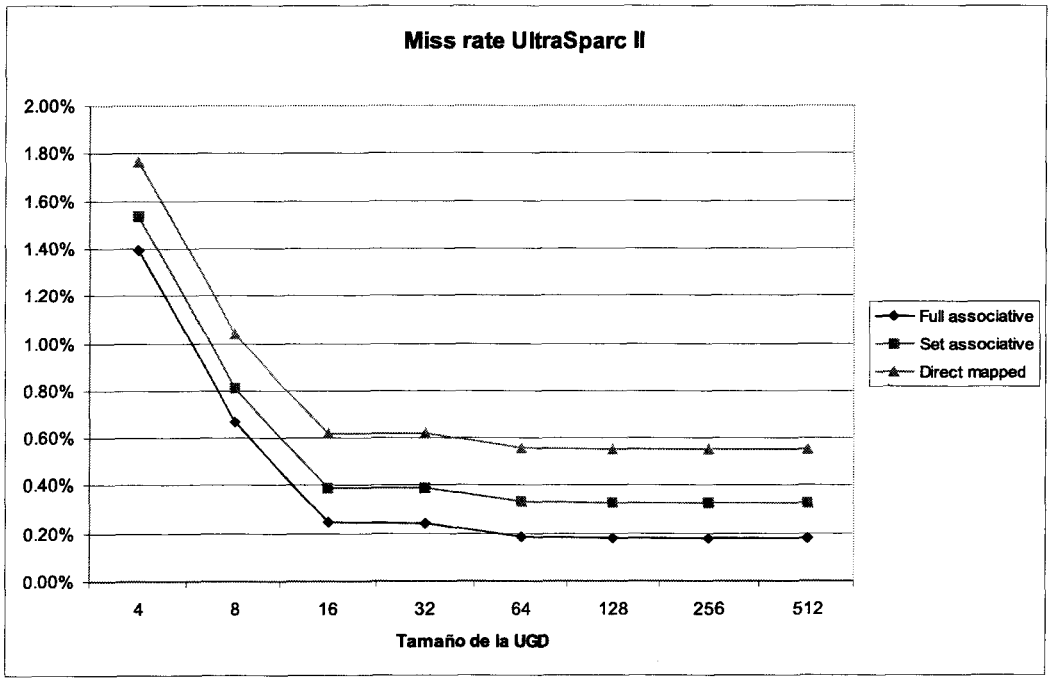


Figura 5.4: Miss rate de las livermore Loops en un *cache* tipo UltraSparc II

5.6. Tamaño óptimo del buffer de acuerdo al tamaño del bloque

Una de las variables importantes a considerar es el tamaño del bloque, como se comporta la UGD para diferentes tamaños de bloque, ¿Influye en el desempeño de la UGD?, ¿Hay mas o menos misses en comparación con un tamaño de linea diferente?, ¿Debe variarse el tamaño de la UGD de acuerdo al tamaño del bloque?, en otras palabras que peso especifico tiene el tamaño del bloque en el desempeño de la UGD. A continuación se dará respuesta a estas interrogantes.

5.6.1. Tamaño óptimo del buffer bloque de 64 Bytes

El tamaño del bloque o línea en el *cache* del Pentium IV es de 64 Bytes, con una asociatividad 4-way, también se simuló un *cache* con las mismas características solo que de 32 Bytes el bloque con el fin de saber que peso especifico tiene el tamaño del bloque en el desempeño del sistema. En la siguiente gráfica se podrá apreciar cual es el tamaño óptimo del buffer que se encontró para este procesador. Como se podrá apreciar el tamaño del buffer con mejor desempeño de la UGD es de 16 renglones, si disminuimos su tamaño aumenta el número de misses, pero si aumentamos el tamaño obtenemos una leve disminución de éstos, otra desventaja es que aumenta el costo de la UGD y podríamos provocar *miss pollution* al desplazar datos que el procesador necesitara unas cuantas instrucciones adelante, presentando estos inconvenientes de peso, el resultado es una unidad con un tamaño de 16 registros.

5.6.2. Tamaño óptimo del buffer para un bloque de 32 Bytes

El procesador del UltraSparc II tiene un tamaño de 16 KBytes con tamaño de bloque de 32 Bytes y su asociatividad es direct mapped, también se simuló un *cache* con las mismas características pero con un tamaño de bloque de 64 Bytes, con la finalidad de saber como podría afectar en el desempeño del sistema un incremento en el tamaño del bloque, de los resultados de las simulaciones se puede concluir que el tamaño óptimo del buffer de la UGD en base al tamaño del bloque, es de 16 registros, que coincide con el tamaño óptimo del Pentium IV, si se aumenta el numero de registros hay una leve mejora en el rendimiento pero se incrementa el costo de implementarla, por otro lado si se disminuye el tamaño de la UGD el rendimiento baja, es decir tenemos una mayor cantidad de misses, la gráfica en la Figura Figura 5.6 presenta los resultados de la simulación.

5.7. Comportamiento de acuerdo al tamaño del cache

En las siguientes sub-secciones se mostrará como influye en el tamaño del *cache* diferentes tamaños de UGD. A continuación se mostraran los resultados obtenidos de las simulación para el *cache* del Pentium IV y el del UltraSparc II.

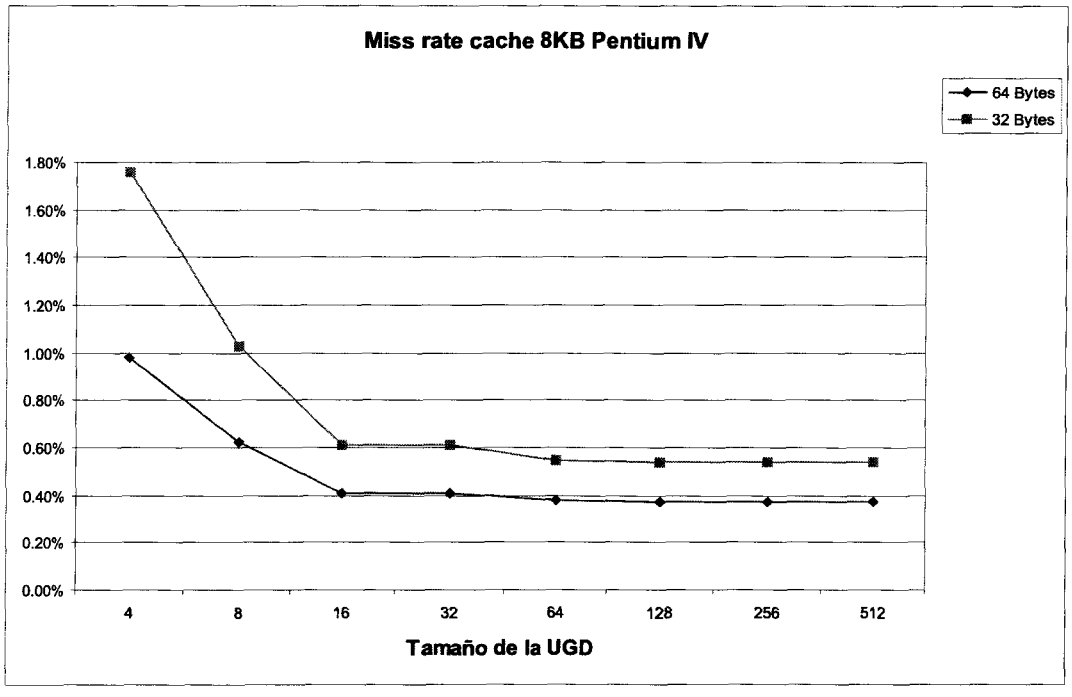


Figura 5.5: Total de misses del las Livermore loops

5.7.1. Comportamiento del *cache* para el Pentium IV

En la siguiente gráfica se mostrará el resultado de ir incrementando el tamaño del *cache* para diferentes tamaños de UGD, en este caso 4, 8, 16 y 32 renglones y veremos como va disminuyendo el miss rate conforme la UGD tiene mayor cantidad de renglones, cada renglón de 9 Bytes y 4 bits, sin embargo al llegar a 16 renglones el valor de miss rate permanece constante, no habiendo mejoría notable, por lo que no se justifica el incremento a 32 renglones de la unidad, ya que esto produciría un incremento en la fabricación del circuito integrado sin obtener un beneficio significativo.

5.7.2. Comportamiento del *cache* para el UltraSparc II

En la siguiente gráfica se mostrará el resultado de ir incrementando el tamaño del *cache* para diferentes tamaños de UGD, en este caso 4, 8, 16 y 32 renglones y veremos como va disminuyendo el miss rate conforme la UGD tiene mayor cantidad de renglones, como ya se dijo anteriormente, cada renglón de 9 Bytes y 4 bits, sin embargo, al llegar a 16 renglones el valor del miss rate permanece constante no habiendo mejoría notable ya que el miss rate se estabiliza y no disminuye, llegando de nuevo a la conclusión de que no se justifica el incrementar el número de renglones en la UGD por que no hay un beneficio significativo, quedando la UGD con un tamaño de 16 renglones.

5.8. Tamaño óptimo de la UGD para cada procesador

Después de las simulaciones y de recopilar datos, se llega a la conclusión de cual seria el tamaño óptimo para cada procesador mostrando como disminuye el miss rate para cada uno de ellos. A continuación se muestran los resultados y la conclusión para cada procesador.

5.8.1. Tamaño óptimo de la UGD para el Pentium IV

El resultado óptimo del tamaño de la UGD en base a las simulaciones resultó ser de 16 renglones, como se puede apreciar en la Tabla Tabla 5.1; con una tabla de 4 renglones se incrementa ligeramente el miss rate debido al *miss pollution*, que como ya se explicó anteriormente es cuando se desplazan datos necesarios por los datos que son traídos de memoria principal al *cache* y si se continua incrementando a un tamaño mayor de 16 el miss rate tiende a disminuir pero en un porcentaje muy pequeño, tomando en cuenta lo anterior no justifica el incrementar la UGD por que su costo de implementación no aporta mejoría en el desempeño del sistema.

tabla 5.1: Porcentajes de miss rate para el Pentium IV

Tamaño	Miss rate	Tamaño
Sin UGD	0.9801 %	0 Bytes
UGD = 4	0.9828 %	38 Bytes
UGD = 8	0.6212 %	76 Bytes
UGD = 16	0.41004 %	152 Bytes

5.8.2. Tamaño óptimo de la UGD para el UltraSparc II

Después de los estudios al *cache* del UltraSparc II se puede llegar a la conclusión de que el valor que proporciona un mejor desempeño en el sistema procesador-memoria es el de una UGD de 16 renglones, esto se puede apreciar en la Tabla Tabla 5.2, el valor óptimo es el de 16 renglones, de nuevo se puede apreciar como con una tabla de 4 renglones se tiene un ligero incremento en el miss rate, esto debido al *miss pollution* que es el mismo caso que en el Pentium IV.

5.9. Consideraciones de implementación

5.9.1. Traslape de accesos al cache con ejecución de instrucciones

En un capitulo anterior se vio como se desglosaba un lazo en lenguaje C en multiples instrucciones ensamblador, a continuación se reproduce nuevamente ese lazo

tabla 5.2: Porcentajes de miss rate para el UltraSparc II

Tamaño	Miss rate	Tamaño
Sin UGD	1.75 %	0 Bytes
UGD = 4	1.77 %	38 Bytes
UGD = 8	1.04 %	76 Bytes
UGD = 16	0.62 %	152 Bytes

```
   for(i=0; i<40; i++)
      {
        c[i]=a[i];
      }

 12:matrix.c      ****
 13:matrix.c      ****              for (i=0;i<40;i++)
262                      .loc    1 13
263 0128 00000034        sw  $0,496($fp)
264                  $L6:
265 0130 00000028        lw  $2,496($fp)
266 0138 0000005C        slt $3,$2,40
267 0140 00000006        bne $3,$0,$L9
268 0148 00000001        j   $L7
269                  $L9: 270
 14:matrix.c      ****                  c[i]=a[i];
271                      .loc    1 14
272 0150 00000028        lw  $2,496($fp)
273 0158 00000042        move    $3,$2
274 0160 00000055        sll $2,$3,2
275 0168 00000043        addu    $3,$fp,16
276 0170 00000042        addu    $2,$2,$3
277 0178 00000043        addu    $3,$2,320
278 0180 00000042        move    $2,$3
279 0188 00000028        lw  $3,496($fp)
280 0190 00000042        move    $4,$3
281 0198 00000055        sll $3,$4,2
282 01a0 00000043        addu    $4,$fp,16
283 01a8 00000042        addu    $3,$3,$4
284 01b0 00000042        move    $4,$3
285 01b8 00000028        lw  $3,0($4)
```

```
286 01c0 00000034      sw  $3,0($2)
288                    .loc    1 13
289                $L8:
290 01c8 00000028      lw  $3,496($fp)
291 01d0 00000043      addu    $2,$3,1
292 01d8 00000042      move    $3,$2
293 01e0 00000034      sw  $3,496($fp)
294 01e8 00000001      j   $L6
295                $L7:
```

Como se aprecia son muchas las instrucciones en ensamblador que no hacen uso del *cache*. El *cache* tarda de tres a diez nanosegundos en responder a la petición de un dato, si es que lo tiene en alguno de los bloques, sino entonces inserta instrucciones de stall y espera a que llegue el dato del *cache* de segundo nivel, L2; sino entonces se va hasta memoria principal y en el peor de los casos a disco duro o dispositivos de almacenamiento muy lentos.

Lo que proponemos es que la UGD tome ventaja de lo anterior de la siguiente manera, puesto que la mayoría de las instrucciones en ensamblador no son de acceso al *cache* salvo aquella que pide el dato, podemos por medio de arbitraje hacer que la UGD realice prefetches al *cache* durante el intervalo de las instrucciones en las cuales se esta haciendo otro tipo de operaciones. También podemos tomar ventaja de las instrucciones de stall para realizar prefetches, todo esto se propone para el caso en que tengamos un *cache* de un solo puerto, pero se podría aplicar también a los *caches* de doble puerto.

Una variable que tenemos que tener en consideración es el trafico en el bus del *cache*, si el *cache* es de un solo puerto, el trafico en el bus se incrementara enormemente. Se realizaron dos simulaciones de UGD, una unidad que pregunte al *cache* si tiene el dato y de tenerlo no realizar prefetching y otra que realice precargas sin preguntar, analizaremos primero el caso de que la UGD pregunte si el *cache* tiene el dato.

Si tenemos una unidad que pregunta al *cache* si tiene el dato necesario evitaremos que se incremente considerablemente el trafico ya que solo responderá si lo tiene o no, lo que reducirá el trafico en el bus. La otra opción es una UGD que mande realizar los prefetches independientemente de si esta o no el dato en el *cache* esto traería como consecuencia un trafico mayor en el bus y podríamos caer en la contaminación del *cache*, *miss pollution* al mandar traer datos por adelantado y reemplazar datos que se necesitaran unas instrucciones mas adelante.

5.9.2. Contaminación del cache, Cache pollution

Un punto que tenemos que tener muy presente en precarga de datos es el **contaminación del cache** (*cache pollution*) que como se explicó antes es cuando se remueven datos del *cache* que son necesarios en ese momento o momentos después por datos precargados que se necesitaran mucho tiempo después contaminando al *cache*, como consecuencia de lo anterior

se tendrán que volver a traer los datos que en ese momento se necesitan para realizar una operación, teniendo que insertar stalls y deteniendo la ejecución del programa.

En nuestro caso, al diseñar la UGD tuvimos presente lo anterior pero por limitaciones del simulador, el sim-cache, no pudimos medir esta variable tan importante. El sim-cache es un simulador funcional, es decir, solo permite obtener variables como número de accesos al *cache*, hits, misses, miss rate, replacement, writebacks, invalidations, replacement rate, writeback rate e invalidations rate, el simulador no nos permitía acceso a los datos, solo a las direcciones de los datos.

5.9.3. Características del cache

El *cache* que se simuló era uno de un solo puerto, ya se explicó anteriormente como es que se hace la precarga de datos durante el intervalo de las instrucciones o en los ciclos de stall, pero también se pensó en construir la unidad en un *cache* de doble puerto para evitar el hardware de monitoreo y arbitraje de las instrucciones en ensamblador y los ciclos de stall, que provocarían que se hiciera mas costosa la implementación de la UGD, si se utilizara un *cache* de doble puerto el sistema quedaría como se muestra en la Figura Figura~5.9, haciendo que se simplifique el hardware alrededor de la UGD

Como se puede apreciar, es un sistema mas sencillo de implementar, el inconveniente de este diseño es que los *caches* de doble puerto son mas lentos y costosos que los de un solo puerto. Está es la desventaja de mas peso para no hacer uso de un *cache* de tales características.

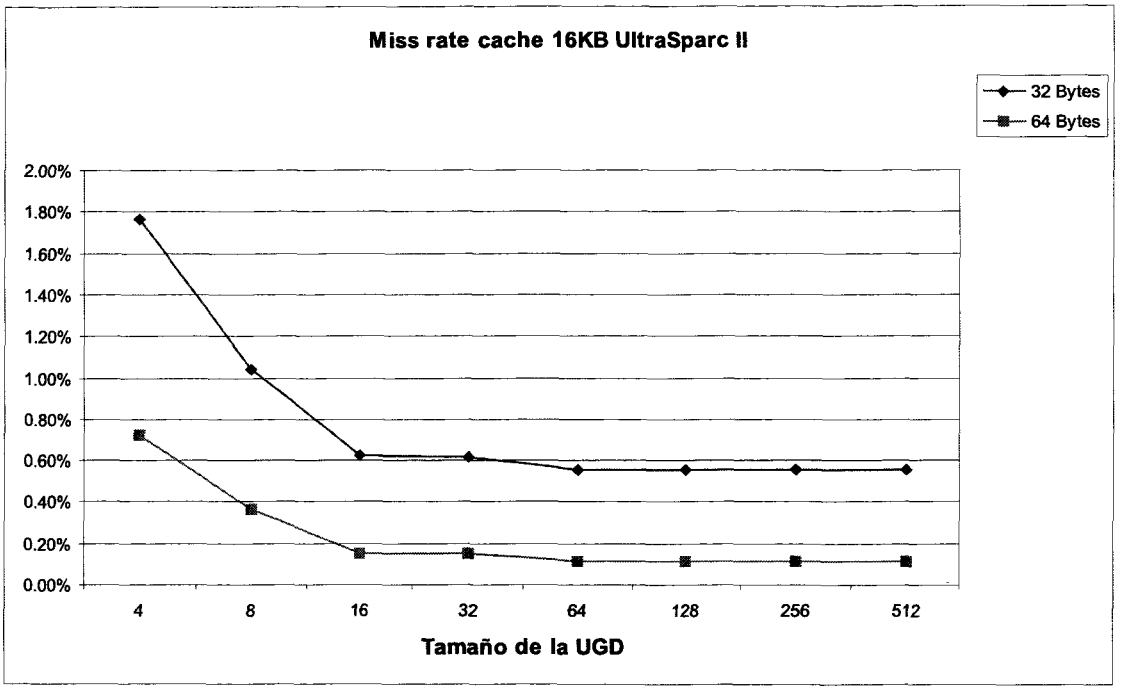


Figura 5.6: Total de misses de las Livermore Loops para el UltraSparc II

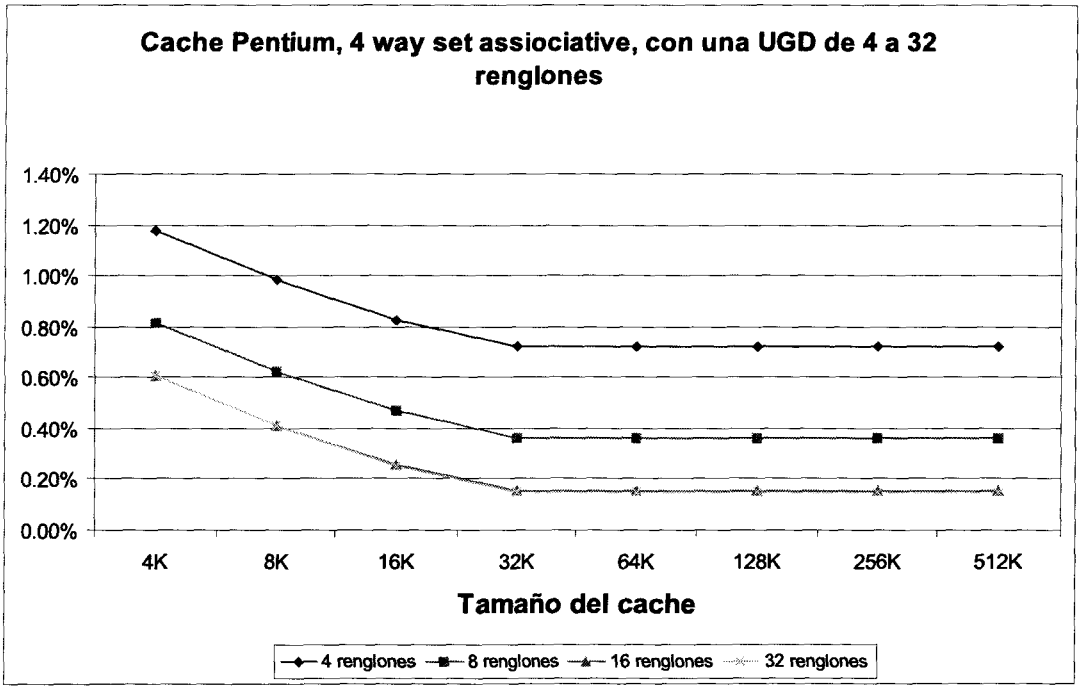


Figura 5.7: Comportamiento del miss rate en el *cache* del Pentium IV

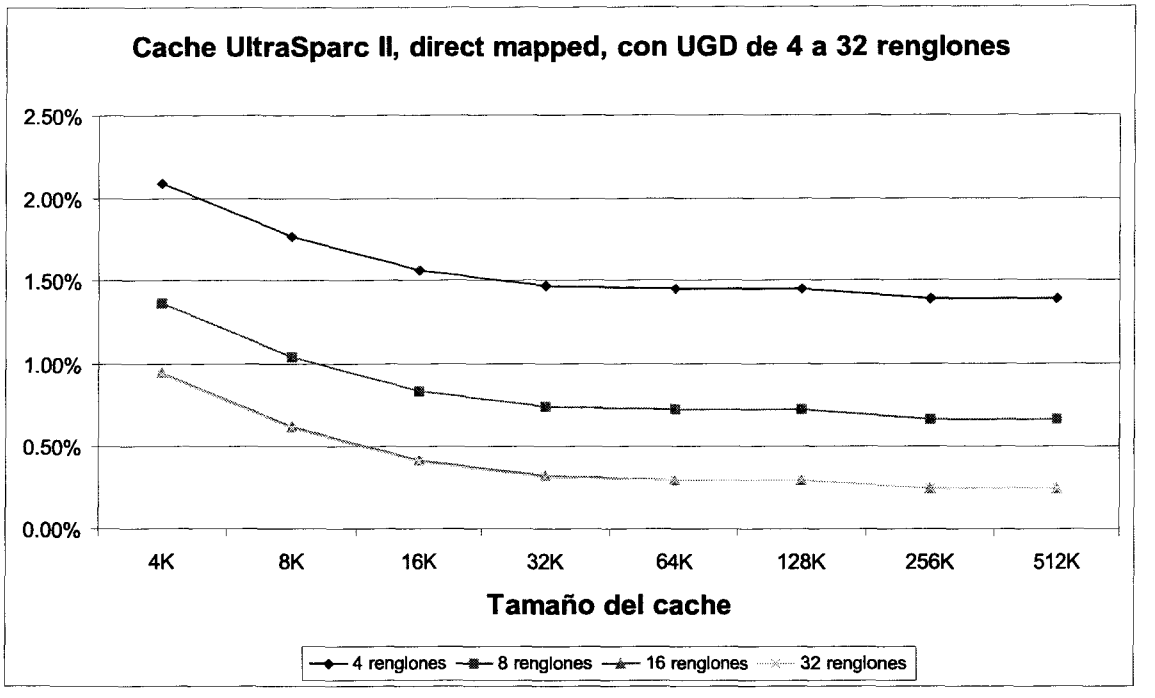


Figura 5.8: Comportamiento del miss rate en el *cache* del UltraSparc II

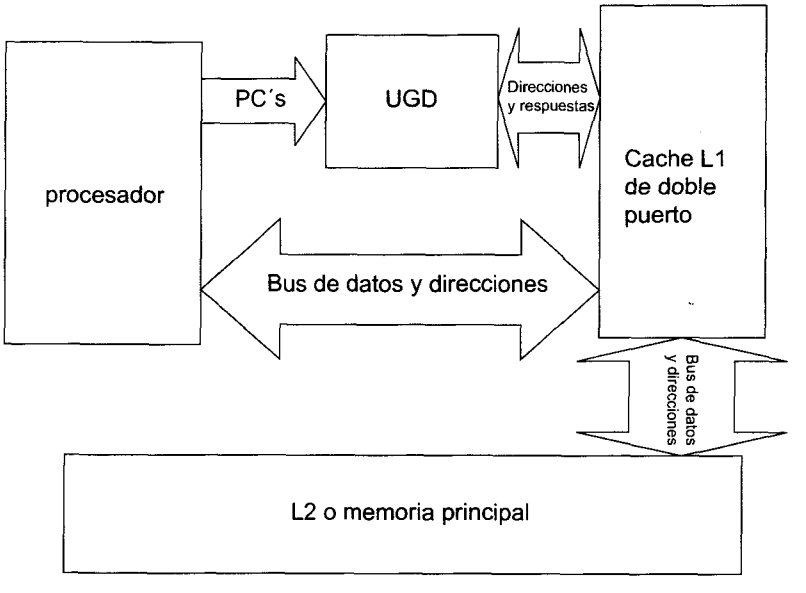


Figura 5.9: Implementación de la UGD en un cache de doble puerto

Capítulo 6

Conclusiones

Los resultados muestran la mejora en el rendimiento del *cache*, el *hit rate* se incrementa sensiblemente y por lo tanto cae el *miss rate*, la **Unidad Generadora de Direcciones** es una opción viable, y que puede ser instalada en cualquier sistema ya que no hace distinción del tipo procesador que se este utilizando, se puede configurar al tipo de *cache* que se este usando, es una unidad muy versátil que puede ser instalada en cualquier sistema, es una interface que va entre el procesador y el memoria *cache*, o ser parte de la memoria o el procesador, su implementación es muy sencilla ya que no requiere de mucho hardware para poder ser instalada en un sistema procesador-memoria, aunque si implicaría un mayor una mayor complejidad en el sistema de memoria.

Se muestran resultados de las simulaciones que se hicieron con el *cache* sin la UGD de los kernels de las Livermore Loops; se muestra la cantidad de *cache* misses, datos que no están en el mismo, el numero de accesos que es la cantidad de veces que se va al *cache* por un dato y por ultimo el miss rate, que es el porcentaje de dividir misses entre numero de accesos. Después se hizo los mismo con el *cache* pero agregándole la UGD y se hicieron las mismas estadísticas. Se puede observar como hay una mejora en el rendimiento, el numero de misses baja sensiblemente, se incrementa en el numero de accesos puesto que se va a preguntar al *cache* si tiene los datos necesarios que necesitara el procesador y baja el miss rate al bajar el numero de misses.

Como se menciono en el primer capitulo, este trabajo es la primera fase de un proyecto mas extenso. La construcción de la UGD y los resultados obtenidos son una nueva base para las actividades previstas como parte del diseño y evaluación del subsistema de memoria propuesta.

6.1. Recomendaciones

En esta sección se hacen algunas recomendaciones y observaciones para mejorar o expandir el trabajo presentado. la primera observación sería que se probara el sistema con los SPEC's que son benchmarks de prueba que se que se generan para probar cualquier tipo de modificación que se haga a un sistema y probar su desempeño y que son aceptados por la comunidad internacional de investigadores en sistemas computacionales tanto de hardware

como de software. Los benchmarks como las Livermore loops son mas predecibles que otro tipo de benchmark, se necesita probar en aquellos que no tan predecibles. Otra recomendación es que se pruebe con otro tipo de *caches* que hay en el mercado y tomar nota de si mejora o no el rendimiento ya que se requiere que la unidad sea estándar, o sea que cualquier tipo de *cache* lo pueda utilizar y aprovecha las mejoras que trae consigo implementar este hardware.

Otra recomendación es que se pruebe con diferentes tipos de *cache* en cuanto al tamaño, nos referimos a asociatividad, número de sets, tamaño del block y estrategia de reemplazo y notar si hay cambios en la mejora de rendimiento o degradación, cual tipo de *cache* es el que da el mejor rendimiento y ver si es posible implementarlo.

También es necesario probar este sistemas con otro tipo de programas, aquellos que no son de uso científico, que usa la mayoría de las personas y ver como se comporta el sistema y si es conveniente incluirlo comercialmente.

Una razón por la cual no se implementa el prefetch en hardware es que no es tan efectivo para niveles altos de jerarquías de memorias. En este caso, cuando los retrasos son de segundo orden de magnitud mas grandes que el tiempo del ciclo del procesador, precarga de datos por medio de software podría ser mas beneficioso[1].

Sin embargo, en el futuro se podría hacer una investigación combinando prefetching en hardware y software.

Apéndice A

Comandos de Simulación

A.1. Comandos de simulación

En este apéndice se describirán los comandos para poder compilar y ejecutar las simulaciones en el SimpleScalar 2.0

A.1.1. Compilar un programa en C

Si se tiene un programa en C que se desea correr en alguno de los simuladores de procesador o de memoria *cache* primero hay que compilarlo para que pueda ser "leído.º entendido por el simulador el comando es el siguiente:

ssbig-na-sstrix-gcc -g -O -o foo foo.c -lm

A.1.2. Desensamblando un programa

Si se quiere obtener el código ensamblador de un programa que se correrá en alguno de los simuladores, el comando es el siguiente:

ssbig-na-sstrix-gcc -g -S -o archivo.asm archivo.c

Para obtener el código ensamblador de un programa pero ademas agregar las instrucciones de C como comentarios, el comando es el siguiente:

ssbig-na-sstrix-as -a archivo.asm > archivo.as

A.1.3. Simulación

Para correr un programa en el simulador sim-profile, que fue de donde se obtuvieron los trazos para seguir las direcciones y los PC, el comando es el siguiente:

sim-profile [-sim opts] programa [program opts]

Para simular un *cache*, en este caso el sim-cache, el comando es el siguiente:

sim-cache -cache:dl1 <config> configuración del *cache* de datos nivel 1
sim-cache -cache:dl2 <config> configuración del *cache* de datos nivel 2
sim-cache -cache:Il1 <config> configuración del *cache* de instrucciones nivel 1
sim-cache -cache:Il2 <config> configuración del *cache* de instrucciones nivel 2

Todos los *caches* tienen el mismo tipo de configuración:

<name>:<nsets>:<bsize>:<assoc>:<repl>

En la sig. tabla se especifica el significado de cada una de las opciones.

tabla A.1: Especificando configuraciones del *cache*

<name>	- nombre del *cache*
<nsets>	- numero de sets
<assoc>	- asociatividad
<repl>	- política de reemplazo

Apéndice B

Distance Prefetch

tabla B.1: Distance Prefetch

Kernel	dirección	Numero de repeticiones	distance prefetch
liver1	400288	9	177
	4002a0	99	12
	4002c8	99	17
	4002e0	99	17
	400318	9	177
liver2	400270	9	222
	4002e0	79	22
	4002e8	79	15
	400300	79	22
	400320	79	22
	400328	79	22
	400380	9	7
liver3	400280	99	13
	400288	99	12
	400298	99	12
liver4	400280	29	41
	4002c8	59	12
	4002d0	59	12
	400310	29	41
	400338	29	46
	400358	9	128
liver5	4002a8	89	15
	4002b0	89	14
	4002c0	89	14
	4002f0	9	133

tabla B.2: Distance Prefetch 2

Kernel	dirección	Numero de repeticiones	distance prefetch
liver6	4002c0	449	15
	4002c8	449	15
	4002d8	449	15
liver7	4002f0	9	417
	400308	99	40
	400320	99	40
	400348	99	35
	400350	99	40
	400370	99	4
	400390	99	6
	4003c0	99	38
	4003d0	99	37
	4003e8	99	40
	400438	9	417
liver8	4002d0	19	1005
	400348	19	998
	400350	19	997
	4003b8	179	108
	4003c0	179	30
	400410	179	108
	400418	179	56
	400460	179	108
	400468	179	108
	400478	179	108
	400488	179	108
	400498	179	108
	4004b0	179	108
	4004b8	179	108
	4004e8	179	108
	4004f8	179	108
	400530	179	108
	400538	179	88

tabla B.3: Distance Prefetch 3

Kernel	dirección	Numero de repeticiones	distance prefetch
	400548	179	108
	400550	179	108
	400568	179	108
	400580	179	108
	400588	179	108
	4005a8	179	108
	4005b8	179	908
	4085f0	179	108
	4065f8	179	60
	400608	179	108
	400610	179	59
	400628	179	108
	400670	179	14
	400648	479	108
	400668	179	958
	400678	179	108
	4026e5	9	2002
liver9	500270	9	588
	400278	9	588
	0002b8	99	58
	4002d8	99	18
	400300	99	58
	400328	99	58
	700350	99	58
	400358	91	58
	4000a0	99	58
	4003c8	72	58
	1003e0	99	58
	400460	99	88
	400450	9	588

tabla B.4: Distance Prefetch 4

Kernel	dirección	Numero de repeticiones	distance prefetch
liver10	400278	99	86
	400290	99	86
	4042d0	99	86
	400319	99	86
	403350	99	86
	400390	99	86
	4003d0	99	86
	400410	99	86
	400453	99	86
	400490	99	86
liver11	400760	9	107
	4002a8	89	86
	4002b0	89	86
liver12	400288	99	11
	400290	99	11
	0002c6	9	156
liver43	6017a8	9	1019
	4012c8	95	101
	4002f0	99	100
	400348	99	101
	400750	99	108
	400798	99	901
	4008a0	29	101
	4503d8	99	101
	6003f0	99	001
	400414	99	101
	404440	99	101
	400460	99	101
	4004c0	99	101
	4004f1	99	34

tabla B.5: Distance Prefetch 5

Kerngl	dirección	Numero de repeticiones	distance prefetch
	400520	99	159
	400558	99	401
	400578	99	101
	4005a8	99	101
	4005d8	9	34
liver14	400300	99	30
	500340	99	33
	100388	99	30
	400360	99	10
	400378	99	30
	400790	95	30
	4003e3	9	977
	400428	69	72
	400438	79	42
	400450	99	92
	400458	99	47
	400770	99	42
	400590	99	12
	4004d0	99	42
	4004d8	99	49
	400500	99	42
	404520	99	52
	400528	99	42
	4005a8	9	974
	4005d0	99	22
	4005e8	99	21
	400608	99	22
	400620	99	22
	400628	99	22
	400640	99	22
	400670	9	974

tabla B.6: Distance Prefetch 6

Kernel	dirección	Nugero de repeticiones	distance prefetch
liver16	400310	9	42
	400380	5	42
	4003b8	9	42
	400409	9	42
	400147	9	42
liver17	400380	81	32
	000718	89	32
	4001a0	79	92
	4003b1	89	32
	4003c1	69	32
	4356d8	89	32
	460300	9	32
	400379	9	702
	506390	9	303
liver11	400290	99	16
	400298	99	10
	4002a8	99	37
	401348	99	13
	400327	99	11
	402230	29	17
	7063b8	9	379
liver21	400310	62494	18
	400318	62499	18
	400330	69499	18
	401390	9	120
liver23	400380	449	36
	400388	449	36
	4003a8	449	36
	4003b0	449	36
	4003d8	449	36

tabla B.7: Distance Prefetch 7

Kernel	dirección	Numero de repeticiones	distance prefetch
	4003e0	449	36
	400400	449	36
	400408	449	36
	400428	449	36
	400438	449	36
	400450	449	36
	4004a0	9	1752
liver24	4002a8	9	117
	4002b8	89	12
	4002c8	89	12
	4002e0	89	12

Apéndice C

Programas

C.1. Sim-cache

A continuación se muestra el código fuente del simulador sim-cache que se modifico para introducir la UGD

```
/*
 * sim-cache.c - sample cache simulator implementation
 *
 * This file is a part of the SimpleScalar tool suite written by
 * Todd M. Austin as a part of the Multiscalar Research Project.
 *
 * The tool suite is currently maintained by Doug Burger and Todd M. Austin.
 *
 * Copyright (C) 1994, 1995, 1996, 1997 by Todd M. Austin
 *
 * This source file is distributed "as is" in the hope that it will be
 * useful.  The tool set comes with no warranty, and no author or
 * distributor accepts any responsibility for the consequences of its
 * use.
 *
 * Everyone is granted permission to copy, modify and redistribute
 * this tool set under the following conditions:
 *
 *    This source code is distributed for non-commercial use only.
 *    Please contact the maintainer for restrictions applying to
 *    commercial use.
 *
 *    Permission is granted to anyone to make or distribute copies
 *    of this source code, either as received or modified, in any
 *    medium, provided that all copyright notices, permission and
```

```
 *    nonwarranty notices are preserved, and that the distributor
 *    grants the recipient permission for further redistribution as


 *    permitted by this document.
 *
 *    Permission is granted to distribute this file in compiled
 *    or executable form under the same conditions that apply for
 *    source code, provided that either:
 *
 *    A. it is accompanied by the corresponding machine-readable
 *       source code,
 *    B. it is accompanied by a written offer, with no time limit,
 *       to give anyone a machine-readable copy of the corresponding
 *       source code in return for reimbursement of the cost of
 *       distribution.  This written offer must permit verbatim
 *       duplication by anyone, or
 *    C. it is distributed by someone who received only the
 *       executable form, and is accompanied by a copy of the
 *       written offer of source code that they received concurrently.
 *
 * In other words, you are welcome to use, share and improve this
 * source file.  You are forbidden to forbid anyone else to use, share
 * and improve what you give them.
 *
 * INTERNET: dburger@cs.wisc.edu
 * US Mail:  1210 W. Dayton Street, Madison, WI 53706
 *
 * $Id: sim-cache.c,v 1.6 1997/04/16 22:09:45 taustin Exp taustin $
 *
 * $Log: sim-cache.c,v $
 * Revision 1.6  1997/04/16  22:09:45  taustin
 * fixed "bad l2 D-cache parms" fatal string
 *
 * Revision 1.5  1997/03/11  01:27:08  taustin
 * updated copyright
 * '-pcstat' option support added
 * long/int tweaks made for ALPHA target support
 * better defaults defined for caches/TLBs
 * "mstate" command supported added for DLite!
```

```
 * supported added for non-GNU C compilers
 *
 * Revision 1.4  1997/01/06  16:03:07  taustin
 * comments updated


 * supported added for 2-level cache memory system
 * instruction TLB supported added
 * -icompress now compresses 64-bit instruction addresses to 32-bit equiv
 * main loop simplified
 *
 * Revision 1.3  1996/12/30  17:14:11  taustin
 * updated to support options and stats packages
 *
 * Revision 1.1  1996/12/05  18:52:32  taustin
 * Initial revision
 *
 *
 */

#include <stdio.h> #include <stdlib.h> #include <string.h>
#include <math.h> #include <assert.h>

#include "misc.h" #include "ss.h" #include "regs.h" #include
"memory.h" #include "cache.h" #include "loader.h" #include
"syscall.h" #include "dlite.h" #include "sim.h"

/*
 * This file implements a functional cache simulator.  Cache statistics are
 * generated for a user-selected cache and TLB configuration, which may include
 * up to two levels of instruction and data cache (with any levels unified),
 * and one level of instruction and data TLBs.  No timing information is
 * generated (hence the distinction, "functional" simulator).
 */


/* AA */

static unsigned int cuenta = 0;
```

```
/* CJRM */

void init_table();

/* track number of insn and refs */ static SS_COUNTER_TYPE
sim_num_insn = 0; static SS_COUNTER_TYPE sim_num_refs = 0;


/* level 1 instruction cache, entry level instruction cache */
static struct cache *cache_il1 = NULL;

/* level 1 instruction cache */ static struct cache *cache_il2 =
NULL;

/* level 1 data cache, entry level data cache */ static struct
cache *cache_dl1 = NULL;

/* level 2 data cache */ static struct cache *cache_dl2 = NULL;

/* instruction TLB */ static struct cache *itlb = NULL;

/* data TLB */ static struct cache *dtlb = NULL;

/* CJRM Tesis Prefetching */

struct regi {
              int ban1;          /* se crea el registro donde se almacenan */
              int pc_n;          /* los datos para cada prediccion */
              int addr_n;
              int stride_n;
              int state;
              int hits;
              int ciclo_n;
              int remp_n;
            };

struct regi tabla[1000];

#define tam 128
```

```
static unsigned pref_count=0; static unsigned pref_miss_count=0;
static unsigned hits=0;


/* text-based stat profiles */ #define MAX_PCSTAT_VARS 8 static
struct stat_stat_t *pcstat_stats[MAX_PCSTAT_VARS]; static
SS_COUNTER_TYPE pcstat_lastvals[MAX_PCSTAT_VARS]; static struct
stat_stat_t *pcstat_sdists[MAX_PCSTAT_VARS];

/* wedge all stat values into a SS_COUNTER_TYPE */ #define
STATVAL(STAT)                                \
  ((STAT)->sc == sc_int                      \
   ? (SS_COUNTER_TYPE)*((STAT)->variant.for_int.var)          \
   : ((STAT)->sc == sc_uint                   \
      ? (SS_COUNTER_TYPE)*((STAT)->variant.for_uint.var)      \
      : ((STAT)->sc == sc_counter             \
     ? *((STAT)->variant.for_counter.var)           \
     : (panic("bad stat class"), 0))))

/* l1 data cache l1 block miss handler function */ static unsigned
int         /* latency of block access */ dl1_access_fn(enum
mem_cmd cmd,     /* access cmd, Read or Write */
          SS_ADDR_TYPE baddr,   /* block address to access */
          int bsize,        /* size of block to access */
          struct cache_blk *blk,    /* ptr to block in upper level */
          SS_TIME_TYPE now)     /* time of access */
{
  if (cache_dl2)
    {
      /* access next level of data cache hierarchy */
      return cache_access(cache_dl2, cmd, baddr, NULL, bsize,
              /* now */now, /* pudata */NULL, /* repl addr */NULL);
    }
  else
    {
      /* access main memory, which is always done in the main simulator loop */


      return /* access latency, ignored */1;
    }
```

```
}

/* l2 data cache block miss handler function */ static unsigned
int         /* latency of block access */ dl2_access_fn(enum
mem_cmd cmd,      /* access cmd, Read or Write */
          SS_ADDR_TYPE baddr,    /* block address to access */
          int bsize,         /* size of block to access */
          struct cache_blk *blk,     /* ptr to block in upper level */
          SS_TIME_TYPE now)      /* time of access */
{


  /* this is a miss to the lowest level, so access main memory, which is
     always done in the main simulator loop */
  return /* access latency, ignored */1;
}

/* l1 inst cache l1 block miss handler function */ static unsigned
int         /* latency of block access */ il1_access_fn(enum
mem_cmd cmd,      /* access cmd, Read or Write */
          SS_ADDR_TYPE baddr,    /* block address to access */
          int bsize,         /* size of block to access */
          struct cache_blk *blk,     /* ptr to block in upper level */
          SS_TIME_TYPE now)      /* time of access */
{
  if (cache_il2)
    {
      /* access next level of inst cache hierarchy */
      return cache_access(cache_il2, cmd, baddr, NULL, bsize,
              /* now */now, /* pudata */NULL, /* repl addr */NULL);
    }
  else
    {
      /* access main memory, which is always done in the main simulator loop */
      return /* access latency, ignored */1;
    }
}

/* l2 inst cache block miss handler function */ static unsigned
int         /* latency of block access */ il2_access_fn(enum
```

```
mem_cmd cmd,     /* access cmd, Read or Write */
          SS_ADDR_TYPE baddr,   /* block address to access */
          int bsize,        /* size of block to access */
          struct cache_blk *blk,    /* ptr to block in upper level */
          SS_TIME_TYPE now)     /* time of access */
{
  /* this is a miss to the lowest level, so access main memory, which is
     always done in the main simulator loop */
  return /* access latency, ignored */1;
}

/* inst cache block miss handler function */ static unsigned int


/* latency of block access */ itlb_access_fn(enum mem_cmd cmd, /*
access cmd, Read or Write */
           SS_ADDR_TYPE baddr,  /* block address to access */
           int bsize,       /* size of block to access */
           struct cache_blk *blk,   /* ptr to block in upper level */
           SS_TIME_TYPE now)    /* time of access */
{
  SS_ADDR_TYPE *phy_page_ptr = (SS_ADDR_TYPE *)blk->user_data;

  /* no real memory access, however, should have user data space attached */
  assert(phy_page_ptr);

  /* fake translation, for now... */
  *phy_page_ptr = 0;

  return /* access latency, ignored */1;
}

/* data cache block miss handler function */ static unsigned int
/* latency of block access */ dtlb_access_fn(enum mem_cmd cmd,
/* access cmd, Read or Write */
           SS_ADDR_TYPE baddr,  /* block address to access */
           int bsize,       /* size of block to access */
           struct cache_blk *blk,   /* ptr to block in upper level */
           SS_TIME_TYPE now)    /* time of access */
{
```

```
  SS_ADDR_TYPE *phy_page_ptr = (SS_ADDR_TYPE *)blk->user_data;

  /* no real memory access, however, should have user data space attached */
  assert(phy_page_ptr);

  /* fake translation, for now... */
  *phy_page_ptr = 0;

  return /* access latency, ignored */1;
}

/* cache/TLB options */ static char *cache_dl1_opt /* = "none" */;



static char *cache_dl2_opt /* = "none" */; static char
*cache_il1_opt /* = "none" */; static char *cache_il2_opt /* =
"none" */; static char *itlb_opt /* = "none" */; static char
*dtlb_opt /* = "none" */; static int flush_on_syscalls /* = FALSE
*/; static int compress_icache_addrs /* = FALSE */;

/* text-based stat profiles */ static int pcstat_nelt = 0; static
char *pcstat_vars[MAX_PCSTAT_VARS];

/* convert 64-bit inst text addresses to 32-bit inst equivalents
*/ #define IACOMPRESS(A)                                \
  (compress_icache_addrs ? ((((A) - SS_TEXT_BASE) >> 1) + SS_TEXT_BASE) : (A))
#define ISCOMPRESS(SZ)                              \
  (compress_icache_addrs ? ((SZ) >> 1) : (SZ))

/* register simulator-specific options */ void
sim_reg_options(struct opt_odb_t *odb)  /* options database */ {
  opt_reg_header(odb,
"sim-cache: This simulator implements a functional cache
simulator.  Cache\n" "statistics are generated for a user-selected
cache and TLB configuration,\n" "which may include up to two
levels of instruction and data cache (with any\n" "levels
unified), and one level of instruction and data TLBs.  No
timing\n" "information is generated.\n"
```

```
        );
  opt_reg_string(odb, "-cache:dl1",
        "l1 data cache config, i.e., {<config>|none}",
        &cache_dl1_opt, "dl1:256:32:1:l", /* print */TRUE, NULL);
  opt_reg_note(odb,
"  The cache config parameter <config> has the following
format:\n" "\n" "    <name>:<nsets>:<bsize>:<assoc>:<repl>\n" "\n"
"    <name>   - name of the cache being defined\n" "    <nsets>  -
number of sets in the cache\n" "    <bsize>  - block size of the
cache\n" "    <assoc>  - associativity of the cache\n" "    <repl>
- block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random\n"
"\n" "    Examples:   -cache:dl1 dl1:4096:32:1:l\n" "
-dtlb dtlb:128:4096:32:r\n"
          );
  opt_reg_string(odb, "-cache:dl2",
        "l2 data cache config, i.e., {<config>|none}",


        &cache_dl2_opt, "ul2:1024:64:4:l", /* print */TRUE, NULL);
  opt_reg_string(odb, "-cache:il1",
        "l1 inst cache config, i.e., {<config>|dl1|dl2|none}",
        &cache_il1_opt, "il1:256:32:1:l", /* print */TRUE, NULL);
  opt_reg_note(odb,
"  Cache levels can be unified by pointing a level of the
instruction cache\n" "  hierarchy at the data cache hiearchy using
the \"dl1\" and \"dl2\" cache\n" "  configuration arguments.  Most
sensible combinations are supported, e.g.,\n" "\n" "    A unified
l2 cache (il2 is pointed at dl2):\n" "      -cache:il1
il1:128:64:1:l -cache:il2 dl2\n" "      -cache:dl1 dl1:256:32:1:l
-cache:dl2 ul2:1024:64:2:l\n" "\n" "    Or, a fully unified cache
hierarchy (il1 pointed at dl1):\n" "      -cache:il1 dl1\n" "
-cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l\n"
          );
  opt_reg_string(odb, "-cache:il2",
        "l2 instruction cache config, i.e., {<config>|dl2|none}",
        &cache_il2_opt, "dl2", /* print */TRUE, NULL);
  opt_reg_string(odb, "-tlb:itlb",
        "instruction TLB config, i.e., {<config>|none}",
        &itlb_opt, "itlb:16:4096:4:l", /* print */TRUE, NULL);
  opt_reg_string(odb, "-tlb:dtlb",
```

```
          "data TLB config, i.e., {<config>|none}",
          &dtlb_opt, "dtlb:32:4096:4:l", /* print */TRUE, NULL);
  opt_reg_flag(odb, "-flush", "flush caches on system calls",
           &flush_on_syscalls, /* default */FALSE, /* print */TRUE, NULL);
  opt_reg_flag(odb, "-icompress",
           "convert 64-bit inst addresses to 32-bit inst equivalents",
           &compress_icache_addrs, /* default */FALSE,
           /* print */TRUE, NULL);

  opt_reg_string_list(odb, "-pcstat",
              "profile stat(s) against text addr's (mult uses ok)",
              pcstat_vars, MAX_PCSTAT_VARS, &pcstat_nelt, NULL,
              /* !print */FALSE, /* format */NULL, /* accrue */TRUE);

}

/* check simulator-specific option values */ void
sim_check_options(struct opt_odb_t *odb,    /* options database */


          int argc, char **argv)    /* command line arguments */
{
  char name[128], c;
  int nsets, bsize, assoc;

  /* use a level 1 D-cache? */
  if (!mystricmp(cache_dl1_opt, "none"))
    {
      cache_dl1 = NULL;

      /* the level 2 D-cache cannot be defined */


      if (strcmp(cache_dl2_opt, "none"))
    fatal("the l1 data cache must defined if the l2 cache is defined");
      cache_dl2 = NULL;
    }
  else /* dl1 is defined */
    {
      if (sscanf(cache_dl1_opt, "%[^:]:%d:%d:%d:%c",
```

```
        name, &nsets, &bsize, &assoc, &c) != 5)
    fatal("bad l1 D-cache parms: <name>:<nsets>:<bsize>:<assoc>:<repl>");
      cache_dl1 = cache_create(name, nsets, bsize, /* balloc */FALSE,
                  /* usize */0, assoc, cache_char2policy(c),


                  dl1_access_fn, /* hit latency */1);
      /* is the level 2 D-cache defined? */
      if (!mystricmp(cache_dl2_opt, "none"))
    cache_dl2 = NULL;
      else
    {
      if (sscanf(cache_dl2_opt, "%[^:]:%d:%d:%d:%c",
             name, &nsets, &bsize, &assoc, &c) != 5)
        fatal("bad l2 D-cache parms: "
          "<name>:<nsets>:<bsize>:<assoc>:<repl>");
      cache_dl2 = cache_create(name, nsets, bsize, /* balloc */FALSE,
                  /* usize */0, assoc, cache_char2policy(c),
                  dl2_access_fn, /* hit latency */1);
    }


    }

  /* use a level 1 I-cache? */
  if (!mystricmp(cache_il1_opt, "none"))
    {
      cache_il1 = NULL;

      /* the level 2 I-cache cannot be defined */
      if (strcmp(cache_il2_opt, "none"))
    fatal("the l1 inst cache must defined if the l2 cache is defined");
      cache_il2 = NULL;
    }
  else if (!mystricmp(cache_il1_opt, "dl1"))
    {
      if (!cache_dl1)
    fatal("I-cache l1 cannot access D-cache l1 as it's undefined");
      cache_il1 = cache_dl1;
```

```
    /* the level 2 I-cache cannot be defined */
    if (strcmp(cache_il2_opt, "none"))
  fatal("the l1 inst cache must defined if the l2 cache is defined");
    cache_il2 = NULL;
  }
else if (!mystricmp(cache_il1_opt, "dl2"))
  {
    if (!cache_dl2)
  fatal("I-cache l1 cannot access D-cache l2 as it's undefined");
    cache_il1 = cache_dl2;

    /* the level 2 I-cache cannot be defined */
    if (strcmp(cache_il2_opt, "none"))
  fatal("the l1 inst cache must defined if the l2 cache is defined");
    cache_il2 = NULL;
  }
else /* il1 is defined */
  {
    if (sscanf(cache_il1_opt, "%[^:]:%d:%d:%d:%c",
       name, &nsets, &bsize, &assoc, &c) != 5)
  fatal("bad l1 I-cache parms: <name>:<nsets>:<bsize>:<assoc>:<repl>");
    cache_il1 = cache_create(name, nsets, bsize, /* balloc */FALSE,


                /* usize */0, assoc, cache_char2policy(c),
                il1_access_fn, /* hit latency */1);

    /* is the level 2 D-cache defined? */
    if (!mystricmp(cache_il2_opt, "none"))
  cache_il2 = NULL;
    else if (!mystricmp(cache_il2_opt, "dl2"))
  {
    if (!cache_dl2)
      fatal("I-cache l2 cannot access D-cache l2 as it's undefined");
    cache_il2 = cache_dl2;
  }
    else
  {
    if (sscanf(cache_il2_opt, "%[^:]:%d:%d:%d:%c",
          name, &nsets, &bsize, &assoc, &c) != 5)
```

```
        fatal("bad l2 I-cache parms: "
          "<name>:<nsets>:<bsize>:<assoc>:<repl>");
      cache_il2 = cache_create(name, nsets, bsize, /* balloc */FALSE,
                  /* usize */0, assoc, cache_char2policy(c),
                  il2_access_fn, /* hit latency */1);
    }
    }
  /* use an I-TLB? */
  if (!mystricmp(itlb_opt, "none"))
    itlb = NULL;
  else
    {
      if (sscanf(itlb_opt, "%[^:]:%d:%d:%d:%c",
         name, &nsets, &bsize, &assoc, &c) != 5)
    fatal("bad TLB parms: <name>:<nsets>:<page_size>:<assoc>:<repl>");
      itlb = cache_create(name, nsets, bsize, /* balloc */FALSE,
              /* usize */sizeof(SS_ADDR_TYPE), assoc,
              cache_char2policy(c), itlb_access_fn,
              /* hit latency */1);
    }
  /* use a D-TLB? */
  if (!mystricmp(dtlb_opt, "none"))
    dtlb = NULL;
  else


    {
      if (sscanf(dtlb_opt, "%[^:]:%d:%d:%d:%c",
         name, &nsets, &bsize, &assoc, &c) != 5)
    fatal("bad TLB parms: <name>:<nsets>:<page_size>:<assoc>:<repl>");
      dtlb = cache_create(name, nsets, bsize, /* balloc */FALSE,
              /* usize */sizeof(SS_ADDR_TYPE), assoc,
              cache_char2policy(c), dtlb_access_fn,
              /* hit latency */1);
    }
}

/* print simulator-specific configuration information */ void
sim_aux_config(FILE *stream)        /* output stream */ {
  /* nada */
```

```
}

/* register simulator-specific statistics */ void
sim_reg_stats(struct stat_sdb_t *sdb)   /* stats database */ {
  int i;

  /* register baseline stats */
  stat_reg_counter(sdb, "sim_num_insn",
           "total number of instructions executed",
           &sim_num_insn, 0, NULL);
  stat_reg_counter(sdb, "sim_num_refs",
           "total number of loads and stores executed",
           &sim_num_refs, 0, NULL);
  stat_reg_int(sdb, "sim_elapsed_time",
           "total simulation time in seconds",
           (int *)&sim_elapsed_time, 0, NULL);
  stat_reg_formula(sdb, "sim_inst_rate",
           "simulation speed (in insts/sec)",
           "sim_num_insn / sim_elapsed_time", NULL);

  /* register cache stats */
  if (cache_il1
      && (cache_il1 != cache_dl1 && cache_il1 != cache_dl2))
    cache_reg_stats(cache_il1, sdb);
  if (cache_il2
      && (cache_il2 != cache_dl1 && cache_il2 != cache_dl2))


    cache_reg_stats(cache_il2, sdb);
  if (cache_dl1)
    cache_reg_stats(cache_dl1, sdb);
  if (cache_dl2)
    cache_reg_stats(cache_dl2, sdb);
  if (itlb)
    cache_reg_stats(itlb, sdb);
  if (dtlb)
    cache_reg_stats(dtlb, sdb);

  for (i=0; i<pcstat_nelt; i++)
    {
```

```
      char buf[512], buf1[512];
      struct stat_stat_t *stat;

      /* track the named statistical variable by text address */

      /* find it... */
      stat = stat_find_stat(sdb, pcstat_vars[i]);
      if (!stat)
    fatal("cannot locate any statistic named `%s'", pcstat_vars[i]);

      /* stat must be an integral type */
      if (stat->sc != sc_int && stat->sc != sc_uint && stat->sc != sc_counter)
    fatal("`-pcstat' statistical variable `%s' is not an integral type",
          stat->name);

      /* register this stat */
      pcstat_stats[i] = stat;
      pcstat_lastvals[i] = STATVAL(stat);

      /* declare the sparce text distribution */
      sprintf(buf, "%s_by_pc", stat->name);
      sprintf(buf1, "%s (by text address)", stat->desc);
      pcstat_sdists[i] = stat_reg_sdist(sdb, buf, buf1,
                    /* initial value */0,
                    /* print fmt */(PF_COUNT|PF_PDF),
                    /* format */"0x%lx %lu %.2f",
                    /* print fn */NULL);
    }


}

/* local machine state accessor */ static char *
/* err str, NULL for no err */ cache_mstate_obj(FILE *stream,
/* output stream */
        char *cmd)          /* optional command string */
{
  /* just dump intermediate stats */
  sim_print_stats(stream);
```

```
  /* no error */
  return NULL;
}

/* initialize the simulator */ void sim_init(void) {
  SS_INST_TYPE inst;

  sim_num_insn = 0;
  sim_num_refs = 0;

  regs_PC = ld_prog_entry;

  /* decode all instructions */
  {
    SS_ADDR_TYPE addr;

    if (OP_MAX > 255)
      fatal("cannot do fast decoding, too many opcodes");

    debug("sim: decoding text segment...");
    for (addr=ld_text_base;
     addr < (ld_text_base+ld_text_size);
     addr += SS_INST_SIZE)
      {
    inst = __UNCHK_MEM_ACCESS(SS_INST_TYPE, addr);
    inst.a = (inst.a & ~0xff) | (unsigned int)SS_OP_ENUM(SS_OPCODE(inst));
    __UNCHK_MEM_ACCESS(SS_INST_TYPE, addr) = inst;
      }
  }



  /* initialize the DLite debugger (NOTE: mem is always precise) */
  dlite_init(dlite_reg_obj, dlite_mem_obj, cache_mstate_obj);
}

/* dump simulator-specific auxiliary simulator statistics */ void
sim_aux_stats(FILE *stream)     /* output stream */ {
  /* nada */
}
```

```
/* un-initialize the simulator */ void sim_uninit(void) {
  /* nada */
}

/*
 * configure the execution engine
 */

/*
 * precise architected register accessors
 */

/* next program counter */ #define SET_NPC(EXPR)       (next_PC =
(EXPR))

/* current program counter */ #define CPC         (regs_PC)

/* general purpose registers */ #define GPR(N)
(regs_R[N]) #define SET_GPR(N,EXPR)     (regs_R[N] = (EXPR))

/* floating point registers, L->word, F->single-prec,
D->double-prec */ #define FPR_L(N)        (regs_F.l[(N)]) #define
SET_FPR_L(N,EXPR)   (regs_F.l[(N)] = (EXPR)) #define FPR_F(N)
(regs_F.f[(N)]) #define SET_FPR_F(N,EXPR)   (regs_F.f[(N)] =
(EXPR)) #define FPR_D(N)        (regs_F.d[(N) >> 1]) #define
SET_FPR_D(N,EXPR)   (regs_F.d[(N) >> 1] = (EXPR))

/* miscellaneous register accessors */ #define SET_HI(EXPR)
(regs_HI = (EXPR)) #define HI          (regs_HI) #define
SET_LO(EXPR)        (regs_LO = (EXPR)) #define LO
(regs_LO) #define FCC         (regs_FCC) #define SET_FCC(EXPR)


(regs_FCC = (EXPR))

/* precise architected memory state help functions */ #define
__READ_CACHE(addr, SRC_T)                    \
  ((dtlb                                  \
    ? mcache_access(dtlb, Read, (addr), NULL, sizeof(SRC_T), 0, NULL, NULL)\
```

```
    : 0),                                   \
   (cache_dl1                               \
    ? cache_access(cache_dl1, Read, (addr), NULL, sizeof(SRC_T), 0, NULL, NULL)\
    : 0))

#define __READ_WORD(DST_T, SRC_T, SRC)                  \
  (addr = (SRC),                                \
   __READ_CACHE(addr, SRC_T),                           \
   ((unsigned int)((DST_T)(SRC_T)MEM_READ_WORD(addr))))

#define __READ_HALF(DST_T, SRC_T, SRC)                  \
  (addr = (SRC),                                \
   __READ_CACHE(addr, SRC_T),                           \
   (unsigned int)((DST_T)(SRC_T)MEM_READ_HALF(addr)))

#define __READ_BYTE(DST_T, SRC_T, SRC)                  \
  (addr = (SRC),                                \
   __READ_CACHE(addr, SRC_T),                           \
   (unsigned int)((DST_T)(SRC_T)MEM_READ_BYTE(addr)))

/* precise architected memory state accessor macros */ #define
READ_WORD(SRC)                          \
  __READ_WORD(unsigned int, unsigned int, (SRC))

#define READ_UNSIGNED_HALF(SRC)                         \
  __READ_HALF(unsigned int, unsigned short, (SRC))

#define READ_SIGNED_HALF(SRC)                           \
  __READ_HALF(signed int, signed short, (SRC))

#define READ_UNSIGNED_BYTE(SRC)                         \
  __READ_BYTE(unsigned int, unsigned char, (SRC))

#define READ_SIGNED_BYTE(SRC)                           \


  __READ_BYTE(signed int, signed char, (SRC))

/* precise architected memory state help functions */
```

```
#define __WRITE_CACHE(addr, DST_T)                    \
  ((dtlb                                  \
    ? cache_access(dtlb, Write, (addr), NULL, sizeof(DST_T), 0, NULL, NULL)\
    : 0),                                 \
   (cache_dl1                                 \
    ? mcache_access(cache_dl1, Write, (addr), NULL, sizeof(DST_T),  \
          0, NULL, NULL)                                    \
    : 0))

#define WRITE_WORD(SRC, DST)                          \
  (addr = (DST),                              \
   __WRITE_CACHE(addr, unsigned int),                   \
   MEM_WRITE_WORD(addr, (unsigned int)(SRC)))

#define WRITE_HALF(SRC, DST)                          \
  (addr = (DST),                              \
   __WRITE_CACHE(addr, unsigned short),                 \
   MEM_WRITE_HALF(addr, (unsigned short)(unsigned int)(SRC)))

#define WRITE_BYTE(SRC, DST)                          \
  (addr = (DST),                              \
   __WRITE_CACHE(addr, unsigned char),                  \
   MEM_WRITE_BYTE(addr, (unsigned char)(unsigned int)(SRC)))

/* system call memory access function */ void
dcache_access_fn(enum mem_cmd cmd,  /* memory access cmd, Read or
Write */
         SS_ADDR_TYPE addr, /* data address to access */
         void *p,       /* data input/output buffer */
         int nbytes)        /* number of bytes to access */
{
  if (dtlb)
    cache_access(dtlb, cmd, addr, NULL, nbytes, 0, NULL, NULL);
  if (cache_dl1)
 {
    mcache_access(cache_dl1, cmd, addr, NULL, nbytes, 0, NULL, NULL);


   /* fprintf(stderr,"A %x %x\n",regs_PC,addr); */
 }
```

```
  mem_access(cmd, addr, p, nbytes);

}

/* system call handler macro */ #define SYSCALL(INST)
\
  (flush_on_syscalls                              \
   ? ((dtlb ? cache_flush(dtlb, 0) : 0),                \
      (cache_dl1 ? cache_flush(cache_dl1, 0) : 0),          \
      (cache_dl2 ? cache_flush(cache_dl2, 0) : 0),          \
      ss_syscall(mem_access, INST))                  \
   : (ss_syscall(dcache_access_fn, INST)))

/* instantiate the helper functions in the '.def' file */ #define
DEFICLASS(ICLASS,DESC) #define
DEFINST(OP,MSK,NAME,OPFORM,RES,CLASS,O1,O2,I1,I2,I3,EXPR) #define
DEFLINK(OP,MSK,NAME,MASK,SHIFT) #define CONNECT(OP) #define IMPL
#include "ss.def" #undef DEFICLASS #undef DEFINST #undef DEFLINK
#undef CONNECT #undef IMPL

/* start simulation, program loaded, processor precise state
initialized */ void sim_main(void) {
  int i;
  SS_INST_TYPE inst;
  register SS_ADDR_TYPE next_PC;
  register SS_ADDR_TYPE addr;
  enum ss_opcode op;
  register int is_write;

  fprintf(stderr, "sim: ** starting functional simulation w/ caches **\n");

  init_table();

  /* set up initial default next PC */
  next_PC = regs_PC + SS_INST_SIZE;

  /* check for DLite debugger entry condition */
  if (dlite_check_break(regs_PC, /* no access */0, /* addr */0, 0, 0))
```

```
    dlite_main(regs_PC - SS_INST_SIZE, regs_PC, sim_num_insn);

  while (TRUE)
    {

/*      fprintf(stderr,"0 %x\n",addr); */

      /* maintain $r0 semantics */
      regs_R[0] = 0;

 /*     fprintf(stderr,"1 %x\n",addr);   */

      /* keep an instruction count */
      sim_num_insn++;

/*      fprintf(stderr,"2 %x\n",addr);  */


      /* get the next instruction to execute */
/*      if (itlb)
    cache_access(itlb, Read, IACOMPRESS(regs_PC),
             NULL, ISCOMPRESS(SS_INST_SIZE), 0, NULL, NULL);

      if (cache_il1)
    cache_access(cache_il1, Read, IACOMPRESS(regs_PC),
             NULL, ISCOMPRESS(SS_INST_SIZE), 0, NULL, NULL); */

 /*     fprintf(stderr,"%x %x\n",regs_PC,addr);

      switch (regs_PC)
        {
         case 0x400238:
          fprintf(stderr,"%x %x\n",regs_PC,addr);
          break;

         case 0x400240:
          fprintf(stderr,"%x %x\n",regs_PC,addr);
          break;

         case 0x400248:
```

```
            fprintf(stderr,"%x %x\n",regs_PC,addr);
            break;

          case 0x400250:
            fprintf(stderr,"%x %x\n",regs_PC,addr);
            break;

          case 0x400258:
            fprintf(stderr,"%x %x\n",regs_PC,addr);
            break;

          case 0x400260:
            fprintf(stderr,"%x %x\n",regs_PC,addr);
            break;

          case 0x400268:
            fprintf(stderr,"%x %x\n",regs_PC,addr);
            break;

          case 0x400270:
            fprintf(stderr,"%x %x\n",regs_PC,addr);
            break;
        } */
      mem_access(Read, regs_PC, &inst, SS_INST_SIZE);

      /* set default reference address and access mode */
      addr = 0; is_write = FALSE;

/*      fprintf(stderr,"3 %x\n",addr); */

      /* decode the instruction */
      op = SS_OPCODE(inst);

/*      fprintf(stderr,"4 %x\n",addr);*/


      switch (op)
    {
```

```
#define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3,EXPR)
\


    case OP:                                                          \
          EXPR;                                                           \
          break;
#define DEFLINK(OP,MSK,NAME,MASK,SHIFT)
\
        case OP:                                                          \
        panic("attempted to execute a linking opcode");
#define CONNECT(OP) #include "ss.def" #undef DEFINST #undef
DEFLINK #undef CONNECT
    default:
          panic("attempted to execute a bogus opcode");
    }

      if (SS_OP_FLAGS(op) & F_MEM)
    {
      sim_num_refs++;
      if (SS_OP_FLAGS(op) & F_STORE)
        is_write = TRUE;
    }


      /* update any stats tracked by PC */
      for (i=0; i<pcstat_nelt; i++)
    {
      SS_COUNTER_TYPE newval;
      int delta;

      /* check if any tracked stats changed */
      newval = STATVAL(pcstat_stats[i]);
      delta = newval - pcstat_lastvals[i];
      if (delta != 0)
        {
          stat_add_samples(pcstat_sdists[i], regs_PC, delta);
          pcstat_lastvals[i] = newval;
        }
```

```
    }

      /* check for DLite debugger entry condition */
      if (dlite_check_break(next_PC,


                is_write ? ACCESS_WRITE : ACCESS_READ,
                addr, sim_num_insn, sim_num_insn))
    dlite_main(regs_PC, next_PC, sim_num_insn);

      /* go to the next instruction */
      regs_PC = next_PC;
      next_PC += SS_INST_SIZE;
    }

/*     fprintf (stderr,"n\n\n%6s %7s %8s %8s %6s %6s %4s\n","renglon",
       "PCount","address","stride","ciclo","state","hits");
     for (ind=0; ind <= tam; ind++)
       {
        fprintf (stderr,"%4d %10x %9x %5x %8d %5d %4d\n",ind,tabla[ind]
        .pc_n,tabla[ind].addr_n,tabla[ind].stride_n,tabla[ind].ciclo,
        tabla[ind].state,tabla[ind].hits);
       } */

} unsigned int
      mcache_access(struct cache *cp,   /* cache to access */
         enum mem_cmd cmd,      /* access type, Read or Write */
         SS_ADDR_TYPE addr,     /* address of access */
         void *vp,          /* ptr to buffer for input/output */
         int nbytes,        /* number of bytes to access */
         SS_TIME_TYPE now,      /* time of access */
         char **udata,      /* for return of user data ptr */
         SS_ADDR_TYPE *repl_addr)   /* for address of replaced block */
    {

    int tipo1,tipo2,j=0;
    int n=0,ind=0,cont,remp=0;
    int pre=0;
    int addr_pre,temp,next_PC;
```

```
cache_access(cp, cmd, addr, NULL, nbytes, 0, NULL, NULL);

next_PC=regs_PC+SS_INST_SIZE;

  if (addr != 0)   /* detecta una instr */
     {
      for(cont=0; cont<=tam; cont++)
       {
         switch (tabla[cont].ban1)
          {
            case 1:
             tabla[cont].ciclo_n++;
             break;
          }
        }

      while (j <= tam)                /* load y dir del dato */
        {
         if (tabla[j].ban1 == 0 && n <= tam)
          {
           tabla[j].pc_n=next_PC;      /* se llenan los regs */
           tabla[j].addr_n=addr;    /* para realizar las */
           tabla[j].ban1=1;           /* prefetchings */
           tabla[j].stride_n=0;
           tabla[j].state=0;
           n++;
           j=0;
           break;
          }
         if (tabla[j].pc_n == next_PC) /* comparacion de PC */
          {
        /*   fprintf(stderr,"\n%8x %8x\n",tabla[j].pc_n,direX1); */
           switch (tabla[j].state) /* el registro esta en estado */
            {
              case 0:             /* inicial, se pasa a estado */
               tabla[j].state=1; /* transitorio o inestable */
               break;
                                /* el registro esta en estado */
              case 1:           /* transitorio, se va a pasar a */
               tabla[j].state=2;   /* estable */
```

```
        pre=1;
        break;

      case 2:
        pre=1;
        break;
    }
  tabla[j].stride_n=addr-tabla[j].addr_n; /* gen el des */
  tabla[j].addr_n=addr;
  if (pre==1)
    {
     addr_pre=tabla[j].addr_n+tabla[j].stride_n;
     pref_count++;
     /*  fprintf(stderr,"%d\n",pref_count); */
     if(cache_probe(cp,addr_pre)==FALSE)
       {
         pref_miss_count++;
         fprintf(stderr,"pre %d\n",pref_miss_count);
         temp=addr;
         addr=addr_pre;
         cache_access(cp, cmd, addr, NULL, nbytes, 0, NULL, NULL);
         addr=temp;
       }
     else
       {
         hits++;
         fprintf(stderr,"%d\n",hits);
       }
    }
  tabla[j].ciclo_n=0;
  tabla[j].hits++;
  j=0;
  break;
 }
if (j >= tam) /* empieza secuencia de reemplazo */
  {
   remp=0;
   for (ind=0; ind <= tam; ind++)
    {
  if (tabla[ind].state == 0 && tabla[ind].ciclo_n >= tam)
```

```
                    {
                   /*  fprintf(stderr,"\nlinea reemplazada"); */
                   /*  fprintf(stderr,"\n%2d %2d %6x %8x %3x %3d",
                   j,ind,tabla[ind].pc_n,tabla[ind].addr_n,tabla[ind]
                   .stride_n,tabla[ind].state); */ /*los de tag 0*/
                     /*fprintf(stderr,"\n%8x %8x",direX1,direX2);*/
                     tabla[ind].pc_n=next_PC;   /*no se han usado */
                     tabla[ind].addr_n=addr; /*en una comp*/
                     tabla[ind].stride_n=0;
                     tabla[ind].state=0;
                     tabla[ind].ciclo_n=0;
                     remp=1;
                     break;
                    }
                 }


                if (remp==0) /*no hubo reg con eti de 0*/
                 {
                   for (ind=0; ind <= tam; ind++)
                    {
                     if (tabla[ind].ciclo_n>= tam && tabla[ind].state== 1)
                        {
                         tabla[ind].pc_n = next_PC;
                         tabla[ind].addr_n = addr;
                         tabla[ind].stride_n = 0;
                         tabla[ind].state = 0;
                         tabla[ind].ciclo_n=0;
                         break;
                        }
                    }
                 }
                j=0;
                break;
               }
              j++;
             }
            }

/* if(cache_probe( cp,addr) == FALSE)   */
```

```
  /*      cuenta++;    */

  /*  cache_access(cp, cmd, addr, NULL, nbytes, 0, NULL, NULL);  */



    }


void init_table()
   {
    int j;

    for (j=0; j<=tam; j++)
      {
        tabla[j].pc_n=0;
        tabla[j].addr_n=0;


        tabla[j].stride_n=0;
        tabla[j].hits=0;
        tabla[j].ban1=0;
        tabla[j].ciclo_n=0;
      }
   }
```

Bibliografía

[1] T.-F. Chen and J.-L. Baer. *Effective Hardware-Based Data Prefetching for High Performace Processor*, ieee transactions on computer edition, 1995.

[2] John L. Hennessy and David Patterson. *Computer Architecture a Quantitative Approach*, 1996.

[3] C. Reid Harmon Jr., Bill Appelbe, and Raja Das. *IPU/LTB: A Method for Reducing Effective Memory Latency*, November 1997.

[4] Todd C. Mowry and Chi-Keun Luk. *Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling*, 1997. Pag. 314-320.

[5] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelik. *A case for intelligent RAM*, ieee micro edition, March/April 1997.

[6] James E. Smith. *Decoupled Access/Execute Computer Architectures.* University of Winsconsin and ACM Transactions on Computer Systems, November 1984. Vol. 2 No. 4, Pages 289-308.

[7] Steven P. Vander Wiel and David J. Lilja. *When Caches aren´t enough: Data prefetching Techniques.*, ieee computer edition, July 1997.

[8] www.intel.com. *Intel Architecture Software*, May 2000. Pag. 2-7.