Centro de Investigación y de Estudios Avanzados del I.P.N.
Unidad Zacatenco
Departamento de Ingeniería Eléctrica
Computación

Simulación paralela de un procesador superescalar en sistemas de memoria compartida y memoria distribuida

Alumna: Lorena Santos Espinosa

Director de Tesis: Dr. Arturo Díaz Pérez

México, Distrito Federal

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL I.P.N.

UNIDAD ZACATENCO
DEPARTAMENTO DE INGENIERIA ELECTRICA
SECCION DE COMPUTACION

Simulación paralela de un procesador superescalar en sistemas de memoria compartida y memoria distribuida[1]

TESIS QUE PRESENTA LA

Lic. Lorena Santos Espinosa

PARA OBTENER EL GRADO DE

Maestro en Ciencias

EN LA ESPECIALIDAD DE

Ingeniería Eléctrica

OPCION

Computación

DIRECTOR DE TESIS

Dr. Arturo Díaz Pérez

CIUDAD DE MEXICO, D.F. DICIEMBRE 2002

[1] Este trabajo fue parcialmente financiado mediante el proyecto $CONACyT$ 31892A : Algoritmos y arquitecturas de computadoras con dispositivos reconfigurables

Agradecimientos

Mi agradecimiento a cuantas personas han hecho posible la realización de esta tesis con cita especial al **Dr.Arturo Díaz Pérez** director del presente trabajo, por la confianza que depositó en mí cuando ingresé al Cinvestav, por su apoyo y amistad durante este período de mi vida y por el conocimiento adquirido gracias a él. Porque valió la pena.

Al **Dr. Luis Gerardo de la Fraga** por su participación como sinodal en este trabajo de tesis, por su amistad y por su apoyo en algún momento difícil durante mi estancia en el Cinvestav. Y al **Dr. Jorge Buenabad** por sus comentarios para mejorar este trabajo.

A mi hijo **Luis Alejandro** por su sacrificio y amor para lograr esta meta, por ser el principal motivo para continuar.

A **mi mamá, y a mis hermanos Laura, Luis Angel y Liza** por su apoyo incondicional para realizar este proyecto, por cuidar mi mayor tesoro, mi nené.

A **Joselito** y **Ulises** por su amistad incondicional.

A mis amigos, generación 2000, todos sabemos lo que es esto.

A mi madre

A Luis Alejandro

A mis hermanos

Resumen

El mejoramiento en el rendimiento de los procesadores se debe fundamentalmente a dos aspectos: cambios tecnológicos y cambios arquitecturales. Los cambios tecnológicos se deben fundamentalmente a los avances en la fabricación de semiconductores los cuáles permiten desarrollar dispositivos que cambian de estado rápidamente. Los cambios arquitecturales pretenden por un lado realizar las funciones de un procesador de manera diferente y por otro agregar a un procesador cada vez más funciones. Debido a los altos costos de construcción de un procesador, es necesario realizar una simulación antes de implantar un cambio arquitectural. La simulación mediante programas de prueba permite evaluar el rendimiento obtenido con los cambios propuestos. La simulación utiliza un modelo microarquitectural del procesador el cual simula cada una de sus unidades funcionales. En esta tesis se presenta una plataforma de experimentación para la ejecución de un conjunto de tareas de simulación de un procesador superescalar las cuales serán distribuidas dinámicamente entre un conjunto de computadoras. La estrategia de distribución pretende mantener a los computadoras con una carga de trabajo similar (balance de carga) y aplica un tipo de migración de procesos cuando el sistema no esta balanceado. Así también, dado que existen tareas de simulación muy costosas computacionalmente, se ha desarrollado un simulador paralelo el cual distribuye el trabajo de una sola tarea de simulación entre varios procesadores. El simulador paralelo asigna un proceso a cada etapa de la arquitectura segmentada (*pipeline*) del procesador desacoplándolas por cada ciclo de reloj y sincronizándolas mediante la estrategia de reloj global. El procesador utilizado es SimpleScalar el cual es un procesador superscalar diseñado por el Grupo de Arquitectura de Computadoras de la Universidad de Wisconsin.

Abstract

Improvements in computer architecture are mostly based on technological and architectural changes. Technological changes rely on advances in semiconductor fabrication which permit to built faster switching devices. Architectural improvements promote to realize the same functions but in different manner and to add new functions to a microprocessor. Due to high costs in processor fabrication, it is necessary to simulate an architectural change before implementing it. Simulation through benchmarks (test programs) allows to evaluate performance of proposed architectural changes. Simulation uses a processor's microarchitectural model which describes each of its functional units. In this thesis we present a tool to dynamically distribute different simulation tasks of a superscalar processor among a set of computers. Distribution strategy tries to keep computers with similar loads (load balance) and applies a kind of process migration when the system is not balanced. In addition, since some simulation tasks are computationally intensive, it has been developed a parallel simulator which distributes the simulation work of a single task among several processors. The parallel simulator maps each pipeline stage of the microprocessor to a single process isolating them for each clock cycle and achieving synchronization by a global clock. We used SimpleScalar which is a superscalar processor designed by the Computer Architecture's Group of University of Wisconsin.

Índice General

Índice de Figuras

Índice de Tablas

Introducción

La demanda de un mejor rendimiento en los procesadores requiere de computadoras cada vez más rápidas y con mayores capacidades. El mejoramiento en el rendimiento de los procesadores se debe fundamentalmente a dos aspectos: cambios tecnológicos y cambios arquitecturales. Los cambios tecnológicos se deben básicamente a los avances en la fabricación de semiconductores los cuáles permiten desarrollar dispositivos que cambian de estado más rápidamente. Los cambios arquitecturales pretenden por un lado realizar las funciones de un procesador de manera diferente y por otro agregar a un procesador cada vez más funciones.

Los procesadores modernos, conocidos como procesadores superescalares, son capaces de ejecutar varias instrucciones por cada ciclo de reloj, dividiendo la ejecución de cada instrucción en etapas. Además, realizan predicción de saltos, ejecutan de manera especulativa instrucciones ante la presencia de saltos condicionales, calendarizan la ejecución de instrucciones para eliminar dependencias de datos entre ellas y tienen varios niveles de memoria caché para acelerar el acceso a memoria, entre otras características [13].

Dados los altos costos de fabricación de un procesador, cualquier cambio arquitectural propuesto necesita ser simulado antes de tratar de implantarse. La etapa de simulación sirve para verificar, por un lado, el comportamiento correcto de un procesador y, por otro lado, si el diseño propuesto exhibe un rendimiento adecuado. La simulación puramente funcional permite verificar correctitud; sin embargo, para evaluar rendimiento se requiere un simulador microarquitectural. La simulación microarquitectural de un procesador requiere simular cada una de sus unidades de ejecución (*fetch*, decodificación, ALU, acceso a memoria, predicción de saltos, etc.).

Para evaluar el rendimiento de un procesador se eligen un conjunto de programas (*benchmarks*) los cuales tratan de poner a prueba las mejoras introducidas en un procesador. Entre los *benchmarks* más utilizados se encuentran los kernels, segmentos de código, programas sintéticos y conjuntos de programas reales [23]. De la simulación del procesador sobre programas de prueba se extraen algunas estadísticas de las cuales se obtienen medidas que evalúan el rendimiento del procesador ante el cambio arquitectural propuesto. Entre las medidas que se pueden utilizar están el número promedio de ciclos por instrucción, el número promedio de instrucciones ejecutadas en un ciclo, el

porcentaje de aciertos o fallas en la predicción de saltos y el porcentaje de aciertos o fallas en la memoria caché, entre otras.

Mediante la simulación secuencial convencional es posible simular pequeños programas de prueba en un tiempo razonable. Sin embargo, existen tareas que tienen un alto costo computacional por lo que su simulación es lenta. Una manera de mejorar el tiempo de respuesta para estas aplicaciones es mediante el uso de varios procesadores, es decir de computadoras paralelas [19]. En principio, el uso de varios procesadores para realizar una tarea de simulación puede reportar grandes beneficios. Sin embargo, una simulación microarquitectural de procesadores requiere un acoplamiento fuerte entre todas sus unidades funcionales, y por lo tanto de una comunicación y sincronización adecuada en un simulador paralelo. Este es un problema cuya solución no se puede aplicar en forma general [10]. Es necesario desarrollar técnicas específicas para cada problema de simulación.

Esta tesis presenta un ambiente de trabajo para la ejecución paralela de un conjunto de tareas de simulación de procesadores superescalares. Para ello, se han considerado dos problemas fundamentales. En primer lugar, se considera que se tiene un conjunto de tareas de simulación diferentes, las cuales se pretende ejecutar simultáneamente sobre un conjunto heterogéneo de computadoras. En segundo lugar, se considera una sóla tarea de simulación cuyo tiempo de simulación se quiere reducir.

Para el primer problema, se parte de un conjunto de tareas de simulación conocidas (*benchmarks*) y mediante un balanceador de carga se distribuye dinámicamente cada tarea entre un conjunto de computadoras disponibles. El balanceador de carga trata de mantener la carga de trabajo de cada computadora disponible de manera balanceada. Para lograr lo anterior, toma en cuenta, por un lado, la heterogeneidad de las aplicaciones y de las computadoras de la plataforma de experimentación y hace estimaciones acerca de los tiempos de ejecución esperados de las tareas de simulación en diferentes computadoras. Por otro lado, cuando las estimaciones no coinciden con los tiempos de ejecución reales, el balanceador aplica un tipo de migración de procesos en el cual las tareas cuya ejecución aun no ha iniciado son relocalizadas en otras computadoras a fin de lograr tener un sistema más balanceado.

Para el segundo problema, asumiendo que se tienen tareas de simulación que son muy costosas computacionalmente, se desarrolló un simulador paralelo que distribuye el trabajo de una sola tarea de simulación entre varios procesadores. Este simulador asigna un proceso (hilo de ejecución) a cada etapa de la arquitectura segmentada (*pipeline*) del procesador. Para ello, cada una de las etapas se desacopla de la simulación secuencial asignando copias de las estructuras de datos globales a cada una de ellas. Para sincronizar adecuadamente las diferentes etapas de ejecución de una instrucción se utiliza un reloj global virtual, el cual sincroniza las etapas al inicio y fin de cada ciclo de reloj.

Para los propósitos de este trabajo, se han utilizado los simuladores del conjunto de herramientas de simulación de SimpleScalar, un procesador superscalar diseñado por el Grupo de Arquitectura de Computadoras de la Universidad de Wisconsin [3]. Los simuladores utilizados, *sim_outorder*, *sim_bpred* y *sim_cache*, permiten evaluar el rendimiento de la ejecución de varias instrucciones en forma simultánea, de la unidad de predicción de saltos y de la configuración de la memoria caché, respectivamente. En el simulador de SimpleScalar el ciclo principal de una instrucción simula las diferentes etapas de la arquitectura segmentada (*pipeline*) del procesador: *commit, writeback, execute, scheduler, dispatch,* y *fetch* [7]. Este ciclo se ejecuta una vez por cada instrucción de la tarea de simulación.

Para evaluar el rendimiento del simulador paralelo se usaron los benchmarks SPEC95 precompilados para la arquitectura de SimpleScalar [1].

Este tesis esta organizada en 5 capítulos cuyo contenido se describe a continuación. En el Capítulo 1 se presenta una revisión de conceptos diversos sobre arquitectura de procesadores, así también, se hace una descripción del simulador *sim-outorder* de SimpleScalar y de las tareas de simulación bajo estudio, los *benchmarks* SPEC95. En el Capítulo 2 se presenta la descripción formal del problema de balance de carga, del espacio solución, y se discuten algunas alternativas de solución. En el Capítulo 3 se presenta el sistema de balance de carga desarrollado, se plantea la solución elegida en este trabajo al problema de balance de carga, se describen las heurísticas aplicadas, el modelo de la aplicación, el modelo teórico del sistema y los algoritmos usados para desarrollar el sistema de balance de carga. En el Capítulo 4 se describe la arquitectura segmentada (*pipeline*) del procesador SimpleScalar. Para describir la realización del simulador paralelo de dicho procesador, las estrategias de particionamiento y sincronización desarrolladas se basan en las etapas de su arquitectura segmentada. En el Capítulo 5 se presenta un estudio sobre la unidad de predicción de saltos, la unidad de memoria caché, y los resultados obtenidos con el simulador secuencial y las diferentes versiones desarrolladas del simulador paralelo. Finalmente, se presentan las conclusiones del trabajo realizado.

Capítulo 1

Arquitecturas de procesadores superescalares

Los procesadores superescalares usan el paralelismo a nivel de instrucción, es decir tienen la capacidad de ejecutar más de una instrucción por cada ciclo de reloj [13]. Se traen de la memoria caché más de una instrucción al mismo tiempo envíandolas a la cola de *fetch* de instrucciones y posteriormente se decodifican y, si no tienen dependencias de datos, se ejecutan tantas instrucciones simultáneamente como unidades de ejecución estén disponibles. En este capítulo se presentan estos y otros conceptos de arquitectura de computadoras, la arquitectura del procesador superescalar SimpleScalar, y se describen las pruebas de rendimiento que se utilizan para evaluar arquitecturas de computadoras.

1.1 El concepto de arquitectura de computadoras

El término de "arquitectura de computadoras" fue introducido por IBM en 1964. Amdahl, Blaauw y Brooks lo usaron para referirse a la estructura de la computadora IBM 360, específicamente al conjunto de instrucciones que el programador debe comprender para poder escribir un programa correcto. La idea principal era que los programas deberían poder ejecutarse en todas las computadoras con la misma estructura [13]. Actualmente arquitectura de computadoras se puede definir como la capacidad funcional y la estructura lógica de las computadoras cuyo propósito es coordinar los diferentes niveles de abstracción de los programas a ejecutar por una computadora. La arquitectura de una computadora se puede interpretar también como la interfaz entre el *hardware* y el *software*, como se muestra en la Figura 1.1

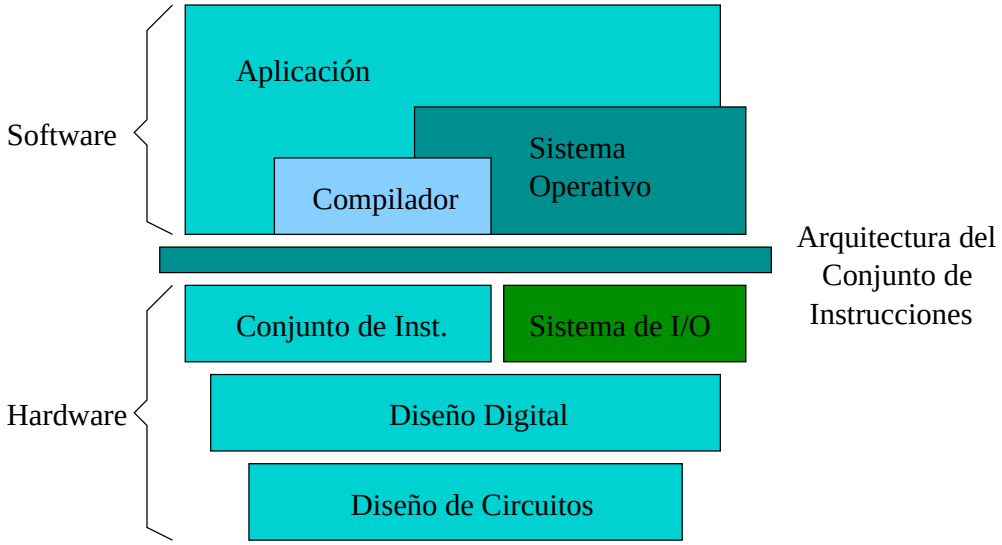


Figura 1.1: Arquitectura de computadoras

1.2 Procesadores superescalares

Fundamentalmente, un procesador lee instrucciones de memoria, las decodifica, las ejecuta y genera resultados que van a registros internos o a la memoria.

1.2.1 Organización de la arquitectura segmentada

En una computadora con arquitectura segmentada, o arquitectura *pipeline*, cada uno de los pasos completa una parte de una instrucción. Estos pasos son llamados etapas o segmentos. Cada etapa esta conectada a la siguiente, de tal forma que para que se ejecute una instrucción debe pasar por cada una de las etapas. El *pipeline* incrementa el *throughput* de las instrucciones, número de instrucciones completadas por unidad de tiempo, ejecutando diferentes etapas de diferentes instrucciones simultáneamente [13].

Etapas del pipeline

Las etapas típicas del *pipeline* de un procesador son las siguientes: (ver Figura 1.2).

- Fetch (Instruction Fetch, IF): Trae la instrucción de la memoria al IR (Instruction Register); incrementa el PC (Program counter) para la dirección de la siguiente instrucción. El Instruction Register se usa para guardar la instrucción que se puede necesitar en los subsecuentes ciclos de reloj. El registro NPC (Next Program Counter) se usa para guardar el siguiente PC.

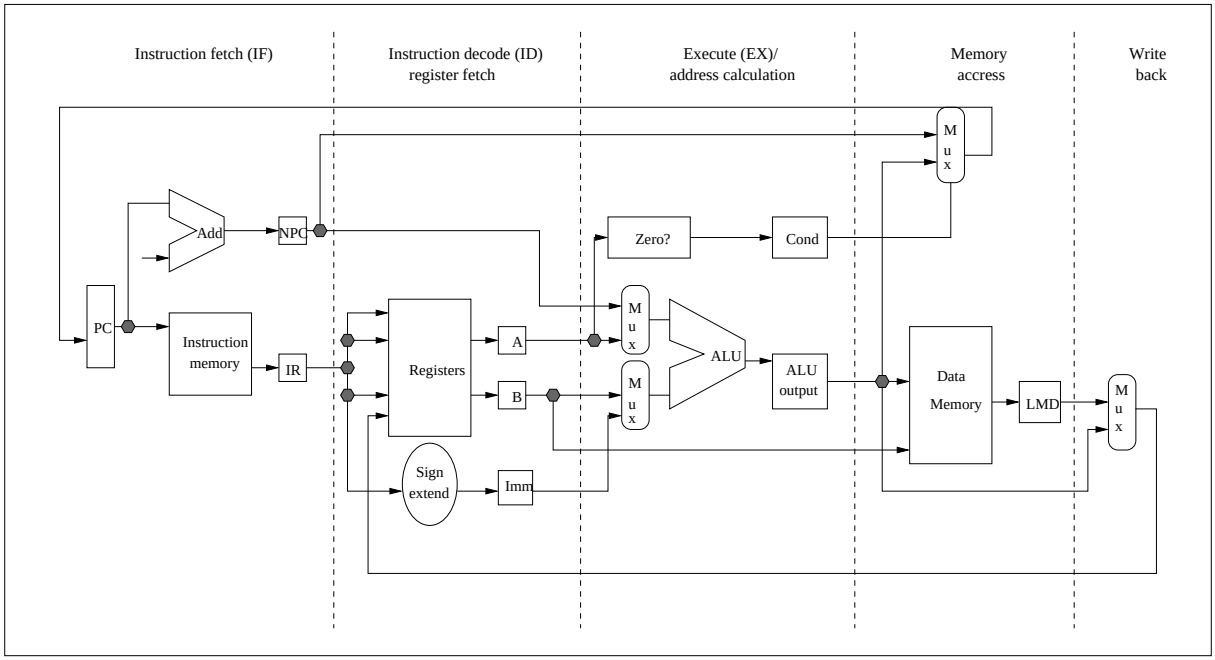


Figura 1.2: Arquitectura segmentada tipica de un procesador.

- Decode (Instruction Decode): Decodifica las instrucciones y lee los registros. La lectura de los registros se hace en paralelo debido a que se encuentran en un lugar fijo.

- Execute (EX): El ALU (Arithmetic Logic Unit) ejecuta la instrucción cuyos operandos quedaron listos en el ciclo anterior, realizando una de cuatro funciones dependiendo del tipo de la instrucción.

  - Referencia a memoria. El ALU suma los operandos para formar la dirección efectiva. El resultado lo pone en el registro ALU output.
  - Instrucción registro-registro. El ALU realiza la operación especificada por el código de operación entre el registro A y el registro B. El resultado lo pone en el registro temporal ALU output.
  - Instrucción registro-inmediato. El ALU realiza la operación especificada por el código de operación entre el registro A y el registro Imm. El resultado lo pone en el registro temporal ALU output.
  - Salto. El ALU suma el NPC al valor en el registro Imm para calcular la dirección de la instrucción a ejecutar después del salto.

- Memory access (MEM): Sólo se ejecutan instrucciones de carga, almacenamiento y salto.

  - Referencia a memoria. Si la instrucción es una carga, se toman los datos de la memoria y se colocan en el registro LMD (Load Memory Data); si es un almacenamiento los datos se toman del registro B y se escribe en memoria.
  - Salto. Si la instrucción es un salto, se modifica el valor de PC por la dirección de destino del salto.

- Writeback (WB): Toma el resultado de memoria o del ALU y lo escribe en los registros.

A continuación se describe más en detalle como se efectúa dicho trabajo en un procesador moderno. En la Figura 1.3 se muestra un diagrama de bloques de la arquitectura típica de un procesador superescalar [16].

- Unidad de Fetch/Decode. En esta unidad se lee un flujo de instrucciones desde la memoria caché de instrucciones L1 transmitiéndolas al decodificador, y las transforma en una serie de micro-operaciones aún en el orden del flujo de instrucciones original. El conjunto de registros de instrucciones puede causar retrasos en los recursos debido a la dependencia entre registros. Para resolver este problema, el procesador cuenta con suficientes registros internos de próposito general. Para asignar estos registros las instrucciones decodificadas son enviadas a la tabla de renombramiento de registros donde las referencias a los registros lógicos se transforman en referencias al conjunto interno de registros. El paso final en la decodificación es preparar las micro-operaciones para la ejecución enviando los resultados a la cola de instrucciones o *buffer* de reordenamiento.

- Buffer de reordenamiento. Las micro-operaciones del flujo de instrucciones se mantienen en el mismo orden en que fueron enviadas desde la unidad de *Decode*. El *buffer* de reordenamiento o cola de instrucciones es un arreglo organizado en registros, los cuales contienen micro-operaciones en espera a ser ejecutados así como las instrucciones que ya han sido ejecutadas pero no han sido retiradas del *buffer*.

- Unidad de despacho y ejecución. Las micro-operaciones almacenadas en el *buffer* de reordenamiento son ejecutadas no necesariamente en el orden en que fueron decodificadas. Considerando la dependencia de datos y la disponibilidad de los recursos, se almacenan las micro-operaciones temporalmente en estaciones de reservación hasta que se obtienen los resultados. La estación de reservación revisa continuamente el *buffer* de reordenamiento para obtener las micro-operaciones que ya están listas para ser ejecutadas. Los resultados de la ejecución de las micro-operaciones son regresados al buffer de reordenamiento almacenándolas en él hasta que las instrucciones son retiradas. La ejecución de las micro-operaciones se realiza en las unidades

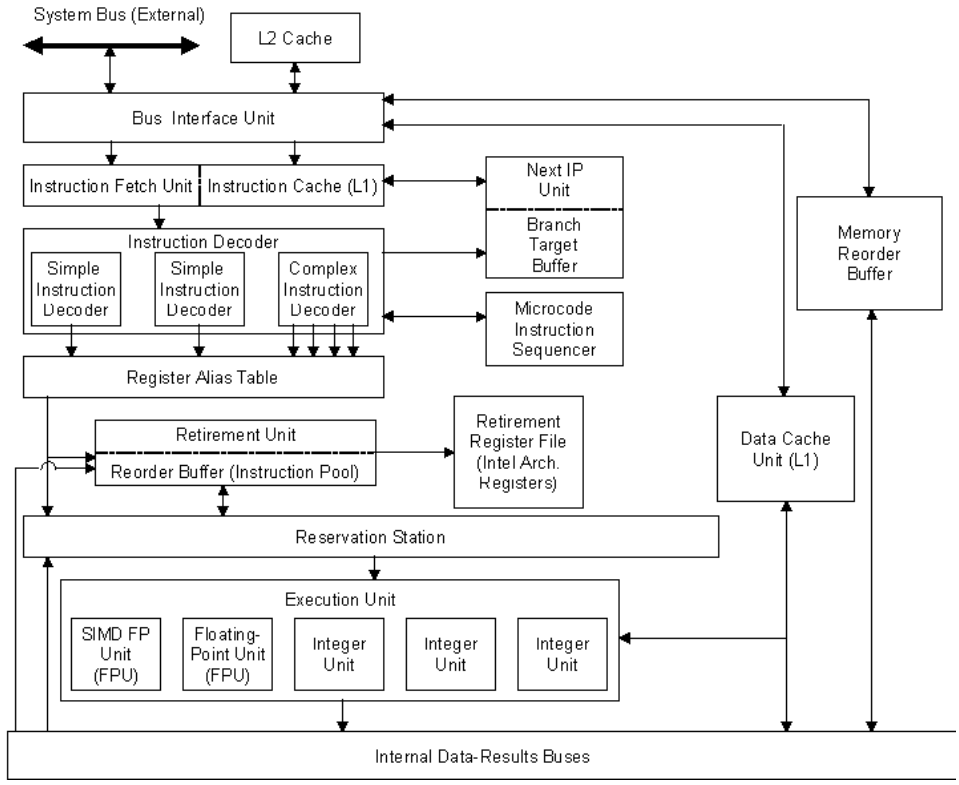


Figura 1.3: Arquitectura típica de un procesador superescalar.

de aritmética de enteros donde una de ellas está diseñada para ejecutar micro-operaciones de salto, en las unidades de punto flotante y en la unidad de carga y almacenamiento (*memory-interface unit*) permitiendo la ejecución de varias instrucciones por ciclo de reloj.

- Unidad de retiro. Esta unidad toma los resultados obtenidos en la ejecución y los retira del *buffer* de reordenamiento. Es similar a la estación de reservación, continuamente verifica el estado del *buffer* para retirar las instrucciones que ya han sido completadas en el orden original del programa. La unidad de retiro puede quitar más de una instrucción por ciclo de reloj. Al retirar las instrucciones escribe los resultados en el archivo de registros y/o en memoria.

1.2.2 Ejecución dinámica

Los procesadores superescalares se caracterizan también por llevar a cabo la ejecución dinámica, o ejecución fuera de orden, de las instrucciones. La ejecución dinámica, permite que las instrucciones se ejecuten sin respetar la secuencia original en el programa, es decir las instrucciones son ejecutadas tan pronto como sus operandos estén disponibles. Cuando las instrucciones pueden ser ejecutadas fuera de orden el tiempo de ejecución total se reduce debido a que se ejecutan más instrucciones

por ciclo de reloj. Esto es posible cuando hay suficiente capacidad en las estaciones de reservación. Sin embargo, los programas no pueden ser ejecutados fuera de orden cuando existen dependencias de datos o cuando se tienen conflictos con los recursos [13].

1.2.3 Dependencias entre instrucciones

Determinar cuando una instrucción depende de otra es importante no sólo para el proceso de programación sino también para determinar cuánto paralelismo a nivel de instrucción existe en un programa y cómo puede ser explotado. En particular, para explotar paralelismo a nivel de instrucción se debe determinar cuales instrucciones pueden ser ejecutadas en paralelo. Si dos instrucciones son paralelas pueden ejecutarse simultáneamente en *pipeline* suponiendo que existen recursos suficientes. Asimismo, dos instrucciones que son dependientes no pueden ser reordenadas. Las instrucciones que pueden ser reordenadas son paralelas y viceversa. Hay tres tipos de dependencias: dependencias de datos, dependencias de nombres y dependencias de control [13].

Una instrucción j tiene dependencia de datos con una instrucción i si:

- La instrucción i produce un resultado que es usado por la instrucción j, ver ejemplo (a) en la Figura 1.4 [15].
- La instrucción j tiene dependencia de datos con la instrucción k y la instrucción k con la instrucción i, ver ejemplo (b) en la Figura 1.4 [15].

Una dependencia de nombres ocurre cuando dos instrucciones usan el mismo registro o localidad de memoria. Hay dos tipos de dependencias de nombres: antidependencia y dependencia de salida.

- Una *antidependencia* entre la instrucción i y la instrucción j ocurre cuando la instrucción j escribe en un registro o localidad de memoria que la instrucción i lee y la instrucción i es ejecutada primero, ver ejemplo (c) en la Figura 1.4 [15].
- Una *dependencia de salida* ocurre cuando la instrucción i y la instrucción j escriben en el mismo registro o localidad de memoria, ver ejemplo (d) en la Figura 1.4 [15].

Una dependencia de control ocurre cuando el orden de ejecución de las instrucciones no puede ser determinado antes de que se ejecute el programa, ver ejemplo (e) en la Figura 1.4 [15].

1.2.4 Predicción de saltos

La predicción de saltos es importante debido a la alta frecuencia con que se realizan saltos en un programa, el objetivo es encontrar rápidamente la instrucción que sigue después de que se realizó

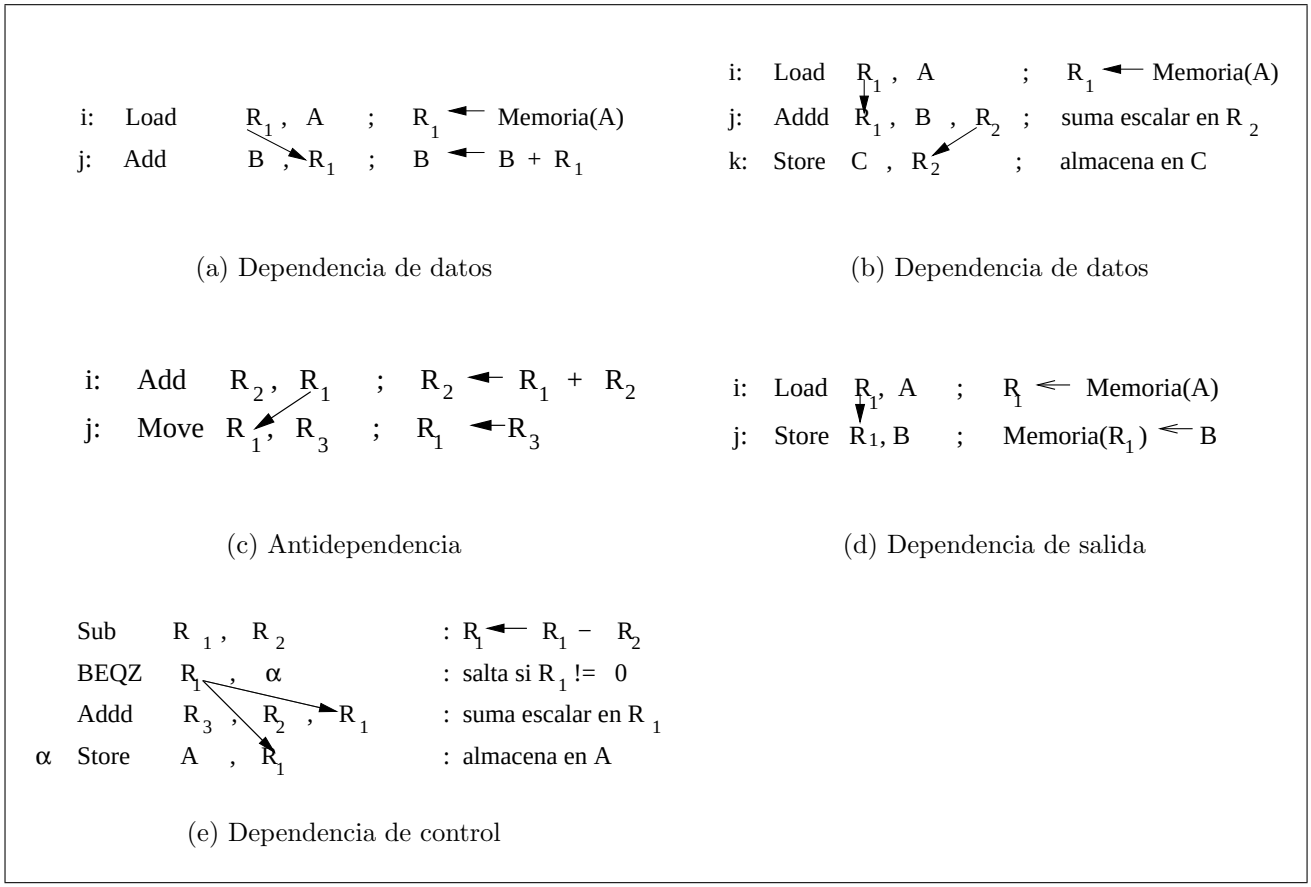


Figura 1.4: Ejemplos de Dependencias entre instrucciones

un salto. La predicción de saltos se puede determinar estáticamente, de acuerdo al código del programa, o dinámicamente en base a la historia de saltos durante la ejecución del mismo. Para obtener la frecuencia de saltos, los tipos de saltos y la probabilidad de tomarlos es necesario trazar programas un gran número de veces. Las estrategias de predicción de saltos dinámicas toman una historia reciente limitada ya que si se toma la historia completa no sería factible de implementar. Por tanto, la predicción de saltos está determinada por la historia reciente [13].

Para tener un mejor rendimiento se tienen algunos esquemas de predicción de saltos:

- *predict-not-taken:* Permite que el *hardware* continúe como si el salto no se ejecutara. No se debe cambiar el estado de la máquina hasta que el resultado del salto se sabe.

- *taken:* Ejecuta la instucción de salto y las instrucciones siguientes a él ya no se ejecutan, entonces se calcula la dirección de la instrucción que se ejecutara de acuerdo a la predicción después del salto.

- *perfect:* El procesador puede tomar arbitrariamente el conjunto de instrucciones a ejecutar,

renombra todos los registros, determina que instrucciones se pueden ejecutar y cuales deben esperar. Predice todos los saltos. Tiene las suficientes unidades funcionales para permitir que todas las instrucciones listas se puedan ejecutar.

- *two-level:* Típicamente el predictor de dos niveles utiliza el BTB (*branch target buffer*) donde se tiene información sobre los saltos más recientes (si fueron tomados o no) y la dirección de la instrucción que se ejecutó después del salto [4].

- *bimodal:* Divide el contador de la tabla en dos. Se realiza la selección tomando un patrón de la historia y dos contadores, uno para cada mitad. En otro contador se van registrando únicamente las direcciones de los saltos para realizar la selección final en uno de los dos contadores. La predicción final es hecha de acuerdo al contador seleccionado y sólo éste puede ser actualizado con el resultado del salto, el estado del contador no seleccionado no debe alterarse [21].

- *combinado:* Es una combinación de los predictores *bimodal* y *two-level*.

1.2.5 Ejecución especulativa

Las instrucciones de salto limitan el paralelismo debido a que no se pueden ejecutar instrucciones hasta que no se tenga el resultado de la instrucción que se debe ejecutar después del salto. Dado que de cada cinco o seis instrucciones una es un salto, es importante contar con alguna técnica para superar esta limitante y así poder explotar más el paralelismo. La ejecución especulativa permite la ejecución de una instrucción antes de que el procesador sepa que debe ser ejecutada evitando la dependencia de control. La especulación se puede hacer de forma estática por el compilador con soporte en *hardware*; aquí el compilador elige la instrucción y el hardware ayuda a deshacer el resultado cuando la especulación fue incorrecta. Otra forma es usando las instrucciones condicionales realizando la especulación dinámicamente pero el hardware usa un predictor de saltos para realizar el proceso de especulación [20].

1.2.6 Jerarquía de memoria

La transferencia de información desde la memoria principal a la memoria caché es por medio de unidades de bloques o líneas de caché.

Los bloques en la memoria caché son llamados *blocks frames*, para distinguirlos de los correspondientes bloques en memoria principal. *Blocks frames* son denotados como: $\overline{B_i}$ para $i = 0, 1, 2, \ldots, m$. Los bloques son denotados como B_j para $j = 0, 1, 2, \ldots, n$. Los mapeos pueden estar definidos del

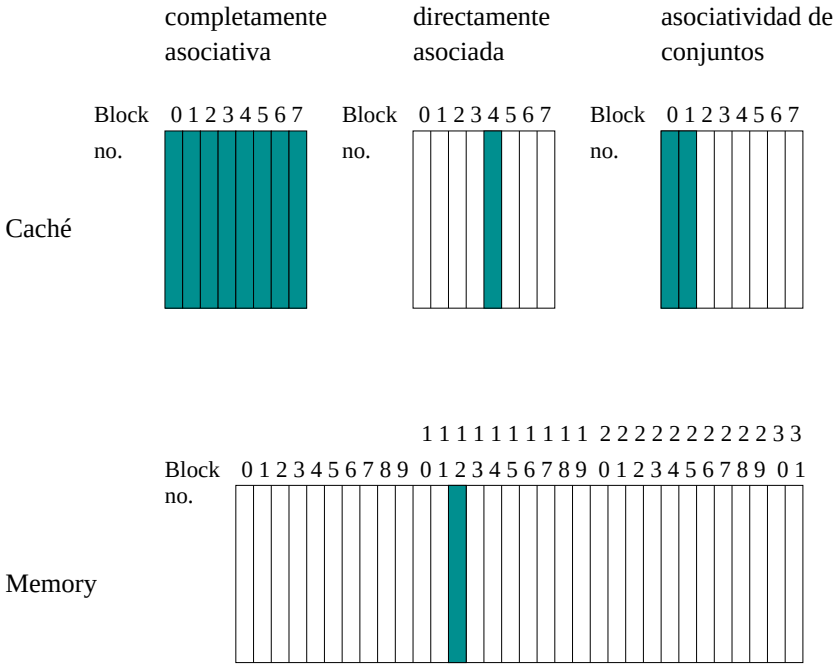


Figura 1.5: Categorías de organización de la memoria caché

conjunto $\{B_j\}$ al conjunto $\overline{B_i}$, donde $n >> m,\ n = 2^s,\ y\ m = 2^r$. Cada bloque o *block frame* tiene b palabras, donde $b = 2^w$.

En esta sección se muestran tres categorías de organización de la memoria caché (ver figura 1.5):

- Directamente asociada. Si cada bloque se puede poner solamente en un lugar en la memoria caché, la memoria caché se dice que está directamente asociada. La asociación es de la siguiente forma:

$$(block\ address)\ MOD\ (Number\ of\ blocks\ in\ cache)$$

- Completamente asociativa. Si un bloque se puede poner en cualquier lugar en la memoria caché, la caché se dice que es completamente asociativa [12].

- Asociatividad de conjuntos. Si un bloque se puede poner en un conjunto limitado de lugares en la memoria caché, la memoria caché se dice que tiene asociatividad de conjuntos. Un conjunto es un grupo del conjunto de bloques en la caché. Primero un bloque se asocia sobre un conjunto, y entonces el bloque se puede poner en cualquier parte dentro de ese conjunto. El conjunto usualmente se elige con un bit de selección:

$$(block\ address)\ MOD\ (Number\ of\ sets\ in\ cache)$$

Si hay n bloques en un conjunto, la organización de la caché es llamada asociatividad de conjuntos de n formas.

TLB (translation look-aside buffer) es una memoria caché dedicada para la traducción de direcciones virtuales a direcciones físicas, así un acceso a memoria raramente requerirá de un segundo acceso para traducir los datos.

1.3 SimpleScalar

SimpleScalar es un conjunto de herramientas de simulación secuencial desarrollado por el grupo de computadoras de la Universidad de Winsconsin-Madison en 1992, está públicamente disponible y ofrece simulación detallada de microprocesadores superescalares. SimpleScalar es una infraestructura para modelar sistemas de computadoras que facilita la implantación del modelo en *hardware* capaz de simular aplicaciones completas. Durante la simulación el modelo mide la correctitud del modelo de *hardware* y el rendimiento del software que se ejecuta en él. El paquete de SimpleScalar consiste de:

- El código fuente de los simuladores (ver sección 1.3.2), y algunos pequeños programas de prueba en lenguaje C (*tests*) y precompilados para SimpleScalar.
- Herramientas de GNU versión 2.5.2 para SimpleScalar, estas herramientas no son necesarias para ejecutar los simuladores, pero se requieren para compilar los *benchmarks* desarrollados por el usuario.
- El compilador GCC versión 2.6.3, y el programa f2c, para convertir código en Fortran a lenguaje C. Estas herramientas son necesarias para generar *benchmarks*.
- El conjunto de *benchmarks* SPEC95 precompilados para SimpleScalar para computadoras de arquitectura *big-endian*[1]
- El conjunto de *benchmarks* SPEC95 precompilados para SimpleScalar para computadoras de arquitectura *little-endian*[2]
- Paquete para generar estadísticas, creando una base de datos, por medio de un arreglo de distribución se aplica una fórmula estadística, obteniendo los valores de las variables estadísticas en la base de datos. Las estadísticas se generan cuando se ejecutan los simuladores.

[1]La dirección de un dato se almacena dentro de una palabra en la posición más significativa.
[2]La dirección de un dato se almacena dentro de una palabra en la posición menos significativa.

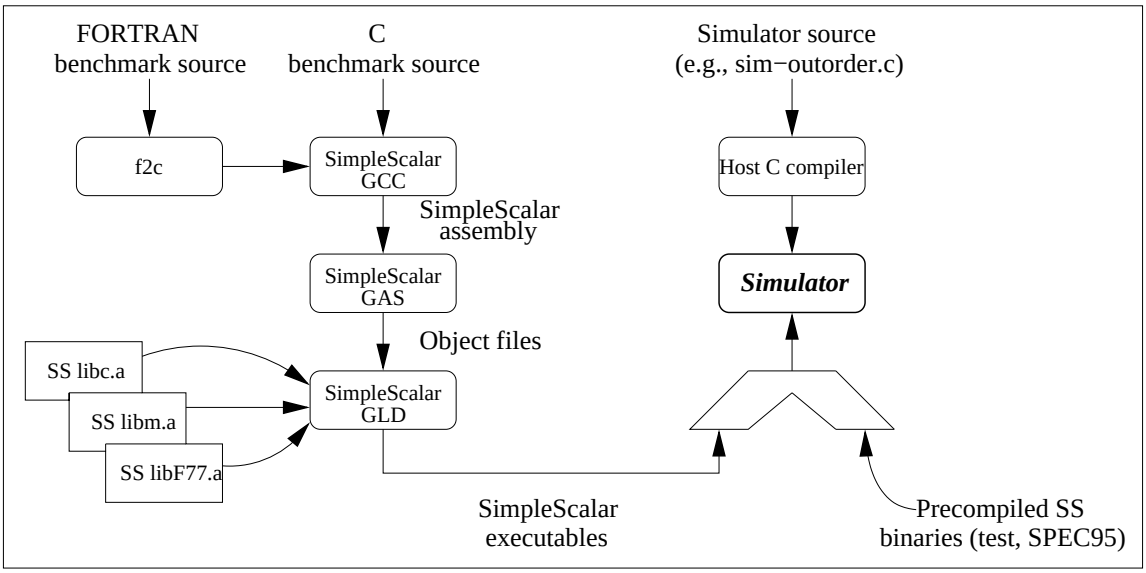


Figura 1.6: Descripción del conjunto de herramientas de SimpleScalar

- Depurador Dlite!. Es un depurador muy sencillo, se puede utilizar con todos los simuladores excepto con el simulador sim-fast. Básicamente permite colocar puntos de interrupción, desplegar instrucciones y el estado de los registros, desensamblar instrucciones indicando la dirección.

1.3.1 Ambiente de simulación del procesador SimpleScalar

En la Figura 1.6 se muestra el ambiente de simulación de SimpleScalar [2]. Los *benchmarks* escritos en Fortran son convertidos a lenguaje C usando el convertidor $f2c$. De tal forma que los *benchmarks* escritos en lenguaje C y los *benchmarks* escritos en Fortran, convertidos a C, son compilados usando el compilador de SimpleScalar GCC, el cual genera código en ensamblador para SimpleScalar. El ensamblador y cargador de SimpleScalar, GAS, junto con las bibliotecas necesarias producen código ejecutable para SimpleScalar, que es el programa de entrada para el simulador (los simuladores que forman parte de SimpleScalar se describen en las secciones siguientes, los simuladores son compilados con el compilador ANSI C de la computadora donde se ejecutan los simuladores). También es posible usar los *benchmarks* SPEC95 o los programas de prueba incluidos en SimpleScalar.

1.3.2 Simuladores de SimpleScalar

SimpleScalar cuenta con cinco tipos de simuladores:

- simuladores funcionales:

  - **sim-fast**. Es el simulador más rápido, no realiza ninguna estadística de rendimiento la simulación sólo es funcional. Ejecuta cada instrucción de forma serial, no realiza simulación de instrucciones en paralelo y no hay memoria caché.

  - **sim-safe**. También realiza simulación funcional. Pero revisa los permisos de acceso de cada referencia a memoria. Ninguno de los dos simuladores acepta argumentos en la línea de comandos que modifiquen su comportamiento, son útiles para entender el funcionamiento interno de los simuladores.

- Simuladores de la jerarquía de memoria:

  Estos simuladores son ideales para una simulación rápida de la memoria caché si el efecto del rendimiento de la memoria caché en tiempo de ejecución es el parámetro de rendimiento a observar.

  - **sim-cache**. Permite configurar el tamaño de la memoria caché, la asociatividad y la política de reemplazo.

  - **sim-cheetah**. Está orientado a la configuración de una asociatividad completa y políticas de remplazo algunas veces óptimas. Esta política usa conocimiento futuro para seleccionar un reemplazo; se escoge el bloque que será referenciado en el futuro. Se utiliza sólo como referencia, pues ningún procesador real aplica esta política.

- **sim-profile**. Genera descripciones detalladas de las clases de instrucciones y direccionamientos, símbolos de texto, accesos a memoria, saltos, y símbolos de segmentos de datos.

- **sim-bpred**. Este es un simulador de predicción de saltos, donde se pueden elegir diferentes banderas dando una secuencia de argumentos. Con los argumentos para especificar el predictor podemos modificar el número de entradas, el número de niveles, el número de entradas de cada nivel, el tamaño de la meta-tabla para un predictor combinado, el tamaño de la pila, el número de conjuntos y la asociatividad.

- **sim-outorder**. Es el simulador más complicado y detallado de los simuladores de SimpleScalar. sim-outorder simula la ejecución fuera de orden del procesador *pipeline*. Usa todos los argumentos y formatos usados en los simuladores de memoria caché, y el predictor de saltos se puede especificar como en el simulador de predicción de saltos.

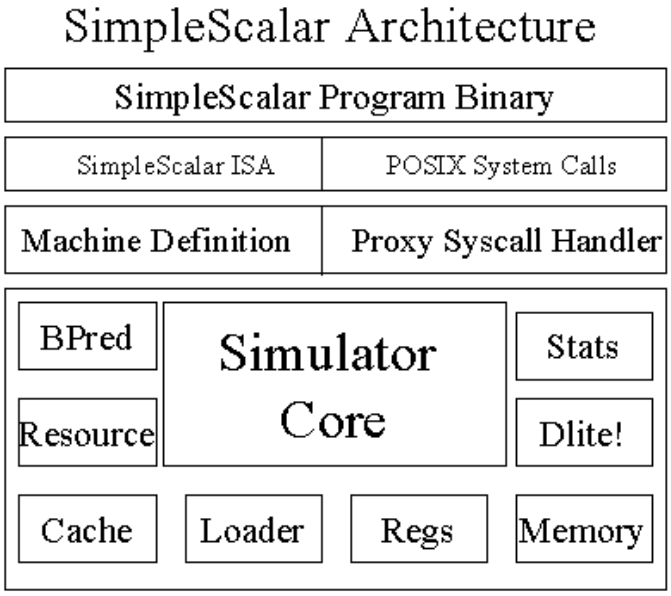


Figura 1.7: Arquitectura del procesador superescalar SimpleScalar

1.4 El procesador SimpleScalar

En la Figura 1.7 se muestra la arquitectura del procesador superescalar de SimpleScalar. Esta arquitectura está derivada del procesador MIPS-IV. Los llamados al sistema son manejados por un *proxy* que intercepta los llamados hechos por el programa binario decodificándolos haciendo el correspondiente llamado al sistema operativo de la computadora en donde se ejecutan los simuladores y copiando los resultados del llamado en la memoria del programa simulado. SimpleScalar cuenta con instrucciones de control, instrucciones de carga y almacenamiento e instrucciones de aritmética entera y de punto flotante [3].

1.4.1 Ejecución dinámica

SimpleScalar simula la ejecución de las instrucciones fuera de orden y está basado en la unidad de actualización de registros, RUU. Este esquema usa un *buffer* de reordenamiento para renombrar registros automáticamente y almacenar resultados con dependencia de datos. En cada ciclo el buffer de reordenamiento retira las instrucciones que han sido completadas de acuerdo a la secuencia original del programa escribiendo en el conjunto interno de registros.

1.4.2 Predicción de saltos

En SimpleScalar se tienen diferentes tipos de predictores de saltos: not-taken, taken, perfect, bimod, two-level y combinado.

En los predictores se puede configurar el número de entradas de la tabla del predictor bimodal, en el de dos niveles se puede especificar el número de entradas en cada nivel, el tamaño del *stack*, se puede configurar en el BTB (branch target buffer) el número de conjuntos que se puede tener y la asociatividad.

1.4.3 Jerarquía de memoria

En SimpleScalar se puede configurar la memoria caché especificando el tipo de asociatividad: directamente asociativa, completamente asociativa o asociatividad de conjuntos, y el número de conjuntos; la política de reemplazamiento: FIFO o aleatorio; el tamaño de la memoria caché y el tamaño de los bloques para el TLB. Se tienen dos niveles de memoria, se puede configurar el tamaño de los dos niveles para la caché de datos y la caché de instrucciones y el número de entradas para el TLB de datos y el número de entradas para el TLB de instrucciones.

1.4.4 Organización de la arquitectura segmentada

La ejecución de una instrucción del procesador *pipeline* se ejecuta una vez por cada instrucción del programa.

Etapas del pipeline en sim-outorder

Las etapas del pipeline en sim-outorder son las siguientes: (ver Figura 1.8).

- Fetch: Está implementado en la función *ruu_fetch()*. Carga la cola IFQ (Instruction Fetch Queue) con instrucciones que toma de la caché de instrucciones (I-cache). Revisa el estado del predictor de saltos, si se realizó alguna predicción se accesa en esa dirección en el siguiente ciclo.

- Dispatch: Está implementado en la función *ruu_dispatch()*. Toma las instrucciones desde la IFQ las decodifica y las asigna a la RUU (Register Update Unit) y a la LSQ (Load/Store Queue) donde se guardan todos los resultados y las instrucciones esperan a que todos los operandos esten listos.

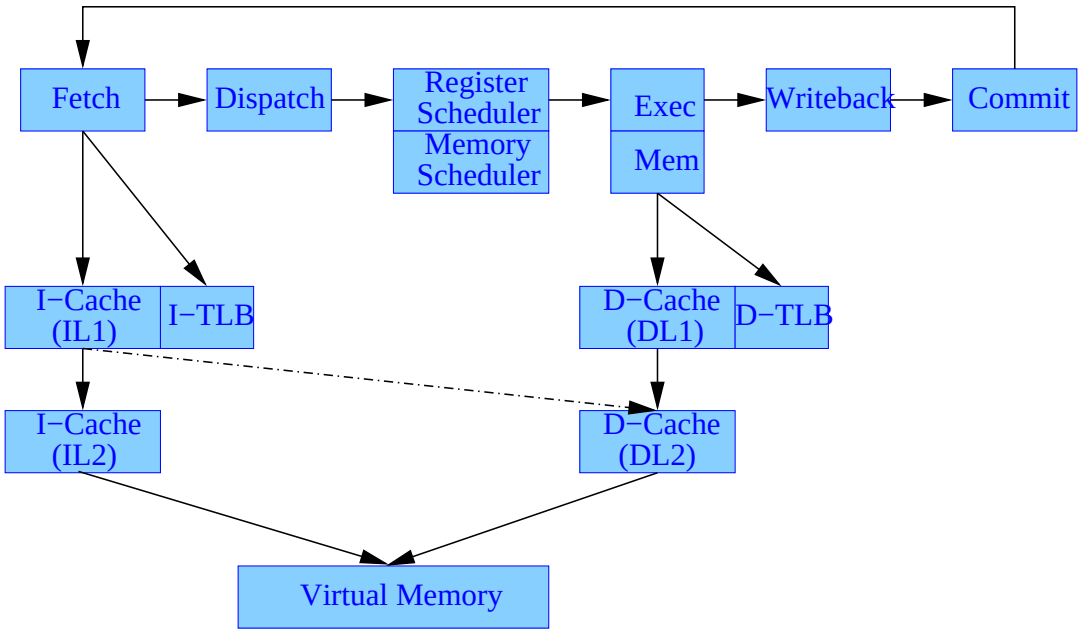


Figura 1.8: Pipeline en sim-outorder

- Scheduler: Está implementado en la función *lsq_refresh y ruu_issue()*. Localiza en LSQ y RUU las instrucciones listas, es decir, las instrucciones cuyas dependencias en memoria y en los registros ya fueron resueltas para que puedan ser ejecutadas.

- Execute: Está implementado en la función *ruu_issue()*. Se toman las instrucciones a ejecutarse de RUU y LSQ realizando el acceso desde la memoria caché de datos (D-cache). Se verifica que la unidad funcional esté disponible para comenzar la ejecución de la operación, si es así se realiza la asignación de la unidad funcional, se manda un evento a writeback y se inicia la ejecución de la instrucción.

- Writeback: Está implementado en la función *ruu_writeback()*. Obtiene los resultados completos de las instrucciones ejecutadas desde las unidades funcionales y los escribe en el conjunto interno de registros determinando si cualquier instrucción con dependencia de datos ha resuelto dichas dependencias enviando la instrucción a la cola de instrucciones listas.

- Commit: Está implementado en la función *ruu_commit()*. Reordena las instrucciones en el orden del programa, se accede a la memoria caché de datos, para retirar las instrucciones del conjunto interno de registros, de RUU, LSQ y de la unidad funcional.

1.5 Evaluación de arquitecturas de computadoras

Cuando se realiza un cambio arquitectural es necesario realizar una etapa de simulación que exhiba un rendimiento adecuado antes de implantarlo para reducir costos. Se considera que una computadora es más rápida que otra cuando un programa se ejecuta en un tiempo menor, también se puede considerar que una computadora es más rápida que otra cuando ésta termina más tareas por unidad de tiempo. Por lo tanto, el tiempo de ejecución, es el tiempo que toma una tarea desde que inicia hasta que termina y el *throughput* es la cantidad total de trabajo hecha en un tiempo dado. Para medir el rendimiento de dos computadoras X y Y (X es más rápida que Y) lo expresamos de la siguiente forma:

$$\text{rendimiento} = \frac{\text{tiempo de ejecución } Y}{\text{tiempo de ejecución } X}$$

Debido a que el rendimiento y el tiempo de ejecución son recíprocos, si incrementamos el rendimiento decrementamos el tiempo de ejecución. Para evaluar un sistema nuevo se pueden realizar pruebas de comparación del tiempo de ejecución de una carga de trabajo determinada. Otra forma es realizando pruebas con programas o segmentos de código como por ejemplo ciclos, que sean representativos para medir el rendimiento.

1.5.1 Pruebas de rendimiento

Para la evaluación del rendimiento existen algunos programas que fueron diseñados con este propósito:

1. Programas reales: Se ejecutan para resolver problemas reales, por ejemplo los compiladores de texto, los procesadores de texto como *TeX*. Estos programas tienen entrada, salida y opciones que un usuario puede seleccionar cuando los ejecuta.

2. Kernels: Se extraen las partes importantes de programas reales para evaluar el rendimiento, como los ciclos, y a diferencias de los programas reales los usuarios no pueden ejecutarlos, estos programas sólo existen para evaluar el rendimiento. En los *kernels* se analizan características específicas de una computadora para explicar las diferencias de rendimiento en los programas reales.

3. Programas pequeños de prueba: Estos programas tienen entre 10 y 100 líneas de código y produce un resultado que el usuario conoce antes de ejecutarlo. Estos programas son pequeños, fáciles de escribir y ejecutar en casi cualquier computadora.

4. Programas sintéticos: La filosofía de estos programas es similar a la de los *kernels* se analiza el promedio de la frecuencia de las operaciones y los operandos de un gran conjunto de programas. La diferencia es que estos programas no son parte de una aplicación real.

Recientemente, se ha hecho popular tener un conjunto de *benchmarks* para medir el rendimiento de los procesadores con diferentes aplicaciones del mundo real, tan buenas como las pruebas individuales. Además, tienen la ventaja de que si se tiene alguna debilidad en alguna de las pruebas ésta se reduce por la presencia de otras pruebas. En las siguientes secciones se describen algunos bancos de *benchmarks* usados para evaluar el rendimiento de arquitecturas de computadoras [13].

1.5.2 Benchmarks SPEC95

SPEC (Standard Performance Evaluation Corporation) establecida en 1988 como una corporación dedicada a establecer y mantener un conjunto estandarizado de *benchmarks* que pueden ser aplicados a computadoras de alto rendimiento. SPEC95 fue diseñado para proporcionar una medida comparable del funcionamiento de un sistema que ejecuta una carga conocida de trabajo de cómputo-intensivo. Las pruebas se realizan para evaluar los siguientes componentes de un sistema: procesador, jerarquía de memoria y compilador. SPEC95 está dividido en dos conjuntos de benchmarks [23]:

- SPEC INT95. Conjunto de 8 benchmarks de aritmética de enteros (ver Tabla 1.1).
- SPEC FP95: conjunto de 10 benchmarks de aritmética de punto flotante (ver Tabla 1.1).

Los *benchmarks* SPEC95 pueden ejecutarse en los siguientes sistemas: AIX 413/RS6000, AIX 413/RS6000, HPUX/PA-RISC, SunOS 4.1.3/SPARC, Linux 1.3/x86, Solaris 2/SPARC, Solaris 2/x86, DEC Unix 3.2/Alpha, FreeBSD 2.2/x86 y Windows NT/x86.

Los benchmarks SPEC95 están precompilados para la arquitectura del SimpleScalar y son aplicaciones del mundo real.

1.5.3 Benchmarks SPEC2000

SPEC diseñó SPEC2000 para tener una medida de rendimiento de cómputo intensivo del procesador, la jerarquía de memoria y el compilador de un sistema de cómputo con própositos de análisis de comparación. Los *benchmarks* SPEC2000 son aplicaciones reales, se desarrollaron para actualizar los *benchmarks* SPEC95. Por lo tanto, pueden ser utilizados en los mismos sistemas que los *benchmarks* SPEC95 y en procesadores más recientes como en MacOS X e Intel Itanium. SPEC2000 está dividido en dos conjuntos de benchmarks [14]:

INT95	Area de aplicación
go	Juegos de inteligencia artificial
m88ksim	Simulación
gcc	Programación y compilación
compress	Compresión
li	Interprete de lenguaje lisp
ijpeg	Imágenes
perl	Interprete de lenguaje perl
vortex	Bases de datos

FP95	Area de aplicación
tomcatv	Translación geométrica
swim	Predicción de tiempo
su2cor	Física cuántica
hydro2d	Astrofísica
mgrid	Electromagnetismo
applu	Fluidos dinámicos
turb3d	Simulación
apsi	Predicción de tiempo
fpppp	Química
wave	Electromagnetismo

Tabla 1.1: Benchmarks SPEC95

INT2000	Area de aplicación
gzip	Compresión
vpr	FPGA
gcc	Compilador de lenguaje C
mcf	Combinatoria
crafty	Juego: Queso
parser	Procesador de palabras
eon	visualización
perlbmk	Lenguaje de programación perl
gap	Interprete para teoría de grupos
vortex	Bases de datos orientadas a objetos
bzip2	Compresión
twolf	Simulador de lugar y ruta

FP2000	Area de aplicación
wupwise	Física
swim	Telecomunicaciones
mgrid	Electromagnetismo
applu	Ecuaciones diferenciales parciales
mesa	Biblioteca de gráficas 3D
galgel	Fluidos dinámicos
art	Reconocimiento de imágenes/Redes neuronales
equake	Simulación de propagación de ondas sísmicas
facerec	Procesamiento de imágenes
ammp	Química computacional
lucas	Teoría de números
fma3d	Simulación rápida de elementos finitos
sixtrack	Diseño acelerador de física nuclear
apsi	Meteorología

Tabla 1.2: Benchmarks SPEC2000

- SPEC INT2000. Conjunto de 12 benchmarks de aritmética de enteros (ver Tabla 1.2).
- SPEC FP2000: conjunto de 14 benchmarks de aritmética de punto flotante (ver Tabla 1.2).

Capítulo 2

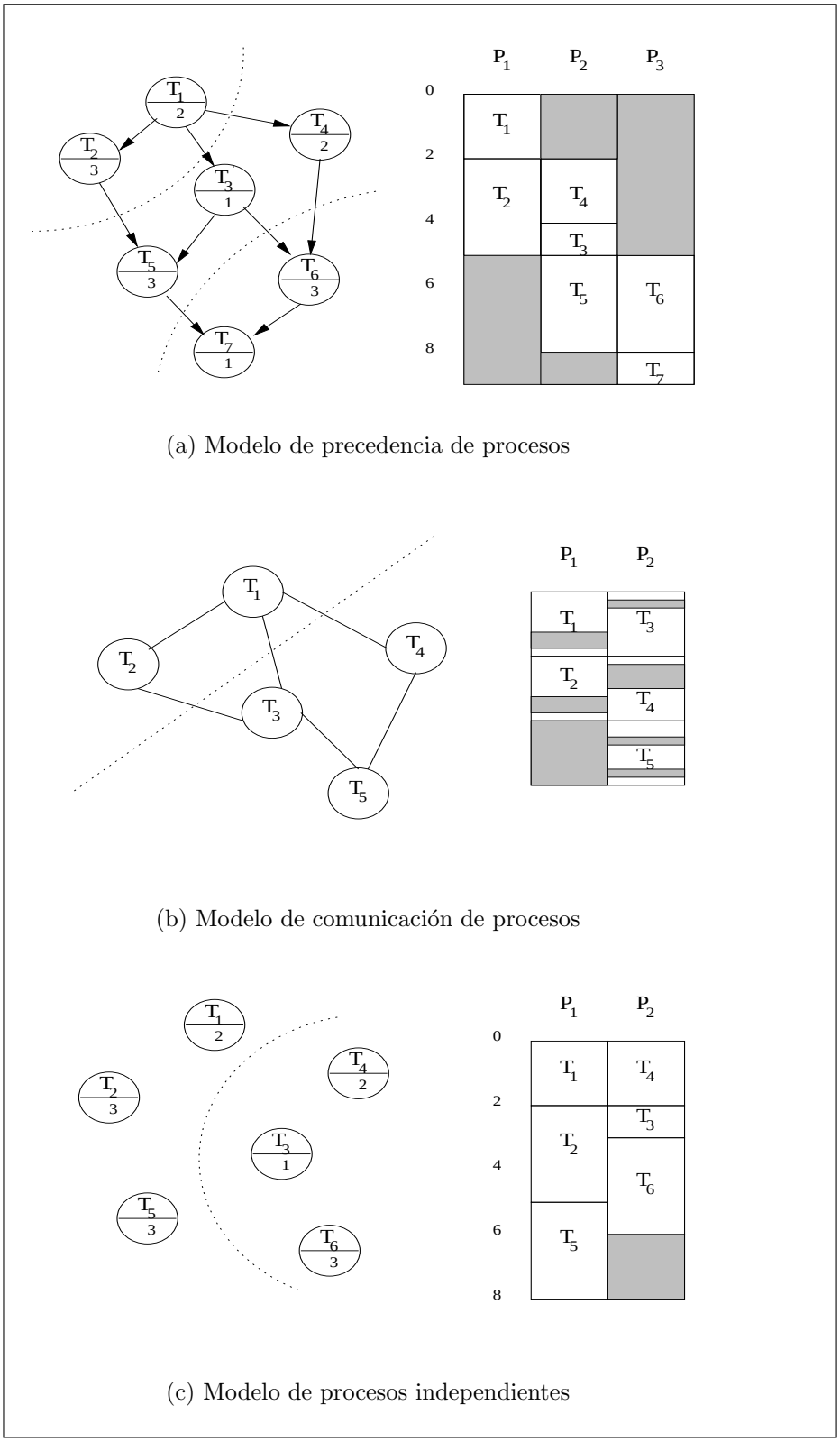
El problema de balance de carga

Si se tiene un conjunto de tareas de simulación a realizar, es conveniente utilizar todos los recursos disponibles para hacerlo de la manera más rápida posible. Para esto se requiere de un balanceador de carga que distribuya dinámicamente las tareas de simulación entre todos los procesadores disponibles. En este capítulo se presentan diversos conceptos relacionados con el problema de balance de carga: los modelos de aplicaciones y los modelos de balance de carga, las medidas de rendimiento, la descripción formal del problema y la estimación de los tiempos de ejecución en un sistema heterogéneo. Finalmente, se presentan algunas estrategias para solucionar el problema de balance de carga.

2.1 Modelos de aplicaciones

Los algoritmos paralelos y distribuidos se representan por un conjunto de procesos múltiples gobernados por reglas que regulan la interacción entre procesos. Las interacciones entre procesos se pueden expresar por tres modelos diferentes [1]:

- **Modelo de precedencia de procesos.** En la Figura 2.1 (a) se muestra el modelo de precedencia de procesos. Este puede ser representado como un grafo dirigido acíclico, en el cual cada nodo representa un proceso, con tiempo de ejecución conocido indicado en la figura debajo del proceso correspondiente, y cada flecha indica la relación de precedencia entre procesos. A la derecha se muestra la forma en que se realizaría la asignación en tres procesadores, considerando los tiempos de ejecución y la precedencia de procesos. Lo que se busca en esta asignación es mantener ocupados a los procesadores mientras se satisfagan las relaciones de precedencia.



(a) Modelo de precedencia de procesos

(b) Modelo de comunicación de procesos

(c) Modelo de procesos independientes

Figura 2.1: Modelos de aplicaciones

- **Modelo de comunicación de procesos.** En la Figura 2.1 (b) se muestra el modelo de comunicación de procesos, la cual se puede representar como un grafo no dirigido donde cada nodo representa un proceso. Los arcos en este caso no indican precedencia sino interacción entre procesos; esto es, los nodos se comunican e intercambian información. A la derecha se muestra como se realizaría la asignación en dos procesadores considerando que los procesos se comunican. En este caso se trata de favorecer la localidad de interacciones haciendo que los procesos que intercambian mucha información queden ubicados en procesadores vecinos o en el mismo procesador.

- **Modelo de procesos independientes.** En este modelo no se tiene mucha información sobre las tareas a ejecutarse. Los procesos son libres de moverse entre los procesadores como se muestra en la Figura 2.1 (c). A la derecha se muestra como se realizaría la asignación en dos procesadores considerando que los procesos son independientes. Lo que se buscaría en este caso es mantener a los procesadores ocupados la mayor parte del tiempo. Los procesos son independientes uno de otro y probablemente lo que se puede conocer es una estimación de sus tiempos de ejecución.

La elección del modelo de balance de carga depende de los procesos de la aplicación. Si algún proceso requiere información del proceso que le antecede, la asignación se debe hacer en base al modelo de precedencia de procesos; así, usando este modelo es posible que los procesadores permanezcan desocupados por largos períodos de tiempo. Si los procesos requieren intercambiar información o comunicarse unos con otros la asignación se debe hacer tomando el modelo de comunicación de procesos. En este tipo de asignación, debido a la interacción entre procesos, es posible que se presenten varios períodos de tiempo en que los procesadores esten desocupados. Cuando se tiene poca información sobre los procesos es posible asignarlos en cualquiera de los procesadores tratando de mantenerlos siempre ocupados. Si la asignación no fue la adecuada es posible reasignar los procesos.

En el caso de tener un conjunto de tareas de simulación de procesadores en donde lo único que se conoce son estimaciones de los tiempos de ejecución de las tareas, es conveniente utilizar el modelo de procesos independientes.

2.1.1 Medidas de rendimiento

- Los procesos se asignan a los procesadores para maximizar la utilización de los procesadores y para minimizar los tiempos de terminación. El tiempo de terminación lo podemos ver como el tiempo que toman los procesos en la cola de espera para ser atendidos y el tiempo que tardan en ejecutarse. Esto se puede expresar como sigue:

$$Tiempo\ de\ terminación\ =\ tiempo\ de\ espera\ en\ la\ cola\ +\ tiempo\ de\ ejecución$$

- Otra medida de rendimiento que se puede usar es la aceleración del tiempo total de terminación. La aceleración, S, es una medida que depende de la aplicación, de las condiciones del sistema y de la política de asignación.

$$S\ =\ \frac{OSPT}{CPT}$$

$OSPT$ = Tiempo de Procesamiento Secuencial Optimo

CPT = Tiempo de Procesamiento Concurrente

- La eficiencia o porcentaje de utilización de un procesador también es una medida de la fracción de tiempo que el procesador es empleado útilmente, definida como el cociente de la aceleración entre el número de procesadores que forman parte del sistema. En un sistema paralelo ideal la eficiencia es igual a 1. En la práctica la eficiencia está entre 0 y 1, dependiendo del porcentaje del trabajo útil realizado por N procesadores. Así la eficiencia se define de la siguiente manera:

$$E\ =\ \frac{S}{N}$$

S = Aceleración

N = Número de procesadores

- Cuando se tiene un sistema heterogéneo compuesto de computadoras con características diferentes, es posible considerar a la carga de trabajo de un procesador como una medida de rendimiento. Estimando los tiempos de ejecución T_{ij} del proceso i en la computadora j, podemos definir a la carga de trabajo de un procesador como la suma de los tiempos de ejecución T_{ij} de las tareas Pi asignadas a la computadora M_j. Así la carga de trabajo se define como sigue:

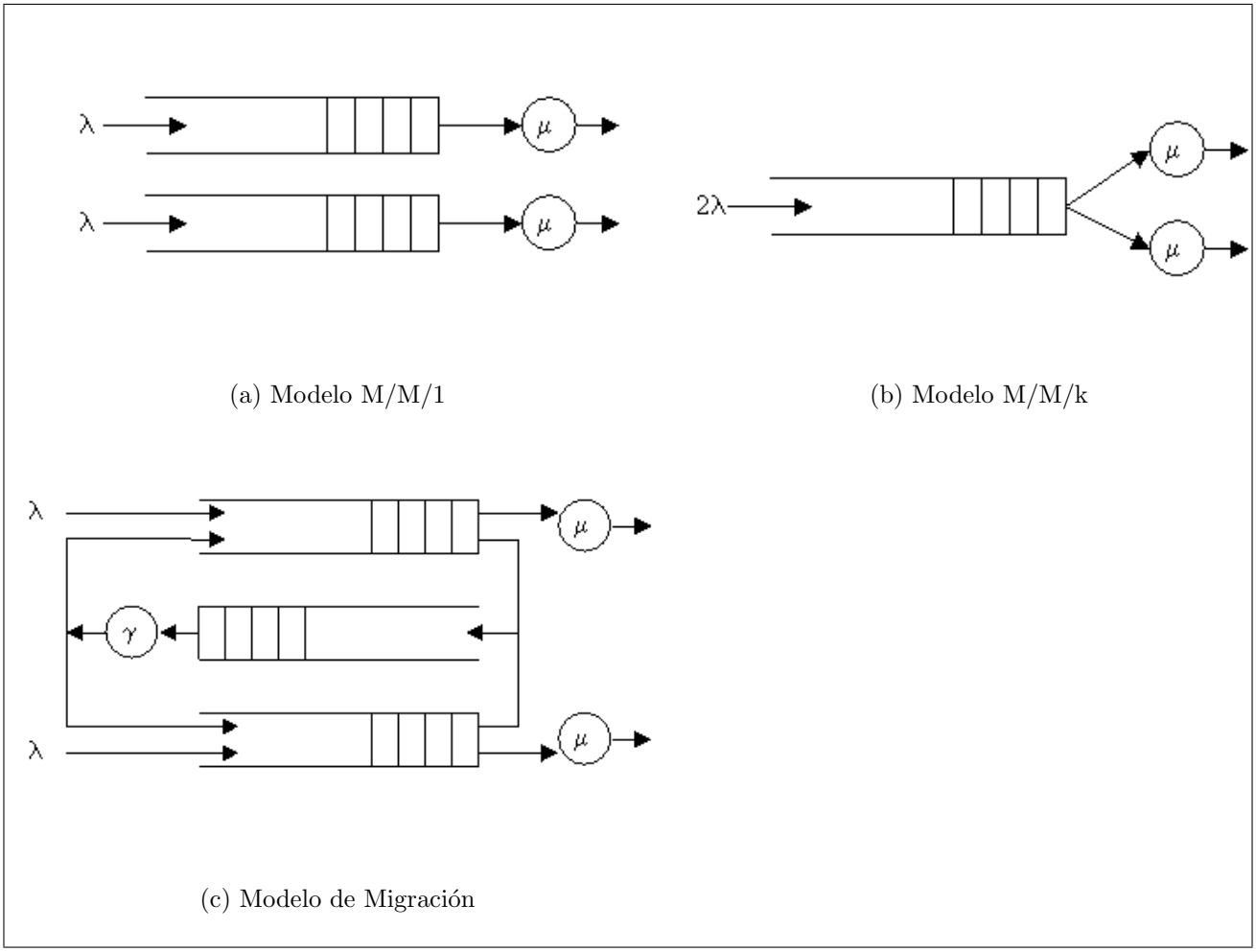
$$Carga\ de\ trabajo\ =\ \sum\ T_L\ ,\qquad T_L \in Conjunto\ de\ tareas\ asignadas\ a\ M_j$$

Para resolver el problema de balance de carga, es necesario tomar una medida para determinar la forma en que se asignarán las tareas que se van a ejecutar. En general, el tiempo de terminación,

la aceleración y la eficiencia son útiles para medir el rendimiento de un sistema después de que se han ejecutado las tareas. Nuestro interés es tener una medida que nos indique en donde realizar la asignación de forma dinámica, antes de que se inicie cualquier tarea. Para lo anterior, se debe considerar que la llegada de nuevas tareas depende de las necesidades de los usuarios, que el sistema es heterogéneo y, que las tareas a ejecutarse son independientes. Por lo tanto, la medida que se usa para resolver el problema de balance de carga, es la carga estimada de trabajo.

2.2 Modelos de balance de carga

Si se quiere ejecutar un conjunto de procesos aprovechando todos los recursos disponibles, se requiere de un balanceador de carga, de tal forma que la carga de trabajo se distribuya dinámicamente entre todos los procesadores de manera similar o equilibrada. Utilizando como medida de rendimiento la



(a) Modelo M/M/1

(b) Modelo M/M/k

(c) Modelo de Migración

Figura 2.2: Modelos de Balance de Carga

carga de trabajo, históricamente se han estudiado tres grandes modelos para abordar el problema de balance de carga [1]:

- **Modelo M/M/1.** En la Figura 2.2 (a) se muestra el modelo M/M/1, en este modelo cada procesador tiene su propia cola de espera. Una vez hecha la asignación los procesos deben ejecutarse en el procesador asignado y son atendidos en el orden en que llegaron. Este modelo se aplica en los supermecados en donde hay varias cajas (procesadores) y los clientes (procesos) se forman en donde probablemente van a ser atendidos más rápidamente.

- **Modelo M/M/k.** En la Figura 2.2 (b) se muestra el modelo M/M/k, para el caso de 2 procesadores, en este modelo sólo hay una cola de espera. Los procesos serán atendidos como van llegando realizando la asignación cuando alguno de los procesadores está disponible. En los bancos se usa este modelo; hay varias cajas (procesadores) y solo hay una cola en donde se van formando los usuarios (procesos) como van llegando y son atendidos conforme las cajas se van desocupando.

- **Modelo de migración.** En la Figura 2.2 (c) se muestra el modelo de migración, en este modelo cada procesador tiene su propia cola de espera, los procesos serán atendidos como fueron llegando pero, si al revisar el estado del sistema la decisión tomada ya no es la adecuada, es decir se ha desbalanceado el sistema, se efectuará la migración de procesos, los cuales podrán ser reasignados formándose en la cola adicional. Este modelo no es común encontrarlo en la vida real, sin embargo, es factible de aplicarse en sistemas paralelos y distribuidos.

$X/Y/c$**:** es una cola con un rango de llegada de procesos X, un tiempo de servicio Y y c procesadores

M**:** es una distribución Markoviana [1]

Si se toma como medida de rendimiento el tiempo de terminación de las tareas con relación a la carga general de un sistema, se pueden obtener las cotas superior e inferior para el tiempo de terminación promedio en los modelos $M/M/1$ y $M/M/2$.

Esto se puede expresar mediante las siguientes ecuaciones:

$$TT_1 = \frac{1}{\mu - \lambda}$$

$$TT_2 = \frac{\mu}{(\mu - \lambda)(\mu + \lambda)}$$

donde,

[1]Los modelos de Markov son modelos estocásticos para la representación de un proceso, los cuales generan una secuencia de propiedades probabilísticas de la secuencia de dicho proceso.

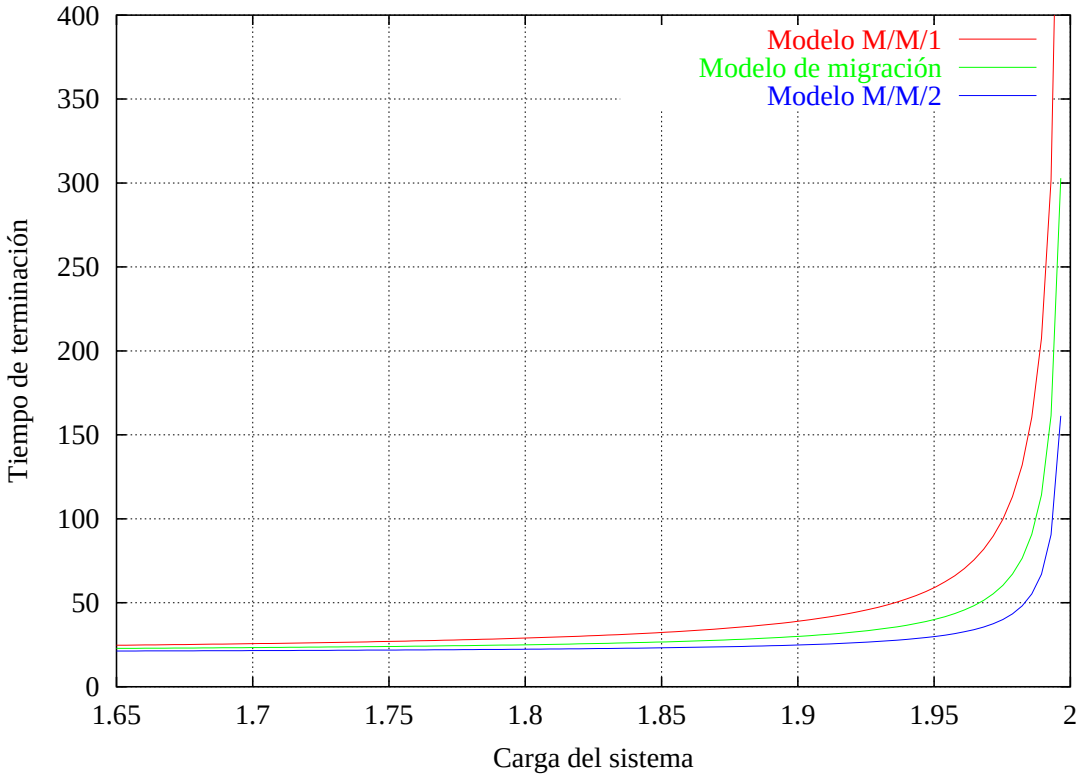


Figura 2.3: Comparación de rendimiento de los modelos de balance de carga

λ **y** μ**:** son los rangos de llegada y de servicio de cada nodo

γ**:** es el rango de migración el cual depende del ancho de banda del canal, el protocolo de migración de procesos y del contexto y estado de los procesos

En la Figura 2.3 se muestra una comparación del rendimiento de los tres modelos. Cuando la carga de trabajo es poca, con los tres modelos se tiene un buen rendimiento ya que los procesos son atendidos rápidamente. Sin embargo conforme la carga de trabajo se incrementa los tiempos de terminación también se incrementan y se puede ver que el modelo que tiene el mejor comportamiento es el modelo M/M/2 debido a que el proceso es asignado al primer procesador que se desocupe. El modelo con rendimiento más bajo es el modelo M/M/1 dado que los procesadores tienen que atender a los procesos que tengan en la cola de espera aún cuando la decisión de asignación ya no sea la adecuada. Puede suceder que algún procesador permanezca sin trabajo mientras otros estén ocupados. En el modelo de migración se tiene un buen rendimiento pues se trata de mantener a los procesadores ocupados y a la posibilidad de reasignar los procesos cuando algún procesador tenga una diferencia considerable de carga de trabajo en relación a los demás procesadores.

El modelo M/M/2 tiene el mejor rendimiento pero tiene la desventaja de que es necesario un controlador central que se encargue de indicar a que procesador se va a asignar el proceso que se va a ejecutar, entonces todos los procesadores dependerían de este controlador y podría originarse un cuello de botella, mientras los procesadores esperan a que el controlador pueda asignarles un nuevo proceso. Para el caso de sistemas distribuidos, esto no es particularmente adecuado dado que el tráfico hacia el controlador central crece con la cantidad de recursos a administrar. Para el caso del problema objeto de esta tesis se utilizará el modelo de migración de procesos.

2.3 Descripción formal del problema

Bajo condiciones ideales, en donde toda o casi toda la información del sistema, recursos y tareas es conocida, el problema de balance de carga se puede plantear de la manera siguiente:

Sean P y M un conjunto de tareas de simulación y un conjunto de computadoras, respectivamente.

$$P = \{\ P_1, P_2, ..., P_k\ \} \qquad y \qquad M = \{\ M_1, M_2, ..., M_l\ \}$$

La asignación del proceso P_i en la computadora M_j se puede expresar como una función de asignación f, que toma valor 1 si se realizó la asignación de P_i a M_j y 0 en otro caso.

$$f(P_i, M_j) = \begin{cases} 1 & \textit{si el proceso } P_i \textit{ es asignado a la computadora } M_j \\ 0 & \textit{en caso contrario} \end{cases} \tag{2.1}$$

Las funciones de asignación f posibles para este problema deben satisfacer la restricción de asignar un proceso a una sola computadora. Esto puede expresarse mediante la siguiente relación:

$$\sum_{j=1}^{l} \quad f(P_i, M_j) \ = \ 1\ , \qquad 1 \ \leq \ i \ \leq \ k \tag{2.2}$$

El conjunto de asignaciones de los procesos P_i en las computadoras M_j se puede expresar en forma matricial de la manera siguiente:

$$\begin{array}{c} \quad M_1\ M_2\ \ldots\ \ldots\ M_l \\ \begin{array}{c} P_1 \\ P_2 \\ \vdots \\ P_k \end{array} \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{2.3}$$

El proceso P_i puede ser asignado a cualquiera de las l computadoras, por tanto, se tienen l funciones que satisfacen la restricción (2.2) para el proceso P_i. Dado que es necesario asignar k procesos, entonces las funciones que satisfacen la restricción (2.2) son l^k. De todas ellas se busca una que proporcione el mejor balance de carga. Para poder formalizar lo anterior, es necesario definir una función que evalúe el costo de una asignación.

Considere un sistema heterogéneo en donde T_{ij} representa el tiempo que toma ejecutar la tarea P_i en la computadora M_j. El costo C_j^f, de la asignación f a la computadora M_j se puede expresar como:

$$C_j^f \ = \sum_{i=1}^{k} \ \ T_{ij} \ * \ f_{ij} \tag{2.4}$$

De esta manera, dada la asignación f se puede encontrar el costo de asignación en cada una de las l computadoras:

$$C_1^f \ , \ C_2^f \ , \ldots , \ C_l^f \tag{2.5}$$

Idealmente, en el problema de balance de carga se busca que los valores de C_j^f, $1 \leq j \leq l$ sean similares. Se define el costo de la asignación total, C^f, como el máximo de los costos de las asignaciones a las computadoras:

$$C^f \ = \ máx\{ \ C_j^f \ \} \tag{2.6}$$

Así, se puede plantear un problema de optimización en donde de todas las posibles funciones de asignación interesa aquella que tenga el costo mínimo:

$$f^* \ = \ min\{C^f\} \tag{2.7}$$

2.4 Estimación de los tiempos de ejecución en un sistema heterogéneo

Para solucionar el problema planteado en la sección anterior es necesario hacer estimaciones sobre los valores de T_{ij}, los cuales son los tiempos de ejecución de los procesos P_i en la computadora M_j, expresados en las mismas unidades. Sin embargo, dado que se está considerando un sistema heterogéneo, es necesario normalizar los valores a un mismo punto de referencia. Los valores de referencia a utilizar son el tiempo del proceso más ligero (el de menor tiempo de ejecución) y la velocidad de la computadora más lenta.

- Se estima el peso relativo del conjunto de procesos P mediante la función $W(P_i) = m$, y se asume que las cargas están normalizadas con respecto del proceso más ligero, P^L, entonces:

$$W(P^L) = 1 \;\; y \;\; W(P_i) = \frac{W(P_i)}{W(P^L)}$$

- Las computadoras M_1 M_2 $\ldots$ $\ldots$ M_l tienen características diferentes, por lo que se les asignan velocidades relativas con respecto a la computadora más lenta. Mediante la función $V(M_j) = r$, se asume que las velocidades están normalizadas con respecto del procesador más lento, M_k; entonces:

$$V(M_k) = 1 \;\; y \;\; V(M_j) = \frac{V(M_j)}{V(M_k)}$$

De este modo, las estimaciones de los tiempos de ejecución T_{ij} se pueden obtener de la manera siguiente:

$$T_{ij} = \frac{W(P_i)}{V(M_j)}$$

2.5 Estrategias de solución del problema

El problema de balance de carga es un problema de optimización. Existen diferentes estrategias para resolver este tipo de problemas, a continuación se muestran algunas.

- **Búsqueda exhaustiva.** Si se utiliza búsqueda exhaustiva para resolver este problema sería demasiado lento ya que entre más computadoras l y más tareas de simulación k se tengan, es mayor el número de posibles asignaciones, por lo que el tiempo para decidir cómo asignar una tarea a una computadora sería demasiado largo.

- **Algoritmos de aproximación.** Otra forma de resolver el problema es mediante algoritmos de aproximación como: *branch and bound*, búsqueda tabú, algoritmos genéticos, recocido simulado, etc [5]. Sin embargo, con este tipo de algoritmos generalmente no siempre se puede obtener una respuesta de manera rápida.

- **Heurísticas.** Dadas las condiciones del sistema, se requiere tomar decisiones rápidamente para realizar la asignación. Para tomar decisiones buenas, no necesariamente exactas, en un tiempo razonable, es conveniente el uso de heurísticas que encuentren soluciones aceptables en un tiempo razonable. Mario Farias Elinos [9] realizó estudios comparativos de diferentes métodos para resolver problemas de optimización, observando que los resultados obtenidos con los algoritmos heurísticos son satisfactorios, debido a que consumen menor tiempo de ejecución y el resultado es aceptable con respecto a otros métodos de optimización.

En el capítulo 3 se presentan las heurísticas utilizadas para resolver el problema de distribuir dinámicamente las tareas de simulación estre todos los procesadores disponibles junto con la descripción del sistema de balance de carga.

Capítulo 3

Sistema de balance de carga

Dado un conjunto de tareas de simulación a ejecutarse, es conveniente repartir cada tarea a los procesadores disponibles de tal manera que se obtenga un alto grado de utilización de los mismos, permitiendo, si es necesario, que los procesos se muevan de manera dinámica a procesadores con cargas ligeras. Para resolver este problema se ha diseñado un sistema de balance de carga. En este capítulo se presenta una descripción general de la implementación del sistema de balance de carga, las heurísticas y la infraestructura utilizadas, los algoritmos de balance de carga desarrollados, la migración de procesos y un simulador del sistema de balance de carga.

3.1 Infraestructura

La implementación del sistema de balance de carga se desarrolló en una plataforma de experimentación formada por dos computadoras de multiprocesamiento y cuatro computadoras con un solo procesador con las características siguientes:

- Características de las computadoras de un procesador
  - **Procesador:** Pentium III (Coppermine)
  - **Velocidad:** 833 MHz
  - **SO:** Red Hat Linux 7.1 2.96-79
- Características de las computadoras de multiprocesamiento
  - **4 Procesadores:** Pentium III (Cascades)
  - **Velocidad:** 750 MHz | 550MHz

Computadora	Velocidad	VN	NP	Factor
Presley	550MHZ	1	4	4
Elvis	750MHZ	1.36	4	5.45
Aqua[1..4]	833MHz	1.51	1	1.51

Benchmark	P. absoluto	P. relativo
test-lswlr	0.07	1
test-llong	0.23	0.29
test-fmath	0.24	0.3
test-math	0.81	1.01
test-printf	3.99	4.99
li	23.46	29.32
mgrid	2275.968	2844.96
hydro2d	3812.42	4765.52

Tabla 3.1: Valores estimados de velocidad y peso

Benchmark	Presley	Elvis	Aqua(1-4)
test-lswlr	0.25	0.18	0.66
test-llong	0.07	0.05	0.19
test-fmath	0.07	0.05	0.2
test-math	0.25	0.19	0.67
test-printf	1.25	0.92	3.3
li	7.33	5.38	19.36
mgrid	711.24	521.58	1878.42
hydro2d	1191.38	873.68	3146.51

Tabla 3.2: Tiempos de ejecución estimados

– **SO:** Linux NET4.0 for Linux 2.4

Debido a que se trata de una plataforma heterogénea, es necesario como se discutió en la sección 2.4, normalizar las velocidades (VN) con respecto al procesador más lento como se muestra en la Tabla 3.1 y considerar el número de procesadores por computadora (NP).

3.1.1 Aplicaciones utilizadas

La carga de trabajo en el sistema de balance de carga desarrollado es un conjunto de tareas de simulación de procesadores superescalares, algunas son aplicaciones reales, que forman parte de los benchmarks SPEC95 (ver Sección 1.5.2) y otras son pequeños programas de prueba en lenguaje C y precompilados para SimpleScalar. Los simuladores que se utilizan son todos los simuladores de SimpleScalar (ver Sección 1.3.2).

En la Tabla 3.1 se muestran los pesos de las tareas de simulación y los pesos relativos con respecto a la tarea más ligera.

Finalmente, los tiempos de ejecución estimados T_{ij}, se muestran en la Tabla 3.2. Los tiempos

de ejecucion se obtuvieron con los valores estimados de velocidad y peso (ver Tabla 3.1) como se definió en la sección 2.4.

3.2 Algoritmos de balance de carga

El problema descrito en el capítulo 2 relativo a balance de carga se traduce en un problema de optimización. Para tener un buen sistema de balance de carga es necesario tener soluciones de forma rápida. Por ello, la estrategia usada para solucionar el problema es el uso de heurísticas. Las heurísticas generales utilizadas son:

1. Asignar las tareas más pesadas (con el mayor tiempo de ejecución estimado) a los procesadores más eficientes.

2. Asignar un nuevo proceso a la computadora con la carga asignada más ligera. La carga de trabajo de una computadora está determinada por la suma de los tiempos de ejecución (estimados) de las tareas a ejecutarse en la cola (ver sección 2.1). Por lo tanto, como se trata de un sistema heterogéneo es necesario que sea expresada en medidas normalizadas.

Como estrategia para solucionar el problema de balance de carga primero se abordó el problema como un modelo determinista y se desarrolló el algoritmo de balance de carga estático. Posteriormente, se consideró el escenario real, donde las tareas son independientes y llegan en el momento en que un usuario realice la simulación de alguna aplicación. Por lo tanto, se desarrollaron los algoritmos de asignación de un nuevo proceso y terminación de un proceso. Finalmente, dado que los tiempos de ejecución son estimados, en algún momento los tiempos reales pueden ser mayores o menores en relación a los estimados por lo que el sistema podría desbalancearse. Por lo tanto, se desarrolló un algoritmo general de migración de procesos.

3.2.1 Balance de carga estático

En el problema de balance de carga estático es necesario realizar la asignación de la carga de trabajo a los procesadores disponibles antes de que se ejecute cualquier proceso. La asignación se realiza tratando de reducir los tiempos de terminación de procesos. Para realizar la asignación de los procesos, las cargas y las velocidades de las computadoras se normalizaron de acuerdo a los descrito en la sección 2.4. De esta forma, se desarrolló el Algoritmo 1 de balance de carga estático. Así, la carga de trabajo es un conjunto de procesos ordenados de forma decreciente, en tiempo de ejecución (tiempos estimados), del más pesado al más ligero. El conjunto de procesadores también

es ordenado de forma decreciente de acuerdo a su eficiencia (velocidades relativas), del procesador más veloz al procesador más lento. Dado que la carga y los procesadores son fijos y están ordenados de antemano no se considera tiempo de llegada y terminación de procesos. Las tareas de simulación (procesos) son independientes, tienen tiempos de ejecución diferentes (estimados) y el número de procesadores depende de su disponibilidad, el cual asigna los procesos en orden decreciente a las computadoras con menor carga. En caso de tener más de una posibilidad se considera la velocidad relativa de las computadoras y se elige la más veloz.

Algorithm 1 Algoritmo de balance de carga estático

Require: $P = \{P_i, P_{i-1}, \ldots, \}, M = \{M_j, M_{j-1}, \ldots, \}, W(P), V(M)$
Ensure: $f : P * M \rightarrow \{0, 1\}$
  Ordenar las tareas de forma decreciente: $\{P_i, P_{i-1}, \ldots, \}$
  Ordenar las computadoras de forma decreciente: $\{M_j, M_{j-1}, \ldots, \}$
  Definir $L_i = 0,\ 1 < i < k$ la carga actual de la computadora M_i
  while $P \neq \emptyset$ **do**
    Sea P_n = la tarea más pesada del conjunto P.
    $P \leftarrow P - \{P_n\}$ {Se elimina P_n del conjunto P}
    Sea M_t la computadora con la carga más ligera.
    $L_t \leftarrow L_t + T_{nt}$ {Se actualiza la carga de M_t agregándole el tiempo de ejecución de P_n}
    $f(P_n, M_t) = 1$ {Se realiza la asignación de P_n en la computadora M_t}
  end while

3.2.2 Balance de carga dinámico

En los sistemas paralelos y distribuidos no es realista suponer que se conocen las características de los procesos que se van a ejecutar. En general, no es posible conocer con exactitud los requerimientos de comunicación y cómputo. Debido a esto se debe establecer una calendarización adaptable, lo cual permita que las decisiones de asignamiento se realicen de manera local. Por lo tanto, la asignación de procesos debe hacerse de forma dinámica y descentralizada. En la asignación de procesos dinámica se considera el ingreso de nuevos procesos y la terminación de otros, así como la posibilidad de migrar los procesos de acuerdo al estado del sistema. Este problema también se aborda por medio de las heurísticas mencionadas antes. Para el balance de carga dinámico se desarrolló el Algoritmo 2 para cuando arriba un nuevo proceso al sistema y el Algoritmo 3 para cuando un proceso termina su ejecución. Cuando arriba un nuevo proceso se obtiene el estado del sistema y se determina cual es la computadora con la carga de trabajo más ligera, para realizar en ella la asignación del nuevo proceso. Si se presenta el caso de que dos o más computadoras tengan la misma carga se resuelve por la computadora más veloz. Después de asignado el proceso se procede a actualizar la carga

de trabajo de la computadora donde se realizó la asignación, agregándole el tiempo de ejecución (estimado) del nuevo proceso a su carga de trabajo. Cuando un proceso termina su ejecución, se actualiza la carga de trabajo de la computadora donde había sido asignado dicho proceso; se decrementa entonces el tiempo de ejecución (estimado) del proceso que ha terminado. El tiempo de ejecución real es conocido cuando la tarea ha terminado. Por lo tanto, no puede ser utilizado para determinar la carga de un procesador. En lugar de ello, se toma el tiempo estimado.

Algorithm 2 Algoritmo de asignación de un nuevo proceso al sistema

Require: P_n nuevo proceso,$M, W(P), V(M)$
Ensure: $f : P * M \rightarrow \{0, 1\}$
Sean L_i, $1 \leq i \leq k$, las cargas de las computadoras correspondientes.
Sea M_t = la computadora con la carga más ligera, en caso de empate, resolver por la más poderosa.
$f(P_n, M_t) \leftarrow 1$ {Se asigna P_n en la computadora M_t}
$L_t \leftarrow L_t + T_{nt}$ {Se actualiza la carga de M_t agregándole el tiempo de ejeución de P_n}
Terminar.

Algorithm 3 Algoritmo de terminación de un proceso

Require: P_n proceso que terminó, M, $f : P * M \rightarrow \{0, 1\}$
Ensure: $f(P_n, M_t) = 0$
Sea M_t = la computadora donde se asignó el proceso P_n.
$f(P_n, M_t) \leftarrow 0$ {Se elimina P_n de la carga de trabajo de M_t}
$L_t \leftarrow L_t - T_{nt}$ {Se actualiza la carga de M_t quitándole el tiempo de ejecución de P_n}
Terminar.

3.3 Migración de procesos

La migración de procesos es una facilidad para relocalizar de manera dinámica un proceso después de que fue asignado a una computadora particular para su ejecución. Dadas las condiciones del sistema, para la migración se prefiere tomar sólo los procesos de las colas y no los procesos que ya están en ejecución para no incrementar el trabajo adicional (*overhead*).

Basado en las políticas de distribución de carga se pueden desarrollar los siguientes tipos de algoritmos para la migración de procesos:

- **Iniciado por el emisor.** Si un proceso tiene una carga muy alta, tratará de compartirla con algún otro procesador. En este tipo de algoritmo la transferencia de procesos se hará usando un valor de umbral en la cola de procesos listos. Esto es, cuando la carga de trabajo

de algún procesador, es mayor que un valor de umbral, este proceso buscará algún receptor para realizar la migración de procesos. Este algoritmo es adecuado cuando la carga general del sistema es ligera, dado que el emisor encontrará eventualmente un nodo receptor para realizar la migración.

- **Iniciado por el receptor.** Si un proceso tiene una carga muy baja, ofrecerá sus servicios al sistema. Cuando la carga de trabajo de algún procesador es menor que un valor de umbral, este proceso buscará algún emisor para obtener carga de trabajo. Este tipo de algoritmo es adecuado cuando la carga del sistema es alta, porque el receptor eventualmente encontrará un emisor para incrementar su propia carga.

- **Híbrido**. Este algoritmo es una mezcla de los anteriores, permitiendo un papel activo de emisores y receptores. Cuando la carga global del sistema sea alta se iniciará por el receptor, en cambio cuando la carga sea ligera es adecuado iniciar por el emisor.

Algorithm 4 Algoritmo de migración de un proceso.

Require: P_n el proceso a migrar es el último que llegó, $M, W(P), V(M), T(periodo)$
$L_{inf} = Prom(L_{inf}) - Umb, L_{sup} = Prom(L_{inf}) + Umb$
Ensure: $f : P * M \rightarrow \{0, 1\}$
Obtener la carga actual L_i de cada computadora M_i
{Migración iniciada por el receptor}
while $L_i < L_{inf}$ **do**
Sea M_i la computadora con poca carga de trabajo
Sea M_m la computadora con mayor carga L_m
$L_m \leftarrow L_m - T_{nm}$ {Se actualiza la carga de M_m quitándole el tiempo de ejecución de P_n}
$L_i \leftarrow L_i + T_{ni}$ {Se actualiza la carga de M_i agregándole el tiempo de ejecución de P_n}
end while
{Migración iniciada por el emisor}
while $L_i > L_{sup}$ **do**
Sea M_i la computadora con mucha carga de trabajo
Sea M_m computadora con menor carga L_m
$L_m \leftarrow L_m + T_{nm}$ {Se actualiza la carga de M_m agregándole el tiempo de ejecución de P_n}
$L_i \leftarrow L_i - T_{ni}$ {Se actualiza la carga de M_i quitándole el tiempo de ejecución de P_n}
end while

El algoritmo 4 presenta la estrategia general de migración. Para saber en qué momento se debe iniciar la migración de procesos se verifica el estado del sistema en intervalos regulares de tiempo T. Evaluando si la carga de trabajo está dentro de los valores de umbral L_{inf} y L_{sup}, los cuales se definen considerando el promedio de las cargas de trabajo de las computadoras ± un valor de umbral. Así, si la carga de trabajo está por debajo del valor de L_{inf} se inicia la migración por el

receptor, si la carga de trabajo es mayor que L_{sup}, la migración es iniciada por el emisor. En otro caso, significa que el sistema está balanceado y no es necesario migrar procesos. Después de realizar la migración es posible que el sistema continúe desbalanceado pero es más fácil y rápido esperar a que el supervisor después de un tiempo, T, lo detecte e iniciar nuevamente la migración de procesos que tratar de desarrollar un algoritmo complicado que trate de balancear en una sola etapa todo el sistema. Los estudios de simulación realizados y que se presentan en la siguiente sección muestran que esto fue suficiente para tener un sistema balanceado.

3.4 Simulador del proceso de balance de carga

La simulación es un proceso a través del cual una actividad real se puede representar con un programa de computadora, evaluar su rendimiento y/o desempeño para la toma de decisiones o para obtener tiempos comparativos entre dos sistemas. Antes de implementar el sistema de balance de carga se desarrolló un simulador del sistema para validar los algoritmos propuestos. El simulador fue desarrollado con la herramienta de simulación Arena [18].

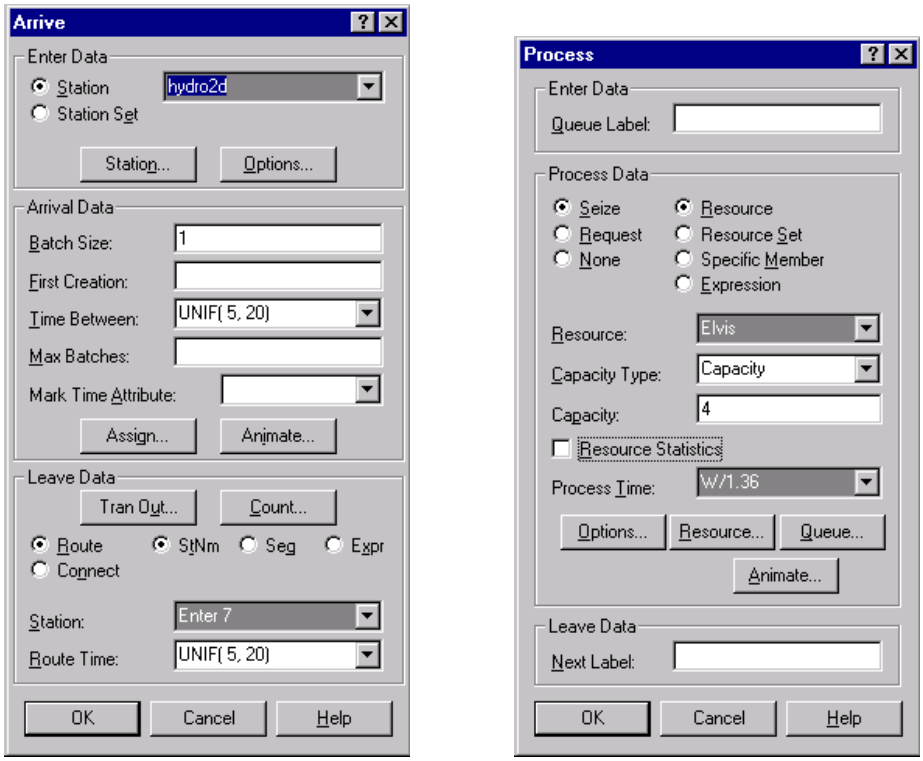
3.4.1 Modelo de simulación

El modelo de simulación es una representación válida que se utiliza para conocer el comportamiento del sistema. El propósito es que se trabaje con un programa de computadora y no con el sistema real. Así generalmente es más fácil, barato y rápido obtener respuestas pues es más fácil manipular los datos de entrada de una simulación que del sistema real. Más aún, con un simulador es posible generar algunas situaciones que en la realidad sería difícil de producir; el hacerlo llevaría mucho tiempo, el rendimiento de las computadoras se decrementaría y los usuarios podrían resultar afectados.

Elementos del simulador de balance de carga

Un modelo de simulación requiere diferentes elementos para su desarrollo. En el simulador del sistema de balance de carga se utilizaron los siguientes:

- Entidades. Las entidades en nuestro sistema son las tareas de simulación a ser ejecutadas (*benchmarks*). Las entidades se representan con el módulo *Arrive* como se muestra en la Figura 3.1 (a). Este módulo está dividido en tres partes. En la primera, se tienen los datos de entrada, a la derecha se indica el nombre del *benchmark*. En la figura se refiere al *benchmark hydro2d* y en este caso se indica que se trata de una estación. Cuando se indica que es un



(a) Benchmarks (b) Computadoras

Figura 3.1: Elementos del Simulador de Balance de Carga

conjunto de estaciones los datos de entrada van hacia un submodelo de estaciones (macro). En la segunda parte, se específica el número de entidades que llegan por lote. Para definir el tiempo de llegada entre un lote y otro se usa una función de distribución uniforme con rango $[a, b] = [5, 20]$ y $media = (a + b)/2$, esto es con el fin de simular que las tareas llegan en diferentes tiempos. Al seleccionar *route*, en la tercera parte se indica que la entidad va a ser transferida a través de una ruta del módulo *Arrive* al módulo siguiente. Con *StNm* se indica que la entidad es transferida a la estación especificada *Enter 7*. *Route time* es el tiempo que le va a llevar a la entidad en llegar a su destino, para ello se usó la misma función que en la segunda parte. Una descripción similar se realizó para cada uno de los *benchmarks* simulados.

- Variables.

  Las variables son información que reflejan algunas características del sistema. Las variables que se usaron son las que acumulan la carga de trabajo de cada una de las computadoras, $S_1, \ldots, S_6$, las de factor de carga, $F_1, \ldots, F_6$, el período de tiempo en que el supervisor recolecta el estado del sistema, T, y los límites para verificar si es necesario iniciar la migración de

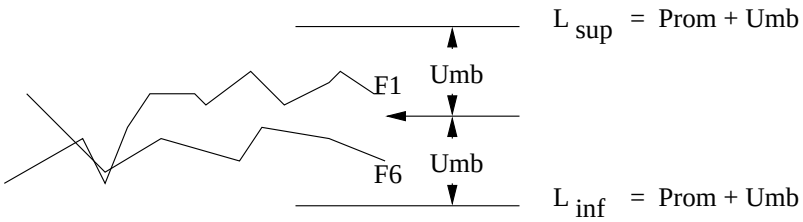


Figura 3.2: Límites para los cuales un sistema se encuentra balanceado

procesos, L_{inf} y L_{sup}. La actualización de las cargas de trabajo y los factores de carga se realiza de la forma siguiente, a la carga de trabajo S_i, con $1 < i < l$ se le agrega el tiempo de ejecución (estimado) del *benchmark* que se va a ejecutar: $S_1 = S_1 + W$ y el factor de carga está determinado como sigue: $F_i = S_i \ / \ (V_i * NP)$, donde S_i es la carga de la computadora, V_i es su velocidad relativa y NP es el número de procesadores de la computadora. En la Figura 3.2 se observa la forma en que se calcularon los límites. Después de un tiempo T, se obtienen los factores de carga, $F_1, \ldots, F_6$, se calcula el promedio, $Prom$, y se obtiene un valor de umbral con una distribución de Poisson, $Umb = POIS(\lambda)$, con $\lambda = Prom$.

Los límites se calculan de la manera siguiente:

$$L_{inf} = Prom - Umb$$
$$L_{sup} = Prom + Umb$$

- Atributos. Un atributo es una característica de una entidad, pero con un valor específico que puede ser diferente para cada entidad. En el sistema el atributo que se definió es el peso de las tareas, W, se le asignó a cada uno de los *benchmarks* el tiempo de ejecución estimado como se explica en la sección 2.4.

- Recursos. Un recurso puede representar un grupo de varios servidores, llamados cada uno, una unidad de ese recurso. En nuestro sistema el recurso es el grupo de las seis computadoras, *Aqua1, ..., Aqua4, Elvis y Presley.* En la Figura 3.1 (b) se muestra como se definió la computadora Elvis como unidad del recurso. Para esto se uso el módulo *Process* que está dividido en tres partes. En la segunda parte con *Seize* se indica que el recurso es requerido por las entidades para su procesamiento. En *Resource*, se da el nombre del recurso, en el caso de la unidad de la figura se trata de *Elvis*, con la capacidad de procesar 4 entidades al mismo tiempo. En *Process time:* se define el tiempo que le lleva al recurso procesar las entidades, $W/1.36$. W es el peso de las tareas y para el caso de la figura la velocidad relativa de la computadora *Elvis* es 1.36 (ver sección 2.4).

- Colas. Son necesarias para cuando una entidad (tarea) tenga que esperar a ser atendida, tienen nombres y es posible indicar su capacidad. En el sistema se tiene una cola para cada computadora y una cola para que las entidades esperen la decisión del balanceador para ser asignadas.

- Eventos. En el sistema se usan dos tipos de eventos. *Arrival*; para cuando llega una nueva tarea y *Departure*; para cuando una tarea termina su ejecución y abandona el sistema.

- Reloj. El valor de tiempo en la simulación es soportado en una variable llamada *simulation clock*. No es similar al tiempo real, el reloj de simulación no fluye continuamente, si no hay cambios entre los eventos no cambia el tiempo [11].

3.4.2 Construcción del modelo

En el simulador mostrado en la Figura 3.3, a los procesos (*benchmarks*) se les asignan pesos relativos de acuerdo al tiempo de ejecución en la máquina más lenta, se usaron funciones de distribución uniforme para representar el tiempo que requiere generar una tarea después de otra y el tiempo que tarda en llegar al balanceador de carga. Cuando un nuevo proceso llega el balanceador revisa la carga de trabajo de cada computadora y realiza la asignación en la computadora que tenga la carga más ligera, en caso de tener procesadores con cargas iguales se resuelve por la computadora más eficiente.

El simulador cuenta con un supervisor, el cual verifica el estado del sistema dado un intervalo de tiempo, éste calcula los límites inferior y superior que determinan si la carga está balanceada o no, dependiendo de los pesos relativos de las tareas. Si se determina un desbalance, se realiza la migración usando el Algoritmo 4 de migración de procesos (ver sección 3.4).

Los contadores con números blancos indican la carga de trabajo en cada computadora, y los otros contadores indican un factor de carga considerando la velocidad y el número de procesadores. En las computadoras de multiprocesamiento se indica con otro contador el número de procesadores que se encuentran atendiendo tareas en ese momento.

Migración de procesos

Para probar el algoritmo de migración de procesos fue necesario forzar al sistema a desbalancearse (ver Figura 3.4). Para ello, en el módulo *Process* se modificó *Process time*. Por ejemplo, en la computadora *Presley* su tiempo de procesamiento es el tiempo de ejecución estimado, W, entonces se cambió por una función de distribución uniforme con rango $[a, b] = [W/2, W * 2]$, dando como resultado que el procesador terminará antes o después del tiempo estimado la ejecución de las

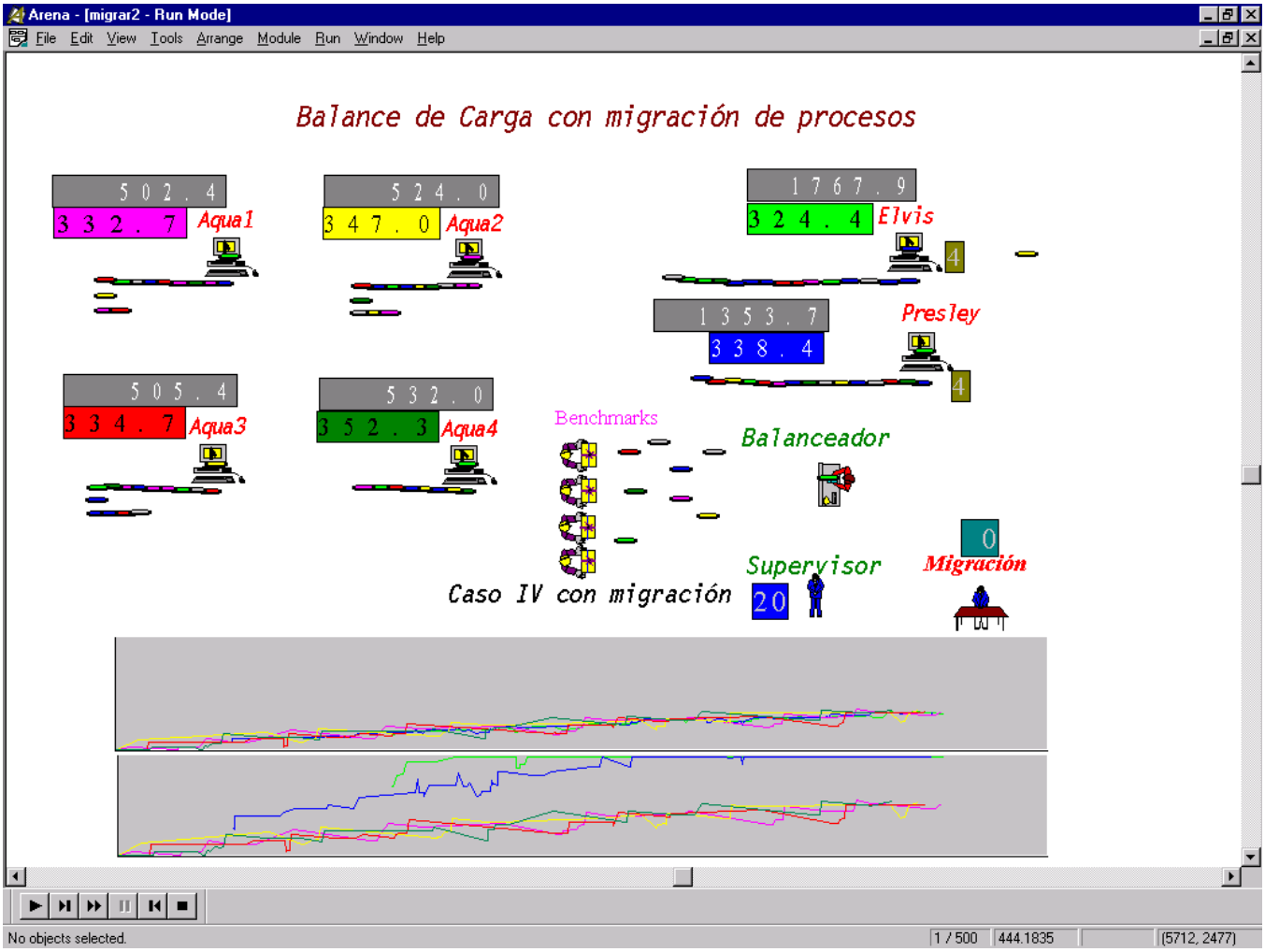


Figura 3.3: Simulador de Sistema de Balance de Carga con Migración de Procesos.

tareas. También, se usó una función de distribución uniforme con rango $[a, b] = [1, 4]$ para variar la capacidad de procesamiento en las computadoras de multiprocesamiento.

Cuando se determina que es necesario realizar la migración de procesos se procede a elegir el proceso a migrar. Si la migración se inició por el emisor se busca cual es la computadora que tiene la carga más baja para que ahí sea reasignado el proceso, en el caso de que la migración sea iniciada por el receptor se busca cual es la computadora con la carga más alta, de ahí se toma el proceso a migrar. El proceso a migrar es el último proceso en llegar.

3.4.3 Resultados

En la Figura 3.3 se muestra el sistema de balance de carga de acuerdo a los algoritmos presentados en las secciones anteriores. La computadora con la carga de trabajo más alta es la computadora

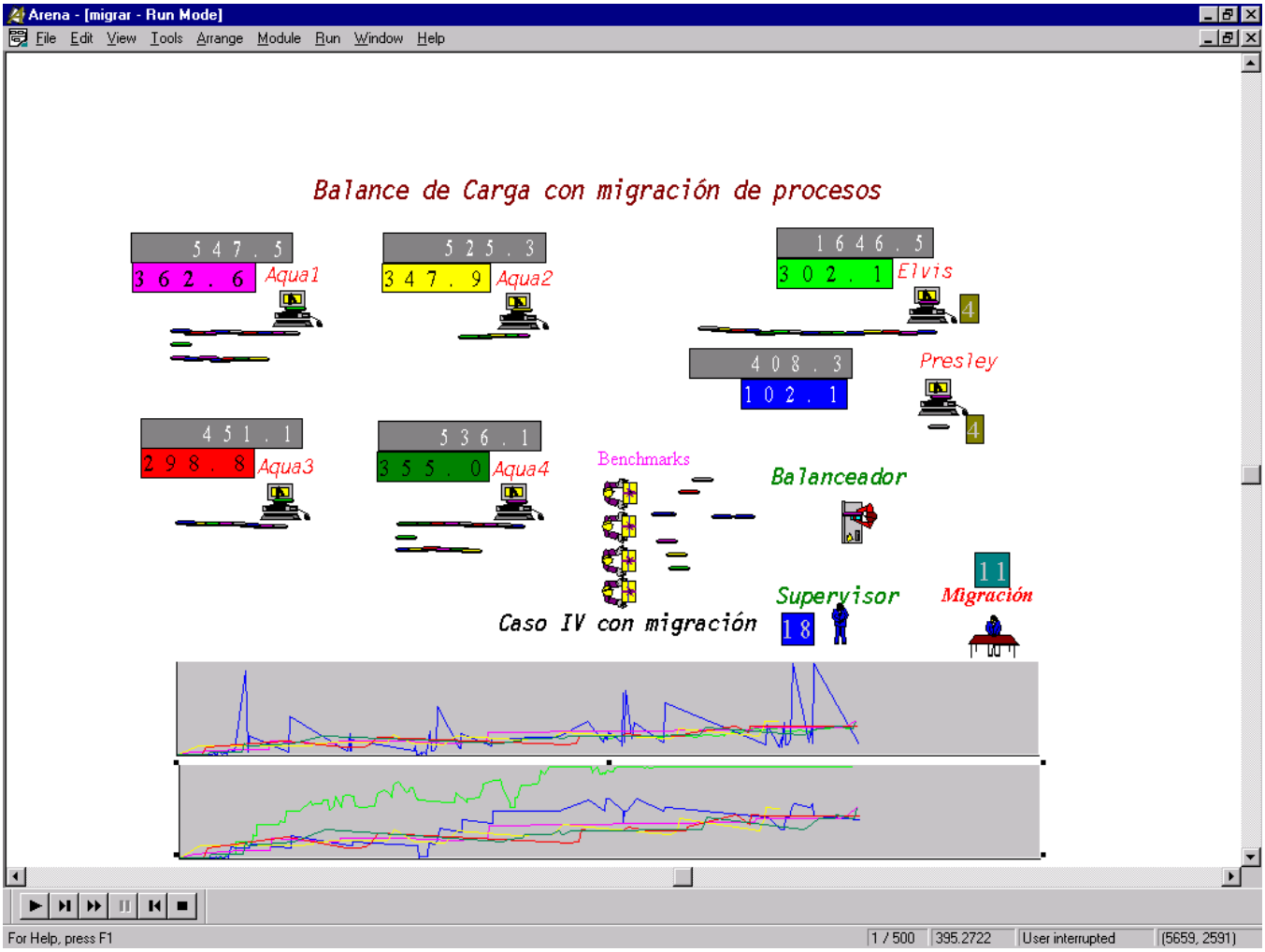


Figura 3.4: Sistema de Balance de carga forzado para probar migración de procesos

Elvis esto es debido a que se trata de una computadora con cuatro procesadores que además son más veloces que los de la computadora *Presley* (ver sección 3.2). Sin embargo, si se observan los contadores del factor de carga se puede visualizar fácilmente que el sistema está balanceado.

Para visualizar mejor el estado del sistema se generaron dos gráficas: la primera, corresponde al factor de carga, en donde se considera la velocidad relativa de las computadoras (ver sección 2.4) y el número de procesadores. Por lo tanto, en esta gráfica se puede visualizar que el sistema está balanceado. En la segunda gráfica se muestra la carga total de las computadoras, se pueden observar dos líneas que están por arriba de las demás, esto es porque corresponden a las cargas de trabajo de Elvis y Presley, debido a que las computadoras tienen cuatro procesadores y se les asignan cargas de trabajo mayores.

En la Figura 3.4 se muestra el sistema de balance de carga desbalanceado, se observa que de 18 veces que el supervisor revisó el estado del sistema 11 veces fue necesario migrar procesos. La

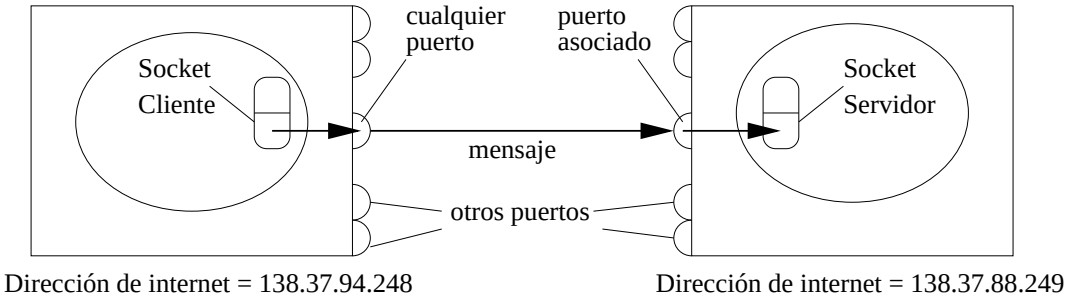


Figura 3.5: Sockets y puertos

primera gráfica, presenta la variación de carga de cada procesador, los picos indican que se ha presentado un desbalance de carga en la computadora Presley, por tanto, se inicia la migración de procesos y el sistema se balancea nuevamente. En la segunda gráfica se puede ver que la carga de Presley ya no es tan alta como en la Figura 3.3 pues se está forzando a comportarse de manera diferente que en condiciones normales.

Las simulaciones realizadas mostraron que el algoritmo de migración descrito en la sección 3.4 fue suficiente para mantener al sistema balanceado.

3.5 Sistema de balance de carga

El sistema de balance de carga fue implementado en la plataforma heterógenea descrita en la sección 3.1. En esta sección se discutirán algunos detalles sobre la implementación del mismo.

3.5.1 Comunicación cliente-servidor

La comunicación entre procesos se realiza con un modelo cliente-servidor por medio de sockets basados en TCP (Transmission Control Protocol) que son orientados a conexión y los datos son transferidos sin guardarlos en registros. Los sockets deben inicializarse definiendo quien es el servidor y quien es el cliente; ya que se establece la comunicación cualquier socket envía y recibe datos. Para que los procesos envíen y reciban mensajes los sockets deben estar vínculados a puertos locales y a las direcciones IP de las computadoras en donde se ejecutan (ver Figura 3.5). Por lo tanto, se utiliza un puerto para cada servidor, de tal forma que los procesos comparten el sistema de archivos [6].

Cuando ocurre un error en la comunicación, los procesos que usan los sockets son informados de tal error. Si alguna de las computadoras ya no está disponible (por ejemplo, cuando el usuario la

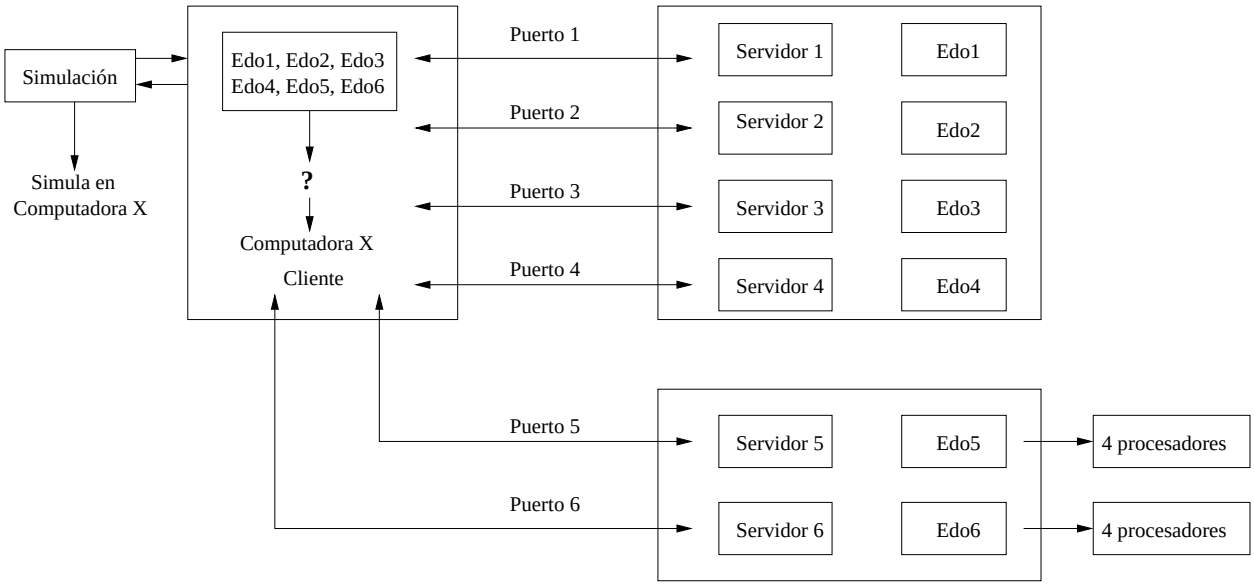


Figura 3.6: Esquema cliente-servidor para el problema de balance de carga.

apagó o la reinició con otro sistema operativo), al tratar de realizar la conexión esto es detectado, por tanto se le da de baja del sistema hasta que este disponible nuevamente.

3.5.2 Implementación del sistema de balance de carga

El sistema de balance de carga está formado por un cliente y n servidores (computadoras disponibles), es decir se trata de un modelo cliente-servidor orientado al servidor (ver Figura 3.6). El cliente se localiza en la computadora donde se requiere ejecutar una tarea de simulación, desde donde se envía una solicitud a cada uno de los servidores que se encuentran en cada una de las computadoras que forman parte del sistema. En cada servidor se activa un demonio que devuelve el estado de la computadora, su carga de trabajo. Se recolecta así el estado de todo el sistema y se ejecuta la tarea de simulación en la computadora con la carga más ligera, en caso de empate se resuelve por la computadora más eficiente. Todo esto es transparente al usuario; para poder ejecutar una tarea de simulación sólo tiene que escribir el nombre del programa *simulador*, el simulador que va a utilizar (ver sección 1.3.2) y la tarea de simulación que quiere ejecutar (ver sección 1.5.2). Por ejemplo, si se quiere medir el rendimiento de la memoria caché con una aplicación de aritmética de punto flotante, por ejemplo *mgrid*, el usuario sólo debe escribir en la línea de comandos lo siguiente:

$$. \setminus simulador\ sim - cache\ mgrid$$

Se realizó, por tanto, un sistema de balance de carga con migración de procesos, donde para realizar la asignación de procesos se tomaron pesos y velocidades relativas para estimar los tiempos de ejecución T_{ij} pero considerando también los tiempos reales. A continuación se muestra el pseudocódigo del servidor (*Server*), del cliente (*Client*) y el demonio (*Daemon*):

Server

INICIO
// Declaración de variables
entero puerto
entero tipo **// (1/2) 1:consulta de la carga, 2: actualizar carga**
entero estado **// (1/2) 1:Inicia proceso, 2:Termina proceso** real carga
// Crea socket y espera solicitudes
LEER puerto
$CREAR\_socket(puerto)$
ESCUCHAR
if (Petición)
    {conexión(puerto)}
else
    {$CERRAR\_socket$
    SALTO FIN}
// Se realiza la conexión
conexión(puerto)
// Se obtiene del cliente el tipo y el estado
$LEER\_socket(tipo, estado)$
// Activa demonio para obtener la carga de la computadora
carga = daemon(puerto,tipo,carga)
// Envía la carga al cliente
$ESCRIBE\_socket(carga)$
FIN

El **Server** se mantiene en espera de solicitudes por parte de **Client**. Cuando hay una solicitud realiza la conexión y activa al **Daemon** que es quien obtiene las cargas de todas las computadoras. Finalmente, envía las cargas a **Client**.

Client verifica cuales computadoras están disponibles y envía solicitudes a los **Servers** de estas computadoras, para obtener el estado del sistema a través de *sockets*. Una vez que lo tiene, busca cuál es la computadora, **M**, que tiene la carga más ligera y ejecuta la tarea de simulación en esta computadora.

Client

INICIO

// Declaración de variables

entero puerto,mq

entero tipo **// (1/2) 1:consulta de la carga, 2: actualizar carga**

entero estado **// (1/2) 1:Inicia proceso, 2:Termina proceso**

real carga, carga[i], $1 < i < 6$

cadena maquina

if(tipo=1){ **//Lee** Cargas(1,estado)

//Busca la computadora con la carga más ligera

min=mínimo(carga[1],...,carga[6])

M = computadora con la carga más ligera }

EJECUTA tarea de simulación en la computadora M

else { **if**(tipo=2){ **//Actualiza las cargas**

Cargas(2,estado)

$CERRAR\_socket$ }}

//Obtiene el estado general del sistema (la carga de trabajo de todas las computadoras)

Cargas(tipo,estado)

$carga[1] = edo\_sistema(puerto1, maquina1, tipo, estado)$

$carga[2] = edo\_sistema(puerto2, maquina2, tipo, estado)$

$carga[3] = edo\_sistema(puerto3, maquina3, tipo, estado)$

$carga[4] = edo\_sistema(puerto4, maquina4, tipo, estado)$

$carga[5] = edo\_sistema(puerto5, maquina5, tipo, estado)$

$carga[6] = edo\_sistema(puerto6, maquina6, tipo, estado)$

edo_sistema(puerto,maquina,tipo,estado)

// Envía solicitud al servidor

if(Acepta)

{ $CREAR\_socket(puerto)$ }

// Verifica cuales computadoras están disponibles

if(máquina no disponible)

{ $mq = -1$ } **computadora dada de baja**

// Envía el tipo y el estado al servidor

$ESCRIBE\_socket(tipo, estado)$

// Recibe la carga del servidor

$LEER\_socket(carga)$

FIN

Daemon

INICIO

// Declaración de variables

entero puerto

entero tipo **// (1/2) 1:consulta de la carga, 2: actualizar carga**

entero estado **// (1/2) 1:Inicia proceso, 2:Termina proceso**

real carga, carga[i], $1 < i < 6$

cadena bench **//Benchmark a ejecutarse**

real Tbench[j], $1 < j < 18$

// Verifica si existe el benchmark

// Asigna el tiempo de ejecución estimado

Tbench[j] = benchmark[bench]

// Obtiene la carga de la computadora conectada a puerto

if(tipo = 1){ **switch**(puerto){

caso puerto1: $LEE\_archivo(carga1)$

caso puerto2: $LEE\_archivo(carga2)$

$\vdots$

caso puerto6: $LEE\_archivo(carga6)$} }

//Actualiza la carga

if(tipo=2){ **switch**(puerto){

caso puerto1: **if**(estado=1)

$\{carga = carga[1] + Tbench[j]\}$

else { **if**(estado = 2) $\{carga = carga[1] - Tbench[j]\}\}$

$ESCRIBE\_archivo(carga1)$

caso puerto2: **if**(estado=1)

$\{carga = carga[2] + Tbench[j]\}$

else { **if**(estado = 2) $\{carga = carga[2] - Tbench[j]\}\}$

$ESCRIBE\_archivo(carga2)$

$\vdots$

caso puerto6: **if**(estado=1)

$\{carga = carga[6] + Tbench[j]\}$

else { **if**(estado = 2)

$\{carga = carga[6] - Tbench[j]\}\}$

$ESCRIBE\_archivo(carga6)$

}

FIN

El **Daemon** es activado por el **Server**. Cuando se activa obtiene la carga de trabajo de cada una de las computadoras disponibles y asigna el tiempo de ejecución del *benchmark* que se va a ejecutar dependiendo de la computadora de acuerdo a la Tabla 3.2. En **Daemon** también se actualizan las cargas cuando inicia o termina una tarea de simulación.

Capítulo 4

Simulador paralelo para SimpleScalar

Mediante la simulación secuencial convencional es posible simular pequeños programas de prueba. Sin embargo, la simulación de una aplicación grande requiere de un tiempo de simulación muy largo. Una manera de superar esta limitante es mediante el uso de varios procesadores, es decir de computadoras paralelas [13]. De esta manera, cuando se tiene una tarea de simulación suficientemente grande el trabajo a realizar se distribuye entre varios procesadores. Para ello, se desarrolló un simulador paralelo del funcionamiento del procesador superescalar. En este capítulo se presenta una descripción general de las arquitecturas paralelas, y como la programación paralela como una forma de construir aplicaciones que trabajen en computadoras de multiprocesamiento para memoria compartida. Con estos antecedentes, se presenta entonces la estrategia de particionamiento, la orquestación y las principales estructuras de datos globales del simulador paralelo.

4.1 Arquitecturas paralelas

Una computadora paralela es "una colección de elementos de procesamiento que se comunican y cooperan entre sí para resolver problemas grandes de manera rápida" (Almasi and Gottlieb 1989) [7]. De acuerdo a esta definición, se puede ver a la arquitectura paralela como la extensión de una arquitectura de computadora convencional. Una arquitecura de computadora tiene dos facetas. En la primera, se definen abstracciones críticas, esencialmente los límites entre el hardware y el software y los límites entre el usuario y el sistema. En la segunda faceta, se define la organización de la estructura que realiza estas abstracciones para dar alto rendimiento. Escencialmente, la arquitectura paralela extiende los conceptos usuales de una arquitectura de computadoras con una arquitectura de comunicación. En la arquitectura de comunicación, en la primera faceta se definen la comunicación básica y las operaciones de sincronización. En la segunda faceta, se define la

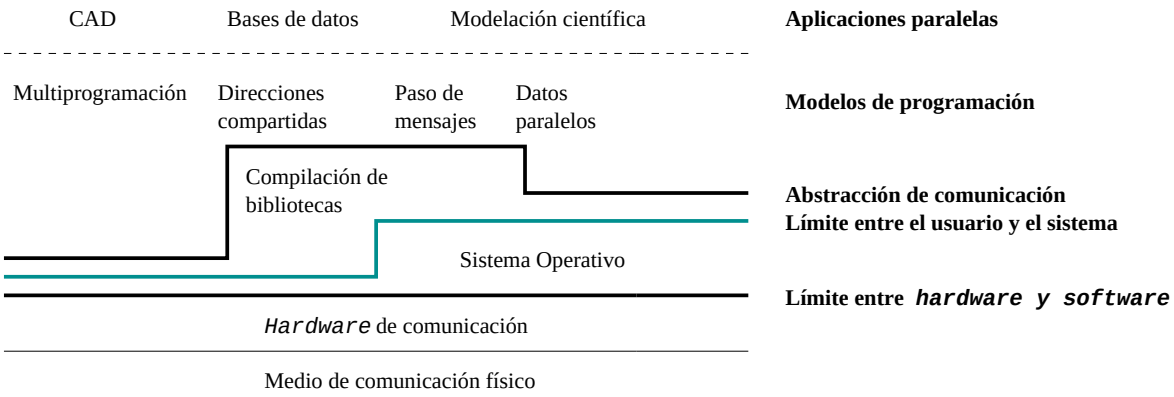


Figura 4.1: Capas de abstracción en arquitecturas de computadoras paralelas

organización de la estructura que realiza estas operaciones. En la Figura 4.1 se muestra que el límite entre el *hardware* y el *software* es fijo para indicar que el *hardware* disponible en diferentes diseños es más o menos de complejidad uniforme. Debido al desarrollo de los compiladores y bibliotecas ya no es necesario que el *hardware* sea esencialmente idéntico al modelo de programación.

Hay 3 modelos principales de programación paralela:

- Direcciones compartidas. Es similar a usar un boletín, donde se pueden comunicar uno o más colegas enviando información a localidades compartidas conocidas. Las actividades individuales son orquestadas notificando quién está haciendo cada tarea.

- Paso de mensajes. Es similar a llamadas telefónicas o cartas, las cuales transmiten información desde un remitente específico a un receptor específico. Estos son eventos bien definidos donde la información es enviada o recibida, así las actividades individuales son orquestadas por medio de estos eventos. Sin embargo, las localidades no compartidas son accesibles a todos.

- Procesamiento paralelo de datos. Está basado en la cooperación, donde varios agentes realizan una acción sobre elementos separados de un conjunto de datos de forma simultánea e intercambian información globalmente. La reorganización global de datos puede ser a través de accesos a direcciones compartidas o mensajes, ya que el modelo de programación sólo define el efecto global de las etapas paralelas.

La *abstracción de comunicación* se refiere a las primitivas de comunicación a nivel de usuario utilizadas para realizar el modelo de programación. Típicamente el modelo de programación se incorpora al lenguaje de programación, donde las primitivas están dadas directamente por el *hardware*, por el sistema operativo o por el *software* del usuario.

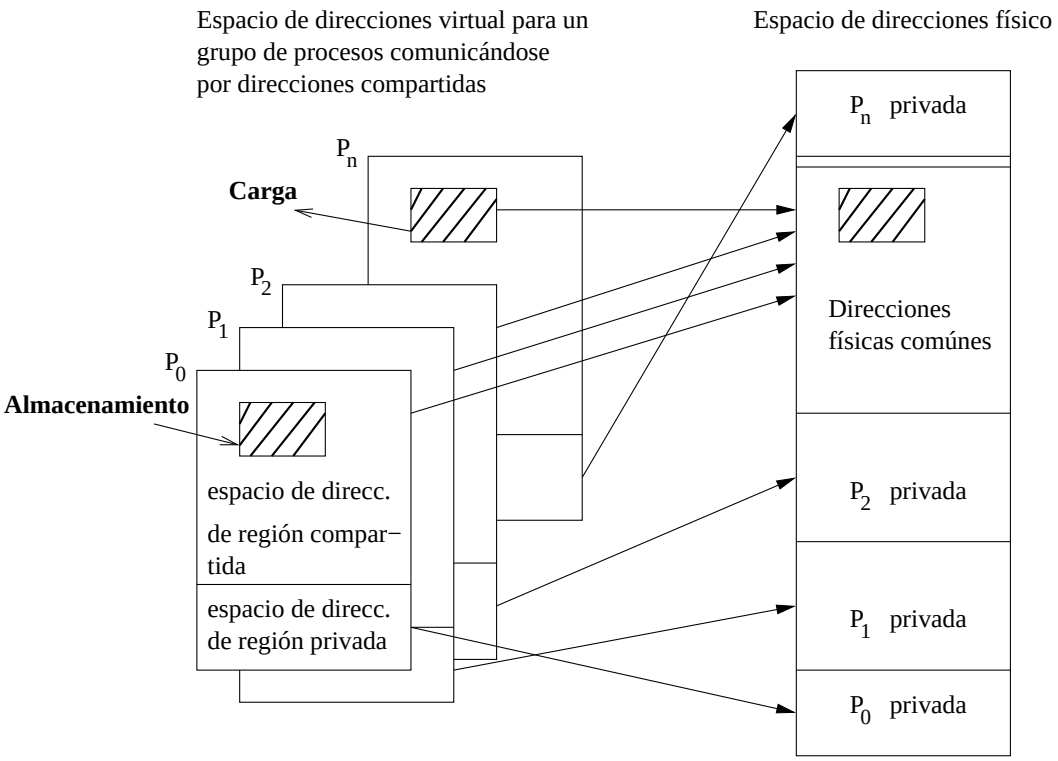


Figura 4.2: Modelo de memoria típico para programas paralelos de memoria compartida

4.1.1 Memoria compartida

Una de las clases más importantes de computadoras paralelas, son los multiprocesadores de memoria compartida. La propiedad principal en éste tipo de computadora es que la comunicación ocurre implícitamente como resultado de un acceso convencional a la memoria de instrucciones (carga y almacenamiento). En la Figura 4.2 se muestra un grupo de procesos que tienen una región común de direcciones físicas direccionadas hacia su espacio de direcciones virtual. Además los procesos tienen una región privada que contiene el *stack* y los datos privados.

Formalmente, un proceso es la instancia de un programa en ejecución el cual reside en un espacio de direccionamiento virtual y consiste de uno o más hilos (*threads*) de control. La cooperación y coordinación entre *threads* se consigue a través de lectura y escritura a variables compartidas y por apuntadores haciendo referencia a direcciones compartidas. La arquitectura de comunicación usa las operaciones a memoria convencional para proveer la comunicación através de direcciones compartidas como operaciones atómicas especiales para sincronización. Los procesos completamente independientes comparten la parte del núcleo del espacio de direcciones, aunque sólo es accedido por el sistema operativo. Sin embargo, el modelo del espacio de direcciones compartidas es utilizado dentro del sistema operativo para coordinar la ejecución de los procesos. Aunque la memoria

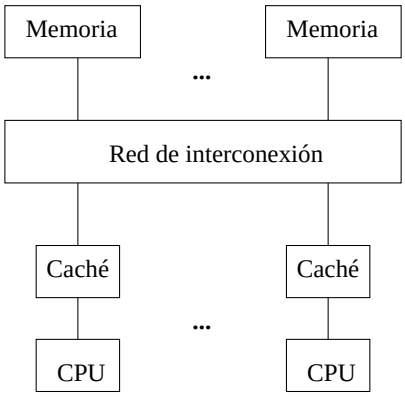


Figura 4.3: Estructura de multiprocesadores de memoria compartida

compartida puede ser usada para comunicación entre una colección arbitraria de procesos, la mayoría de los programas paralelos están completamente estructurados para usar el espacio de direcciones virtual. Esta simple estructura implica que las variables privadas en el programa estén presentes en cada proceso y que las variables compartidas tienen la misma dirección en cada *thread*.

La comunicación del *hardware* en multiprocesadores de memoria compartida es una extensión natural del sistema de memoria en la mayoría de las computadoras. Esencialmente, todos los sistemas de computadora utilizan un procesador y un conjunto de controladores de I/O para acceder a los módulos de memoria a través de algún tipo de interconexión de *hardware*. La capacidad de la memoria se incrementa simplemente agregando módulos de memoria. Esta capacidad puede o no incrementar el ancho de banda de memoria disponible, dependiendo de la organización del sistema específico. Hay dos formas posibles de incrementar la capacidad de procesamiento: Tener un procesador más rápido disponible o agregar más procesadores. Con más procesadores más procesos se pueden ejecutar cada vez y el *throughput* se incrementa (ver sección 1.5). Si una aplicación sencilla es programada para hacer uso de varios *threads*, el usar más procesadores puede acelerar la aplicación.

En la Figura 4.3 se muestra la estructura de las computadoras de multiprocesamiento de memoria compartida, los procesadores y los módulos de memoria están conectados por medio de una red de interconexión. Los procesadores comparten la memoria principal, pero tienen su propia memoria caché. La red de interconexión es una colección de estructuras de árboles de interruptores y memorias. Esta organización evita el congestionamiento que puede ocurrir si hay sólo un simple bus compartido. De tal forma, cuando dos procesadores hacen referencia a diferentes localidades de memoria, el contenido de estas localidades estará en la caché de cada procesador. Sin embargo, los problemas pueden surgir cuando dos procesadores hagan referencia a la misma localidad de memoria al mismo tiempo. Si ambos procesadores leen la misma localidad de memoria, cada uno puede

cargar una copia de los datos en su memoria caché, pero si uno de los procesadores escribe, se crea un problema de consistencia en la memoria caché. Por lo tanto, en cada una de las otras memorias cachés se deben actualizar los datos. Para ello, en cada multiprocesador se debe implementar un protocolo de consistencia en la caché a nivel de *hardware* [8] [22].

4.2 Programación paralela para memoria compartida

Un programa paralelo consiste de uno o más *threads* de control operando sobre los datos. Un modelo de programación establece que datos pueden ser llamados por los *threads*, que operaciones se pueden realizar con esos datos, y el orden entre estas operaciones.

El trabajo de paralelización implica determinar la parte del trabajo que se puede hacer en paralelo. Esto requiere determinar cómo distribuir el trabajo y los datos entre los procesadores, y como manejar el acceso a los datos, la comunicación, y la sincronización. El objetivo es obtener un alto rendimiento sin realizar mucho esfuerzo y con pocos recursos. Es decir, se quiere obtener una buena aceleración sobre el mejor programa secuencial que resuelva el mismo problema. Para ello, es necesario tener un buen balance de carga entre los procesadores y evitar que se incremente el trabajo adicional (*overhead*) de comunicación y sincronización.

Las tres medidas fundamentales de rendimiento son las siguientes:

- Latencia. Tiempo que toma ejecutar una operación.
- Ancho de banda. La velocidad con que se realizan las operaciones.
- Costo. El impacto que tienen estas operaciones en el tiempo de ejecución del programa.

Los programas concurrentes dependen absolutamente del uso de componentes compartidos. Esto es porque la única forma para que los procesos trabajen juntos para resolver un problema es comunicándose, y para comunicarse es necesario que un proceso escriba sobre una variable compartida o un canal de comunicación compartido. Así la comunicación es programada escribiendo y leyendo variables compartidas o enviando y recibiendo mensajes. La comunicación da lugar a la sincronización. Hay dos formas básicas para sincronizar procesos paralelos: exclusión mutua y la sincronización condicional. La exclusión mutua ocurre cuando dos procesos necesitan turnarse el acceso a objetos compartidos. La sincronización condicional ocurre cuando un proceso necesita esperar por un evento generado por otro proceso.

4.2.1 Problemas clásicos en la programación concurrente

Los dos problemas clásicos en la programación concurrente son:

- **Sección crítica.** Una sección crítica es una secuencia de declaraciones que acceden a objetos compartidos; es la implementación de las acciones atómicas en *software*. En el problema de la sección crítica, n procesadores ejecutan repetidamente una sección crítica y otra sección no crítica. La sección crítica es precedida por un protocolo de entrada y seguida por un protocolo de salida. Asumiendo que cada proceso que entra a una sección crítica eventualmente tiene que salir. Por lo tanto, un proceso debe terminar su ejecución fuera de la sección crítica. El protocolo de entrada y salida debe satisfacer cuatro propiedades:

  - Exclusión mutua. La mayor parte del tiempo algún estado está en la sección crítica. Un mal estado sería que dos procesos se encuentren al mismo tiempo en la sección crítica.
  - Ausencia de *deadlock*. Si dos o más procesos están tratando de entrar a sus secciones críticas, al menos uno de ellos tendrá éxito. Un mal estado es cuando todos los procesos están esperando para entrar, pero ninguno lo hace.
  - Ausencia de una espera innecesaria. Si un proceso está tratando de entrar a su sección crítica y los otros procesos están ejecutándose en su sección no crítica o terminan su ejecución, el primer proceso no debe esperar para entrar a su sección crítica. Un estado erróneo se presentaría cuando un proceso quiere entrar a su sección crítica y no puede hacerlo aunque no hay otro proceso en la sección crítica.
  - Entrada eventual. Un proceso que está esperando para entrar a su sección crítica eventualmente puede hacerlo.

  Para generalizar la solución al problema de sección crítica para n procesos, se pueden usar n variables. Además sólo hay dos estados interesantes: si el proceso está dentro de su sección crítica o si está en su sección no crítica. Una variable es suficiente para distinguir estos dos estados, independientemente del número de procesos, a esta variable se le llama *candado (lock)*.

  En el pseudocódigo de secciones críticas usando candados inicialmente se indica con $lock = false$ que ningún proceso se encuentra en su sección crítica. Cuando uno de los procesos requiere entrar a su sección crítica se verifica que el otro proceso no se encuentre dentro. Si $lock = true$ indica que la sección crítica está ocupada y tiene que esperar hasta que se desocupe. Cuando el proceso que se encuentra dentro de su sección crítica termina lo indica haciendo $lock = false$ y sale de la sección crítica. Este pseudocódigo se puede generalizar para resolver el problema de sección crítica para cualquier número de procesos.

Secciones críticas usando candados

```
// Declaración de variables
bool lock = false
process1 {
while(true){
        < await (! lock) lock = true >    //Entra
        sección crítica
        lock = false                      //Sale
        sección no crítica }
}
process2 {
while(true){
        < await (! lock) lock = true >    //Entra
        sección crítica
        lock = false                      //Sale
        sección no crítica }
}
```

Sincronización con barreras

```
// Declaración de variables
entero i
entero n //número de procesos
process Worker [i = 1 to n]{
while(true){
        código para implementar tarea i
        wait < n tareas estén completas > } // Barrera
}
```

- **Barreras.** Una barrera es un punto en la sincronización que todos los procesos deben alcanzar antes de que cualquier proceso continúe; las barreras son necesarias en muchos programas paralelos. Muchos problemas pueden resolverse usando algoritmos iterativos; un atributo importante en la mayoría de los algoritmos iterativos paralelos es que cada iteración depende de los resultados de la iteración anterior. Una forma de estructurar de manera eficiente estos algoritmos es creando los procesos cada vez que se inicie el trabajo de cómputo, y sincronizándolos al final de cada iteración. En el pseudocódigo de sincronización con barreras, el punto final de cada iteración representa una barrera a la que todos los procesos tienen que llegar antes de que cualquiera de ellos pase.

4.2.2 Lenguajes de alto nivel

La mayoría de los programas paralelos de alto rendimiento son escritos usando un lenguaje secuencial más una biblioteca. Esto es, porque los programadores ya están familiarizados y tienen experiencia con estos lenguajes. Además, las bibliotecas están disponibles en las platafomas de cómputo paralelo comúnmente usadas. Enseguida se describen los aspectos más importantes de algunos de estos lenguajes de alto nivel.

- **Fortran 90.** Es la evolución de Fortran 77 con nuevas características:
  - Operaciones con arreglos
  - Facilidades para cálculos numéricos
  - Sintaxis para permitir soporte de enteros cortos, valores lógicos empacados, conjuntos de caracteres largos, números reales y complejos de alta precisión.

  Comunicaciones

  - Comunicación implícita. Las operaciones sobre secciones de arreglos conformables pueden requerir movimiento de datos.
  - Comunicación global. Corrimiento de secciones no contiguas, permutación.
  - Comunicación especializada. Corrimiento circular.

- **High Performance Fortran (HPF).**
  - Introduce directivas para distribución de datos que le permitan al programador el control sobre la localidad de datos.
  - El mapeo de procesadores virtuales a procesadores reales depende de la implementación del lenguaje (compilador).
  - Las directivas de distribución de datos pueden tener impacto en el rendimiento del programa pero no en sus resultados.
  - Las directivas de distribución son recomendaciones al compilador, no son instrucciones del lenguaje.

- **C*.**
  - Introduce la noción de una forma (*shape*) que especifica la manera en que se organizan los datos paralelos.

  - Las variables paralelas tienen asociada una forma (*shape*).
  - Las operaciones paralelas ocurren dentro del contexto de la proposición *with*. Activa las posiciones de una forma, definiendo un contexto en el cual las variables de este tipo pueden ser manipuladas en paralelo.
  - Tiene operadores de reducción que reducen valores paralelos a valores secuenciales.
  - La proposición *where* permite realizar operaciones sobre un subconjunto de los elementos de la variable paralela.

- **OCCAM.** Lenguaje de programación para *transputers*
  - Un cómputo paralelo se ve como una colección de procesos que se ejecutan asíncronamente y que se comunican a través de un protocolo síncrono de paso de mensajes.
  - Tienen procesos primitivos de entrada, salida y asignamiento.
  - Un canal es una liga de comunicación síncrona, punto a punto, de un sólo sentido entre un par de procesadores.

4.2.3 Multithreading

Multithreading surge como una técnica para reducir la latencia. El tener múltiples *threads* dentro de un programa, le dá al procesador la habilidad de intercambiar información entre los *threads* reduciendo la latencia de memoria, la latencia de operaciones como las operaciones de I/O y las operaciones de sincronización. El procesador puede también intercambiar instrucciones en un ciclo básico desde los diferentes *threads* para reducir las interrupciones en el *pipeline* debido a las dependencias entre las instrucciones dentro de un *thread* simple, principalmente para incrementar la utilización del procesador.

Para soportar la explotación del paralelismo a nivel *thread*, se proponen dos tipos de arquitecturas: Sistemas multiprocesador y sistemas multithreaded. Los multiprocesadores tienen varios procesadores superescalares y proporcionan mecanismos de comunicación vía memoria compartida. Los *threads* son estáticamente particionados y ejecutados en un procesador separado. Por lo tanto, es difícil para un multiprocesador explotar dinámicamente el paralelismo a nivel de *thread* entre los procesadores. Por otro lado, los sistemas *multithreaded* proporcionan soporte para diferentes contextos y cambio de contexto rápido entre los *threads* para compartir los recursos del procesador de forma dinámica [17].

Pthreads

Un *thread* es un proceso ligero. Algunos sistemas operativos proporcionan mecanismos para permitir a los programadores escribir aplicaciones *multithreaded*. Un grupo de gente a mediados del año 1990 definió un conjunto estándar de rutinas de bibliotecas para el lenguaje C para programación *multithreaded*. Este grupo trabajó bajo el patrocinio de una organización llamada POSIX (Portable operating systems interface). La biblioteca desarrollada fue llamada Pthreads. La biblioteca está disponible actualmente para varios sistemas operativos de UNIX, así como para algunos otros.

La biblioteca Pthreads contiene docenas de funciones para manejo y sincronización de *threads*. Para usar Phreads en un programa en C se siguen 4 pasos: Primero, se debe incluir el encabezado estándar para la biblioteca de Pthreads. Segundo, declarar las variables para el descriptor de atributos de *threads* y uno o más descriptores de *threads*. Tercero, inicializar los atributos para ejecución. Finalmente, crear los *threads*.

La comunicación de los *threads* se realiza usando variables globales en las funciones ejecutadas por los *threads*. Los *threads* se pueden sincronizar usando espera ocupada, candados, semáforos, o variables condicionales.

4.3 Creación del simulador paralelo

Para crear un programa paralelo a partir de uno secuencial se deben seguir 4 pasos:

- **Descomposición**. Dividir el cómputo en tareas. Con ello, se identifica concurrencia y se decide el nivel con el cual se explotará. Es importante tener suficientes tareas para mantener a los procesadores ocupados pero no en exceso.

- **Asignamiento**. Mecanismo que especifica cómo dividir el trabajo entre procesos. Junto con la descomposición se le conoce como particionamiento. Es necesario tener un buen balance de carga para reducir los costos de comunicación y sincronización.

- **Orquestación**. Se necesitan mecanismos para acceder a los datos y mecanismos de comunicación y sincronización entre procesos. La arquitectura, el modelo de programación y el lenguaje de programación juegan un papel importante. En este paso se debe reducir el trabajo adicional *overhead* para controlar el paralelismo reduciendo costos de comunicación y sincronización.

- **Mapeo**. Cada proceso se asigna a un procesador de manera que se trata de maximizar la utilización de los procesadores y minimizar los costos de comunicación y sincronización. Se

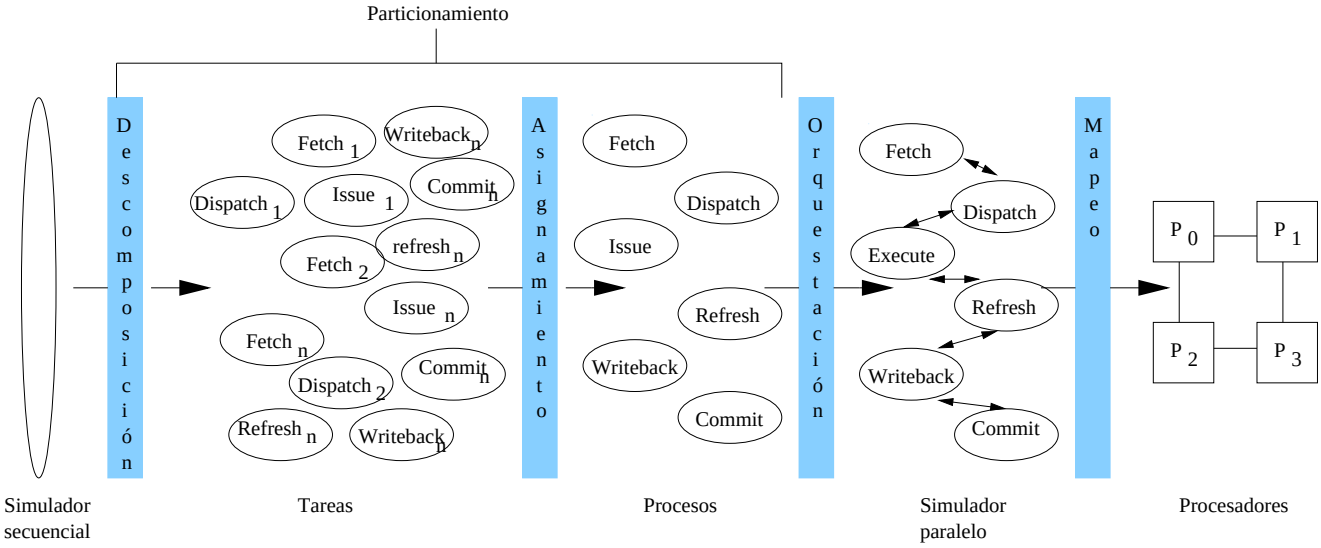


Figura 4.4: Pasos para la creación del simulador paralelo

debe tratar de colocar los procesos relativos en el mismo procesador. El problema de mapeo no aparece en computadoras con un solo procesador o en computadoras con memoria compartida.

En la Figura 4.4 se muestran los pasos que se siguieron para la creación del simulador paralelo. En las siguientes secciones se describen con detalle cada uno de los pasos.

4.3.1 Estrategia de particionamiento

En esta etapa se debe definir la cantidad de computación envuelta en un proceso de *software*, es decir el tamaño de grano o granularidad. Así, el tamaño del grano determina el segmento de programa básico elegido para procesar en paralelo. Entonces, se debe dividir el problema en varias pequeñas tareas. La ejecución de una instrucción del procesador *pipeline* se ejecuta una vez por cada instrucción del programa, y está estructurado como sigue:

$for(;;)\{$
$\quad ruu\_commit();$
$\quad ruu\_writeback();$
$\quad lsq\_refresh();$
$\quad ruu\_issue();$
$\quad ruu\_dispatch();$
$\quad ruu\_fetch();\}$

Para realizar la paralelización se tomó como estrategia partir la ejecución secuencial creando un *thread* por cada etapa de la arquitectura *pipeline* de SimpleScalar. Para realizar la simulación del *pipeline* en forma realmente paralela es necesario desacoplar las diferentes etapas del *pipeline* traslapando en el tiempo la ejecución de etapas diferentes de instrucciones diferentes [19].

Una alternativa de particionamiento es crear un *thread* para cada una de las unidades de la simulación microarquitectural. De tal forma, se tendría un *thread* por cada unidad de ejecución: ALU, contador de programa, acceso a memoria, etc. Con esta estrategia de particionamiento se tiene un grano muy fino, por lo tanto, se generan muchas tareas. Por otro lado, para poder partir la ejecución secuencial de la simulación microarquitectual de SimpleScalar se tendría que reescribir gran parte del código.

Otra alternativa de particionamiento es crear un *thread* por cada instrucción del programa a simular. Por lo tanto, la granularidad estará determinada por el tamaño de la tarea de simulación, por lo que al igual que en la alternativa de las unidades funcionales no se tienen los suficientes procesadores para explotar una granularidad tan fina.

Dado que usualmente no se dispone de suficientes procesadores para explotar el paralelismo de un grano tan fino, se tomó como estrategia de particionamiento un *thread* por cada función del ciclo que simula las diferentes etapas del *pipeline*.

4.3.2 Orquestación del programa paralelo

El análisis del simulador secuencial indicó la presencia de variables globales, con información necesaria de una etapa a otra. En el simulador paralelo, para la utilización de estos datos entre los diferentes *threads* se generó una estructura de datos compartida en donde se incluyen todas las variables globales generando así un paquete de datos para realizar consultas o actualizaciones. De tal forma, la comunicación entre los *threads* se realiza de manera coordinada, introduciendo el paquete en un *buffer* compartido.

Para la sincronización del programa paralelo se plantean dos tipos de sincronización: la sincronización con reloj global y la sincronización productor-consumidor.

Sincronización con reloj global

Por medio de un reloj global virtual se identifica cuando inicia y cuando termina cada una de las etapas del *pipeline*. De tal forma, el diseño es completamente síncrono. En la Figura 4.5 se muestra la sincronización con reloj global. Para implementar el reloj global se utiliza un *buffer* de 5 localidades, una para cada *thread*, excepto para la última etapa que ya no necesita guardar su resultado. Cuando una etapa termina su ejecución se genera un paquete con todas las variables

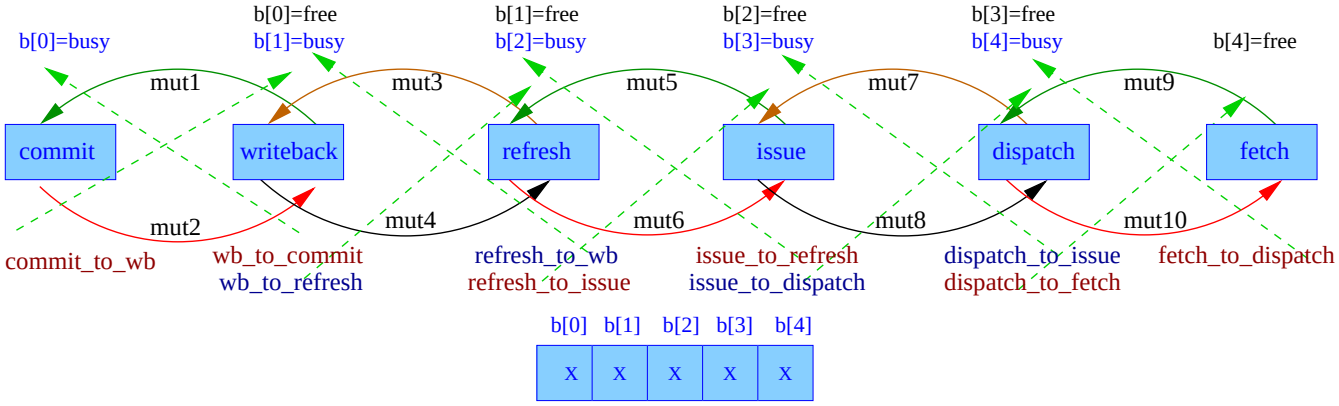


Figura 4.5: Sincronización de 6 *threads* con estrategia: reloj global.

globales y se coloca en el *buffer*. Si la localidad correspondiente está libre introduce el paquete en el *buffer* si no es así significa que todavía se está ejecutándo la instrucción anterior y debe esperar hasta que se le indique que ha finalizado. Dado que por cada instrucción de la tarea de simulación se debe ejecutar el ciclo del *pipeline*, cada etapa requiere de una fase de sincronización con la siguiente. Entonces, la longitud del ciclo de reloj está determinada por la velocidad de la fase más lenta. Por lo tanto, no se puede ejecutar una nueva instrucción en cada etapa hasta que las demás etapas hayan concluído.

El pseudocódigo de sincronización con reloj global mostrado en la Figura 4.6 presenta la sincronización entre dos *threads*: *commit* y *writeback*. Inicialmente *writeback* no puede ejecutarse hasta que *commit* coloque un paquete en el *buffer*, entonces se queda esperando. *commit* verifica si su localidad se encuentra ocupada si es así se espera a que *writeback* la desocupe. Cuando la localidad está desocupada se ejecuta *commit* y coloca su paquete en el *buffer* avisando a *writeback*. La sincronización de los 6 *threads* con reloj global se realiza generalizando este pseudocódigo.

Sincronización productor-consumidor

Para realizar la paralelización es necesario traslapar en el tiempo la ejecución de etapas diferentes de instrucciones diferentes desacoplando las etapas del *pipeline* por cada ciclo de reloj. Para ello se usa la sincronización mediante el esquema: productor-consumidor. Definiendo un *buffer* compartido por cada par de etapas a sincronizar en donde el paquete que se produce o consume está integrado por el conjunto de variables globales. El productor y el consumidor se alternan para acceder al buffer [1]. En la Figura 4.7, se considera a *commit* como productor y a *writeback* como consumidor, los cuales usan las variables *empty* y *full*. Para resolver el problema de la sección crítica se usan candados, cuando el productor quiere guardar en el *buffer* que está lleno entonces tiene que esperar la señal del consumidor *writeback_to_commit* y cuando el consumidor quiere acceder al *buffer* vacío

Sincronización reloj global

```
ruu_commit1(void * q){
for(;;) {
// verifica si su localidad está ocupada
        pthread_mutex_lock(mut1);
          if(b[0] == busy)
            pthread_cond_wait( writeback_to_commit, mut1);
        pthread_mutex_unlock( mut1 );
              ruu_commit();
// coloca un paquete y avisa a la siguiente etapa
        pthread_mutex_lock( mut2 );
          put_buffer();
            pthread_cond_signal(commit_to_writeback);
        pthread_mutex_unlock ( mut2 ); }
}
ruu_writeback1(void * q){
for(;;) {
// verifica si la localidad de la etapa anterior está libre
        pthread_mutex_lock ( mut2 );
          if (b[0]=free)
            pthread_cond_wait(commit_to_writeback, mut2);
          pthread_mutex_unlock(mut2);
//obtiene un paquete y avisa a la etapa anterior
        pthread_mutex_lock( mut1 );
          get_buffer();
            pthread_cond_signal(writeback_to_commit);
            pthread_mutex_unlock(mut1);}
               ruu_writeback();
}
```

Figura 4.6: Pseudocódigo de estrategia de sincronización con reloj global

entonces tiene que esperar la señal del productor *commit_to_writeback*.

En la Figura 4.8 se muestra el pseudocódigo de Sincronización productor-consumidor entre dos *threads* del simulador paralelo: *commit* y *writeback*. En este caso *commit* es el productor y *writeback* es el consumidor. *commit* debe verificar el estado del *buffer* antes de producir un nuevo paquete si el *buffer* no esta lleno, produce, lo depósita en *buffer* y en caso de que el consumidor se quedó en espera deberá notificar que ya se encuentra un nuevo paquete en el *buffer*. En caso de que el *buffer* se encuentre lleno *commit* deberá esperar la señal del consumidor, en este caso *writeback*. Cuando *writeback* requiere consumir un paquete verifica si el *buffer* no está vacío. En este caso, deberá esperar que el productor introduzca un nuevo paquete en el *buffer* y le notifique al consumidor. Cuando *writeback* consuma un paquete deberá notificar al productor en caso de que este se haya quedado en espera, debido a que el *buffer* estaba lleno. Este pseudocódigo se debe generalizar para la sincronización de los demás *threads* de manera similar.

4.3.3 Mapeo

El simulador paralelo se probó en multiprocesadores con 4 procesadores en donde mecanismos de *hardware* y del sistema operativo hacen la calendarización de procesos de forma automática. Esto es, dado que cada procesador es similar a los otros, puede ejecutar cualquier *thread* que se encuentre disponible.

De acuerdo a la estrategia de particionamiento utilizada se tienen 6 *threads*. Debido a que se cuenta con 4 procesadores es posible mapear 4 *threads* que se ejecutan simultáneamente, quedando dos *threads* pasivos. Sin embargo, cuando los *threads* necesiten utilizar algún otro recurso, como el acceso a memoria, es posible mapear a los otros *threads*, maximizando así la utilización de los procesadores. Debido a esto, es necesario tener un número mayor de *threads* que de procesadores, pero sin excederse. Evitando así intervalos de tiempo de los procesadores sin utilizar.

4.4 Estructuras de datos globales

El principal problema para lograr desacoplar las etapas del *pipeline* es la presencia de datos globales. Para poder acceder a estos datos se utilizó una estructura compartida. A esta estructura le llamamos *shared_var* y se muestra en la figura 4.9. La estructura compartida está integrada por todas las variables globales necesarias en las diferentes etapas del *pipeline*.

En las Tablas 4.1 y 4.2 se dá una breve definición y descripción de cada una de las variables globales y las etapas del *pipeline* en donde son utilizadas. Las estructuras de datos globales pasan de una etapa a otra produciendo una copia de ellas para cada etapa del *pipeline*.

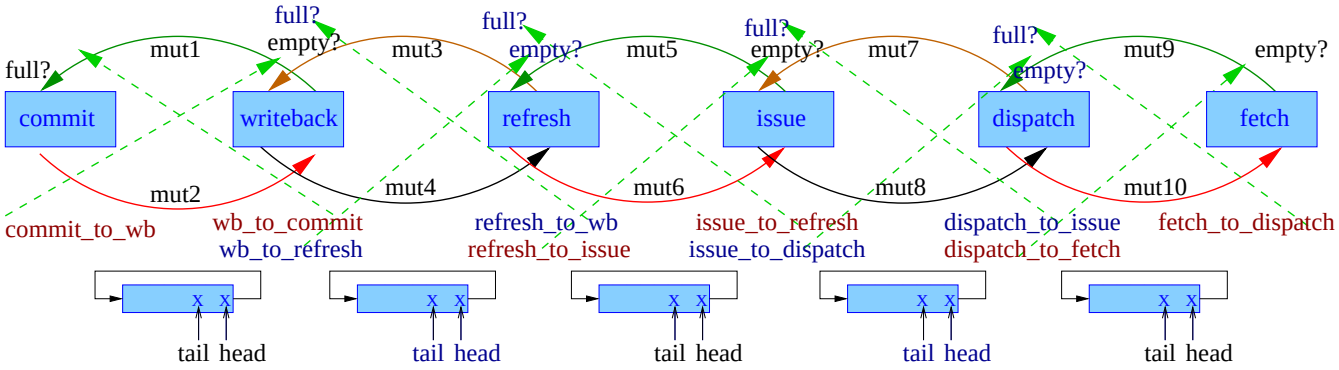


Figura 4.7: Sincronización de 6 *threads* con estrategia: productor-consumidor.

Sincronización productor-consumidor

```
ruu_commit1(void * q){
for(;;) {
// verifica si el buffer está lleno
        pthread_mutex_lock(mut2);
          if(shv_num == shv_size){
            full++;
            pthread_cond_wait( writeback_to_commit, mut2);
            full–; }
        pthread_mutex_unlock( mut2 );
              ruu_commit();
// produce un paquete y avisa al consumidor
        pthread_mutex_lock( mut1 );
          producer();
          if(empty > 0)
            pthread_cond_signal(commit_to_writeback);
            ncommit++;
        pthread_mutex_unlock ( mut1 ); }}
ruu_writeback1(void * q){
for(;;) {
// verifica si el buffer está vacío
        pthread_mutex_lock ( mut1 );
          if (!shv_num){
            empty++;
            pthread_cond_wait(commit_to_writeback, mut1);
            empty–; }
          consumer();
          pthread_mutex_unlock(mut1);
              ruu_writeback();
//consume un paquete y avisa al productor
        pthread_mutex_lock( mut2 );
          if (full > 0){
            pthread_cond_signal(writeback_to_commit);
            pthread_mutex_unlock(mut2); }}
```

Figura 4.8: Pseudocódigo de estrategia de sincronización productor-consumidor

```
struct shared_var{
        struct RUU_station *RUU;
        struct RUU_station *LSQ;
        struct cache *cache_dl1;
        struct cache *cache_il1;
        struct cache *dtlb;
        struct cache *itlb;
        struct res_pool *fu_pool;
        struct CV_link create_vector[SS_TOTAL_REGS];
        struct CV_link spec_create_vector[SS_TOTAL_REGS];
        SS_TIME_TYPE create_vector_rt[SS_TOTAL_REGS];
        SS_TIME_TYPE spec_create_vector_rt[SS_TOTAL_REGS];
        struct RS_link *ready_queue;
        SS_COUNTER_TYPE sim_num_insn;
        struct stat_stat_t *pcstat_stats[MAX_PCSTAT_VARS];
        struct stat_stat_t *pcstat_sdists[MAX_PCSTAT_VARS];
        SS_WORD_TYPE spec_regs_R[SS_NUM_REGS];
        SS_ADDR_TYPE recover_PC;
        SS_ADDR_TYPE fetch_regs_PC;
        SS_ADDR_TYPE fetch_pred_PC;
        struct fetch_rec *fetch_data;
        struct RS_link last_op; };
```

Figura 4.9: Buffer de variables compartidas

Variable	Definición	Descripción	Etapas
RUU	Unidad de actualización de registros. Buffer apuntador a la estructura: *RUU_station*	Las estaciones guardan los resultados y esperan a que todos los operandos esten listos	commit writeback dispatch
LSQ	Cola de carga y almacenamiento. Buffer apuntador a la estructura: *RUU_station*	Permite cargar y almacenar el programa en orden, indicando el estado de acceso de carga y almacenamiento	commit refresh issue dispatch
*cache_dl*1	cache de datos apuntador a la estructura: cache	Se utiliza para almacenar y acceder a la memoria caché de datos	commit issue
*cache_il*1	cache de instrucciones. Apuntador a la estructura: cache	Se utiliza para almacenar y acceder a la memoria caché de instrucciones	fetch

Tabla 4.1: Variables globales

Variable	Definición	Descripción	Etapas
dtlb itlb	Buffer de traducción de direcciones (instrucciones y datos) apuntador a la estructura: cache	Mapea una dirección virtual a una dirección física de la caché de datos y de instrucciones	commit issue fetch
fu_pool	Cola de unidades funcionales Apuntador a la estructura: *res_pool*	Se usa para asignar y consultar el estado de las unidades funcionales (recursos)	commit issue
create_vector	Arreglo de estructuras: *CV_link* para registros	Almacena los registros válidos	writeback
create_vector_rt	Arreglo para registros de especulación	Se usa para indicar cual fue el último registro creado	writeback
ready_queue	Cola de instrucciones listas Apuntador a la estructura: *RS_link*	Almacena las instrucciones que están listas para ejecutarse	issue
sim_num_isns	Contador	Cuenta el número de instrucciones cargadas	dispatch
pcstat_stats *pcstat_sdists*	Arreglo apuntador a la estructura: *stat_stat_t*	Realiza la decodificación: convierte el valor de las variables a una expresión válida	dispatch
spec_regs_R	Archivo de registros	Se usa para recuperar registros cuando hay falla en la especulación	dispatch
recover_PC	Contador de programa	Número de instrucciones decodificadas	dispatch
fetch_regs_PC *fetch_pred_PC*	Variables de estado	Se usan cuando no se ejecutó alguna de las etapas por dependencia de datos por lo que no se toma una nueva instrucción	dispatch fetch
fetch_data	Cola del fetch Apuntador a la estructura: *fetch_rec*	Cola de instrucciones en el fetch listas para decodificarlas	dispatch fetch
last_op	Estructura *RS_link*	Ultima instrucción decodificada para implementar la ejecución en orden	dispatch

Tabla 4.2: Variables globales

Capítulo 5

Resultados de la simulación de procesadores

En este capítulo se presenta un estudio sobre dos de los simuladores de SimpleScalar *sim-bpred* y *sim-cache*, se evaluaron 4 *benchmaks* de SPEC95 para cada simulador dos *benchmarks* de aritmética entera y dos *benchmarks* de aritmética de punto flotante. Para cada *benchmark* se probaron 11 configuraciones diferentes para el simulador *sim-bpred* y 10 para el simulador *sim-cache*.

5.1 Simulación de la unidad de predicción de saltos

La frecuencia con que se dan los saltos demanda que se estudie sobre la predicción de saltos. El objetivo de todos estos mecanismos es permitir al procesador encontrar el resultado de un salto rápidamente. La efectividad del esquema de predicción de saltos, depende no solo de la precisión, sino también del costo de un salto cuando la predicción es correcta y cuando la predicción es incorrecta. Estas penalizaciones a los saltos depende de la estructura de la arquitectura, el tipo de predictor, y las estrategias utilizadas para la recuperación de predicciones erradas.

5.1.1 Configuraciones para la unidad de predicción de saltos

Para la unidad de predicción de saltos se utilizaron los siguientes *benchmarks* de aritmética entera:

- *benchmark: gcc*

  Area de aplicación: Programación y compilación

  Tarea específica: Compila archivos preprocesados generando código ensamblador. gcc está basado en el compilador GNU C versión 2.5.3 distribuida por La Fundación Libre de

Software. El benchmark toma el tiempo que le toma al compilador convertir un número de archivos fuente preprocesados en archivos de salida en código optimizado para sistemas Sparc.

- *benchmark: compress*

  Area de aplicación: Compresión

  Tarea específica: Reduce el tamaño de los archivos llamados. Cuando es posible cada archivo es reemplazado por uno con extensión .z. Si los archivos no son especificados, la entrada estándar es comprimida a la salida estándar. Los archivos comprimidos pueden ser restaurados a su forma original usando uncompress o zcat.

Se utilizaron también los siguientes *benchmarks* de aritmética de punto flotante:

- *benchmark: mgrid*

  Area de aplicación: Electromagnetismo

  Tarea específica: Demuestra la capacidad de resolver multi-redes de campos potenciales en 3 dimensiones. Resuelve solo ecuaciones de coeficientes constantes y sólo en una red uniforme cúbica.

- *benchmark: hydro2d*

  Area de aplicación: Astrofísica

  Tarea específica: Programa astrofísico para resolver ecuaciones hidrodinámicas para cálculos usados en jets galácticos.

Se usaron los diferentes argumentos con que cuenta el simulador *sim-bpred* para representar diferentes tipos de predictores, con los mismos archivos de entrada y parámetros, con el fin de obtener estadísticas que indiquen cómo obtener mayor exactitud en la predicción de saltos. Las configuraciones evaluadas fueron las siguientes:

- **A.** Predictor de saltos de dos niveles con 512 entradas en cada nivel y una historia de amplitud 4.
- **B.** Predictor de saltos de 2 niveles con 2 entradas en el primer nivel y 512 en el segundo nivel.
- **C.** Predictor de saltos bimodal con 1024 entradas y BTB (branch target buffer) de 256 conjuntos y una asociatividad de 2.

Benchmark	gcc			compress		
Pruebas	Saltos	Hits	Misses	Saltos	Hits	Misses
A	54992191	48830404	6161787	338736	328466	10270
B	54992191	47614273	7377918	338736	328878	9858
C	54992191	48830404	6161787	338736	328466	10270
D	54992191	48830404	6161787	338736	328466	10270
E	54992191	48830404	6161787	338736	328466	10270
F	54992191	48320048	6672143	338514	328264	10250
G	54992191	48830404	6161787	338514	328264	10250
H	54992191	48830404	6161787	338514	328055	10459
I	54992191	49223861	5768330	338514	328264	10250
J	54992191	48830404	6161787	338736	328466	10270
K	54992191	48830404	6161787	338736	328878	9858
Benchmark	mgrid			hydro2d		
A	57771503	56322678	1448825	132712071	131140640	1571431
B	57771503	56322678	1448825	132712133	130208038	2504095
C	57771498	56322665	1448833	131140625	132712065	1571440
D	57771503	56309729	1461774	131791165	132712065	920900
E	57771498	56330407	1441091	132712071	131140640	1571431
F	57771503	56322678	1448825	131140625	132712065	1571440
G	57771498	56322665	1448833	130740092	132712065	1971973
H	57771503	56322678	1448825	132712071	131140640	1571431
I	57771498	56322665	1448833	131140625	132712065	1571440
J	57771503	56322678	1448825	131400190	132712065	1311875
K	57771498	56322665	1448833	132712133	131140674	1571459

Tabla 5.1: Resultados sobre aciertos y fallas de las 11 configuraciones evaluadas sobre 4 benchmarks

- **D.** Predictor bimodal con 2048 entradas y BTB (branch target buffer) de 256 conjuntos y una asociatividad de 64.
- **E.** Predictor combinado con 2048 entradas.
- **F.** Predictor bimodal con 1024 entradas.
- **G.** Predictor combinado con 256 entradas.
- **H.** Predictor bimodal con 2048 entradas.
- **I.** Predictor bimodal con 4096 entradas.
- **J.** Predictor bimodal con 2048 entradas con 4 entradas en el stack.
- **K.** Predictor bimodal con 2048 entradas con 32 entradas en el stack.

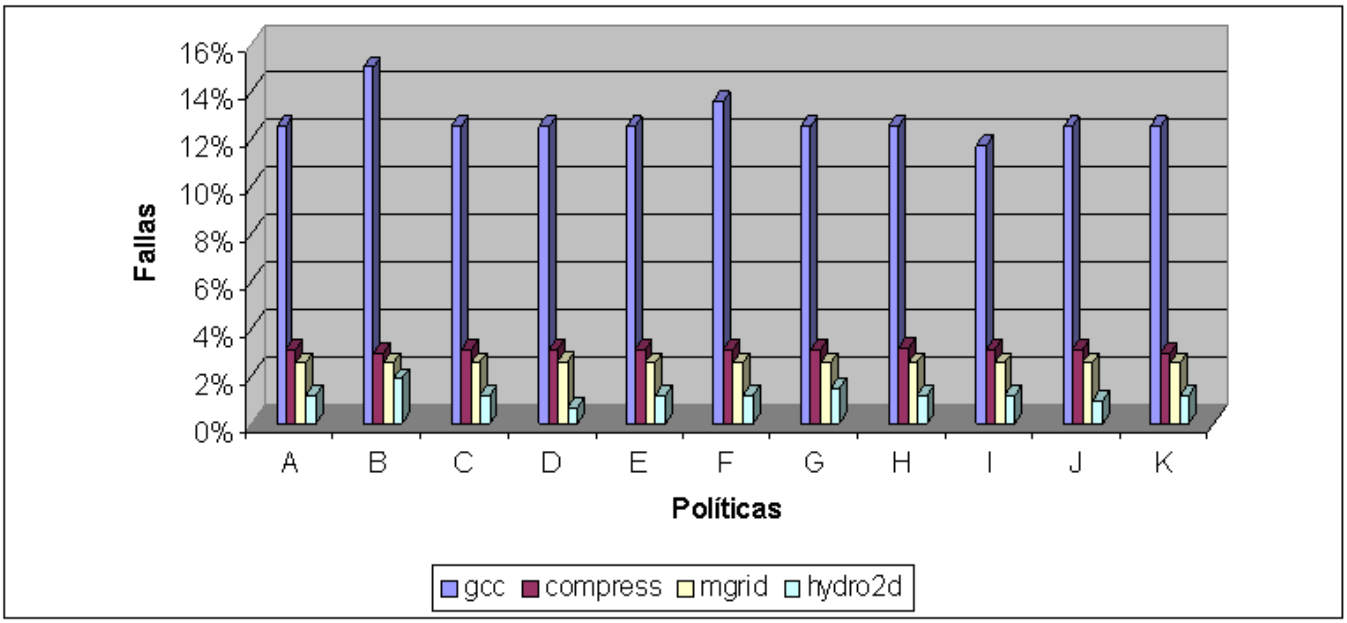


Figura 5.1: Fallas en la predicción de saltos

5.1.2 Resultados

En la Tabla 5.1 se muestran algunos datos estadísticos generados con el simulador *sim-bpred* para cada una de la pruebas aplicadas a los 4 *benchmarks* en estudio.

Los datos que determinan que tan preciso es el predictor de saltos son las fallas (*misses*) en la predicción de saltos. En la gráfica de la Figura 5.1 se presentan gráficamente los resultados de errores de la Tabla 5.1. Se puede ver que en general, para todas las configuraciones usadas, la predicción es buena pues el porcentaje de error está por debajo del 16% del total de saltos.

Para el *benchmark gcc* el porcentaje de *misses* es muy uniforme en todas las pruebas, en la prueba I se obtuvo mayor precisión en la predicción de saltos, como se puede observar en la gráfica 5.1 tiene aproximadamente el 11% de error (ver Tabla 5.1). Esto es debido a que tiene mayor número de entradas en la tabla del predictor. Con la prueba B se obtuvo el rendimiento más bajo.

Para el *benchmark compress* el porcentaje de *misses* es menor al 3%, es casi el mismo para todas las pruebas a excepción de la prueba B y la prueba K (ver Tabla 5.1) pero la diferencia con *gcc* no es significativa considerando el número de saltos.

En general, en todas las pruebas para el *benchmark mgrid* el porcentaje de *misses* es aproximadamente del 2.5% pero se tiene un mejor rendimiento que para los *benchmarks* anteriores considerando el número total de saltos (ver Tabla 5.1).

Con el *benchmark hydro2d* independientemente de las diferentes pruebas se obtuvo un rendimiento muy bueno. El mejor rendimiento se obtuvo con la prueba D, donde se tuvo solamente el

0.69% de error en la predicción. Con la prueba B se obtuvo el rendimiento más bajo.

Aún cuando los programas con mayor frecuencia de saltos son los enteros, los *benchmarks* de aritmética entera probados no tuvieron una mayor presición que los de punto flotante, debido a que aún cuando se probaron las mismas configuraciones para todos los *benchmarks* los archivos de entrada son diferentes dada la diversidad en las aplicaciones.

5.2 Simulación de la unidad de memoria caché

El rendimiento de un procesador esta fuertemente ligado al rendimiento del sistema de acceso a memoria. Para acelerar el tiempo de acceso se ha colocado dentro del procesador una memoria de acceso rápido, llamada memoria caché.

En la memoria caché se guardan los bloques de memoria que han sido recientemente referidos por el procesador. La memoria caché se puede organizar tomando en cuenta la longitud de bloque, la asociatividad y la política de reemplazo. El rendimiento de una organización específica se evalúa por la cantidad de fallas (misses) cuando un dato solicitado no se encuentra en la memoria caché.

5.2.1 Configuraciones para la unidad de memoria caché

Para la unidad de memoria caché se utilizaron los siguientes *benchmarks* de aritmética entera:

- *benchmark: li*

  Area de aplicación: Programación

  Tarea específica: Intérprete de Xlisp

- *benchmark: perl*

  Area de aplicación: Programación

  Tarea específica: Intérprete para el lenguaje perl

Y los siguientes *benchmarks* de aritmética de punto flotante:

- *benchmark: swim*

  Area de aplicación: Telecomunicaciones

  Tarea específica: El programa resuelve el sistema de ecuaciones de transmisión de ondas bajo el agua usando aproximaciones de diferencias finitas. Se puede llegar a generar una matriz de 1024x1024.

- *benchmark: fppp*

  Area de aplicación: Química cuántica

  Tarea específica: Obtiene dos derivaciones de electrones integrales por medio de las series Gaussianas de química cuántica. Los átomos, son localizados en una región relativamente compacta de espacio.

Se evaluaron 10 configuraciones diferentes de la memoria caché, tomando en cuenta el tamaño de la memoria caché y la asociatividad. Para todas las configuraciones se usaron los mismos archivos de entrada de acuerdo al *benchmark* en estudio. Las configuraciones evaluadas fueron las siguientes:

- **A.** Nivel de instrucciones 1, con 256 conjuntos de 32 bytes, asociatividad de 1 (8KB).
- **B.** Nivel de instrucciones 1, con 4 conjuntos de 8 bytes, asociatividad de 2 (64KB).
- **C.** Nivel de instrucciones 1, con 16 conjuntos de 16 bytes, asociatividad de 4 (1KB).
- **D.** Nivel de instrucciones 1, con 32 conjuntos de 8 bytes, asociatividad de 4 (1KB).
- **E.** Nivel de instrucciones 1, con 32 conjuntos de 8 bytes, asociatividad de 8 (2KB).
- **F.** Nivel de instrucciones 1, con 16 conjuntos de 16 bytes, asociatividad de 16 (4KB).
- **G.** Nivel de instrucciones 1, con 64 conjuntos de 8 bytes, asociatividad de 16 (8KB).
- **H.** Nivel de instrucciones 1, con 64 conjuntos de 16 bytes, asociatividad de 8 (8KB).
- **I.** Nivel de instrucciones 1, con 64 conjuntos de 32 bytes, asociatividad de 8 (16KB).
- **J.** Nivel de instrucciones 1, con 128 conjuntos de 32 bytes, asociatividad de 8 (32KB).

En todas las configuraciones se indica el tamaño de la memoria caché al final, entre paréntesis. El tamaño de la memoria caché, se obtiene del producto del número de conjuntos por el tamaño de bloque y por la asociatividad.

5.2.2 Resultados

En la Tabla 5.2 se tienen algunos de los datos estadísticos obtenidos con el simulador de la unidad de caché *sim-cache* para cada prueba.

Es importante que el porcentaje de fallas (*misses*) no sea alto al acceder a la memoria caché para tener un buen rendimiento en el procesador. Con los datos mostrados en la Tabla 5.2 se generó

Benchmark	li			perl		
Pruebas	Accesos	Hits	Misses	Accesos	Hits	Misses
A	269181	258551	10630	331065	2883068	467997
B	269181	31884	237297	331065	156835	3194230
C	269181	218458	50723	331065	2107109	1243956
D	269181	182399	86782	331065	1188614	2162451
E	269181	231738	37443	331065	1579104	1771961
F	269181	257580	11601	331065	2605717	745348
G	269181	257356	11825	331065	2407283	943782
H	269181	262373	6808	331065	3126274	224791
I	269181	266891	2290	331065	2782750	568315
J	269181	267615	1566	331065	3320931	30134
Benchmark	swim			fppp		
A	849922285	839907028	10015257	1872316391	1439083433	433232958
B	849922285	1227200	848695085	1872316391	1950036	1870366355
C	849922285	716151744	133770541	1872316391	953415670	918900721
D	849922285	609218375	240703910	1872316391	46121310	1826195081
E	849922285	759169791	90752494	1872316391	63874941	1808441450
F	849922285	849883056	39229	1872316391	991944441	880371950
G	849922285	849886272	36013	1872326391	181529328	1690797063
H	849922285	849902565	19720	1872316411	1024062188	848254223
I	849922285	849913679	8606	1872316391	1463577377	408739014
J	849922285	849918056	4229	1872316391	1621841186	250475205

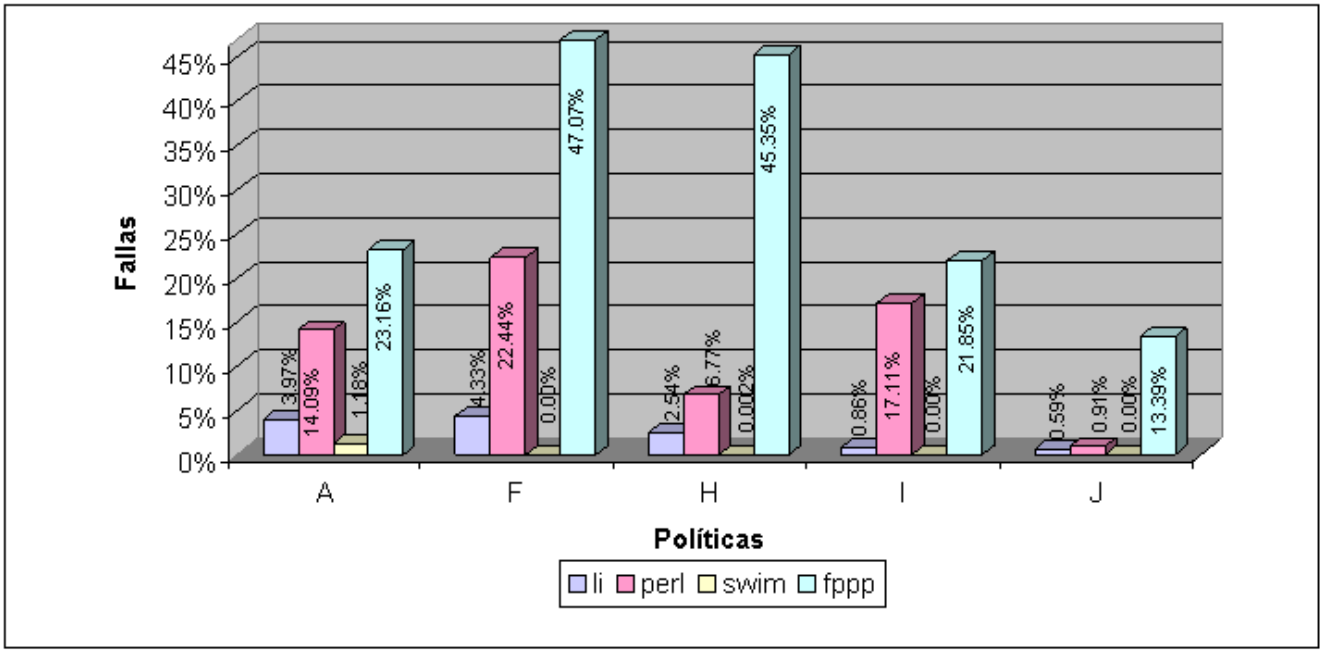
Tabla 5.2: Resultados sobre aciertos y fallas en el acceso a memoria caché sobre las 10 configuraciones con 4 benchmarks

la gráfica de la Figura 5.2. Se puede observar que es muy importante la configuración que tenga la memoria caché. Debido a que los resultados muestran un porcentaje de error que va del 0% al 100% se presentan en dos gráficas para tener una mejor visualización. En la Figura 5.2 (a) se tienen las pruebas que dieron mejores resultados. Tienen un porcentaje de error que va del 0% al 47% y en Figura 5.2 (b) están las pruebas con rendimiento más bajo, donde el porcentaje de error al acceder la memoria caché va del 0% al 100%.

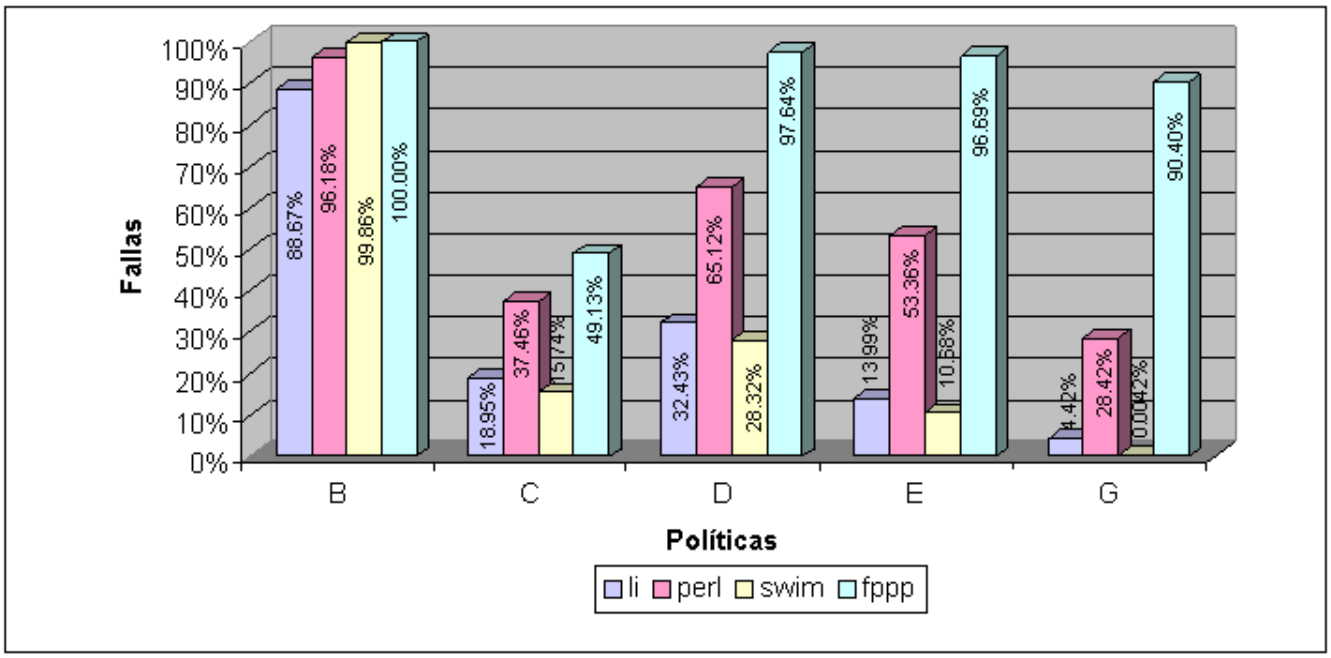
En general, para todos los *benchmarks* el peor rendimiento se obtuvo con la prueba B, esto es debido a que la longitud de bloque y el tamaño del conjunto son muy pequeños. El mejor rendimiento se obtuvo con la prueba J, pues se tiene un mayor tamaño en la memoria caché y una buena asociatividad en relación con las demás pruebas. Para casi todas las pruebas con el *benchmark li* se tuvo un rendimiento bueno, pero es el *benchmark* que tiene el menor número de accesos a la memoria caché.

Para el *benchmark perl* se tiene un rendimiento más bajo que para el *benchmark li* debido a que se tiene un mayor número de accesos a la memoria caché y esto depende del archivo de entrada, pues el archivo *lisp* es bastante más simple que el archivo *perl* donde se tienen más iteraciones.

El *benchmark swim* es el que tiene el mejor rendimiento, en la mayoría de las pruebas tiene un



(a) Mejores casos



(b) Peores casos

Figura 5.2: Fallas en la predicción de la memoria caché

porcentaje de error casi nulo. En las pruebas en donde no es así se debe a la configuración de la memoria caché que es más pequeña y de menor asociatividad que en las otras pruebas.

El peor rendimiento se tiene con el *benchmark fppp* pues se trata de un programa donde se presenta mucha aleatoriedad en la obtención de los resultados. Sin embargo, con una buena configuración de la memoria caché, como en el caso de la prueba J, se obtiene un buen rendimiento.

En general, entre mayor sea el tamaño de la memoria caché es menor el número de fallas. Sin embargo, por las pruebas realizadas se puede concluir que también son factores importantes la longitud del bloque y la asociatividad de la memoria.

5.3 Resultados del simulador paralelo

Se desarrollaron 3 versiones del simulador paralelo. En la primera versión se partió la ejecución secuencial creando 2 hilos de ejecución, cada uno con 3 etapas del ciclo *pipeline* de SimpleScalar. El primer *thread*, está integrado por las etapas ruu_commit, $ruu\_writeback$ y $lsq\_refresh$. El segundo *thread*, está integrado por las etapas ruu_issue, $ruu\_dispatch$ y ruu_fetch. En la segunda versión se crearon 4 hilos de ejecución. El primer *thread*, está integrado por la etapa ruu_commit, el segundo *thread* está integrado por las etapas $ruu\_writeback$ y $lsq\_refresh$. El tercer *thread* está integrado por la etapa $ruu\_dispatch$ y ruu_issue. Finalmente, el cuarto *thread* esta integrado por la etapa ruu_fetch. En la última versión se partió la ejecución secuencial creando 6 hilos de ejecución, uno por cada etapa del *pipeline*. Así, el primer *thread* está integrado por la etapa ruu_commit, el segundo *thread* está integrado por la etapa $ruu\_writeback$, el tercer, *thread* por la etapa $lsq\_refresh$, el cuarto *thread* por la etapa $ruu\_dispatch$, el quinto por la etapa ruu_issue y el sexto *thread* por la etapa ruu_fetch. Para cada versión se probaron dos *tests*: test-math y test-printf, dos *benchmarks* de aritmética entera: li.ss y gcc.ss y dos *benchmarks* de aritmética de punto flotante: mgrid.ss e hydro2d.ss.

5.3.1 Tiempos de ejecución de los benchmarks

El tiempo del simulador secuencial está determinado únicamente por el tiempo de cómputo:

$$T\_\text{sec} \;=\; T\_\text{comp}$$

Sin embargo, el tiempo de ejecución del simulador paralelo, está determinado por el tiempo de cómputo más el tiempo de sincronización:

No. Prueba	Benchmark	Sim. secuencial	No. de instrucciones
1	test-math	0.815	189668
2	test-printf	3.955	1261460
3	li.ss	252.633	62275608
4	gcc.ss	817.705	264280116
5	mgrid.ss	1137.984	422323517
6	hydro2d.ss	3805.670	974466159

Tabla 5.3: Tiempos de ejecución del simulador secuencial sim-outorder y número de instrucciones de cada aplicacion

No. Prueba	Benchmark	Simulador paralelo 2th			Porcentaje
		T_par	T_comp	T_sync	
1	test-math	0.66	0.407	0.252	19%
2	test-printf	4.13	1.977	2.153	−4.4%
3	li.ss	247.03	126.316	120.713	10.1%
4	gcc.ss	769.54	408.850	360.690	9.5%
5	mgrid.ss	987.54	568.992	418.548	13.2%
6	hydro2d.ss	2905.72	1902.835	1002.885	26.2%
No. Prueba	Benchmark	Simulador paralelo 4th			Porcentaje
		T_par	T_comp	T_sync	
1	test-math	0.48	0.203	0.277	41.10%
2	test-printf	2.96	0.988	1.972	25.15%
3	li.ss	141.06	63.158	77.902	44.16%
4	gcc.ss	577.77	204.426	373.344	29.34%
5	mgrid.ss	867.54	284.496	583.044	23.76%
6	hydro2d.ss	2412.56	951.417	1461.143	36.60%
No. Prueba	Benchmark	Simulador paralelo 6th			Porcentaje
		T_par	T_comp	T_sync	
1	test-math	0.35	0.135	0.215	57.05%
2	test-printf	2.58	0.659	1.921	34.76%
3	li.ss	94.06	42.105	51.955	62.76%
4	gcc.ss	463.43	136.284	327.146	43.32%
5	mgrid.ss	687.54	189.664	497.876	39.58%
6	hydro2d.ss	2103.20	634.278	1468.922	44.73%

Tabla 5.4: Tiempos de ejecución de los simuladores paralelos de 2 threads, 4 threads y 6 threads

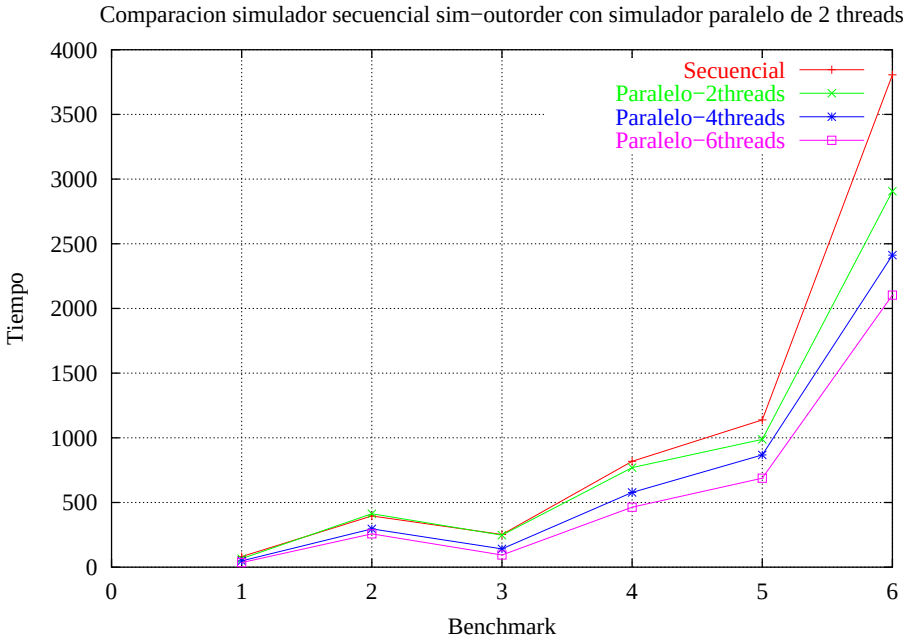


Figura 5.3: Tiempos de ejecución

$$T\_\text{par} = T\_\text{comp} + T\_\text{sync}$$

En la Tabla 5.3 se muestran los tiempos de ejecución (segundos) de los *benchmarks* de prueba para el simulador secuencial de SimpleScalar *sim-outorder* y el número de instrucciones de cada aplicación. En la Tabla 5.4 se muestran los tiempos de ejecución (segundos) de los *benchmarks* de prueba del simulador paralelo de 2 *threads*, del simulador paralelo de 4 *threads* y del simulador paralelo de 6 *threads*. Se puede observar en la gráfica de la Figura 5.3 y en las Tablas 5.3 y 5.4 que el promedio de la reducción en los tiempos de ejecución fue del 12.26% para el simulador paralelo de 2 *threads*, del 33.35% para el simulador paralelo de 4 *threads* y del 47.03% para el simulador paralelo de 6 *threads*.

En las gráficas de las Figuras 5.4, 5.5, 5.6, 5.7, 5.8 y 5.9 se muestra para cada uno de los *benchmarks* el tiempo de ejecución secuencial, el tiempo de ejecución paralelo, el tiempo de sincronización y el tiempo de cómputo. En la Tabla 5.4 y en las Figuras 5.4, 5.5, 5.6, 5.7, 5.8 y 5.9 se puede observar que el tiempo de ejecucion se disminuye para cada una de las versiones conforme se incrementa el número de *threads*. Sin embargo, el tiempo de sincronización se va incrementando. Esto es, debido al número de instrucciones que se están simulando dependiendo de la aplicación que se prueba. Por lo tanto, el tiempo de ejecución paralelo no disminuye tanto como el tiempo de cómputo, que sería el tiempo de ejecución ideal.

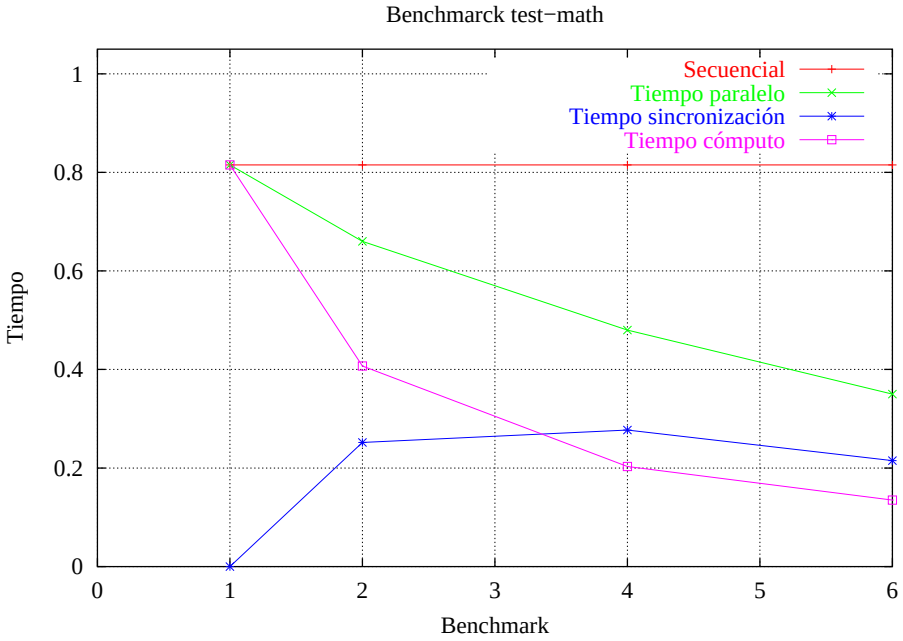


Figura 5.4: Tiempos de ejecución del benchmark test-math

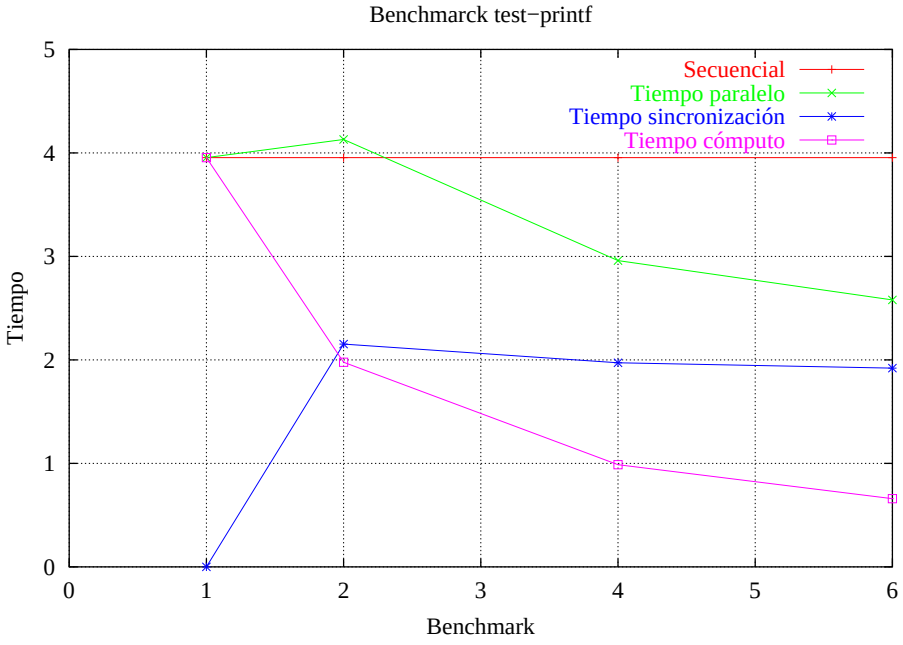


Figura 5.5: Tiempos de ejecución del benchmark test-printf

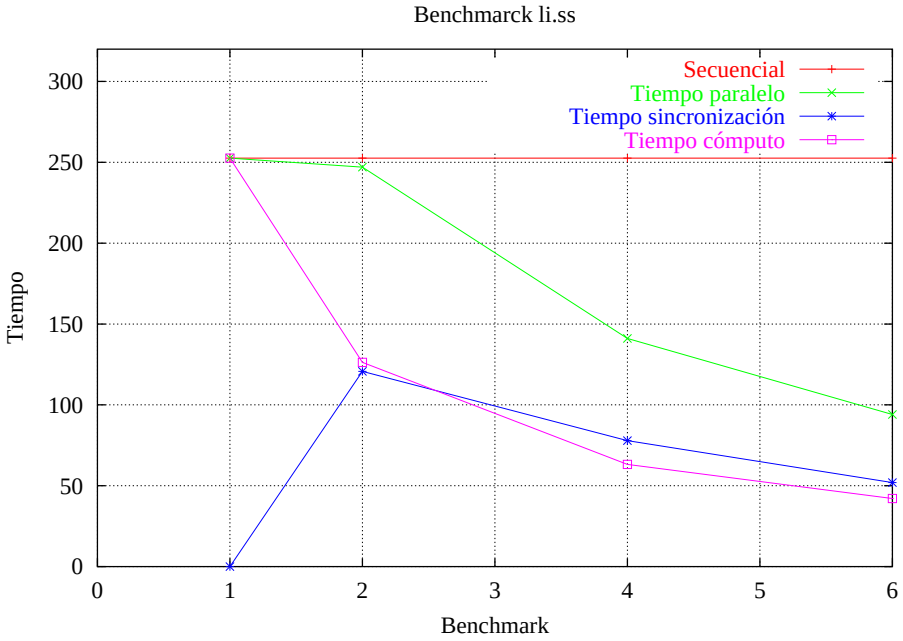


Figura 5.6: Tiempos de ejecución del benchmark li.ss

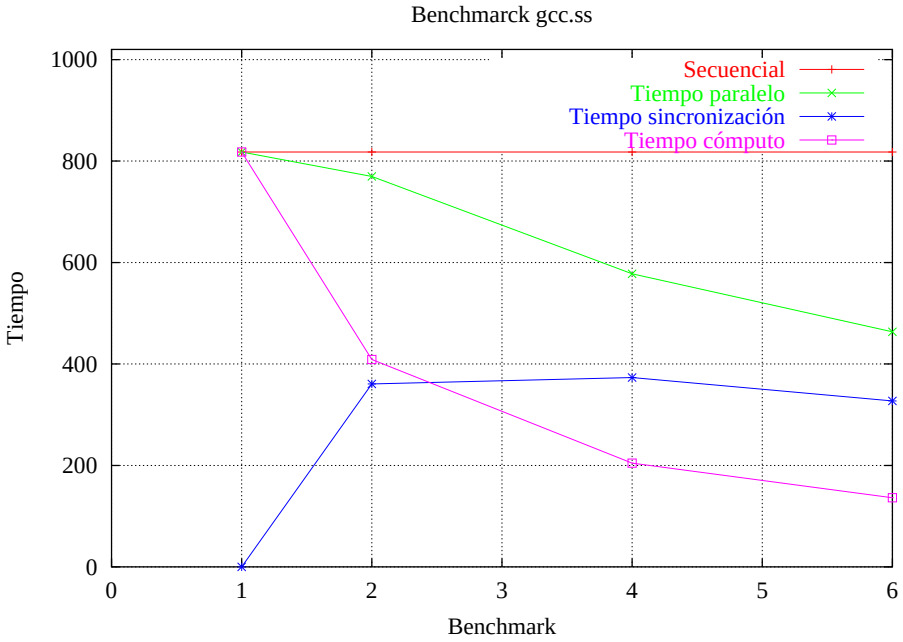


Figura 5.7: Tiempos de ejecución del benchmark gcc.ss

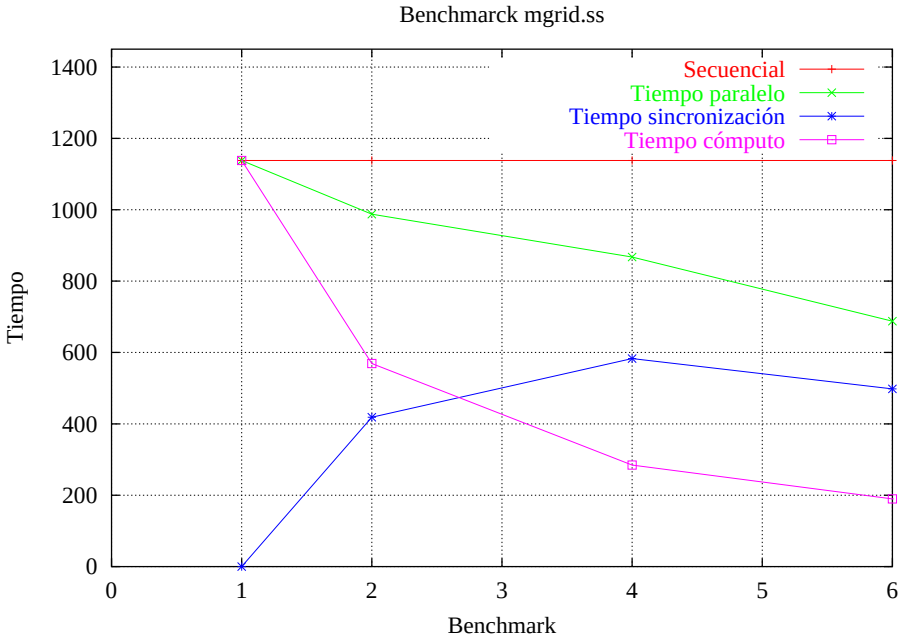


Figura 5.8: Tiempos de ejecución del benchmark mgrid.ss

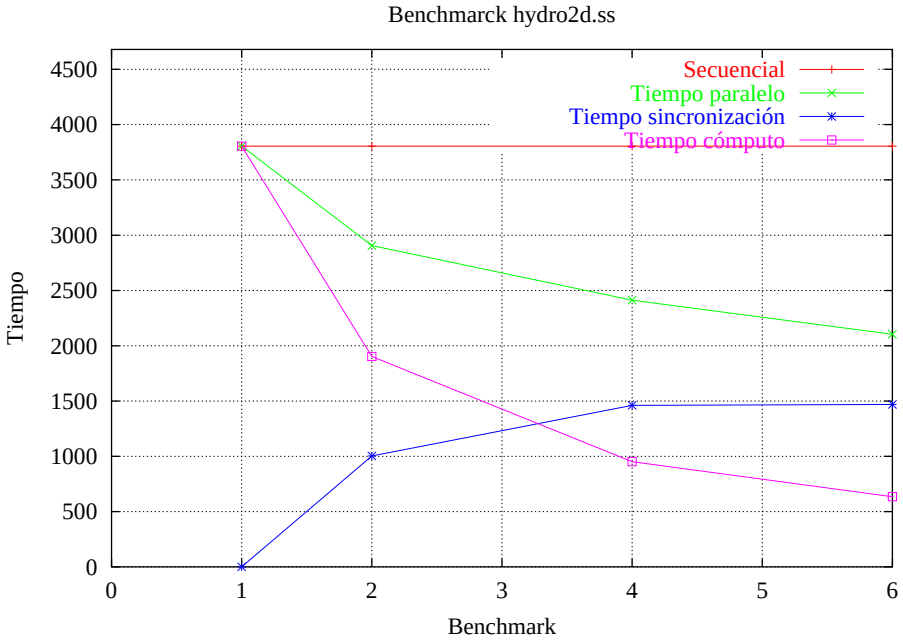


Figura 5.9: Tiempos de ejecución del benchmark hydro2d.ss

Conclusiones

La etapa de simulación es fundamental para verificar el funcionamiento correcto y el buen rendimiento de un procesador superescalar. En esta tesis se presentó una plataforma de experimentación para la ejecución de un conjunto de tareas de simulación de un procesador superescalar. Dicha plataforma considera dos aspectos diferentes del trabajo de simulación. Por un lado, se consideró que se parte de un conjunto de programas de prueba (*benchmarks*) conocidos. Por otra parte, se consideró que existen tareas de simulación con tiempos de ejecución largos. Para la evaluación del rendimiento del procesador SimpleScalar se realizaron simulaciones a partir de programas de prueba conocidos como SPEC95, precompilados para dicho procesador.

Dado que la evaluación se realiza con un conjunto de programas de prueba, se desarrolló una herramienta de balance de carga la cual permite distribuir las tareas de simulación entre un conjunto de computadoras que constituyen la plataforma de experimentación. Las computadoras utilizadas son heterogéneas, van desde computadoras con un solo procesador hasta computadoras con varios procesadores. La herramienta de balance de carga considera la heterogeneidad de la plataforma de experimentación y asigna velocidades de ejecución relativas a cada computadora. Así también, para la distribución de las tareas se tomaron en cuenta las estimaciones de los tiempos de ejecución de diferentes tareas de simulación sobre computadoras diferentes de la plataforma. Las velocidades relativas y los tiempos de ejecución relativos se utilizaron como una medida de referencia común para establecer la carga de trabajo de cada computadora.

En la distribución de las tareas de simulación se utilizó la heurística que distribuye las tareas más costosas computacionalmente a las computadoras más capaces. Los resultados de simulación y experimentales mostraron que esta estrategia es suficiente para tener un sistema balanceado en la mayoría de los casos. Los casos de desbalance se encontraron cuando las estimaciones de los tiempos de ejecución no coinciden con los tiempos reales observados. Para balancear el sistema en estos casos, se utilizó un tipo de migración de procesos el cual reasigna a otros procesadores las tareas cuya ejecución no ha iniciado. No se garantiza que en una sola fase de migración se logre tener un balance; sin embargo, los resultados de simulación mostraron que una vez detectado un desbalance del sistema, el balance se logra nuevamente en, a lo más, dos fases de migración.

Dada la dificultad de reproducir realmente las situaciones de desbalanceo del sistema, se utilizó un simulador del balanceador de carga para verificar su funcionamiento correcto. El desbalance se logró aplicando una variación a los tiempos de ejecución estimados con una varianza muy grande.

En general, los resultados de simulación y experimentales mostraron que con heurísticas simples y estrategias de balanceo y migración no muy sofisticados es suficiente para lograr un sistema balanceado.

En este trabajo se realizó también un estudio sobre la unidad de predicción de saltos y la unidad de memoria caché. El estudio realizado para la unidad de predicción de saltos indica que en general ésta es buena ya que el porcentaje de error para todas las pruebas está por debajo del 16%. Para los programas numéricos la predicción de saltos tiene un alto grado de acierto, las fallas son inferiores al 4%. Para programas de manipulación simbólica el porcentaje de aciertos de la predicción de saltos se reduce; el porcentaje de fallas está entre el 11% y el 16% . Los resultados experimentales para predictores combinados y bimodales son muy similares y solo se presenta una ligera variación cuando se incrementa el número de entradas en el buffer de direcciones de salto (BTB-branch target buffer).

Para la unidad de memoria caché, los resultados dependen de la configuración de la memoria caché así como de la aplicación sobre la cual se está realizando la simulación. Se observó un relación directamente proporcional entre la probabilidad de acierto y el tamaño de la memoria caché y de la longitud del bloque. Así también, entre mayor sea la asociatividad de la memoria, mayor es el porcentaje de aciertos. De igual forma que en el caso de la unidad de predicción de saltos, el porcentaje de acierto en la memoria caché es notablemente mayor para programas numéricos que para programas de manipulación simbólica.

Los resultados obtenidos con el simulador secuencial *sim-outorder* de SimpleScalar reflejan tiempos de simulación muy largos. Por lo tanto, para la simulación de tareas que requieren mayor trabajo de cómputo, es necesaria la reducción de los tiempos de ejecución. Una alternativa para ello es el uso de paralelismo. Dado que en simulación microarquitectural de procesadores se requiere un acoplamiento fuerte entre todas sus unidades funcionales, se desarrollaron técnicas *ad hoc* para resolver necesidades de comunicación y sincronización en un simulador paralelo. Se desarrolló un simulador paralelo, para una computadora de memoria compartida, del funcionamiento de un procesador superescalar basado en la arquitectura *pipeline* del simulador secuencial de SimpleScalar. La estrategia general fue crear un proceso por cada etapa de la arquitectura *pipeline* del procesor SimpleScalar. La sincronización se realizó mediante un reloj global el cual sincroniza cada etapa (proceso) al inicio y final. La orquestación se realizó obteniendo copias de las estructuras globales para cada etapa.

Para realizar la paralelización se desarrollaron tres versiones. En la primera versión se crearon dos hilos de ejecución, en la segunda versión se crearon 4 hilos de ejecución y en la última versión se crearon 6 hilos de ejecución. Las pruebas para todos los simuladores se realizaron en una arquitectura multiprocesador de 4 procesadores.

La versión del simulador paralelo con 2 hilos mostró que los tiempos de ejecución se reducen 12.26%. La versión del simulador paralelo con 4 hilos mostró que los tiempos de ejecución se reducen 33.35%. Finalmente, la versión del simulador paralelo con 6 *threads* mostró que los tiempos de ejecución se reducen 47.03%. La reducción en los tiempos de ejecución no es como uno desearía, el ideal 50%, 75% y 83%, respectivamente, porque el tiempo de sincronización es muy alto, dado que por cada instrucción es necesario realizar dos fases de sincronización. Por lo tanto, el tiempo de ejecución paralelo depende del número de instrucciones de la aplicación que se simula. De hecho, en este tipo de trabajos de simulación se observó que, cuanto mayor es el tiempo de cómputo, mayor es el tiempo de comunicación y sincronización, por lo que es difícil desarrollar simuladores paralelos escalables, esto es, que exhiban un rendimiento adecuado conforme se incremente el número de procesadores.

La limitante en el rendimiento del simulador paralelo observado sugiere el replanteamiento de la estrategia de particionamiento del simulador. El diseño implementado tiene como máximo grado de paralelismo 6 procesos, pues éste es el número de etapas de la arquitectura *pipeline* del procesador SimpleScalar. Si se dispusiera de una computadora con más de seis procesadores es necesario plantear una nueva estrategia de particionamiento para explotar un paralelismo de grano más fino, lo cual permitiría maximizar el uso de los procesadores.

Una alternativa que podría mejorar la eficiencia de utilización es utilizar un *thread* por cada unidad funcional del procesador, sin embargo, como fue mostrado en este trabajo, los costos de sincronización son importantes y, por tanto, habría que encontrar un balance adecuado entre grado de paralelismo y costos de sincronización.

Otra mejora podría ser el reducir los costos de sincronización evitando el uso de un reloj global y desacoplando por completo las etapas de la arquitectura segmentada. Por ejemplo, es posible utilizar un mecanismo tipo productor-consumidor de tal manera que los hilos tengan que sincronizarse sólo hasta que un *buffer* intermedio entre cada etapa quede completamente lleno o vacío.

Este trabajo puede ser completado mediante un estudio del rendimiento de la unidad de especulación del procesador SimpleScalar. Sin embargo, tal estudio requeriría de un conjunto de benchmarks más amplio y variado para tener resultados aceptables. Los benchmarks utilizados aquí corresponden al conjunto SPEC95. Para actualizar el conjunto de benchmarks se pueden utilizar los SPEC2000, compilarlos para el procesador SimpleScalar y hacer las evaluaciones corres-

pondientes. Desafortunadamente, a diferencia de los SPEC95, los SPEC2000 solo se pueden obtener mediante contratos de licenciamiento, por lo que no fue posible su uso en esta tesis.

Bibliografía

[1] Gregory R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addisson-Wesley, 2000.

[2] Doug Burger and Todd M. Austin,“ The SimpleScalar Tool Set, Version 2.0, ” University of Wisconsin-Madison Computer Sciences Departament Technical Report # 1342, June 1997.

[3] Todd Austin, Eric Larson, Dan Ernst, “ SimpleScalar: An infraestructure for Computer System Modeling, ” IEEE Computer, Vol.35, No.2, pp. 59-67, February 2002

[4] Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale N. Patt, “ An analysis of correlation and predictability: what makes two-level branch predictors work, ” ACM SIGARCH Computer Architecture News, Vol.26, No.3, pp. 52-61, June 1998

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, (MIT Electrical Engineering and Computer Science Series), The MIT Press, ISBN:0262031418, 1990.

[6] George Coulouris, Jean Dollimore and Tim Kindberg, *Distributed systems. Concepts and design*, Addison-Wesley, 2001

[7] David E. Culler and Jaswinder Pal Singh, *Parallel Computer Architecture A Hardware/Software Approach*,1999.

[8] Yeimkuan Chang and Bhuyan L.N., “ An efficient tree cache coherence protocol for distributed shared memory multiprocessors ”, IEEE Transactions on Computers, Vol. 48, No.3, pp. 352-360, March 1999

[9] Mario Farias Elinos, *Reasignación de tareas de un sistema multiprocesador*, CINVESTAV, Departamento de Ingeniería Eléctrica, Sección Computación, 1999

[10] Fujimoto, M. Richard. *Parallel and Distributed Simulation Systems*. John Wiley & Sons. October 1999.

[11] David R. Jefferson, " Virtual time," ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 7, No. 3, pp. 404-425, July 1985

[12] Erik G. Hallnor, and Steven K. Reinhardt, " A fully associative software-managed cache design, " ACM SIGARCH Computer Architecture News, Vol.28, No.2, pp. 107-116, May 2000

[13] John L. Hennessy and David A. Patterson *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, INC. San Francisco, California. 1996

[14] John L. Henning, " SPEC CPU2000: Measuring CPU Performance in the New Millenium.," IEEE Computer, pp. 28-35, July 2000.

[15] Kai Hwang, *Advanced Computer Architecture: Paralelism, Scalability, Programmability*, McGraw-Hill, Inc. 1993.

[16] Intel Architecture Software Developer's Manual, Volumen:Basic Architecture, Order Number 243190, 1999.

[17] Hantak Kwak, Ben Lee, Ali R. Hurson, Suk-Han Yoon and Woo-Jong Hahn, " Effects of Multithreading on Cache Performance, " IEEE Transactions on Computers, Vol. 48, No 2, pp. 176-184, February 1999

[18] W. David Kelton, Randall P. Sadowski and Deborah A. Sadowski, *Simulation with Arena*, WCB/McGraw-Hill, 1998.

[19] Vipin Kumar, Anauth Grama, Anshul Gufta, and George Karypis, *Introduction to Parallel Computing, Design and analysis of algorithms*, The Benjamin/Cummings Publishing Company, Inc., 1995.

[20] Ganesh Lakshminarayana, and Anand Raghunathan, " Incorporating speculative execution into scheduling of control-flow-intensive designs ", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.19 No.3 , pp.308 -324 March 2000

[21] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge, " The Bi-Mode Branch Predictor," IEEE/ACM Proceedings; Thirtieth Annual International Symposium Microarchitecture, 1997, pp. 4-13

[22] Tao Li and Lizy Kurian John, " ADir/sub p/NB: a cost-effective way to implement full map directory-based cache coherence protocols ", IEEE Transactions on Computers, Vol.50, No.9 , pp. 921-934, Sept. 2001

[23] Benchmarks CPU95; http://www.spec.org/cpu95