Paralelismo a Nivel de Instrucción

Esteban F. Corrales
Universidad de Costa Rica, Facultad de Ingeniería,
Escuela de Ciencias de la Computación e Informática, San Pedro, Costa Rica
esteban.corrales@ucr.ac.cr

y

Kenneth Cascante
Universidad de Costa Rica, Facultad de Ingeniería,
Escuela de Ciencias de la Computación e Informática, San Pedro, Costa Rica
kennethcr@gmail.com

Abstract

Instruction level parallelism is a technique that seeks that combination of instruction of low level running by a processor can be ordered so at moment to be processed simultaneously do no affect the final result of a program, and rather increase speed and maximize hardware capabilities. Instruction pipelines allow that for each clock cycle of processor multiple instructions can be in different stages of execution.

Keywords: ILP, Pipeline, Scheduler, Parallelism, Instruction

Resumen

El paralelismo a nivel de instrucción consiste en un técnica que busca que la combinación de instrucciones de bajo nivel que ejecuta un procesador puedan ser ordenadas de forma tal que al ser procesadas en simultáneo no afecten el resultado final del programa, y más bien incrementen la velocidad y aprovechen al máximo las capacidades del hardware. Un pipeline (canalizador) de instrucciones es el que permite que por cada ciclo de reloj del procesador múltiples instrucciones se encuentren en distintas fases de ejecución.

Palabras clave: ILP, Pipeline, Calendalizador, Paralelismo, Instrucción

1 Introducción

La evolución en la arquitectura de procesadores permitió que la ejecución de instrucciones en simultáneo pasara de ser algo imaginario a algo real. En sus inicios los procesadores solo podían ejecutar una instrucción por ciclo de reloj. La multitarea y la ejecución de procesos en paralelo, era el objetivo por lograr. Claro está que en esos momentos las limitaciones de hardware eran muchas, y el tamaño la memoria de acceso aleatorio era una de las principales.

Con el avance en la arquitectura de las computadoras, la investigación y el diseño sofisticado de procesadores se construye paso a paso la capacidad de multiprocesamiento y la ejecución de múltiples instrucciones por ciclo de reloj. El ciclo de reloj consiste en la unidad de medida utilizada para medir la velocidad en la que un procesador ejecuta las instrucciones básicas.

Con el arribo de máquinas con procesadores multinúcleo el tema del paralelismo ha tomado mayor relevancia; el manejo de hilos de ejecución, métodos de sincronización y concurrencia consisten en temas fundamentales que no pueden ser obviados en la creación de aplicaciones de alto nivel.

Existen diversos tipos de paralelismo, entre los que destacan el paralelismo a nivel de bit, paralelismo por datos, paralelismo por control y el paralelismo a nivel de instrucción.

El paralelismo a nivel de instrucción se utilizó por primera vez en las arquitecturas computacionales como un medio para agilizar el código de máquina secuencial ordinario. La pregunta clave en su momento fue: “¿qué tan rápido puede ejecutarse un programa en un procesador con paralelismo a nivel de instrucción?”

Gracias a la investigación, se obtuvo la conclusión de que la respuesta dependía de 3 premisas:

1. El paralelismo potencial en el programa.
2. El paralelismo disponible en el procesador.
3. Capacidad del programador de extraer el paralelismo del programa secuencial original.
4. Capacidad del programador para encontrar la mejor calendarización en paralelo, dadas las restricciones de programación.

Si las instrucciones de un programa dependen en gran parte unas de otras, entonces no hay hardware ni técnicas de paralelización que puedan hacer que el programa se ejecute con rapidez en paralelo. Las técnicas empleadas buscan alta cohesión y bajo acoplamiento en cuanto a instrucciones se refiere.

Comprender los límites que la paralelización a nivel de instrucción puede tener ha dado paso a gran cantidad de investigación, ya que existen múltiples razones que dificultan que la misma se dé la manera más adecuada, eficiente y con alto desempeño.

Una gran cantidad de programas poseen distintas salidas que dependen de otros datos, lo cual no permite la predicción de las instrucciones que deben seguir ejecutándose, y mucho menos decidir cuáles se pueden ejecutar en paralelo. Es por esté motivo que el trabajo en este ámbito se ha centrado en la flexibilización de las restricciones de programación, incluyendo la introducción de nuevas características arquitectónicas.

Las aplicaciones numéricas como los cálculos científicos tienden a presentar mayor paralelismo. Estas aplicaciones manejan grandes estructuras de datos en conjunto, en las cuales las operaciones muy a menudo son independientes unas de otras, y ello les permite ser ejecutadas en paralelo. La escritura código para estas aplicaciones es interesante y considerable, ya que se constituye de un gran número de instrucciones independientes que se asignan a un gran número de recursos.

Tanto la extracción del paralelismo y la calendarización para la ejecución en paralelo pueden realizarse de forma estática en el software o de forma dinámica en el hardware, y también ocurre que las máquinas con calendarización por hardware se vean beneficiadas con la calendarización basada en software.

Especial importancia posee el análisis de dependencias de datos básicas entre instrucciones, ya que resulta fundamental para la extracción de paralelismo y optimizaciones de código.

2 Arquitecturas de Procesadores

Arquitectura de una Computadora.

Definir la arquitectura teniendo en cuenta sus requerimientos funcionales, logrando el mejor desempeño dentro de las restricciones de precio, energía, tecnología disponible (estado del arte) comprende del diseño de:

2.1.1 Arquitectura del Conjunto de Instrucciones:

- Almacenamiento de operandos en la CPU (reg-reg actualmente)
- Modos de direccionamiento
- Tipo y tamaño de operandos
- Operaciones
- Instrucciones de control de flujo
- Codificación

La definición del conjunto de instrucciones sobre las cuáles el procesador trabaja consiste en una de las tareas prioritarias que el diseñador de procesadores debe tomar en cuenta ya que una vez que se encuentran establecidas es casi imposible cambiarlas sin tener que desechar todo. Las operaciones más comunes que todo procesador posee son:

- Instrucciones para el movimiento de datos (MOV)
- Instrucciones aritmético-lógicas (ADD, SUB, MUL, DIV, AND, OR, NOT)
- Instrucciones de comparación (CMP)
- Instrucciones de salto (BEQZ, BNEZ)
- Instrucciones de acceso a memoria (LW, SW)
- Instrucciones de entrada/salida, entre otras.

2.2.2 Organización Funcional

- Sistema de memoria
- Interconexiones de memoria
- Diseño del procesador interno
-

2.2.3 Hardware que Permite la Implantación de Dichos Puntos

Al pensar en el paralelismo a nivel de instrucción, por lo general, se piensa en un procesador que emite varias operaciones en un solo ciclo de reloj. De hecho, es posible que una máquina emita sólo una operación por reloj y, aún así lograr paralelismo a nivel de instrucción mediante el concepto de pipelining [1].

2.2 Pipeline de Instrucciones y Retrasos de Saltos (Branch Delays)

Prácticamente todos los procesadores, ya sea que se encuentren en supercomputadoras de alto rendimiento, máquinas de escritorio, consolas de videojuego, sistemas empotrados, etc., utilizan un pipeline de instrucciones. En el mismo puede obtenerse una nueva instrucción en cada ciclo de reloj, mientras que las siguientes instrucciones pueden seguir ingresando conforme el reloj avanza. En la Fig. 10,1 se muestra un pipeline de instrucciones simples compuesto por 5 etapas:

[1] Se utilizará el término de pipelining que es más utilizado en el ámbito computacional, aunque estemos ante un anglicismo ya que de lo contrario el sentido de algunas ideas podrían prestarse a confusión.

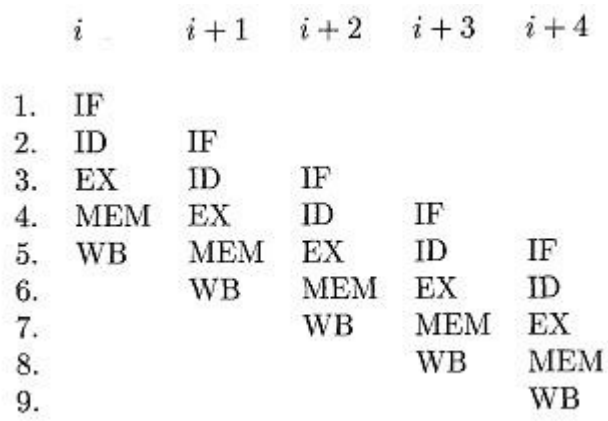

Figura 1: Pipeline de 5 etapas ejecutando 5 instrucciones consecutivas.

IF: Instruction Fetch: obtiene la instrucción.

ID: Instruction Decode: decodifica la instrucción.

EX: Execute: ejecuta la operación.

MEM: Memory: accede la memoria (lecturas y escrituras)

WB: Write Back: escribe el resultado con lo que se finaliza la ejecución de instrucción.

La figura muestra cómo las instrucciones i, i + 1, i + 2, i + 3, i + 4 pueden ser ejecutadas al mismo tiempo.

Si el resultado de una instrucción se encuentra disponible, y una instrucción siguiente necesita de ellos para ser ejecutada, el procesador será entonces capaz de emitir una instrucción cada ciclo de reloj.

Precisamente los problemas se dan cuando los resultados tardan en ser procesados, y también cuando se presentan instrucciones de salto (en inglés: branches) ya que hasta que son obtenidas y decodificadas (lo cual no es inmediato) permiten la ejecución, pués el procesador no puede ubicar la instrucción que sigue. Algunas técnicas en procesadores obtienen y decodifican de manera especulativa las instrucciones que siguen luego de los saltos, a este tipo de técnicas se le llama predicción de branches.

También existen procesadores avanzados que utilizan hardware para predecir resultados de saltos con base al historial de ejecución, y con ello ganan tiempo en ciclos de reloj. Aún así existe la posibilidad de que se den fallos de predicción lo cual ocasiona retrasos inherentes.

2.3 Ejecución Mediante Pipeline

Algunas instrucciones requieren de varios ciclos de reloj para ser ejecutadas. Un ejemplo común es la operación de cargado de memoria (load). Aún cuando el procesador toma los datos de la memoria de más alta velocidad: memoria caché, al ser está de menor tamaño, puede no tener el dato solicitado, y entonces se origina un fallo de caché. El mismo debe ser resuelto y por lo general tarda una buena cantidad de ciclos de reloj en realizarse.

Se dice que la ejecución de instrucciones está “pipelineada” cuando las instrucciones siguientes no son dependientes del resultado de las anteriores, lo cual les permite proceder. Es por ello que aún cuando el procesador solo puede emitir una operación por ciclo de reloj, varias operaciones podrían estar en sus etapas de ejecución al mismo tiempo.

Se debe tener en cuenta que no todas las instrucciones se “pipelinean” por completo. Sumas y multiplicaciones lo hacen, pero divisiones al ser de mayor complejidad no.

La mayoría de los CPU de propósito general detectan en forma dinámica las dependencias entre instrucciones consecutivas y automáticamente detienen la ejecución en caso de que sus operandos no se encuentren disponibles. Otros procesadores, especialmente los incrustrados en dispositivos portátiles, dejan la comprobación de dependencias al software con el fin de mantener un hardware simple con bajo consumo de energía. En este caso, el compilador es responsable de la inserción de instrucciones "no-operation" en el código, para asegurar que los resultados estén disponibles cuando se requieran.

2.4 Emisión de Varias Instrucciones

Mediante la emisión de varias instrucciones por cada ciclo de reloj, los procesadores logran mantenerlas en ejecución de manera simultánea. El promedio de instrucciones capaces de correr en simultáneo dependen del tamaño de emisión de instrucciones y del número promedio de etapas que tiene el pipeline.

Al igual que pipelining, el paralelismo en máquinas que emiten varias instrucciones puede administrarse, ya sea mediante software o hardware.

2.4.1 Por Software: VLIW (Very-Long-Instruction-Word, Palabra de Instrucción Larga)

Poseen palabras de instrucciones más grandes de lo normal. Esto permite que sean codificadas para ser emitidas en un solo ciclo de reloj. El compilador decide qué operaciones se van a emitir en paralelo y codifica explícitamente la información en código de máquina.

2.4.2 Por Hardware: Máquinas Superescalares.

Poseen un conjunto de instrucciones regular, con una semántica de ejecución secuencial ordinaria. Detectan de manera automática las dependencias entre las instrucciones y las emiten a medida que sus operandos se encuentran disponibles.

Existen procesadores con ambas capacidades de administración.

2.5 Tipos de Calendarizadores

Existen dos tipos de calendarizadores: simples y sofisticados.

Los simples ejecutan las instrucciones en el orden en el que se obtienen. Si encuentra una instrucción dependiente, ésta y todas las instrucciones que le siguen deben esperar hasta que las dependencias sean resueltas. Estas máquinas, evidentemente, pueden beneficiarse de tener un calendarizador estático que coloque las operaciones independientes una enseguida de la otra, en el orden de ejecución.

Los calendarizadores más sofisticados pueden ejecutar instrucciones "fuera de orden". Las instrucciones se detienen de manera independiente y no pueden ejecutarse hasta que se hayan producido todos los valores de los cuales dependen. Inclusive se benefician de los calendarizadores estáticos, ya que sólo tienen un espacio limitado para colocar en búfer las instrucciones que deben detenerse. La calendarización estática permite colocar las instrucciones independientes cerca unas de otras, con lo cual se logra mayor paralelización y mejor utilización del hardware.

Además, sin importar qué tan sofisticado sea un calendarizador dinámico, no puede ejecutar instrucciones que no haya obtenido. Cuando el procesador tiene que tomar un salto inesperado, sólo puede encontrar paralelismo entre las instrucciones recién obtenidas. El compilador puede mejorar el rendimiento del calendarizador dinámico, al asegurar que estas instrucciones puedan ser ejecutadas en paralelo.

3 Restricciones en la Calendarización del Código

La calendarización de código es una forma de optimización en los programas, que se aplica al código máquina producido por el generador de código. Se encuentra sujeta a tres tipos de restricciones:

1. Restricciones de dependencia de control: todas las operaciones ejecutadas en el programa original deben ser ejecutadas en el programa optimizado.

2. Restricciones de dependencia de datos: las operaciones en el programa optimizado deben producir los mismos resultados que las operaciones correspondientes en el programa original
3. Restricciones de recursos: el calendarizador no debe excederse en las solicitudes de recursos de la máquina.

Esto va a garantizar que el programa optimizado produce los mismos resultados que el original. Sin embargo, debido a que la calendarización de código modifica el orden en el que se ejecutan las operaciones, el estado de la memoria en cualquier punto dado podría no coincidir con cualquiera de los estados de memoria en una ejecución secuencial. Esto implica un problema si la ejecución de un programa es interrumpida, por ejemplo, debido al lanzamiento de una excepción o puntos de interrupción. Es por ello que los programas optimizados son más difíciles de depurar.

3.1 Dependencias de Datos

Al modificar el orden de ejecución de dos instrucciones que no posean variables en común, sus resultados nunca son afectados. Inclusive si se da lectura de la misma variable, la ejecución no se ve afectada. Los problemas se dan cuando se dan escrituras, ya que los resultados pueden ser alterados al modificar órdenes de ejecución. Existen tres tipos de dependencia de datos:

1. Dependencia verdadera: lectura después de escritura (RAW: read after write): la instrucción j lee antes de que i escriba con lo cuál se da la lectura de un valor viejo. Es el tipo de conflicto de datos más común y se puede resolver mediante “forwarding”.

2. Antidependencia: escritura después de lectura (WAR: write after read): si luego de una escritura se da una lectura. Se lee un valor incorrecto o dato sucio. Es posible eliminarla si se da el almacenamiento de valores en ubicaciones distintas.

3. Dependencia de salida: escritura después de escritura (WAW: write after write): dos escrituras a la misma ubicación comparten una dependencia de salida. Al ocurrir el valor de la ubicación de memoria escrita tendrá un valor incorrecto ya que j escribe antes de que i lo haga con lo cual queda escrito un valor incorrecto.

A la antidependencia y la dependencia de salida se les llama dependencias relacionadas con el almacenamiento, y pueden eliminarse mediante el uso de distintas ubicaciones de memoria para almacenar valores distintos. Señalar que todas las dependencias de datos ocurren tanto para accesos a memoria como en accesos a registros.

3.2 Búsqueda de Dependencias Entre Accesos a Memoria

Para comprobar si dos accesos a memoria comparten una dependencia de datos, sólo se debe saber si pueden referirse a la misma ubicación. La dependencia de datos es indecibible en tiempo de compilación.

El análisis de dependencia de datos es altamente sensible al lenguaje de programación utilizado en el programa. Para lenguajes sin seguridad de tipos como C y C++, en donde un puntero puede convertirse para apuntar a cualquier tipo de objeto, es necesario el análisis sofisticado para probar la independencia entre cualquier par de accesos a memoria basados en apuntadores. En lenguajes con seguridad de tipos, como Java, los objetos de tipos diferentes son necesariamente distintos unos de otros.

Un correcto descubrimiento de dependencias de datos requiere de diferentes formas de análisis.

3.3 Ordenamiento de Fases Entre la Asignación de Registros y la Calendarización de Código

Si los registros se asignan antes de la calendarización de código se obtienen muchas dependencias de almacenamiento que la limitan.

Si fuese al contrario (la calendarización de código se da antes de la asignación de registros), el programa podría requerir demasiados registros, con lo que se verían comprometidas las ventajas del paralelismo a nivel de instrucción. Para hallar la solución adecuada a este problema se deben considerar las características de los programas a compilar.

Aplicaciones no numéricas no poseen tanto paralelismo disponible, por lo que basta con asignar un pequeño número de registros para guardar los resultados temporales. Se describe en tres fases:

a) Se asignan los registros a todas las variables fijas.

b) Calendarización del código.
c) Asignación de registros a variables temporales (generadas por el compilador para almacenamiento de resultados parciales).

Para aplicaciones numéricas con expresiones mucho más extensas se puede utilizar un método jerárquico, en el cuál el código se optimiza de interior a exterior, empezando desde los ciclos más internos. Descrito en 4 fases:

a) Las instrucciones son calendarizadas suponiendo que a cada seudorregistro se le asigna su propio registro físico.
b) Asignación de registros.
c) Recalendarización de código.

Luego el proceso se repite para los ciclos más externos.

3.4 Dependencia del Control

La calendarización de instrucciones dentro de un bloque básico es relativamente simple, debido a que se garantiza que todas se ejecutan una vez que el flujo de control llega al inicio del bloque. Las instrucciones pueden reordenarse en forma arbitraria, siempre y cuando se satisfagan todas las dependencias de datos. La principal desventaja es que los bloques básicos, en especial en programas no numéricos, son en general muy pequeños (en promedio, hay aproximadamente sólo cinco instrucciones en un bloque básico). Además, a menudo las instrucciones en un mismo bloque se encuentran altamente relacionadas y, por ende, tienen poco paralelismo. Es por ello que la explotación del paralelismo en bloques básicos es de vital importancia.

Se dice que una instrucción i_1 es dependiente del control en la instrucción i_2 si el resultado de i_2 determina si se va a ejecutar i_1. La noción de dependencia del control corresponde al concepto de anidar niveles en los programas estructurados por bloques. Ejemplos:

If (c) s_1; else s_2; //s_1 y s_2 dependen del control en c

while (c) s; //s es dependiente del control en c

3.5 Soporte de Ejecución Especulativa

Un programa optimizado debe ejecutar todas las instrucciones del programa original. Inclusive puede ejecutar más, siempre y cuando las adicionadas no cambien la lógica del programa. Si se sabe que existen altas probabilidades de que una instrucción vaya a ser ejecutada, y hay un recurso inactivo disponible para realizar la operación “sin costo”, la misma puede ser ejecutada en forma especulativa, y el programa se ejecutará con mayor rapidez si la especulación resulta ser correcta.

La instrucción más beneficiada con está técnica es la de cargar de memoria, ya que poseen latencias de ejecución extensas. Las direcciones que utilizan por lo general se encuentran disponibles con anticipación. Los problemas que presenta se dan cuando ocurren excepciones por direcciones ilegales, que conllevan a interrupciones inesperadas y aumentos en fallos de caché.

Procesadores de alto rendimiento proporcionan características especiales de soporte a accesos especulativos a memoria, entre los más populares se encuentran:

3.5.1 Preobtención (Prefetch)

Es una instrucción que se inventó con el fin de llevar los datos de la memoria a la caché antes de usarlos. Indica al procesador altas probabilidades de que el programa utilice una palabra de memoria específica en un futuro cercano.

3.5.2 Bits venenosos

Permite la carga especulativa de datos de memoria al archivo de registro. Cada registro se aumenta con un bit venenoso. Si se accede a la memoria ilegal el procesador no produce una excepción inmediata, sino que sólo establece el bit venenoso del registro de destino. La excepción se da cuando se utiliza el contenido del registro con el bit marcado.

3.5.3 Ejecución predicada

Las instrucciones predicadas reducen la cantidad de saltos en un programa. Es como una instrucción normal, sólo que tiene un operando predicado adicional para proteger su ejecución; y se ejecuta sólo si el mismo es verdadero. El costo asociado que poseen es que se obtienen y se codifican, aun cuando podrían no ser ejecutadas.

3.6 Un Modelo de Máquina Básico

$M = <R, T>$, consiste en:

1. *T:* un conjunto de instrucciones: cargas, almacenamiento, operaciones aritméticas, etc., en donde cada una tiene un conjunto de operandos de entrada, de salida y un requerimiento de recursos.

2. *R:* un vector *R = [r1, r2,...]* que representa los recursos de hardware: ALUs, unidades de acceso a memoria, etc. El uso de recursos para cada tipo de operación *t* de la máquina se modela mediante una tabla de reservación de recursos bidimensional, RT_t, en la cual la anchura representa el número de tipos de recursos disponibles, y la altura la duración a través de la cual la operación utiliza los recursos.

4 Calendalizador de Bloques Básicos

El problema de la calendarización de instrucciones en un bloque básico posee una solución óptima de NP-completo. Pero en la práctica, un bloque básico ordinario sólo tiene un pequeño número de instrucciones altamente restringidas, por lo que basta con emplear técnicas simples de calendarización.

4.1 Grafos de Dependencia de Datos

Cada bloque básico de instrucciones de máquina es representado mediante un grafo de dependencia de datos, G= (N, E), el cual tiene un conjunto de nodos N que representan las operaciones en las instrucciones de máquina en el bloque, y un conjunto de aristas dirigidas E que representan las restricciones de dependencia de datos entre las operaciones. Un flanco de n a m etiquetado con d indica que la instrucción m debe empezar al menos d ciclos de reloj después de que inicia la instrucción n.

4.2 Calendarización por Listas de Bloques Básicos

El método más simple para calendarizar bloques básicos implica la acción de visitar cada nodo del grafo de dependencia de datos en "orden topológico priorizado". Como no puede haber ciclos en el grafo, siempre hay por lo menos un orden topológico para los nodos.

Los nodos pueden o no terminar siendo calendarizados en el mismo orden en el que se visitan, pero las instrucciones se colocan en el programa tan pronto como sea posible, por lo que hay una tendencia a que las mismas se calendaricen en el orden visitado.

5 Calendarización de Código Global

Para una máquina con una cantidad moderada de paralelismo a nivel de instrucción, los programas creados por compactación de los distintos bloques básicos individuales tienden a dejar muchos recursos inactivos. Para poder aprovecharlos de la mejor manera es necesario considerar estrategias de generación de código que muevan las instrucciones de un bloque básico a otro. Para ello se crearon los algoritmos de calendarización global, los cuales consideran no sólo las dependencias de datos, sino también las de control. Se deben asegurar dos premisas:

a) Todas las instrucciones en el programa original se ejecutan en el optimizado.

b) Aunque el programa optimizado puede ejecutar instrucciones adicionales en forma especulativa, estas instrucciones no deben tener efectos secundarios no deseados.

5.1 Algoritmos de Calendarización Global

Está bien establecido que más del 90% del tiempo de ejecución de un programa se invierte en menos del 10% del código. Por lo tanto, debe lograrse que los caminos ejecutados con frecuencia lo hagan con mayor rapidez, mientras que los menos frecuentes lo hagan más lento, sin que esto tenga consecuencias relevantes.

Existen una variedad de técnicas que un compilador puede utilizar para estimar las frecuencias de ejecución, de las cuales las mejores se obtienen de manera dinámica. En esta técnica, los calendarizadores registran los resultados de los saltos condicionales, a medida que se ejecutan. Luego los programas se ejecutan con entradas representativas, para determinar cómo se van a comportar en general, obteniéndose de esta forma resultados de alta precisión.

5.1.1 Calendarización Basada en Regiones

Soporta las dos formas más sencillas de movimiento de código:
a) Mover las operaciones hacia arriba, a los bloques básicos de control equivalente.
b) Mover las operaciones en forma especulativa un salto hacia arriba, hasta un predecesor dominante.

5.1.2 Desenrollamiento de Ciclos

En la calendarización basada en regiones, el límite de una iteración de un ciclo es una barrera para el movimiento de código. Las instrucciones en una iteración no pueden traslaparse con las de otra.

Una técnica sencilla pero muy eficaz para mitigar este problema consiste en desenrollar el ciclo un pequeño número de veces antes de la calendarización del código. El mismo crea más instrucciones en el cuerpo del ciclo, lo cual permite a los algoritmos de calendarización global encontrar más paralelismo.

```
for (i = 0; i < N; i++) {
        S(i) ;
}
```

```
for (i = 0; i+4 < N; i+=4) {
        S(i) ;
        S(i+l);
        S(i+2);
        S(i+3);
}
```

5.2 Técnicas Avanzadas de Movimiento de Código

Si nuestra máquina es calendarizada en forma estática y tiene mucho paralelismo a nivel de instrucción, es posible que sea necesario un algoritmo más agresivo, por lo que se pueden añadir extensiones adicionales como:

1. Agregar nuevos bloques básicos a lo largo de los flancos de flujo de control que se originan de bloques con más de un predecesor.
2. En el algoritmo de calendarización basada en regiones, el código a ejecutar en cada bloque básico se calendariza de una vez por todas, a medida que se visita cada bloque.

5.3 Interacción con Calendarizadores Dinámicos

Un calendarizador dinámico tiene la ventaja de que puede crear nuevos calendarizadores de acuerdo a las condiciones en tiempo de ejecución, sin tener que codificar todos antes de tiempo. Si una máquina posee un calendarizador dinámico, la principal función del estático consiste en asegurar que las instrucciones con alta latencia se obtengan antes de tiempo, para que el dinámico pueda emitirlas tan pronto como sea posible.

Los fallos de caché podrían afectar el desempeño del programa. Si existen instrucciones de preobtención de datos disponibles, el calendarizador estático puede ayudar al dinámico de manera considerable, al colocar dichas instrucciones con suficiente anticipación para que los datos se encuentren en caché.

Si la calendarización dinámica no es posible, la estática debe ser conservadora y separar cada par dependiente de datos de instrucciones, en base al retraso mínimo.

Si la calendarización dinámica está disponible, el compilador sólo necesita colocar las operaciones dependientes de datos en el orden correcto, para garantizar que el programa lo esté.

Para obtener un mejor rendimiento, el compilador debe asignar retrasos extensos a las dependencias que tienen probabilidad de ocurrir, y retrasos cortos a las dependencias que no. El mal pronóstico del cálculo en instrucciones de salto consiste en una de las principales causas de pérdida en cuánto a rendimientos de ejecución por lo que se les debe prestar especial atención.

6 Pipelining por Software

El pipelining por software es una técnica que explota la habilidad de una máquina para ejecutar varias instrucciones a la vez. Las iteraciones del ciclo son calendarizadas para comenzar en pequeños intervalos, que permita que el ciclo pueda ejecutarse con rapidez, aprovechando al máximo el paralelismo entre iteraciones. Esto resulta de gran importancia, ya que hay relativamente poco paralelismo entre las instrucciones en una sola iteración.

El desenrollamiento de ciclos mejora el rendimiento, debido a que traslapa el cálculo de las iteraciones, pero aún así el límite del ciclo desenrollado sigue actuando como barrera para el movimiento de código. Es decir la mejora en rendimiento podría ser aún mayor si esto es tratado adecuadamente.

La técnica de pipelining por software superpone una serie de iteraciones consecutivas continuas hasta que se agotan, produciendo un código compacto y eficiente.

Tómese como ejemplo el ciclo:

```
for ( i = 0;  i < n; i++)
        D [ i ] = A[ i ] * B[ i ] + c;
```

Las iteraciones en el ciclo escriben en distintas localidades de memoria, que por sí solas son diferentes de cualquiera de las ubicaciones leídas, por lo tanto, no hay dependencias de memoria y todas pueden proceder en paralelo.

En general, se obtiene un mejor uso del hardware al desenrollar varias iteraciones de un ciclo. El problema que esto conlleva es el aumento en tamaño de código, lo que a su vez puede tener un impacto negativo sobre el rendimiento en general. Es por ello que se debe elegir un número finito de veces para desenrollar un ciclo, que pueda obtener la mayoría de la mejora en el rendimiento sin que se expanda el código demasiado.

El desenrollamiento de un ciclo coloca varias iteraciones del mismo en un bloque básico grande, con lo cual puede utilizarse un algoritmo simple de calendarización por lista para las operaciones a ejecutar en paralelo.

6.1 Pipelining por Software Mediante Ciclos

Consiste en una técnica análoga utilizada en la calendarización del “pipelining” por hardware. La misma proporciona una manera conveniente de obtener un uso óptimo de los recursos y un código compacto al mismo tiempo.

El calendarizador debe ser elegido cuidadosamente a fin de optimizar la tasa de transferencia. Un calendarizador compactado en forma local, aunque disminuye al mínimo el tiempo para completar una iteración, puede producir una tasa de transferencia por debajo de la óptima a la hora de ejecutar el pipeline.

6.2 Objetivos y Restricciones del Pipelining por Software

El objetivo principal del pipelining por software es maximizar la tasa de transferencia de un ciclo de larga duración, manteniendo el tamaño del código generado lo más pequeño posible. Esto es logrado al requerir que el calendalizador relativo de cada iteración sea el mismo, además de que las iteraciones se inicien en un intervalo constante.

Un calendalizador de pipeline por software para un grafo de dependencia de datos $G=(N, E)$ puede especificarse mediante 2 parámetros:

a) Un intervalo de iniciación T

b) Un calendarizador relativo S que especifique, para cada instrucción cuándo se va a ejecutar, en forma relativa al inicio de iteración a la cual pertenece.

Al igual que todos los demás problemas de calendarización, el pipelining por software presenta dos tipos de limitaciones: recursos y dependencias de datos.

6.2.1 Reservación Modular de Recursos

Al existir una cantidad limitada de recursos, una iteración de un ciclo puede requerir n subunidades del mismo, por lo que el calendarizador del pipeline no tendrá conflictos siempre y cuando existan recursos disponibles.

6.2.2 Restricciones de Dependencia de Datos

Las dependencias de datos en el pipelining por software poseen la particularidad de que pueden permitir la formación de ciclos, es decir el resultado de una instrucción puede depender de una instrucción con iteración anterior. Esto impone retrasos en cuanto a la tasa de transferencia de ejecución se refiere.

6.3 Un Algoritmo de Pipeline por Software

El principal objetivo del pipeline por software consiste en encontrar un calendarizador con el mínimo intervalo de iniciación posible. Si se da de esta manera el algoritmo de calendarización puede evitar conflictos de recursos mediante una tabla de reservación de recursos modular en la que se coloca cada instrucción. El problema que se da, es que no se puede conocer el intervalo de iniciación mínimo hasta que no se posea un calendarizador. ¿Cómo resolver esta circularidad?

Se sabe que el intervalo de iniciación debe ser superior al límite calculado a partir del requerimiento de recursos de un ciclo y los ciclos de dependencia. Si se pudiera encontrar un calendarizador que cumple con este límite, se ha hallado el programa óptimo. Si no, es posible intentar hallarlo utilizando intervalos de iniciación más largos, teniendo en cuenta que la heurística en vez de la búsqueda exhaustiva podría no encontrarlo.

La probabilidad de encontrar un calendarizador cerca del límite inferior depende de las propiedades del grafo de dependencia de datos y de la arquitectura de la máquina destino.

Resulta simple encontrar el calendarizador óptimo si el grafo es acíclico y si cada instrucción de sólo necesita una unidad de un recurso. También resulta fácil de encontrar cerca del límite inferior si existen más recursos de hardware disponibles de los que pueden usarse por los grafos con ciclos de dependencia.

7 Conclusión

Como ha podido constatarse el paralelismo a nivel de instrucción emplea una serie de técnicas avanzadas para cumplir con su objetivo de agilizar la ejecución simultánea de instrucciones. Todo esto ha sido posible gracias a las nuevas tecnologías que han surgido en aspectos arquitectónicos y las cuáles han obligado a que sean implementados algoritmos de alto rendimiento que aprovechen al máximo y con eficiencia las mismas.

Existen múltiples problemas por resolver en cuánto a lo que el paralelismo implica, ya que se dan diversos conflictos que no pueden ser resueltos con trivialidad y necesitan de alto análisis realizados por equipos de alto nivel.

El diseño de un pipeline para aprovechar al máximo la ejecución por ciclos de reloj que un procesador puede alcanzar ha sido el responsable de que el incremento en los tiempos de respuesta en la computación de código fuera posible y hoy en día la mayoría de procesadores están diseñados bajo este modelo.

Referencias

[1] Aho, Alfred V; Lamm, Monica S.; Seti, Ravi & Ullman, Jeffrey D: “Compiladores: Principios, técnicas y herramientas”, Segunda Edición, Addison Wesley, México, 2008.

[2] Hennessy, John L; Patterson, David A: “*Computer Architecture: a quantitative approach”, Tercera Edición*, Morgan Kaufman, San Francisco, USA, 2003.

[3] The Art of Assembly Language. Extraído el 18 de junio de 2008 desde *http://webster.cs.ucr.edu/AoA/Windows/HTML/AoATOC.html*

[4] Instruction Level Parallelism and Dynamic Execution. Extraído el 18 de junio de 2008 desde *http://www.cs.ualberta.ca/~amaral/courses/429/webslides/Topic8-ILP-dynamic/sld001.ht*