Análisis, Optimización y Autoparalelismo

Miguel Angel Hernández Orozco

A mis padres.

Tito y Susana, por todo el apoyo y cariño que me dan desde que llegué.

A mis hermanos.

Santi, Efra y Sonia, por el amor y la alegría que compraten conmigo desde que llegaron.

Agradecimientos

A mi familia, en especial a mi tía Alma por sus consejos y su ternura. A mis maestros, en especial a mi directora de tesis Dra Elisa Viso por su esfuerzo y sus enseñanzas. A la gente de Root por su paciencia y apoyo. A mis amigos.

Índice general

Capítulo 1

Introducción

Empezaremos por dar una breve perspectiva de los temas que vamos a tratar en este trabajo y el contexto en el que éstos son importantes.

1.1. Optimización de código

Aunque la optimización no es un paso indispensable en la compilación[1], una compilación ingenua produce código poco eficiente.

Por ejemplo, un compilador podría generar código expresión por expresión. En dicho compilador el código de C mostrado en la figura 1.1(a) se traduciría al código en lenguaje ensamblador de la figura 1.1(b), mientras que el mismo código con algo de optimización se podría también traducir a una versión más eficiente, figura 1.1(c), y así reducir el número de ciclos de reloj necesarios para ejecutar el mismo código.

La idea detrás de la optimización es sencilla: transferir lo más que se pueda del tiempo de ejecución de un programa, al tiempo de compilación; puesto que la compilación de un programa ocurre una sola vez. Esto por lo regular se logra transformando el código original en código más eficiente. Estas transformaciones se deben

[1] En el sentido de que no es necesario para la generación del programa objeto

```
int a, b, c;
c = a + b;
d = c + 1;
```

(a)

```
ldw   a,r1
ldw   b,r2
add   r1,r2,r3
stw   r3,c
ldw   c,r3
add   r3,1,34
stw   r4,d
```

(b)

```
add   r1,r2,r3
add   r3,1,r4
```

(c)

Figura 1.1: Un fragmento de código de C 1.1(a), su compilación inocente 1.1(b) y su compilación optimizada 1.1(c)

de hacer de manera concienzuda pues hay que asegurar que ninguna de éstas afecta la semántica general del programa que se está transformando.

Existen muchas técnicas de optimización y la relevancia de las mismas depende del programa a optimizar. Por ejemplo, un programa con pocos ciclos y bloques básicos grandes, se vería más beneficiado con optimizaciones relacionadas como asignación de registros que con optimizaciones que operan con ciclos.

Otra labor importante de la optimización es brindar una interfaz entre el código y las ventajas propias de la arquitectura objetivo. Es común que el código traducido tal cual de la fuente al lenguaje ensamblador no aproveche correctamente las ventajas que aporte la arquitectura para la que fue compilado. Muchas veces estas ventajas ni siquiera son tomadas en cuenta por el programador, ya que después de todo parte de la labor de los lenguajes de programación es abstraer la arquitectura sobre la que se está trabajando. Es deber del compilador realizar las transfomaciones pertinentes al código objeto para aprovechar al máximo el procesador objetivo. Un ejemplo de esto son las arquitecturas con *entubamiento*.

1.1.1. Entubamiento (*pipelining*)

Este concepto parte de dividir el ciclo de ejecución de cada instrucción en varias etapas; todas las instrucciones se dividen en esas mismas etapas. Cada etapa se ejecutará en serie para cada instrucción, sin embargo distintas etapas de diferen-

tes instrucciones se pueden ejecutar simultáneamente, de tal manera que diferentes instrucciones se puedan ejecutar en "paralelo" en el mismo procesador, y cada instrucción se encuentra en una etapa diferente. El número de etapas puede variar, pero por lo regular la primera y la última etapa son las mismas siempre: `IF` (*instruction fetch*) donde se carga la instrucción a ejecutar y se incrementa el contador de programa; y `WB` (*write back*) donde se guarda en el registro destino el resultado de la operación.

Esta técnica tiene sus bemoles: si cada etapa dura un ciclo de reloj entonces tenemos que, durante cada ciclo, alguna instrucción estará entrando en su etapa de `IF` mientras que otra lo hará en su etapa de `WB`. ¿Qué pasa cuando una instrucción que se está comenzando a ejecutar necesita el resultado de otra instrucción que no ha salido aún de su etapa de `WB`? A este problema se le conoce como riesgo de datos. Existen varias formas de lidiar con este tipo de problemas; algunas consisten en "atorar" la tubería, es decir, no se ejecutan más instrucciones hasta que la instrucción en cuestión salga de su etapa `WB`. Como es de suponer este problema tiene una repercusión en el desempeño.

Otro de los problemas del entubamiento se conoce como *riesgo de control*; éste se presenta cuando existe una bifurcación o salto condicional en el programa. Debemos decidir cuáles son las siguientes instrucciones a ejecutar antes de conocer el resultado del salto. Estos saltos condicionales son muy usuales en ciclos, por dar un ejemplo. Una de las formas de sortear este problema es realizar algo útil mientras se decide qué hacer, para minimizar el impacto que tiene este riesgo en el desempeño. Por ejemplo, si se trata de un ciclo, cada incremento sobre el índice puede realizarse mientras se determina si saltar o no. A esta técnica se le conoce como salto retardado, y consiste en que después de cada instrucción de salto hay una o más "ranuras de retardo", donde se colocan las instrucciones que se tienen que realizar, independientemente de que se efectúe o no el salto; en caso de que este tipo de instrucciones no exista es posible insertar instrucciones vacías[2]. El compilador tiene que tomar en cuenta la existencia de dichas ranuras para aprovecharlas sin afectar la semántica del programa. La figura 1.2 muestra cómo se traduce un ciclo en una máquina con entubamiento y salto retardado con una ranura de retardo. Del lado izquierdo se muestra un ciclo y del lado derecho tenemos la transformación de dicho ciclo a lenguaje ensamblador. La instruccion `blt` salta si el valor del registro `i` es menor que el del registro `valor`. Después de esta instrucción se encuentra la

[2]Las arquitecturas que ofrecen la alternativa de salto retardado cuentan con una instrucción "vacía" que no produce resultado alguno, para ocupar ranuras de retardo que no pueden ser llenadas con otra instrucción.

ranura de retardo, etiquetada con `ds`, donde es posible realizar el incremento del índice de la iteración. El compilador sabe que puede ejecutar el incremento de forma segura, sin alterar la semántica del programa ya sea que se realice el siguiente salto o no.

```
for(int i = 0;i < valor ;i++){        for: ...
    ...                                    blt i, valor, for
}                                     ds:  addi 1, i, i
```

Figura 1.2: Transformación de un ciclo a lenguaje ensamblador de una máquina con entubamiento y salto retardado.

1.2. El papel del análisis

Para que un compilador pueda transformar el código de un programa es indispensable que éste conozca ciertos aspectos de su funcionamiento. De manera intuitiva se puede concluir que mientras más se conozca del programa más profundos pueden ser los cambios realizados al mismo. Aquí es donde interviene el análisis cuyo objetivo es brindar descripciones útiles de la semántica del programa y abstraer su comportamiento. Un programa se puede analizar, dependiendo del objetivo, de manera local, a nivel de expresión o bloques de código; y de manera global, a nivel de métodos o procedimientos.

Agrupamos los diferentes análisis que se realizan al código en tres categorías: análisis de flujo de control, análisis de flujo de datos y análisis de dependencias; siendo el primero, el análisis de flujo de control, una especie de base para los dos posteriores.

El análisis de flujo de control pretende abstraer la estructura del código, diseccionando las construcciones proporcionadas por el lenguaje de programación, representándolas con estructuras más generales, como pueden ser gráficas. En una *gráfica de flujo*, donde las regiones básicas del programa se representan con nodos y el control o la dirección que toma el programa se representa por medio de aristas dirigidas o arcos entre dichos nodos.

El análisis de flujo de datos busca acumular información sobre el paso de los

datos a través del programa. Esta información puede ser muy variada, por ejemplo: si una variable es modificada y en qué puntos ocurren estas modificaciones; o si el valor de alguna variable se usa en algún punto del programa. Debido a esto, este tipo de análisis resulta estar muy enfocado al problema en particular que se quiere resolver, o mejor dicho a la pregunta a la que se quiere responder, y por ende a la optimización en concreto que se quiera realizar.

El análisis de dependencias busca determinar cuáles son las relaciones entre los diferentes elementos del programa. Esto ayuda a realizar transformaciones de manera segura es decir, sin alterar la semántica del programa. Por ejemplo, si determinamos que la expresión `B` depende de que se haya ejecutado anteriormente la expresión `A`, podemos concluir que una transformación que mueve a la expresión `B` antes de `A` es insegura.

En los siguientes capítulos se elaborarán con a detalle los diferentes tipos de análisis y se darán breves ejemplos de las aplicaciones de los mismos.

1.3. Autoparalelismo

En la actualidad existe un amplio número de aplicaciones que requieren gran velocidad de procesamiento. Las simulaciones, operaciones con bases de datos, inteligencia artificial, etc, cada vez son más demandantes en procesamiento y los requerimientos en cuanto al tiempo de respuesta se vuelven más exigentes. Con el tiempo se ha demostrado que una estupenda aproximación para atacar este problema es el *paralelismo*.

El paralelismo consiste en dividir la carga de trabajo en varias unidades de procesamiento, divide y vencerás. Partiendo de la idea que n procesadores realizan el trabajo n veces más rápido, se busca dividir la ejecución de un programa, un algoritmo y hasta una instrucción en n partes, en el mejor de los casos, para que se puedan ejecutar de forma simultánea.

Existen diferentes arquitecturas que proveen varios niveles de paralelismo: desde procesadores con entubamiento para paralelismo a nivel instrucción hasta supercomputadoras con paralelismo masivo (una gran cantidad de procesadores). El diseño de aplicaciones y algoritmos paralelos es un campo de estudio amplio. En la actualidad existen bibliotecas para diferentes lenguajes de programación que

permiten desarollar aplicaciones que aprovechen arquitecturas paralelas.

La mayoría de las aplicaciones están desarrolladas para arquitecturas secuenciales, por lo que es importante el problema de paralelizar un programa automáticamente, en tiempo de compilación. Este concepto se conoce como *autoparalelismo*. El autoparalelismo necesita ciertas técnicas de análisis de flujo y de análisis de dependencia para realizar las transformaciones necesarias al código.

El objetivo de este trabajo es revisar algunas técnicas de autoparalelismo para exponer su importancia y la relación con el análisis de flujo de datos, flujo de control y de dependencias. Con este objetivo en mente se revisarán primero los análisis clásicos realizados por el compilador, para posteriormente mostrar algunas técnicas importantes de paralelización estática y dinámica.

Capítulo 2

Análisis de flujo de control

El propósito principal del análisis de flujo de datos y de control es muy simple: optimización; la optimización en tiempo de compilación de programas escritos en lenguajes de alto nivel. Ésta se hace por medio de transformaciones realizadas al código del programa en cuestión. Existen varias técnicas de optimización que llevan a diferentes transformaciones. El análisis que podamos realizar de un programa es indispensable para la utilización de estas técnicas, como pueden ser: la propagación de constantes, la eliminación de código muerto, la asignación de registros, etc.

Por ejemplo, consideremos la propagación de constantes en donde se busca remplazar el uso de variables, que a partir de cierta asignación no cambian de valor, por la constante que les da valor en dicha asignación. Para esta técnica debemos de garantizar que, en efecto, dicha variable no ha sido instanciada de nuevo y que realizar el remplazo es seguro; esto se hace mediante un análisis del flujo de la variable a través del programa.

El análisis de flujo de control es una base para los demás tipos de análisis pues proporciona una abstracción de la estructura del programa, lo que ayuda a tener una visión general de éste, sus estructuras de control y su comportamiento sin conocer la semántica del mismo.

2.1. Gráficas de Flujo

Así como no se puede hablar de análisis de datos sin hablar de flujo de control, tampoco se puede hablar de flujo de control sin hablar de gráficas dirigidas, pues uno de los resultados del análisis de flujo de control es una gráfica dirigida, conocida como gráfica de flujo. La idea es sencilla: dividir el código del programa en conjuntos de instrucciones, fragmentos atómicos o bloques básicos, representarlos en una gráfica por medio de nodos y representar las transiciones entre los bloques básicos por medio de aristas dirigidas.

```
int fibonacci(int n) {
int f0 = 0,f1 = 1,f2 = 2;
  if(n <= 1) {
    return n;
  }
  else {
    for(int i = 2;i<=n;i++) {
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
    }
    return f2;
  }
}
```

(a) Fibonacci: Recibe un entero positivo n y regresa el n-ésimo elemento de la serie de Fibonacci

```
1     receive n
2     f0 <- 0
3     f1 <- 1
4     if n <= 1 goto L3
5     i  <- 2
6 L1: if i <= n goto L2
7     return f2
8 L2: f2 <- f0 + f1
9     f0 <- f1
10    f1 <- f2
11    i  <- i + 1
12    goto L1
13 L3:return n
```

(b) Representación intermedia del código mostrado en la figura 2.1(a)

Figura 2.1: Transformación de un programa en `C` a su representación `MIR`

Como ejemplo, tomemos el código de Java mostrado en la figura 2.1(a) y construyamos la gráfica de flujo. El primer paso es transformar este código a un lenguaje más simple llamado `MIR` (*Medium Level Intermediate Representation*), usado en teoría de compiladores como una representación intermedia del código; dicho código se muestra en la figura 2.5(b). En la mayoría de los compiladores éste es un paso obligado para el análisis semántico del programa. Si bien es cierto que las estructuras de control son más difíciles de identificar en esta versión del código, también en esta forma es más fácil identificar los bloques básicos y las transiciones entre ellos (éstas corresponden a los desplazamientos).

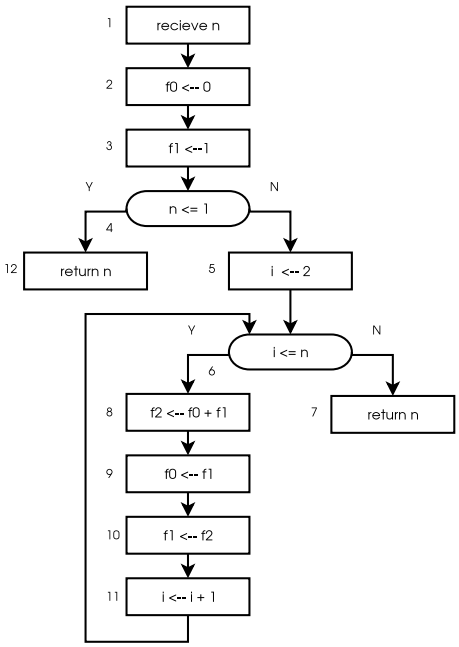


Figura 2.2: Diagrama de flujo correspondiente al código mostrado en la figura 2.1

Para poder ver las cosas claras, y lograr una transición más natural del código a la gráfica, es conveniente representar al código gráficamente por medio de un diagrama de flujo. En dicho diagrama, figura 2.2, se pueden observar las transiciones entre los bloques básicos más fácilmente.

La primer instrucción en un bloque básico puede ser cualquiera de las siguientes:

- El punto de entrada a la rutina.
- El objetivo de un salto o desplazamiento.
- La instrucción inmediata a un desplazamiento o un retorno.

A tales instrucciones se les llama líderes. Para poder identificar los bloques, primero hay que identificar a todos los líderes; y luego para cada líder incluir, en el

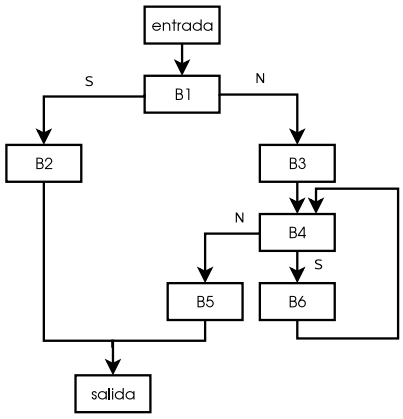


Figura 2.3: Gráfica de flujo correspondiente al código mostrado en la figura 2.2

bloque que éste representa, todas las instrucciones que hay entre éste y el siguiente líder.

En la figura 2.2, claramente los nodos del 1 al 4 forman un bloque, cuyo líder es la instrucción del nodo 1; llamaremos a este bloque B1; los nodos 8 al 11 también forman un bloque, B6. Todos los demás nodos son bloques por sí mismos, de manera que el nodo 12 es B2, el nodo 5 es B3, el nodo 6 es B4 y el nodo 7 es B5.

La aproximación anterior es suficiente para la mayoría de los casos, aunque todavía no se especifica si las llamadas a procedimientos deben ser consideradas líderes. En muchas optimizaciones esto no es necesario pero hay algunas en las que se deben considerar como tales. Una vez identificados los bloques básicos podemos definir al conjunto de nodos de la gráfica de flujo, como $G = \langle N, E \rangle$, donde N se conforma de de un nodo por cada bloque básico, más dos nodos especiales: `entrada` y `salida`. Y el conjunto de aristas E son las aristas $a \rightarrow b$ que corren de un bloque básico a a otro b en el mismo sentido en que las aristas en el diagrama de flujo conectaban la última instrucción de un bloque al líder del siguiente. Finalmente añadimos una arista que va del nodo `entrada` al bloque inicial de la rutina y una arista de cada nodo que salga de la rutina al bloque de `salida`[1].

Para cada nodo podemos definir los conjuntos predecesor y sucesor de manera

[1]Los nodos `entrada` y `salida` son necesarios para definir con mayor claridad los algoritmos de análisis de flujo y son prescindibles en otro contexto.

obvia:

$$pred(n) = \{m \in N \quad \text{tal que} \quad \exists m \rightarrow n \in E\}$$

$$suc(n) = \{m \in N \quad \text{tal que} \quad \exists n \rightarrow m \in E\}$$

Un nodo de unión es aquel que tiene más de un predecesor y un nodo de separación es aquel que tiene más de un sucesor.

2.2. Dominadores

Para poder identificar los ciclos dentro de una gráfica, es conveniente definir primero una relación binaria sobre nodos, llamada *dominio*.

Se dice que un nodo d domina a un nodo i, escrito $d\ dom\ i$, si cualquier camino de ejecución posible de `entrada` a i pasa por d. Claramente dom es reflexiva (todo nodo se domina a sí mismo), transitiva (si $a\ dom\ b$ y $b\ dom\ c$ entonces $a\ dom\ c$), y antisimétrica (si $a\ dom\ b$ y $b\ dom\ a$ entonces $a = b$).

Existen dos subrelaciones del dominio. Decimos que un nodo *domina estrictamente* a otro ($d\ edom\ i$) si y sólo si $d\ dom\ i$ y $d \neq i$. El *dominio inmediato*: para $a\ idom\ b$ sii $a\ edom\ b$ y no existe un nodo c tal que $a\ edom\ c$ y $c\ edom\ b$. Claramente el dominador inmediato de un nodo a es único y lo denotamos $idom(a)$.

La relación dominio inmediato forma un árbol de nodos de la gráfica de flujo cuya raíz es el nodo de entrada, donde para cada nodo x el padre es su dominador inmediato y cuyos caminos muestran todas las relaciones de dominio en la gráfica. A este árbol se le conoce como *árbol de dominio*. El árbol de dominio correspondiente al ejemplo es el que se muestra en la figura 2.4.

Existe una relación conocida como postdominio ($pdom$): $p\ pdom\ i$, si cada camino de ejecución posible desde i a `salida` incluye a p.

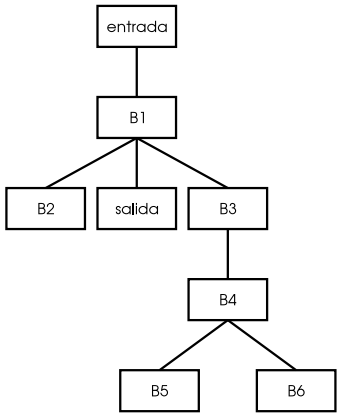


Figura 2.4: Árbol de dominio para la gráfica de flujo de la figura 2.3

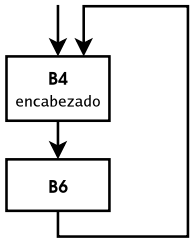
2.3. Ciclos y componentes fuertemente conectados

Para poder identificar los ciclos dada una gráfica de flujo, nos apoyamos en el concepto de dominadores y un concepto adicional llamado *arista hacia atrás* (*back edge*). Una arista hacia atrás, es una arista de la gráfica de flujo cuya cabeza domina a la cola.

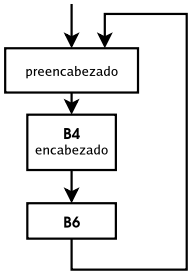
Dada la arista hacia atrás $m \rightarrow n$, un *ciclo natural* de $m \rightarrow n$ es la subgráfica que consiste del conjunto de nodos desde los cuales m puede ser alcanzado por medio de caminos que excluyan a n, más el nodo n; se incluye también el conjunto de aristas que existen entre todos estos nodos. A n se le llama *encabezado* de ciclo.

Muchas optimizaciones requieren mover código que hay dentro del ciclo a un punto antes del encabezado. Para garantizar la existencia de estos lugares se inserta un bloque (inicialmente vacío) antes del encabezado, denominado *preencabezado*, tal que las aristas, que anteriormente apuntaban desde fuera del ciclo a la cabeza, apuntan ahora al preencabezado; se inserta también una arista adicional que va del preencabezado al encabezado.

Se puede intuir que a menos que dos ciclos naturales tengan el mismo encabezado están anidados o son disjuntos. Pero si dos ciclos tienen el mismo encabezado, no es siempre claro si uno está anidado dentro de otro o los dos conforman un mismo ciclo. Dado que sin conocer más del código fuente estas dos situaciones



(a) Un ciclo y su encabezado.



(b) El preencabezado insertado a la gráfica 2.5(a).

son indistinguibles entre sí, muchos análisis tratan a esta estructura como un ciclo simple.

Se puede considerar una estructura más general de ciclo, conocida como *componente fuertemente acoplado*. Este tipo de componente es una subgráfica de G, $G_s = \langle N_s, E_s \rangle$, tal que cada nodo en N_s es alcanzable desde cualquier otro nodo en el mismo conjunto por medio de caminos que contienen únicamente aristas en N_s. Decimos que un compomente fuertemente conectado es *maximal* cuando cualquier componente fuertemente conectado que lo contenga es él mismo.

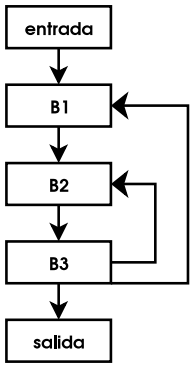


Figura 2.5: Esta gráfica tiene dos componentes fuertemente acoplados, uno de ellos maximal.

2.4. Reducibilidad

La reducibilidad es una propiedad, relativa a ciertos análisis, muy importante de las gráficas de flujo. Otra forma de llamar a una gráfica reducible es gráfica bien formada. Una gráfica es reducible cuando una serie de transformaciones sobre sus subgráficas la reducen a un solo nodo. Esta propiedad siempre se cumple cuando todos los ciclos de la gráfica son naturales. Los ciclos no naturales, los cuales tienen múltiples nodos de entrada, se conocen como *regiones impropias*. La existencia de estas regiones depende de la sintaxis del lenguaje de programación, por ejemplo la instrucción *GOTO* de algunos lenguajes; pero aún en los lenguajes que permiten este tipo de instrucciones el porcentaje de programas que los presentan es reducido (menos del 10 %).

2.5. Análisis de intervalos y árboles de control

En análisis de flujo de control, el análisis de intervalos se refiere a dividir la gráfica de flujo en regiones de varios tipos, consolidando cada región en un nuevo *nodo abstracto* y reemplazando todas las aristas entrantes y salientes a dicha región con nuevas aristas que entran y salen del nodo abstracto. La gráfica resultante de una o más de estas transformaciones se denomina *gráfica abstracta*. Como cada transformación es realizada una a la vez o en paralelo si son disjuntas, podemos anidar las regiones entre sí, en el sentido en el que se va aplicando cada transformación. De esta forma aplicar una de estas secuencias de transformaciones produce un árbol, llamado *árbol de control*, el cual definimos de la siguiente manera:

1. La raíz del árbol de control es una gráfica abstracta representando a la gráfica de control original.

2. Las hojas del árbol de control son bloques básicos de la gráfica original.

3. Los nodos entre la raíz y las hojas son nodos abstractos representando a regiones de la gráfica de control.

4. Las aristas del árbol representan la relación entre cada nodo abstracto y las regiones que de éstas descienden.

Como ejemplo tomamos el análisis de intervalos más simple conocido como el análisis `T1-T2`, el cual se constituye de dos transformaciones: `T1`, que consiste en colapsar un ciclo de un nodo en sí mismo a un nodo abstracto; y `T2`, la cual consiste en colapsar una secuencia de dos nodos, tal que el primero es el único predecesor del segundo, en un solo nodo. La figura 2.6 muestra un ejemplo de este análisis aplicado a una gráfica de flujo sencilla

La figura 2.6 muestra cada paso de la transformación `T1-T2`, realizado a la gráfica 2.3; como podemos ver esta gráfica es reducible, es decir no contiene regiones impropias; la figura 2.7 muestra el árbol de control resultante de esta transformación.

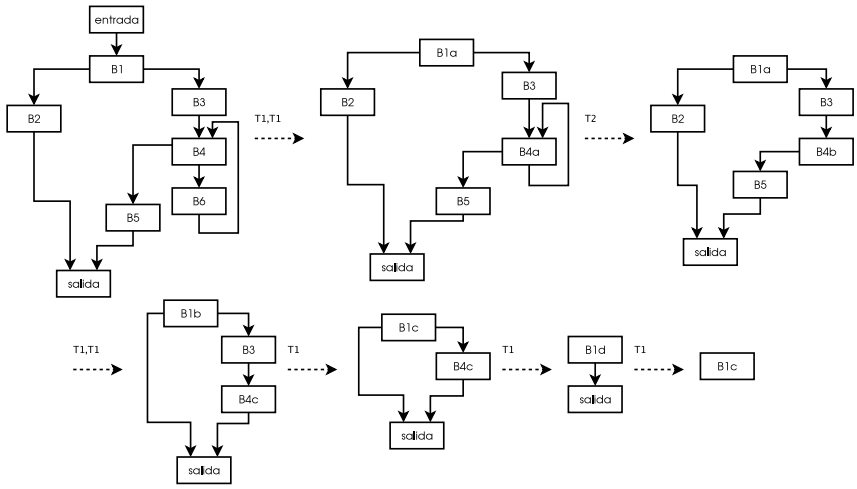


Figura 2.6: El análisis `T1-T2` aplicado a la gráfica 2.3.

2.6. Análisis estructural

El análisis estructural es muy similar al análisis de intervalos y también consiste en reducir regiones de la gráfica de flujo en nodos abstractos. La única diferencia es que este análisis reconoce un mayor número de estructuras. Este análisis se utiliza en particular en el *análisis de flujo de datos de alto nivel*, el cual tiene la ventaja de que para cada construccion de flujo de control del lenguaje de programación, calcula un conjunto de ecuaciones, de una forma muy eficiente. En la gráfica 2.8

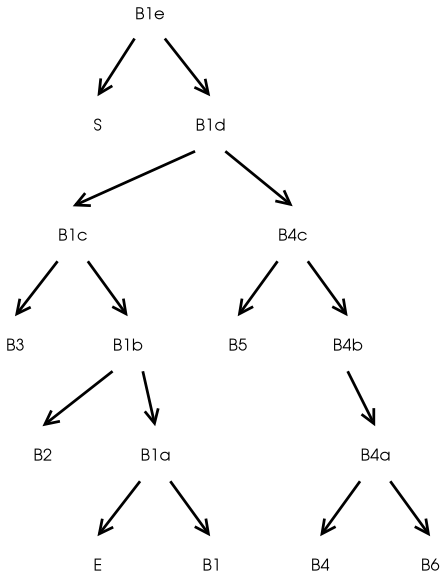


Figura 2.7: El árbol de control que se obtiene del análisis 2.6

se muestran algunas de las construcciones cíclicas y acíclicas que reconoce este análisis.

Es importante enfatizar que el análisis de control por sí solo no aporta suficiente información para realizar optimizaciones al código. Los resultados obtenidos en los diferentes análisis de flujo de control son la base para realizar otros análisis más complejos, como el de flujo de datos o el de dependencias, los cuales sí recopilan la información necesaria para hacer optimización.

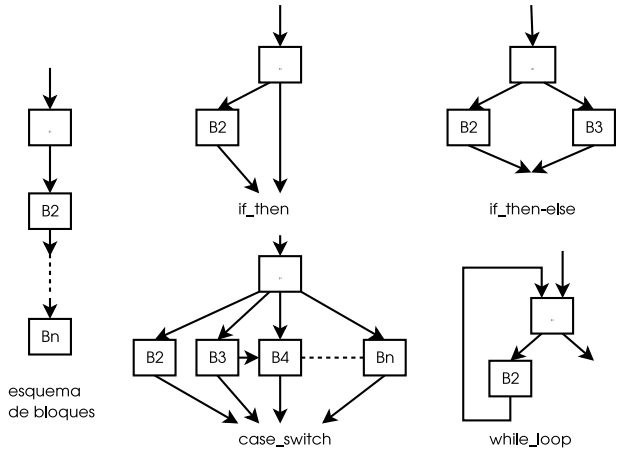


Figura 2.8: Algunas de las construcciones usadas en el análisis estructural

Capítulo 3

Análisis de flujo de datos

En esta capítulo revisaremos el análisis del flujo de los datos de un procedimiento. Éste se encuentra fuertemente relacionado con el análisis de flujo de control que se realice sobre la sección de código correspondiente, ya que depende de ciertas abstracciones logradas por el análisis de control, como la gráfica de flujo o el análisis estructural. Con respecto al análisis de flujo de control, el análisis de flujo de datos tiene metas más concretas con respecto a la optimización. Cada análisis de datos tiene un objetivo, resolver un problema de datos, que por lo regular involucra obtener información sobre el manejo y el flujo de los datos a lo largo del código en cuestión. Esta información es usada posteriormente para realizar transformaciones que puedan resultar en un código más eficiente.

3.1. Conceptos básicos del análisis de flujo de datos

Primero se presentarán algunos conceptos básicos del análisis de flujo de datos, para posteriormente revisar un par de análisis en concreto con sus respectivos ejemplos.

3.1.1. Retículas

Cada análisis de datos trabaja operando elementos de una estrucutra algebráica llamada retícula (en inglés *lattice*). Los elementos de una retícula sirven para representar propiedades abstractas ya sean de variables, expresiones u otras estructuras del lenguaje de programación, independientemente del valor inicial de los datos y en muchos casos también del flujo del programa. Es decir, un análisis de flujo de datos rara vez depende del resultado de una bifurcación `if` o de cuántas veces se ejecuta un ciclo.

Una retícula R consiste de un conjunto de valores y dos operaciones, la intersección $\sqcap$ y la unión $\sqcup$, las cuales satisfacen las siguientes propiedades:

1. Son cerradas: $\forall x, y \in R$, tenemos $x \sqcap y \in R,\ x \sqcup y \in R$.
2. Son conmutativas: $\forall x, y \in R$, tenemos$x \sqcap y = y \sqcap x,\ x \sqcup y = y \sqcup x \in R$.
3. Existe un elemento único llamado *fondo* denotado $\bot$ tal que $\forall x \in R$, tenemos $x \sqcap \bot = \bot$.
4. Existe un elemento único llamado *tope* denotado $\top$ tal que $\forall x \in R$, tenemos $x \sqcup \top = \top$.

La propiedad distributiva no es común a todas las retículas, aunque muchas la presentan.

La mayoría de las retículas utilizan vectores de bits como elementos. El fondo de una retícula de este tipo es el vector que se compone únicamente de ceros y el tope es el que contiene únicamente unos. Es posible mapear el comportamiento del `AND` y el `OR` binario a la intersección y unión respectivamente. Para denotar a la retícula compuesta de vectores de n bits usamos BV^n (de *binary vector*).

Existen varias formas de construir retículas complicadas partiendo de retículas simples. Una de éstas es el producto '$\times$', definido de la siguiente manera

$$R_1 \times R_2 = \{\langle x_1, x_2\rangle | x_1 \in R_1, x_2 \in R_2\}.$$

Y definiendo a la intersección de la siguiente manera:

$$\langle x_1, y_1 \rangle \sqcap \langle x_2, y_2 \rangle = \langle x_1 \sqcap_1 x_2, y_1 \sqcap_2 y_2 \rangle.$$

Donde $\sqcap_1$ y $\sqcap_2$ son las operaciones de intersección para R_1 y R_2, respectivamente. La unión se puede definir de manera análoga. Es fácil demostrar que las cuatros propiedades anteriores se cumplen para $\sqcap$ y $\sqcup$.

Es fácil ver que se puede formar BV^n a partir de n productos de la celda trivial BV^1 sobre sí misma. La figura 3.1 muestra una retícula de vectores de 3 bits.

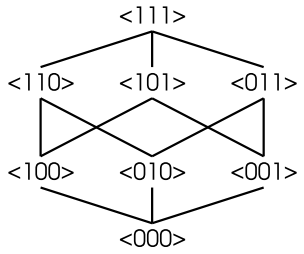


Figura 3.1: Una retícula de vectores de 3 bits

En algunos análisis resulta insuficiente o indeseable utilizar retículas de vectores de bits. Tal es el caso de la propagación de constantes, para la cual se utiliza otro tipo de retícula que tiene como elementos a los enteros, booleanos, el tope y el fondo, como la que se muestra en la figura 3.2. Las reglas para este tipo de retícula no varían radicalmente.

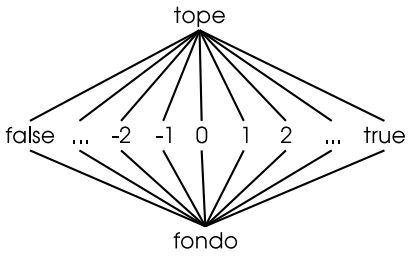


Figura 3.2: Una retícula que tiene como elementos a los enteros, los booleanos, el tope y el fondo

Las operaciones de unión e intersección inducen un orden parcial en los ele-

mentos de la retícula. Podemos usar la intersección para determinar que $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$ o podemos utilizar la unión de manera análoga. Las comparaciones relacionadas con esta noción de orden ($\sqsubset$, $\sqsupset$, $\sqsubseteq$ y $\sqsupseteq$) se definen de forma correspondiente. Usando las definiciones es fácil concluir que:

- $\sqsubseteq$ es transitiva: $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$.
- $\sqsubseteq$ es reflexiva: $x \sqsubseteq x$.
- Es antisimétrica: $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$.

En la figura 3.1 podemos apreciar la relación $\sqsubseteq$ para la retícula BV^3.

3.1.2. Funciones de flujo y puntos fijos

Una función de flujo es un mapeo de la retícula en sí misma. Se dice que una función de flujo es *monótona* si $\forall x, y\, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. Por ejemplo, la función $f : BV^3 \to BV^3$ definida como $f(\langle x_1, x_2, x_3\rangle) = \langle x_1, 1, x_3\rangle$ es monótona.

La altura de una retícula está dada por la longitud de la máxima cadena de elementos estrictamente ascendente.

$$\bot = x_1 \sqsubset x_2 \sqsubset x_3 \sqsubset \cdots \sqsubset x_n = \top$$

La monotonía en las funciones de flujo y la altura finita de la retícula aseguran que un algoritmo de análisis de flujo termina; la monotonía también es útil para poder determinar la complejidad.

El objetivo de una función de flujo es modelar, para un problema dado de análisis de datos, el efecto que alguna o varias construcciones del lenguaje de programación tienen sobre los datos, como transformaciones que mapean elementos de la retícula en sí misma.

Siguiendo la idea de que los elementos de la retícula representan la información obtenida de los datos a lo largo del análisis, es obvio que se requiera que todas las funciones de flujo usadas sean monótonas, ya que no es deseable que alguna de

dichas transformaciones lleve al análisis a la pérdida de información. Dependiendo de los requerimientos del problema, la construcción modelada por una función de flujo puede ir desde una simple expresión hasta el procedimento completo.

Un *punto fijo* de una función $f : R \to R$ es un elemento $z \in R$ tal que $f(z) = z$. Para un conjunto de ecuaciones de flujo de datos un punto fijo, representa una solución a dicho conjunto, pues el aplicar el lado derecho de la ecuación al punto fijo obtenemos el mismo valor.

3.1.3. Soluciones a ecuaciones de flujo de datos

El valor que se desea calcular al resolver ecuaciones de flujo de datos es la *solución que intersecta todos los caminos* (MOP *meet-over-all-paths solution*), para cada nodo de la gráfica. Esta solución surge de tomar la información inicial que entra en el bloque de `entrada` e ir aplicando las funciones correspondientes a los nodos de cada camino de `entrada` al nodo en cuestión, y luego juntar la información de todos los caminos. De manera más formal:

Sea $G = \langle N, F \rangle$ una gráfica de flujo. Denotamos al conjunto de todos los caminos de la `entrada` a `B`$\in N$ como $Path(B)$ y sea p un elemento cualquiera de $Path(B)$, definimos a F_B como la función de flujo que abstrae el paso a través de `B` mientras que F_p representa la composición de todas las funciones correspondientes a los nodos del camino p. Es decir si $B_1 = entrada, \cdots, B_n = B$ son nodos de p entonces $F_p = F_{B_n} \circ \cdots \circ F_{B_1}$.

Si denotamos al elemento de la retícula asociado con el bloque de entrada como *init*, la solución MOP está dada por

$$MOP(B) = \bigsqcap_{p \in Path(B)} F_p(init)$$

Se puede mostrar que para un análisis de flujo arbitrario en el cual sólo se garantiza que las funciones de flujo son monótonas, no es posible calcular una función MOP para cualquier gráfica de flujo. Por ejemplo, los ciclos dentro de la gráfica hacen que el conjunto $Path(B)$ sea infinito para algunos nodos de ésta. Por lo que la mayoría de los algoritmos tratan de calcular una solución maximal dentro de la retícula en cuestión, es decir, el punto fijo maximal (MFP *maximal-fixed-point solution*).

3.2. Problemas de análisis de flujo de datos

Los problemas de análisis de flujo de datos se categorizan en diferentes dimensiones, las cuales incluyen:

- La información que proveen.
- Si son relacionales o involucran atributos independientes.
- Los tipos de retícula que usan y la información representada.
- La dirección del flujo de información, que puede ser: en el sentido de la ejecución del programa, problemas hacia adelante o *forward problems*; en sentido inverso, problemas hacia atrás o *backward problems*; y en ambos sentidos, problemas bidireccionales o *bidirectional problems*.

A continuación se enuncian algunos de los problemas de análisis de flujo más importantes.

Alcance de definiciones

Este problema determina cuáles definiciones de una variable alcanzan qué usos. Se trata de un problema hacia adelante y utiliza vectores de bits, donde cada bit representa una definición.

Expresiones disponibles

Este problema determina cuáles expresiones están disponibles en cada punto del programa. En cada camino de la entrada al punto dado hay una evaluación de la expresión y ninguna de las variables involucradas en la expresión ha cambiado de valor entre tal punto y la evaluación. Ese es un problema hacia adelante que usa una retícula de vectores de bits en la cual cada bit es asociado a una definición de cierta expresión.

Variables vivas

Determina para una variable dada y un punto en el programa si existe un uso de la variable dentro de algún camino entre dicho punto y la salida. Evidentemente éste es un problema hacia atrás. Usa un vector de bits en el cual a cada uso de la variable se le asigna una posición.

Usos expuestos (*Upward Exposed Uses*)

Este problema es el dual de alcance de definiciones. Determina qué usos de variables en puntos particulares son alcanzados por cuáles definiciones. Se trata de un problema hacia atrás que utiliza vectores de bits donde cada posición representa a cada uso de una variable.

Propagación de copias

Este análisis determina que en cada camino, que hay entre una copia, digamos $x \leftarrow y$, y un uso de x, no haya asignaciones de y. Es un problema hacia adelante donde para un vector de bits cada posición representa una copia.

Propagación de constantes

Determina si en cada camino de una asignación de un valor constante a una variable, ($x \leftarrow cons$), a algún uso de x las únicas asignaciones de x son al valor constante $cons$. Es un problema hacia adelante y utiliza vectores con una posición por variable y cuyo dominio de valores es cualquier valor posible de $cons$.

Redundancia parcial

Determina qué cálculos se realizan más de una vez en algún camino de ejecución sin que los operandos hayan sido modificados. Éste es un problema bidireccional.

Existen muchas aproximaciones para resolver problemas de análisis de flujo. En este capítulo se revisan algunas de ellas.

3.3. Análisis iterativo

Para explicar el análisis iterativo utilizaremos un problema hacia adelante, aunque es fácil generalizar esta explicación para cubrir problemas hacia atrás y bidireccionales.

Suponemos que, dada una gráfica de flujo, $G = \langle N, E \rangle$ con bloques de `entrada` y `salida` en N deseamos calcular $in(B)$ y $out(B) \in L$ para cada $B \in N$, donde $in(B)$ representa la información de flujo de datos al entrar en B y $out(B)$ representa la información de flujo de datos a la salida del bloque B. Para dichas funciones damos las siguientes definiciones:

$$in(B) = \begin{cases} init & \text{cuando } B = entrada, \\ \bigsqcap_{p \in pred(B)} out(P) & \text{cuando } B \neq entrada. \end{cases} \qquad out(B) = F_B(in(B)) \tag{3.1}$$

Donde *init* representa el valor inicial apropiado para representar la información a la entrada del procedimiento; F_B representa la transformación de la información correspondiente al ejecutar el código del bloque B; y $\bigsqcap$ modela el efecto de combinar la información obtenida de los predecesores del bloque.

Desde luego estas definiciones también se pueden expresar únicamente utilizando la función *in*.

$$in(B) = \begin{cases} init & \text{cuando } B = entrada \\ \bigsqcap_{p \in pred(B)} F_p(in(B)) & \text{cuando } B \neq entrada \end{cases} \tag{3.2}$$

3.3.1. Alcance de definiciones

Para ejemplificar el análisis iterativo nos valemos de un problema común, el *alcance de definiciones* (*reaching definitions*), donde por *definición* se entiende como la asignación de un valor a una variable. Se dice que una definición alcanza a un punto de un procedimiento (o está viva en un punto dado) si existe un camino de ejecución desde la definición de la variable hasta dicho punto, tal que en este punto la variable definida pueda tener el valor asignado en la definición. El problema consiste en poder determinar qué definiciones alcanzan, mediante un *camino de flujo de control*, cuáles puntos, esto para cada punto del procedimiento, donde un camino de flujo de control es cualquier camino en la gráfica de flujo, sin tomar en cuenta si este camino depende de saltos condicionales imposibles de determinar de manera estática. Esto implica que la solución al problema es conservadora, es decir suponemos verdadero lo que está indeterminado.

Como ejemplo utilizaremos la gráfica de flujo, figura 2.2, del capítulo anterior. Abordaremos este problema de manera local a los bloques, es decir, en vez de determinar los alcances de las definiciones con respecto a los puntos del procedimiento, lo haremos con respecto a sus bloques; de esta forma el análisis es más eficiente y nos provee una perspectiva global.

En este análisis utilizamos una retícula de vectores de 8 bits BV^8. Cada posición del vector se asocia a una definición como se aprecia en el cuadro 3.1. El n-ésimo bit es 1 cuando la definición existe en el bloque y es 0 cuando ésta es inexistente.

índice	Definición	Bloque
1	`m`	`B1`
2	`f0`	
3	`f1`	
4	`i`	`B3`
5	`f2`	`B6`
6	`f0`	
7	`f1`	
8	`i`	

Cuadro 3.1: Los índices de las definiciones; éstas ocurren en tres de los 6 bloques de la gráfica de flujo.

La información que se requiere obtener por cada bloque B es: ¿cuáles definiciones alcanzan la entrada de B (denotada por $RCHin(B)$)? y ¿cuáles alcanzan la salida ($RCHout(B)$)?. Empezaremos por caracterizar dos funciones cuya composición determinará $RCHin$ y $RCHout$

PRSV Determina las definiciones que son preservadas en `B`, es decir las asignaciones cuyas variables no se redefinen a través del bloque. Es fácil encontrar los valores de esta función para cada bloque; primero damos el conjunto de índices y luego los vectores.

$$PRSV(B1) = \{4,5,8\} = \langle 00011001\rangle$$
$$PRSV(B3) = \{1,2,3,5,6,7\} = \langle 1101110\rangle$$
$$PRSV(B6) = \{1\} = \langle 10000000\rangle$$
$$PRSV(i) = \{1,2,3,4,5,6,7,8\} = \langle 11111111\rangle \text{ para } i \neq B1, B3, B6$$

GEN Complementario a `PRSV` definimos `GEN` para denotar las definiciones generadas en cada bloque, es decir que son asignadas y no reasignadas a lo largo de éste. Los valores de `GEN` son:

$$GEN(B1) = \{1,2,3\} = \langle 11100000\rangle$$
$$GEN(B3) = \{4\} = \langle 00010000\rangle$$
$$GEN(B6) = \{5,6,7,8\} = \langle 00001111\rangle$$
$$GEN(i) = \phi = \langle 00000000\rangle \text{ para } i \neq B1, B3, B6$$

Ahora podemos usar GEN y $PRSV$ para definir $RCHout$. Las definiciones que alcanzan la salida del bloque, son las generadas en el bloque y las que entran y son preservadas en el mismo.

$$RCHout(B) = GEN(B) \cup (RCHin(B) \cap PRSV(B)) \text{ para todo bloque } B \qquad (3.3)$$

Luego definimos a $RCHin$ como la unión de todas las definiciones que salen de los predecesores de B:

$$RCHin(B) = \bigcup_{P \in Pred(B)} RCHout(P) \text{ para todo bloque } B. \qquad (3.4)$$

Pero $RCHout(B)$ puede ser definida como una función de flujo aplicada a $RCHin(B)$ como en la ecuación 3.1. Utilizamos la definición de $RCHout$-ecuación 3.3- y los valores de GEN y $PRSV$ para deducir las funciones de flujo F_B para cada bloque. Estos valores están representados en el cuadro 3.2. Como podemos observar

$$
\begin{aligned}
&F_{entry} = id \\
&F_{B1}(\langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 \rangle) = \langle 111 x_4 x_5 00 x_8 \rangle \\
&F_{B_2} = id \\
&F_{B3}(\langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 \rangle) = \langle x_1 x_2 x_3 1 x_5 x_6 x_7 0 \rangle \\
&F_{B4} = id \\
&F_{B5} = id \\
&F_{B6}(\langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 \rangle) = \langle x_1 0001111 \rangle
\end{aligned}
$$

Cuadro 3.2: Funciones de flujo

estas funciones son no decrecientes, por lo que si las aplicamos de forma iterativa eventualmente van a alcanzar un punto fijo. Esta solución es de tipo MFP.

Para encontrar los puntos fijos tenemos que iniciar $RCHin(entrada)$ con un valor adecuado, en este caso $\langle 00000000 \rangle$ y aplicar la ecuación 3.2 hasta obtener los valores de $RCHin$ y $RCHout$ para cada bloque. Luego aplicamos iterativamente las funciones sobre los valores obtenidos hasta encontrar los puntos fijos, es decir que los valores no cambien. En el ejemplo, al iterar dos veces alcanzamos los valores a todas las funciones; el cuadro 3.3 muestra dichos valores para ambas iteraciones. El iterar la ejecución de las ecuaciones emula las construcciones cíclicas del método; al llegar a un punto fijo podemos garantizar que no importa cuántas veces se ejecute un ciclo la información obtenida ya no va a cambiar.

Estas soluciones nos muestran un panorama global acerca de cuáles definiciones alcanzan qué usos. Por ejemplo, se puede notar que la definicion de `f0` en `B1` puede alcanzar a su primer uso en `B6`; o también que a través del camino de ejecución que pasa por `B2` no se definen las variables `i` y `f2`. Una forma de utilizar esta información para optimizar el código es no reservar espacio (o registros) para `i` y `f2` a través de `B2`.

3.4. Análisis basados en árboles de control

Los algoritmos usados en este tipo de análisis, en concreto análisis estructural y análisis de intervalos, son muy similares entre sí pues ambos están basados en los árboles de control vistos en la sección 2.5.

En la primera iteración tenemos

$RCHout(entrada)$	=	$\langle 00000000 \rangle$	$RCHin(entrada)$	=	$\langle 00000000 \rangle$
$RCHout(B1)$	=	$\langle 11100000 \rangle$	$RCHin(B1)$	=	$\langle 00000000 \rangle$
$RCHout(B2)$	=	$\langle 11100000 \rangle$	$RCHin(B2)$	=	$\langle 11100000 \rangle$
$RCHout(B3)$	=	$\langle 11110000 \rangle$	$RCHin(B3)$	=	$\langle 11100000 \rangle$
$RCHout(B4)$	=	$\langle 11110000 \rangle$	$RCHin(B4)$	=	$\langle 11110000 \rangle$
$RCHout(B5)$	=	$\langle 11110000 \rangle$	$RCHin(B5)$	=	$\langle 11110000 \rangle$
$RCHout(B6)$	=	$\langle 10001111 \rangle$	$RCHin(B6)$	=	$\langle 11110000 \rangle$
$RCHout(exit)$	=	$\langle 11110000 \rangle$	$RCHin(exit)$	=	$\langle 11110000 \rangle$

Y en la segunda

$RCHout(entrada)$	=	$\langle 00000000 \rangle$	$RCHin(entrada)$	=	$\langle 00000000 \rangle$
$RCHout(B1)$	=	$\langle 11100000 \rangle$	$RCHin(B1)$	=	$\langle 00000000 \rangle$
$RCHout(B2)$	=	$\langle 11100000 \rangle$	$RCHin(B2)$	=	$\langle 11100000 \rangle$
$RCHout(B3)$	=	$\langle 11110000 \rangle$	$RCHin(B3)$	=	$\langle 11100000 \rangle$
$RCHout(B4)$	=	$\langle 11111111 \rangle$	$RCHin(B4)$	=	$\langle 11111111 \rangle$
$RCHout(B5)$	=	$\langle 11111111 \rangle$	$RCHin(B5)$	=	$\langle 11111111 \rangle$
$RCHout(B6)$	=	$\langle 10001111 \rangle$	$RCHin(B6)$	=	$\langle 11111111 \rangle$
$RCHout(exit)$	=	$\langle 11111111 \rangle$	$RCHin(exit)$	=	$\langle 11111111 \rangle$

Cuadro 3.3: Los resultados después de iterar 1 y 2 veces las ecuaciones.

Estos análisis consisten de dos pasadas sobre el árbol de control. La primer pasada se realiza de abajo hacia arriba; comenzando en los bloques básicos se construye la función de flujo correspondiente al efecto de ejecutar esa porción del árbol de control. La segunda pasada, de arriba hacia abajo, empieza con el nodo abstracto que representa todo el procedimiento, la raíz del árbol, y con la información inicial correspondiente al nodo `entrada` o `salida`. Dependiendo de la dirección del problema, se construyen y evalúan las ecuaciones que propagan la información a través de la región representada por cada nodo del árbol de control utilizando las funciones de flujo construidas en la primer pasada.

3.4.1. Retículas de funciones de flujo

Al igual que los elementos sobre los que realizamos los análisis de flujo, las funciones que modelan las transformaciones también forman una retícula. En dicha

retícula las operaciones de intersección y unión son inducidas por la retícula de elementos. Esta nueva retícula es importante en el análisis estructural de flujo de datos.

En particular dada una retícula R, definimos a R^F como el conjunto de funciones monótonas de R en R es decir:

$$f \in R^F \Leftrightarrow \forall x, y \in R, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y).$$

Las operaciones de unión e intersección son inducidas punto a punto. En el caso de la intersección tenemos:

$$\forall f, g \in R^F, \forall x \in R\, (f \sqcap g)(x) = f(x) \sqcap g(x).$$

El orden se induce de forma similar, de donde concluimos que R^F es, en efecto, una retícula. Finalmente, definimos al tope y al fondo de R^F como:

$$\forall x \in L,\ \bot^F(x) = \bot \text{ y } \top^F(x) = \top.$$

Adicionalmente definimos a la funcion identidad, denotada id, como $id(x) = x\ \forall x \in R$. Y dos operaciones adicionales. La primera de ellas es la composición, donde dadas $f, g \in R^F$ definimos a $f \circ g$ como:

$$(f \circ g) = f(g(x))$$

Es fácil ver que R^F es cerrada bajo composición. Definimos la composición de una función sobre sí misma, dada $f \in R^F$:

$$f^0 = id \text{ y } f^n = f \circ f^{n-1} \text{ para } n \geq 1.$$

Por último, sea $f \in R^F$; la *cerradura de Kleene*, denotada f^* se define como:

$$\forall x \in R f^*(x) = \lim_{n \to \infty} (id \sqcap f)^n(x).$$

Para mostrar que f^* es cerrada sólo hay que utilizar el hecho de que R es de altura finita. Es fácil corroborar que dada una retícula de vectores de bits BV^n, la retícula de funciones correspondiente es distributiva bajo unión e intersección. Además, mientras los valores de las posiciones de los bits cambien de manera independiente, lo cual ocurre en la mayoría de los análisis, se cumple que $f \circ f = f$ y $f^* = id \sqcap f$.

3.4.2. Análisis estructural

Este análisis utiliza la información obtenida en el análisis estructural de flujo de control para producir las ecuaciones de flujo de datos que representan los efectos de las estructuras de control correspondientes.

La diferencia básica entre resolver un problema hacia adelante y un problema hacia atrás es que cada estructura de control tiene una única entrada y varias salidas, lo cual complica la solución de problemas hacia atrás, no serán revisadas en este trabajo ya que se salen de nuestro ámbito de estudio.

Al realizar análisis de datos estructural, la mayoría de las gráficas abstractas que se encuentran en la primer pasada son regiones simples. Para un bloque básico utilizamos la misma funcion de flujo F_B que en el análisis iterativo, sección 3.3, la cual depende únicamente del tipo de problema y el contenido del bloque.

Las ecuaciones para estructuras de control

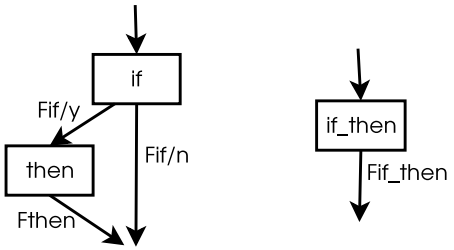


Figura 3.3: Reducción de la construcción `if_then`

Primero tomamos la construcción `if then`, figura 3.3. La función de flujo $F_{\texttt{if then}}$ obtenida está relacionada a los componentes del `if then` de la siguiente forma:

$$F_{\texttt{if then}} = (F_{\texttt{then}} \circ F_{\texttt{if/y}}) \sqcap F_{\texttt{if/n}}.$$

Esto quiere decir: el efecto del bloque `if` tomando la rama `y`, función $F_{\texttt{if}}$, compuesto con el efecto del bloque `then`; esto hay que intersectarlo con el efecto del bloque `if` tomando la otra rama `n`.

Es importante notar que en este caso se decidió distinguir ambas ramas del *if*, esto es necesario únicamente si existe cierta discriminación entre los datos al tomar alguna de las ramas. En caso de no tener tal distinción tendríamos una sola función representando el paso por el bloque `if`, $F_{\texttt{if}}$. La función de flujo asociada al `if_then` quedaría de la siguiente forma:

$$\begin{aligned} F_{\texttt{if\_then}} &= (F_{\texttt{then}} \circ F_{\texttt{if}}) \sqcap F_{\texttt{if}} \\ &= (F_{\texttt{then}} \sqcap id) \circ F_{\texttt{if}}. \end{aligned}$$

Es fácil pasar de un caso al otro. Debido a que el último caso es más frecuente, en adelante nos referiremos a éste de forma general.

La segunda pasada nos indica cómo, dada la información que entra en la construcción `if_then`, propagar la entrada de cada subestructura. Esto es relativamente transparente:

$$\begin{aligned} in(\texttt{if}) &= in(\texttt{if\_then}) \\ in(\texttt{then}) &= F_{\texttt{if}}(in(\texttt{if})) \end{aligned}$$

En la primer ecuación se representa cómo la información que se tiene en la entrada del bloque `if` es la misma que se tiene a la entrada de la gráfica abstracta `if_then`. La segunda nos dice que en la entrada del bloque `then` la información a la entrada del `if` es transformada por el efecto del paso por el mismo bloque.

A continuación extendemos este análisis a la construcción `if_then_else`, figura 3.4. Esto es una generalización sencilla del caso anterior. Para la primera pasada tenemos:

$$F_{\texttt{if\_then\_else}} = (F_{\texttt{then}} \circ F_{\texttt{if}}) \sqcap (F_{\texttt{else}} \circ F_{\texttt{if}})$$

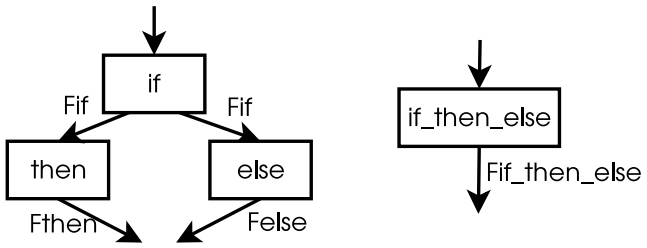


Figura 3.4: Reducción de la construcción `if_then_else`

Y las ecuaciones construidas en la segunda pasada son:

$$in(\texttt{if}) = in(\texttt{if\_then\_else})$$
$$in(\texttt{then}) = F_{\texttt{if}}(in(\texttt{if}))$$
$$in(\texttt{else}) = F_{\texttt{if}}(in(\texttt{if}))$$

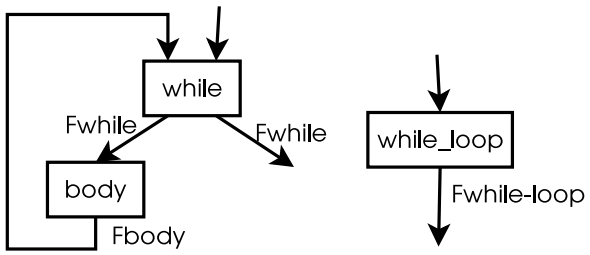


Figura 3.5: Reducción del ciclo `while`

Por último revisamos la construcción `while`, figura 3.5. En la primer pasada la función de flujo que expresa el resultado de iterar en el ciclo una vez y luego regresar a la entrada es: $F_{body} \circ F_{while}$. Para expresar la ejecución del ciclo un número arbitrario de veces, utilizamos la cerradura de Kleene, es decir:

$$F_{\texttt{loop}} = (F_{\texttt{body}} \circ F_{\texttt{while}})^*.$$

En la segunda pasada tenemos para el while

$$in(\texttt{while}) = F_{\texttt{loop}}(in(\texttt{while\_loop}))$$
$$in(\texttt{body}) = F_{\texttt{while}}(in(\texttt{while}))$$

Ya que sólo podemos entrar a `body` a través de `while` y a éste podemos entrar desde afuera o desde una iteración anterior.

Generalización

Es posible dar una generalización para determinar las funciones y ecuaciones de las dos pasadas, tanto para regiones acíclicas como para cíclicas. Primero revisaremos las acíclicas.

Tomamos una región cualquiera A; sean $B_0, B_1, \ldots, B_n$ nodos de A y B_0 la entrada de A. Para cada nodo B_i existe una serie de salidas $B_{i/1}, \ldots, B_{i/e_i}$. Asignamos a cada $B_{i/e}$ una función de flujo $f_{B_{i/e}}$. Sea $P(A, B_{i_k/e_k})$ el conjunto de todos los caminos posibles de la entrada A a la salida del nodo abstracto B_{i_k/e_k} en A.

En la primer pasada, dado el camino

$$p = B_{i_0/e_0}, B_{i_1/e_1}, \ldots, B_{i_k/e_k} \in P(A, B_{i_k/e_k}),$$

la función compuesta para el camino es:

$$F_p = F_{B_{i_k/e_k}} \circ \cdots \circ F_{B_{i_1/e_1}} \circ F_{B_{i_0/e_0}};$$

y la función de flujo correspondiente a todos los caminos posibles del punto de entrada A al nodo B_{i_k/e_k} es:

$$F_{(A,B_{i_k/e_k})} = \bigsqcap_{p \in P(A,B_{i_k/e_k})} F_p. \tag{3.5}$$

En la segunda pasada, para la entrada a B_i, sea $P_p(A, B_i)$ el conjunto de todos los caminos $P(A, B_{j/e})$ tales que $B_j \in Pred(B_i)$ y la salida $B_{j/e}$ lleva a B_i; entonces, la ecuación para la entrada de B_i es:

$$in(B_i) = \begin{cases} in(A) & \text{para } i = 0 \\ \bigsqcap_{P(A,B_{j/e}) \in P_p(A,B_i)} F_{(A,B_{j/e})}(in(A)) & \text{para } i \neq 0 \end{cases} \tag{3.6}$$

Supongamos que tenemos una región cíclica propia C, definida igual que A, con la diferencia de que existe una única arista hacia atrás de algún bloque B_c a B_0. Si

eliminamos esta arista obtenemos una región acíclica, en la cual podemos describir las funciones de flujo correspondientes a todos los caminos de la entrada de C a algún nodo B_{i_k/e_k}, de la misma forma en la que lo hicimos para A. Así obtenemos la colección de funciones $F_{(C,B_{i_k/e_k})}$ y en particular, si $B_{c/e}$ es la cola de la arista hacia atrás, la función

$$F_{iter} = F_{(C,B_{c/e})}$$

representa el resultado de realizar una iteración del cuerpo de la región. Entonces $F_c = F^*_{iter}$ representa el efecto de ejecutar la región y regresar a la entrada un número arbitrario de veces. Para representar la ejecución completa de la región con todo y salida en Bi_k/e_k formamos la función F'

$$F'_{(C,B_{i_k/e_k})} = F_{(C,B_{i_k/e_k})} \circ F_c.$$

De manera correspondiente en la segunda pasada, formamos la ecuación para determinar la entrada a un bloque cualquiera de la región.

$$in(B_i) = \begin{cases} in(C) & \text{para } i = 0 \\ \bigsqcap\limits_{P(C,B_{j/e}) \in P_p(C,B_i)} F'_{(C,B_{j/e})}(in(C)) & \text{para } i \neq 0 \end{cases} \tag{3.7}$$

Ejemplo

Como ejemplo retomaremos el problema de alcance de definiciones. Primero realizamos el análisis estructural de flujo de control sobre la gráfica, figura 2.2, como se muestra en la figura 3.6. En la primer pasada construimos primero la ecuación para la región `while_loop` reducido a `B4a`:

$$F_{\texttt{B4a}} = F_{\texttt{B4}} \circ (F_{\texttt{B6}} \circ F_{\texttt{B4}})^* = F_{B4} \circ (id \sqcap (F_{\texttt{B6}} \circ F_{\texttt{B4}}));$$

luego la ecuación del bloque `B3a`:

$$F_{\texttt{B3a}} = F_{\texttt{B5}} \circ F_{\texttt{B4a}} \circ F_{\texttt{B3}};$$

para el bloque `if_then_else` reducido a `B1a`

$$F_{\texttt{B1a}} = (F_{\texttt{B2}} \circ F_{\texttt{B1}}) \sqcap (F_{\texttt{B3a}} \circ F_{\texttt{B1}});$$

finalmente `entrada_a`:

$$F_{\texttt{entrada\_a}} = F_{\texttt{salida}} \circ F_{\texttt{B1a}} \circ F_{\texttt{entrada}}.$$

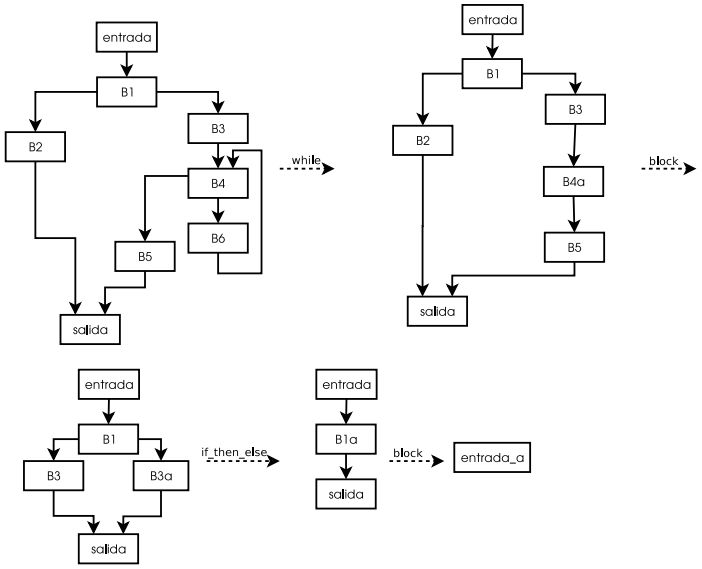


Figura 3.6: Análisis estructural de la gráfica de la figura 2.2

En la segunda pasada construimos las ecuaciones para los componentes del bloque `entrada_a`:

$$\begin{aligned} &in(\texttt{entrada}) = Init,\\ &in(\texttt{B1a}) = F_{\texttt{entrada}}(in(\texttt{entrada})),\\ &in(\texttt{salida}) = F_{\texttt{B1a}}(in(\texttt{B1a})); \end{aligned}$$

para el `if_then_else`:

$$\begin{aligned} &in(\texttt{B1}) = in(B1a),\\ &in(\texttt{B2}) = in(B3) = F_{\texttt{B1}}(in(\texttt{B1a})); \end{aligned}$$

para el bloque reducido a `B3a`:

$$\begin{aligned} &in(\texttt{B3}) = in(\texttt{B3a}),\\ &in(\texttt{B4a}) = F_{\texttt{B3}}(in(\texttt{B4a})),\\ &in(\texttt{B5}) = F_{\texttt{B4a}}(in(\texttt{B4a})); \end{aligned}$$

y finalmente para el `while` reducido a `B4a`

$$\begin{aligned} &in(\texttt{B4}) = (F_{\texttt{B6}} \circ F_{\texttt{B4}})^*(in(\texttt{B4a})),\\ &in(\texttt{B6}) = F_{\texttt{B4}}(in(\texttt{B4})). \end{aligned}$$

El valor inicial de *in*(`entrada`) y las funciones de flujo para los bloques básicos son idénticos a los usados en el ejemplo iterativo. Ahora sí procedemos a calcular los valores de las ecuaciones *in*(*B*) para cada bloque, cuadro 3.4.

$$
\begin{array}{lcl}
in(entrada) & = & \langle 00000000 \rangle \\
in(B1) & = & \langle 00000000 \rangle \\
in(B2) & = & \langle 11100000 \rangle \\
in(B3) & = & \langle 11100000 \rangle \\
in(B4) & = & \langle 11111111 \rangle \\
in(B5) & = & \langle 11111111 \rangle \\
in(B6) & = & \langle 11111111 \rangle \\
in(exit) & = & \langle 11111111 \rangle
\end{array}
$$

Cuadro 3.4: Soluciones a las ecuaciones de flujo

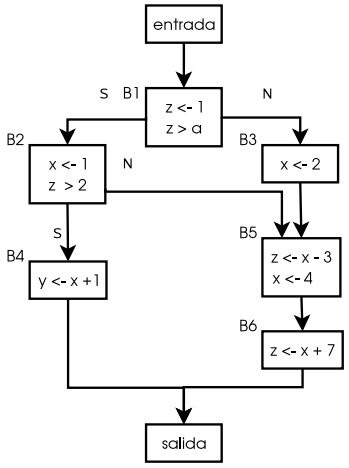
3.5. Asignación estática única

La forma asignación estática única (*Static Single-Assignment* SSA) es una representación intermedia cuyo objetivo es separar los valores que se operan en un programa de las localidades en las que están almacenadas. Esta representación hace posible versiones más efectivas de muchas optimizaciones.

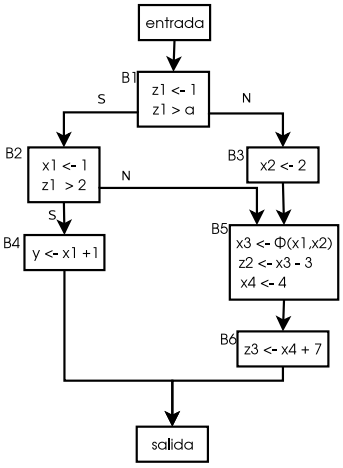
Un método o procedimiento está en forma `SSA` si cada variable con un valor asignado aparece en una sola asignación. En esta representación la relación entre una definición y su uso se vuelve explícita, ya que en un programa en forma `SSA` existe una única asignación por variable y todos los usos de una variable están ligados a su única asignación. Esto simplifica y hace más efectivas muchas transformaciones de optimización: propagación de constantes, numeración de valores, movimiento y eliminación de código invariante, eliminación parcial de redundancia, etc; y también es elemento básico de al menos una técnica de autoparalelización.

Para realizar la traducción de un procedimiento a su forma `SSA` hay que hacer dos cosas:

- Asignar indices las variables: cada nueva asignación introduce un nuevo índice y todos los usos que estén en un bloque cuyo predecesor sea el de



(a) La gráfica de flujo de un programa simple



(b) Representación en forma SSA del programa de la figura 3.7(a)

Figura 3.7: La transformación SSA

la asignación se renombran con dicho índice.

- Usar una función de unión, que denotamos con ϕ, en los puntos de unión (como en la figura 3.7(b)), para ordenar múltipes asignaciones a una variable.

Cada función ϕ tiene tantos argumentos como versiones de la variable se juntan en el punto de unión. La representación estándar SSA de la figura 3.7(a) es la mostrada en la figura 3.7(b). La variable x se separa en 4 (x_1,x_2,x_3 y x_4) y la variable y se separa en 3.

El proceso es: primero, para cada variable, localizar las posiciones donde se ubicarán las funciones ϕ e insertar en esas posiciones funciones ϕ *triviales* ($\phi(x, x, \ldots, x)$) con un número de argumentos igual al número de predecesores (en la gráfica de flujo), con una definición de la variable en cuestión. Luego se renombran las definiciones y usos de la variable para establecer la propiedad SSA, cambiando los argumentos correspondientes de la función ϕ.

Mediante el análisis de flujo de datos es posible obtener información de cómo una región de código manipula sus datos. Utilizando esta información el compi-

lador puede realizar transformaciones al código que involucran desde propagar el valor de una constante a lo largo de sus usos, hasta eliminar secciones de código que nunca se ejecutarán o cuya ejecución es redundante.

En el siguiente capítulo se presenta el análisis de dependencias, el cual es indispensable en transformaciones que involucran el movimiento o reordenamiento de código.

Capítulo 4

Análisis de dependencias

El análisis de dependencias es fundamental cuando realizamos transformaciones al código, ya sea fuente, intermedio u objeto, que requieran mover instrucciones. Por ejemplo, se puede determinar que existe una relación de estricto orden entre un conjunto de instrucciones dentro de un bloque del programa, y también podemos determinar si alterar este orden puede afectar los resultados de la ejecución del programa.

4.1. Dependencia de datos

4.1.1. Relaciones de dependencia

Para denotar que una instrucción $S1$ precede a otra instruccion $S2$ en su orden de ejecución, escribimos $S1 \lhd S2$. Una *dependencia* entre dos instrucciones es una relación que constriñe el orden de ejecución de las mismas. Existen varios tipos de dependencias; para explicarlas nos auxiliaremos con el código de la figura 4.1.

Una "dependencia de control" es una dependencia que surge del flujo del control del programa, como la que se puede apreciar entre $S2$ y $S3$ o también entre $S2$ y $S4$ en la figura 4.1; en este caso $S3$ y $S4$ son ejecutados únicamente si la condición de $S2$ no se satisface. Una dependencia de control entre dos instrucciones $S1$

```
S1      a := b + c
S2      if a > 10 goto L1
S3         d := b * e
S4         e := d + 1
S5 L1: d := e / 2
```

Figura 4.1: Ejemplo de dependencias de datos y de control en un código `MIR`.

y $S2$ se escribe $S1\delta^C S2$, por lo que en el ejemplo tenemos que $S2\delta^C S3$ y $S2\delta^C S4$.

Una dependencia de datos es una constricción que surge a partir del flujo de datos entre dos operaciones, como la que se observa entre $S3$ y $S4$ en la figura 4.1. En este caso $S3$ define el valor de `d` y $S4$ usa dicho valor; también, en $S4$ se define `a` y $S3$ la utiliza. Aquí podemos identificar dos tipos diferentes de dependencia y en ambos casos reordenar el código puede resultar catastrófico. En total existen cuatro tipos de dependencias:

1. Si $S1 \lhd S2$ y la primera define una variable usada por la segunda, entonces existe una dependencia de flujo o una "dependencia verdadera" que denotamos con δ^f; en el código de la figura 4.1 existe una dependencia de flujo entre $S3$ y $S4$: $S3\delta^f S4$.

2. Si $S1 \lhd S2$ y $S1$ usa alguna variable definida en $S2$, entonces existe una "antidependencia" entre los dos, $S1\delta^a S2$. Las instrucciones $S3$ y $S4$ de la figura 4.1 forman también una antidependencia: $S3\delta^a S4$.

3. Si $S1 \lhd S2$ y ambas instrucciones definen la misma variable, decimos que existe una "dependencia de salida", denotada $S1\delta^o S2$. En la figura 4.1 tenemos que: $S3\delta^o S5$.

4. Por último, si $S1 \lhd S2$ y ambas instrucciones leen el valor de la misma variable, decimos que hay una "dependencia de entrada", denotada $S1\delta^i S2$. En la figura 4.1 tenemos que $S3\delta^i S5$. Una dependencia de este tipo no constriñe el orden de ejecución de dos instrucciones, pero es útil para completar las definiciones.

Para representar un conjunto de dependencias podemos usar una "gráfica de dependencia", en la cual los nodos representan las instrucciones y las aristas las dependencias. Para identificar el tipo de dependencia que representa cada arista, éstas

se etiquetan con el superíndice correspondiente al tipo de dependencia, salvo en la dependencia de flujo que tradicionalmente no se etiqueta. La figura 4.2 muestra la gráfica de dependencia correspondiente al código de la figura 4.1. Generalmente las dependencias de control se omiten en la gráfica, a menos que no exista algún otro tipo de dependencia entre los nodos.

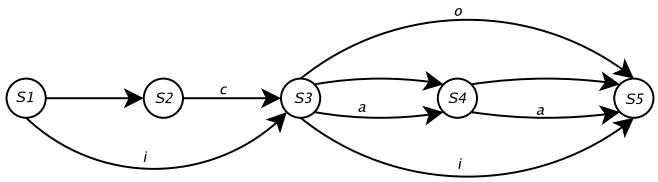


Figura 4.2: La gráfica de dependencia correspondiende al programa de la figura 4.1.

4.1.2. Dependencias en ciclos

Un problema importante es el análisis de dependencias de instrucciones o predicados que involucran variables con subíndices, *arreglos*. En concreto consideramos los usos de arreglos dentro de uno o más ciclos anidados. Para el análisis de dependencias dentro de ciclos, podemos suponer que trabajamos con ciclos en forma canónica, es decir, cada ciclo corre desde 0 (o desde 1) hasta un valor n, incrementando su índice de uno en uno y el ciclo interior es el único que tiene predicados diferentes de declaraciones de ciclos [1]. El código de la figura 4.3 muestra un anidamiento en forma canónica.

```
for(int i1=0;i1<=n1;i1++)
   for(int i2=0;i2<=n2;i2++)
      ...
      for(int ik=0;ik<=nk;ik++)
         predicados
```

Figura 4.3: k-ciclos anidados en forma canónica.

Un *vector de iteración*, $\vec{i}$ es una tupla de k elementos donde cada elemento i_j corresponde al índice de iteración del j-ésimo ciclo, yendo del más externo al más

[1] Existen diversos algoritmos para transformar un ciclo a su forma canónica

interno. Esta estructura se usa para representar cada uno de los puntos o momentos de la ejecución en que los predicados contenidos en el anidamiento son ejecutados.

El *espacio de iteración* de un anidamiento es el poliedro conformado por todos los vectores de iteración correspondientes a dicho anidamiento. La dimensión de éste es igual al número de ciclos en el anidamiento. En el caso del anidamiento de la figura 4.3 el espacio de iteración es:

$$[1 \cdots n_1] \times [1 \cdots n_2] \times \cdots \times [1 \cdots n_k]$$

Usamos "$\prec$" para denotar un ordenamiento lexicográfico entre los vectores de iteración. $\langle i_1, \cdots, i_k \rangle \prec \langle j_1, \cdots, j_k \rangle$ si y sólo si, $\exists l$, $1 \leq l \leq k$, tal que, $i_1 = j_1, \cdots, i_{l-1} = j_{l-1}$ e $i_l < j_l$. En particular, si $\vec{0} \prec \langle i_1, i_2, \cdots, i_k \rangle$ decimos que $\vec{i}$ es lexicográficamente positivo.

Con esta notación tenemos que $\langle i_1, \cdots, i_k \rangle \prec \langle j_1, \cdots, j_k \rangle$ si y sólo si, $\langle i_1, \cdots, i_k \rangle$ precede en orden de ejecución a $\langle j_1, \cdots, j_k \rangle$.

Un *recorrido* por el espacio de iteración de un anidamiento es la secuencia de ejecución de los vectores de iteración, esto es, un ordenamiento lexicográfico sobre los vectores de iteración. Dicho recorrido se puede representar gráficamente mediante una disposición de nodos, que representan a los vectores de iteración, y aristas punteadas que conectan estos nodos, las cuales representan el orden del recorrido. Como ejemplo tomemos el código de la figura 4.4; el espacio de iteración correspondiente al anidamiento es: $[0 \cdots 2] \times [0 \cdots 3]$ y el recorrido para este espacio está esbozado en la figura 4.5.

```
   for(int i=0;i<3;i++)
      for(int j=0;j<4;j++){
S1:      t = x + y;
S2:      a[i][j]=b[i][j] + c[i][j];
S3:      b[i][j]=a[i][j-1]*d[i][j] + t;
      }
```

Figura 4.4: Un ciclo doblemente anidado

Un espacio de iteración no siempre es rectangular, como en la figura 4.5; por ejemplo el anidamiento de la figura 4.6 presenta un espacio de iteración trapezoidal, como se aprecia en la figura 4.7.

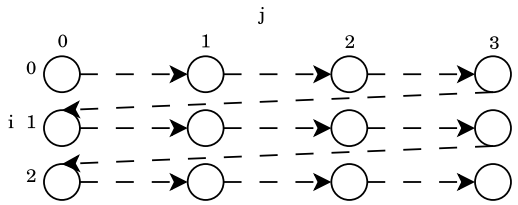


Figura 4.5: El recorrido del espacio de iteración correspondiente a la figura 4.4.

```
   for(int i=0;i<3;i++)
      for(int j=0;j<i+1;j++){
S2:      a[i][j]=b[i][j] + c[i][j];
S3:      b[i][j]=a[i][j];
      }
```

Figura 4.6: Un ciclo doblemente anidado con un espacio de iteración trapezoidal

Al usar variables con subíndices en ciclos anidados, las relaciones de dependencias son más complicadas que para las variables escalares; ya que estas dependencias son funciones tanto de los índices de las variables como de los predicados. Para expresar esta relación juntamos la etiqueta del predicado con los elementos del vector de iteración correspondiente encerrados en corchetes: $S_l[i_1, \cdots, i_k]$. A continuación extendemos la definición de "$\lhd$", $S_1[i_1, \cdots, i_k] \lhd S_2[j_1, \cdots, j_k]$ quiere decir que $S_1[i_1, \cdots, i_k]$ se ejecuta antes que $S_2[j_1, \cdots, j_k]$. Más formalmente $S_1[i_1, \cdots, i_k] \lhd S_2[j_1, \cdots, j_k]$ si y sólo si,

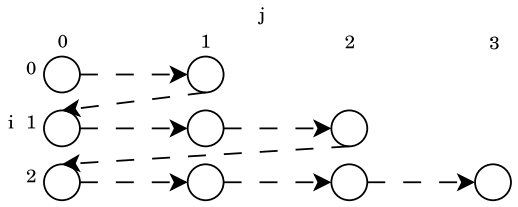


Figura 4.7: El recorrido del espacio de iteración correspondiente a la figura 4.6.

- S_1 está antes que S_2 en el código e $\langle i_1, \cdots, i_l \rangle \preceq \langle j_1, \cdots, j_k \rangle$.
- S_1 está después o es el mismo que S_2 e $\langle i_1, \cdots, i_l \rangle \prec \langle j_1, \cdots, j_k \rangle$.

Para el ejemplo de la figura 4.4, tenemos las siguientes relaciones de orden:

- $S2[i, j-1] \lhd S3[i, j]$,
- $S2[i, j] \lhd S3[i, j]$,

y las relaciones de dependencia correspondientes:

- $S2[i, j-1] \delta^f S3[i, j]$,
- $S2[i, j] \delta^a S3[i, j]$.

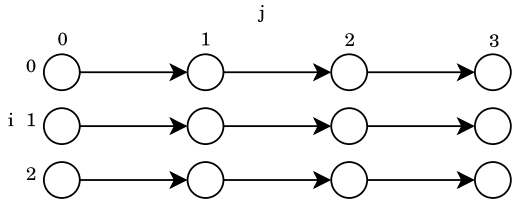


Figura 4.8: La gráfica de dependencias correspondiente al código en la figura 4.4.

Estas relaciones de dependencia también pueden ser representadas gráficamente, como en la figura 4.8. De la misma manera que en las gráficas del espacio de iteración, cada nodo representa un punto en dicho espacio y las aristas representan dependencias entre estos puntos. Las autodependencias, de un nodo en sí mismo, se omiten.

Para poder expresar más información entre dos instrucciones que dependen dentro de un espacio de iteración, es necesario definir una característica adicional de la dependencia. El concepto de *distancia de dependencia* nos sirve para ubicar relativamente a dos instrucciones dependientes dentro del espacio de iteración. Un *vector de distancia* en un anidamiento de `k` ciclos es un vector de `k` dimensiones

$\vec{d} = \langle d_1, \cdots, d_k \rangle$, donde cada d_i es un entero, tal que para algún vector de iteración $\vec{i}$ y el vector $\vec{i} + \vec{d} = \langle i_1 + d_1, \cdots, i_k + d_k \rangle$ existe una dependencia; este vector $\vec{d}$ denota la distancia de dicha dependencia.

Es fácil ver que para dependencias cuya distancia es $\langle 0, 0, \cdots, 0 \rangle$, las transformaciones realizadas a los ciclos no tendrían efecto alguno sobre las mismas, siempre y cuando estas transformaciones no modifiquen el orden de las instrucciones contenidas en el interior del ciclo. Siguiendo esta idea podemos catalogar a las dependencias de dos formas: como *independientes del ciclo*, las antes mencionada; y *acarreadas por el ciclo*, es decir, una dependencia que existe a causa de los ciclos.

Por ejemplo, en la figura 4.4, la dependencia de `S3` en `S2`, que surge del uso de b[i][j] en `S2` y su definición en `S3` es independiente del ciclo; aunque no hubiera ciclos rodeando los predicados, esta dependencia seguiría existiendo. En cambio la dependencia de flujo de `S2` a `S3`, que surge porque `S2` define a un elemento de `a`, el cual es usado en una iteración del ciclo de `j` después, es acarreada por el ciclo, en particular por el ciclo interior; la acción de eliminar el ciclo exterior no provocaría ninguna diferencia en la dependencia.

Para diferenciar, en términos de notación, entre una dependencia acarreada por cierto ciclo `i` y una dependencia independiente del ciclo usamos subíndices `i` y 0 respectivamente. Por ejemplo, para las dependencias en la figura 4.4 tendríamos:

$$S2[i, j-1]\delta_1^f S3[i, j]$$
$$S2[i, j]\delta_0^a S3[i, j]$$

Como hemos visto, el análisis de dependencias sirve para determinar si existe un orden intrínseco para la ejecución de las instrucciones de una región, así como para describir dicho orden. Esta información es imprescindible cuando se quiere cambiar el orden en el que se ejecutan dichas instrucciones. Por ejemplo, ésto puede resultar útil al adecuar una región a la política de ejecución de una máquina con entubamiento.

El encontrar este orden también resulta útil para determinar si es posible ejecutar secciones de código de manera simultánea. Esto se conoce como parelelización la cual se discute a continuación.

Capítulo 5

Autoparalelismo

5.1. Cómputo paralelo

En contraste con el cómputo secuencial, donde una tarea se ejecuta paso por paso siguiendo una secuencia, el cómputo paralelo consiste en ejecutar los pasos de dicha tarea de manera simultánea (o concurrente), con la finalidad de acelerar su ejecución.

Una arquitectura paralela permite ejecutar varias instrucciones al mismo tiempo utilizando ya sea varios procesadores, arquitecturas *multiprocesador*, o un solo procesador con dicha capacidad, procesadores *super escalares*.

En general los procesadores super escalares son circuitos que incluyen más de un componente para realizar cálculos, como la *ALU*. Actualmente la mayoría de los procesadores de propósito general son super escalares.

Una ventaja de las arquitecturas multiprocesador es que resulta más barato construir n procesadores de un mismo tipo x que un procesador n veces más rápido que x. Esto se debe a limitaciones tecnológicas pero también físicas. En general, una computadora multiprocesador acopla dos o más procesadores en un mismo sistema y proporciona un medio de comunicación entre éstos.

5.1.1. Clasificaciones

Podemos clasificar a los sistemas paralelos de varias formas; una de ellas es de acuerdo al esquema que siguen para ejecutar una tarea de manera concurrente:

Single instruction stream multiple data (SIMD): o misma instrucción múltiples datos. Este esquema consiste en ejecutar repetidamente una misma instrucción sobre un conjunto grande de datos. Lo presentan los procesadores super escalares, y resulta útil para procesamiento de gráficos y sonido.

Multiple instruction stream multiple data (MIMD): o múltiples instrucciones sobre múltiples datos. Este esquema lo presentan las arquitecturas multiprocesador. Es más general que el SIMD puesto que no restringe la ejecución paralela a una misma instrucción. En una arquitectura con este esquema cada componente opera con sus propios conjuntos de instrucciones y de datos.

Una misma arquitectura puede presentar ambos esquemas.

Una manera de clasificar a las arquitecturas multiprocesador es mediante el medio de comunicación que utilizan. De esta forma se agrupan en dos categorías:

Fuertemente acoplados: en este tipo de arquitectura los componentes son iguales entre sí y se comunican ya sea por medio de un bus o con memoria compartida.

Débilmente acoplados: los componentes de un sistema de este tipo pueden ser diferentes entre sí y se comunican a través de una red de alta velocidad.

Existen varios esquemas de memoria compartida y los sistemas fuertemente acoplados se pueden clasificar de acuerdo al esquema que utilizan para compartir memoria; dos de ellos son:

Simetric Multiprocesing (SMP): en este esquema procesadores idénticos comparten una misma memoria principal. Las arquitecturas de este tipo presentan un gran cuello de botella, puesto que los accesos a memoria, aparte de lentos, se deben realizar de forma secuencial.

Non-uniform memory access (NUMA): este tipo de sistemas divide la memoria en compartida y memoria local a cada procesador. Cada procesador tiene acceso exclusivo a su memoria local y la comunicación se lleva a cabo a través de la memoria compartida.

5.1.2. Paralelismo

El paralelismo en un programa o aplicación es la propiedad que tiene el código correspondiente de ser ejecutado en paralelo. Éste se puede dar a diferentes niveles, como son:

- De instrucción: el paralelismo a nivel instrucción se presenta cuando instrucciones dentro de una secuencia se pueden ejecutar de forma concurrente.
- De ciclo: cuando las iteraciones de un ciclo se ejecutan de manera simultánea.
- Por regiones: cuando distintas regiones del código pueden ejecutarse en paralelo. Esto involucra por lo general que el programa esté dividido en hilos o subprocesos.

Las diferentes arquitecturas se benefician de diferentes niveles de paralelismo. Por ejemplo los procesadores super escalares se benefician del paralelismo a nivel instrucción. En un procesador de este tipo se analizan las series de instrucciones para identificar cuáles de éstas se pueden ejecutar de manera paralela. Al identificarlas, éstas son despachadas a diferentes unidades de proceso para su ejecución simultánea. Otro ejemplo son las máquinas vectoriales que se benefician casi exclusivamente del paralelismo a nivel ciclo.

El nivel de paralelismo de una aplicación en concreto depende del diseño y de la tarea que se está realizando; hay tareas que son inherentemente secuenciales y otras que tienen un alto grado de paralelismo.

5.1.3. Diseño de programas paralelos

El diseño de un programa paralelo por lo general involucra definir conjuntos de tareas independientes, separarlos en subprocesos (o hilos) y establecer una forma

de comunicación entre ellos. Al definir los conjuntos de tareas se logra distribuir el trabajo entre los procesadores disponibles; una distribución equitativa ayuda a aprovechar al máximo cada procesador. Existen diferentes patrones para distribuir el trabajo; uno de los más simples es el *worker-manager* donde un proceso jefe (*manager*) distribuye el trabajo entre varios obreros (*workers*) para luego recolectar los resultados.

Cuando existen dependencias entre los procesos es necesario que éstos se comuniquen entre sí para resolverlas. Esto resulta costoso puesto que involucra esperas y establece una secuencia implícita entre los subprocesos. Un programa que requiera un alto grado de comunicación entre sus subprocesos es técnicamente secuencial.

Otro aspecto que hay que tomar en cuenta es la arquitectura, ya que no cualquier patrón de diseño se ajusta a cualquier tipo de arquitectura.

5.2. Sistemas multiprocesador

El paralelismo de los procesadores super escalares se aprovecha a nivel del hardware, puesto que éstos cuentan con un *despachador* el cual decide qué instrucciones se pueden ejecutar de manera paralela, por lo que el diseño de aplicaciones paralelas se reduce a arquitecturas multiprocesador. Algunos ejemplos de sistemas multiprocesador son:

5.2.1. Supercomputadoras

Son las primeras computadoras paralelas que existieron y fueron concebidas para correr aplicaciones muy particulares como cálculos científicos, predicciones climáticas o simulaciones físicas. Presentan arquitecturas fuertemente acopladas que involucran una jerarquía de memoria de varios niveles y cientos (a veces miles) de procesadores que acceden a la memoria con un esquema NUMA.

Son altamente costosas por lo que han cedido paso a los cúmulos.

5.2.2. Cúmulos (Clusters)

Los *clusters* o cúmulos son conjuntos de computadoras de propósito general débilmente acopladas, las cuales se comunican mediante una red de alta velocidad. Tienen varios usos como: proveer servicios de alta disponibilidad, como servidor de base de datos; balanceo de cargas, en un servidor de aplicacion; ó máquinas de alto desempeño, como substituto a las supercomputadoras.

5.2.3. CMP

Los *chip level multiprocesor (CMP)*, también conocidos como procesadores *multicore*, son una clase de procesadores que contienen varias unidades de procesamiento o corazones en un mismo circuito. Presentan una arquitectura *SMP* que permite ejecutar varios hilos de ejecución simultáneamente, en sus distintos corazones.

5.3. Autoparalelismo

Llamamos paralelismo implícito a la capacidad que tiene una región de código secuencial de ser ejecutada de forma paralela. Es decir, aunque el programa no haya sido diseñado explícitamente para correr en paralelo, algunas regiones de dicho programa pueden ser ejecutadas de forma paralela.

La paralelización o autoparalelismo consiste en descubrir el paralelismo implícito en código secuencial, para que este tipo de código también se vea beneficiado al correr en una arquitectura paralela, la cual por lo general tiene un desempeño pobre al ejecutar programas secuenciales. El paralelismo se puede descubrir de manera estática o en tiempo de ejecución.

El autoparalelismo estático ocurre en tiempo de compilación y consiste en analizar el código secuencial para posteriormente transformarlo en código paralelo. El análisis por lo general involucra análisis de flujo de datos y de dependencias. Una desventaja de este tipo de paralelización es que las transformaciones son realizadas de manera conservadora. Es decir, para asegurar que se conserva la semántica

del progama se supone que todo segmento de código es secuencial hasta probar lo contrario.

En el paralelismo en tiempo de ejecución se puede descubrir un mayor nivel de paralelismo, pues teniendo más información a la mano se alcanza una certeza mayor sobre el paralelismo de una región. La principal desventaja es la sobrecarga de trabajo provocada por los análisis realizados con este fin. Es posible que dicha sobrecarga pueda superar en tiempo al ahorrado mediante el paralelismo que se encontró.

Ambas técnicas se pueden mezclar para alcanzar mejores resultados.

5.3.1. Ley de Amdahl

No todo es miel sobre hojuelas. Supongamos que un hombre tarda un minuto en cavar un hoyo, ¿tardarían sesenta hombres un segundo en cavar un hoyo igual? Esta argumento es falaz puesto que claramente la tarea de cavar un hoyo no es cien por ciento paralela, de hecho pocas tareas lo son, por lo que el aumento de velocidad al paralelizar un programa no es lineal.

La *ley de Amdahl* para la paralelización proporciona una cota superior para el incremento de velocidad al ejecutar un programa en paralelo.

Sea F la fracción del código que es secuencial y $(1 - F)$ la fracción que puede ser paralelizada; el incremento máximo de velocidad para una arquitectura con N procesadores es:

$$\frac{1}{F + 1\frac{(1-F)}{N}}.$$

Por ejemplo si cierto programa es paralelizable al 90 %, el incremento máximo en su velocidad es en un factor de 10. En la práctica llega un momento en que la ganancia desempeño/costo disminuye rápidamente conforme se incrementa el número de procesadores, N, y $(1 - F)/N$ es chico en comparación a F. Por esta razón la paralelización es útil solamente si se tiene un número pequeño de procesadores o se trabaja con aplicaciones con un alto grado de paralelismo, con F pequeña.

En gran medida el problema que atañe al autoparalelismo es minimizar F.

5.3.2. Estado del autoparalelismo

El uso limitado de las computadoras multiprocesador ha limitado también el alcance del autoparalelismo, pues las técnicas desarrolladas se centran en un conjunto específico de aplicaciones: cómputo científico, simulaciones, etc. La mayoría de los esfuerzos por descubrir y explotar el paralelismo se enfocan en aplicaciones de este tipo. También existen compiladores para lenguajes de alto nivel, usados fuertemente en este tipo de aplicaciones, que incluyen paralelización.

Pruebas de dependencia

Una técnica de paralelización muy difundida son las pruebas de dependencia. Las dependencias entre instrucciones de un código imponen una secuencia sobre el mismo. En particular las dependencias acarreadas por el ciclo obligan a que éste sea ejecutado de manera secuencial. Las pruebas de dependencia son técnicas para, ya sea demostrar que no existen dependencias dentro de un ciclo y por consiguiente todas sus iteraciones se pueden ejecutar de forma concurrente; o para particionar las iteraciones del ciclo en conjuntos independientes entre sí que puedan ejecutarse de forma paralela.

Este problema, que involucra resolver sistemas de ecuaciones y análisis de expresiones se centra en ciclos que contienen operaciones sobre variables con subíndices, los cuales abundan en el cómputo científico.

A continuación se exponen algunas técnicas alternativas que han sido propuestas para paralelizar aplicaciones secuenciales.

5.4. Arreglos SSA

Como vimos anteriormente la forma `SSA` nos proporciona varias ventajas, en concreto para resolver problemas como el de propagación de constantes. Pero hasta

ahora nos hemos limitado a utilizar la forma SSA únicamente en variables escalares; las siguientes líneas describen una implementación de la forma SSA para arreglos, la cual nos servirá más adelante para definir una técnica de autoparalelismo.

```
if(condicion) then
   for(int i=0 ; i<n ; i++)
      x[i] = expresion1;
else
   for(int i=0;i<n;i++)
      x[i] = expresion2;
```

Figura 5.1: Un problema de arreglos SSA trivial

En el programa mostrado en la figura 5.1, podríamos considerar tratar a los arreglos de forma similar a los escalares; tomando en cuenta que en este caso particular todos los elementos de `x` los escribimos en la rama `then` o en la rama `else`, podemos usar una función ϕ con sólo dos argumentos, el arreglo definido en el `then` y el definido en el `else` .

```
for(int i=0 ; i<n ; i++)
   if(condicion[i]) then
      x[i] = expresion1;
   else
      x[i] = expresion2;
```

Figura 5.2: Un problema de arreglos SSA no tan trivial

Sin embargo el ejemplo mostrado en la figura 5.2 muestra una situación más complicada. El hecho de que la condición tambien esté variando con la iteración hace posibles muchas más de dos versiones del arreglo. Esto quiere decir que la función ϕ debe mezclar los elementos de los arreglos uno a uno, dependiendo del valor de la condición. Todo se debe a que este tipo de definiciones no necesariamente "matan" a una definición previa. Por lo cual se requiere de un análisis más complejo para implementar la forma `SSA` para arreglos.

Como último ejemplo tenemos el ciclo de la figura 5.3, donde las asignaciones al arreglo `x` son indirectas por medio de la función *f(i)*, que puede no ser monótona y puede no ser una permutación.

```
/* se inicializa x */
for(int i=0 ; i<n ; i++)
   if(condicion[i])
      x[f(i)] = expresion1;
```

Figura 5.3: Un ciclo con condicional variable e indireción

5.4.1. Dónde colocar la función ϕ

Para colocar la función ϕ se usan las siguientes reglas:

1. Definición ϕ: se coloca una función ϕ inmediatamente después de cada definición del arreglo que no "mate" por completo al valor de éste. A esta función se le llama *definición* ϕ y lo que hace es mezclar los valores de los elementos modificados en la definición con los valores disponibles anteriormente. En variables escalares no se necesitan insertar estas funciones, ya que una nueva definición siempre mata a la anterior.

2. Mezcla ϕ: ésta es la misma regla que en la forma SSA para escalares, en los nodos donde se juntan dos o más definiciones de la variable.

```
/* se inicializa x */
x0 = ...;
for(int i=0 ; i<n ; i++){
   x1 = phi(x4,x0);
   if(condicion[i]){
      x2[f(i)] = expresion1;
      x3 = phi(x2,x1);
   }
   x4 = phi(x3,x1);
}
x5 = phi(x4,x0);
```

Figura 5.4: Un ciclo con condicional variable e inderección

La figura 5.4 muestra el ejemplo de la figura 5.3 con las funciones ϕ insertadas mediante las reglas anteriores.

5.4.2. Semántica de la función ϕ

Si bien en la mayoría de los casos en los que usamos la forma SSA no es necesario profundizar en la semántica e implementación de la función ϕ, ya que ésta generalmente no está presente en el código objeto, para esta técnica de autoparalelización es necesario evaluar esta función en tiempo de ejecución, por lo que es importante describir su semántica de una manera más detallada.

Para poder determinar el resultado de la función ϕ nos apoyamos en otro concepto: los arreglos @. Para cada definición $X_k[i]$, $@X_k[i]$ representa el "momento" más reciente en el cual el i-ésimo elemento del arreglo fue modificado por dicha definición. El valor inicial para todos los elementos de cada arreglo $@X_k$ es $\perp\!\!\!\perp$ que quiere decir que ningún elemento de X_k ha sido modificado aún. En un programa sin ciclos los elementos de los arreglos @ tienen sólo dos estados posibles $@X_k[i] = \perp\!\!\!\perp$ y $@X_k[i] \neq \perp\!\!\!\perp$, pero en programas que involucran uno o más ciclos usamos los ya mencionados vectores de iteración para indicar el momento en el que ocurre cada definición. Esto también incluye a las definiciones ϕ.

El definir $@X_k[j] = \vec{i}$ después de cada definición de $X_k[j]$ recuerda correctamente el momento en que ésta ocurrió, aún cuando j sea resultado de una función de $\vec{i}$, $f(\vec{i})$, que no necesariamente es monótona ni una permutación. El valor de $@X_k$ puede ser modificado en iteraciones subsecuentes, pero podemos tener la certeza de que su valor final será el vector de iteración correspondiente a la última modificación. Esto nos conduce a la noción de *definición dinámica*. Los elementos de un arreglo pueden ser definidos varias veces durante la ejecución del programa y a cada una de estas definiciones le llamamos definición dinámica del elemento. Éstas se especifican con:

- una definición particular y
- el vector de iteración correspondiente.

Para cada elemento del arreglo y punto de ejecución únicamente, la última definición dinámica es *visible*, ya que cada nueva definición oculta a la anterior.

El trabajo de la función ϕ es identificar la definición dinámica visible para cada elemento del arreglo. Por ejemplo, la definición ϕ de la figura 5.4, $X_3 = \phi(X_2, X_1)$ representa una mezcla de los arreglos X_2 y X_1 elemento por elemento. La expresión condicional correspondiente a esta función ϕ es:

$$\phi(X_2^i[j], X_1^i[j]) = \begin{cases} X_2^i[j] & \text{si } @X_2^i[j] \geq @X_1^i[j] \\ X_1^i[j] & \text{si } @X_1^i[j] > @X_2^i[j] \end{cases}$$

Donde X_k^i denota el valor de la definición dinámica X_k en la iteración i del ciclo L. Como dijimos anteriormente el operador '$\geq$' representa el orden lexicográfico de los vectores de iteración; por lo tanto la expresión de arriba asigna, durante cada iteración, a X_3^i el valor más reciente entre X_2^i y X_1^i.

Como la función ϕ hace uso de los arreglos @ de cada uno de sus parámetros, éstos se pueden poner explícitamente como parámetros formales o no, como es el caso de la figura 5.5, la cual muestra cómo insertar los cálculos de $@X_k[i]$ después de cada definición y después de cada función ϕ.

```
/* se inicializa x */
x0 = ...;
@x0 = ...;
for(int i=0 ; i<n ; i++){
   x1 = phi(x4,x0);
   @X1=max(@x4,@x0)
   if(condicion[i]){
      x2[f(i)] = expresion1;
      @x2[f(i)]=(i);
      x3 = phi(x2,x1);
      @x3 = max(@x2,@x1);
   }
   x4 = phi(x3,x1);
   @x3 = max(@x3,@x1);
}
x5 = phi(x4,x0);
@x5 = max(@x4,@x0);
```

Figura 5.5: El mismo ciclo de la figura 5.4 con los arreglos @ insertados

5.4.3. Usando arreglos SSA para autoparalelización

Ahora que podemos calcular los arreglos @ y los valores de las funciones ϕ en tiempo de ejecución, nos es posible utilizar los arreglos SSA para paralelizar una amplia variedad de ciclos.

La ejecución en serie es necesaria fundamentalmente en ciclos con las siguientes características:

- La presencia de dependencias verdaderas de datos, en las cuales un valor dentro de una iteración es determinado mediante un valor calculado en la iteración anterior.
- Cuando tenemos dependencias de salida, esto es que una variable se defina repetidamente a lo largo de las iteraciones, en estos casos es necesario saber qué valor debe de ser visible al final de la iteración.
- Cuando existe una dependencia de control ya que, la modificación de algún elemento del arreglo estará condicionada por una definición dinámica de algún elemento del mismo arreglo durante una iteración anterior.

Sin embargo, gracias a lo mencionado en secciones anteriores, podemos intuir que la función ϕ en la forma SSA para arreglos, es una alternativa para asegurar que los valores corrrectos son visibles al momento de terminación del ciclo, sin requerir ejecución secuencial.

Para ilustrar esto usamos una versión optimizada de la forma SSA del código mostrado en la figura 5.5. Esta transformación, figura 5.6, se logra propagando algunas definiciones y utilizando las definiciones de `max` y de la función ϕ.

Paralelización abstracta

Denominamos a esta paralelización como “abstracta” porque no nos preocupamos por la sobrecarga de tiempo que genera el usar los Arreglos SSA en el programa; nuestra única preocupación es mostrar que en efecto se puede paralelizar el ciclo en cuestión.

```
/* se inicializa x */
x0 = ...;
@x0 = ...;
for(int i=0 ; i<n ; i++){

   if(condicion[i]){
      x2[f(i)] = expresion1;
      @x2[f(i)]=<i>;
   }
}
x5 = phi(x2,x0);
@x5 = max(@x2,@x0);
```

Figura 5.6: Versión SSA optimizada del ciclo mostrado en la figura 5.5.

En el ciclo mostrado en la figura 5.6 existen dos dependencias de salida que impiden la paralelización: la de la definición $X_2[f(i)]$ y la de $@X_2[f(i)]$. Ambas tienen una indirección f que puede resultar en múltiples modificaciones a un mismo elemento del arreglo.

Para lograr la paralelización es necesario expander los arreglos $@X_2$ y X_2 una dimensión más sobre el eje de iteración del ciclo (con el mismo rango que i). De esta forma si $i_1 \neq i_2$ podremos distinguir entre $X_2[f(i_1), i_1]$ y $X_2[f(i_2), i_2]$ aún cuando $f(i_1) = f(i_2)$. Pero ahora tenemos que cambiar la forma de calcular ϕ para trabajar con los arreglos "expandidos". Antes de realizar la expansión teníamos la siguiente condición para calcular el valor final de X_5, $X_5 = \phi(X_2, X_0)$:

$$\phi(X_2^n[j], X_0[j]) = \begin{cases} X_2^n[j] & \text{si } @X_2^n[j] \geq @X_0[j] \\ X_0[j] & \text{si } @X_0[j] > @X_2^n[j] \end{cases}$$

A este nuevo cálculo de ϕ le llamamos *finalización* y consta de dos pasos que se repiten para cada k:

- Para cada k,
  - Localizamos el valor más grande para $@X_2[k, 0 : n]$.
  - Modificamos la expresión de la función ϕ para que tome en cuenta

los elementos adecuados de X_2 o de X_0, determinados por el índice encontrado en el paso anterior.

La figura 5.7 muestra la paralelización abstracta resultante.

```
x0 = ...;
@x0 = ...;
/*
 *EJECUCIÓN En este nivel abstracto suponemos que
 *cada procesador ejecuta concurrentemente las lineas
 *dentro del parallel_for, para algún valor de i
 */

parallel_for(int i=0 ; i<n ; i++){
   if(condicion[i]){
      x2[f(i),i] = expresion1;
      @x2[f(i),i]=<i>;
   }
}

/*
 *FINALIZACIÓN Para calcular el valor final de X5
 *Este ciclo no tiene dependencias y podemos ejecutarlo en paralelo
 */
parallel_for(int i=0 ; i<n ; i++){
  temp = max(@X_2[i]);
  if(temp>0)
      x5[i] = x2[i,temp];
  else
      x5[i] = x0[i];
}
```

Figura 5.7: Versión paralela del ciclo mostrado en la figura 5.6.

5.5. Polítopos

En esta sección se describe otra técnica basada en una representación de la gráfica de dependencias conocida como polítopo. Existen varios aspectos teóricos sobre los que se desarrolla esta técnica, por lo que nos enfocamos en un solo ejemplo para esbozar dicho método.

5.5.1. El producto polinomial

Supongamos que queremos calcular el producto de dos polinomios A y B ambos de grado n, utilizando la notación de cuantificadores de Dijkstra:

$$A = \langle \Sigma k : 0 \leq k < n : a[k] * z^k \rangle$$
$$B = \langle \Sigma k : 0 \leq k < n : b[k] * z^k \rangle$$

Nombramos al producto polinomial C, como:

$$C = A \odot B = \langle \Sigma k : 0 \leq k \leq 2n : c[k] * z^k \rangle$$

Donde los elementos de c están definidos de la siguiente manera:

$$\langle \forall k : 0 \leq k \leq 2n : c[k] := \langle \Sigma i, j : 0 \leq i \leq n \wedge 0 \leq j \leq n \wedge i + j = k : a[i]b[j] \rangle \rangle$$

Notamos que esta definición no impone un orden para calcular las sumas de los elementos de c, nuestro instinto nos dice que se pueden calcular de forma paralela.

5.5.2. Un programa con ciclos anidados

Para poder paralelizar, primero hay que presentar el anidamiento de una forma conveniente. La figura 5.8 muestra la primer versión del ciclo que calcula los coeficientes del polinomio c. Este ciclo presenta varias dependencias acarreadas, implícitas en el índice de `c` que en cada momento depende de `i` y de `j`. Otra forma de realizar este cálculo, resultado de una transformación a la forma *asignación*

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

Figura 5.8: Un ciclo que calcula los coeficientes de *C*

simple, utiliza un arreglo doble para representar los coeficientes, el ciclo correspondiente se muestra en la figura 5.9. Este ciclo no presenta una indirección tan oscura como `c[i+j]`, al terminar el ciclo los coeficientes del polinomio se encuetran en `c[0:n][0]` y `c[n][0:n]` .

```
for(int i = 0; i <= n; i++)
   for(int j = 0; j <= n; j ++)
      if( i==0 || j==0 )
         c[i][j] = a[i]*b[j];
      else
         c[i][j] = c[i-1][j+1] + a[i]*b[j];
```

Figura 5.9: Un ciclo que calcula los coeficientes de *C*

La gráfica de dependencia del anidamiento mostrado en la figura 5.9, es un *polítopo* de dos dimensiones, una por ciclo, esta gráfica se muestra en la figura 5.10. Cada nodo representa un vector de iteración y las dependencias de flujo se pueden deducir de manera trivial. Podemos particionar los vectores de iteración de dos formas: partición *temporal* y partición *espacial*, figuras 5.11(a) y 5.11(b) respectivamente. Los nodos dentro de una misma partición temporal pueden ejecutarse en paralelo y los nodos dentro de una partición espacial tienen que ejecutarse en secuencia, en un mismo procesador.

La partición espacial cubre todas las dependencias. Si ejecutamos cada línea de la partición de manera secuencial en un proceso, no se necesita comunicación entre los procesos.

Las particiones espacial y temporal se pueden combinar mediante un mapeo espacio-tiempo, una transformación afín al sistema de coordenadas en el que uno de los ejes está dedicado al tiempo, i.e. representa un ciclo secuencial, y el otro al espacio, i.e. representa un ciclo paralelo; esto con el fin de hacer explícito el tiempo

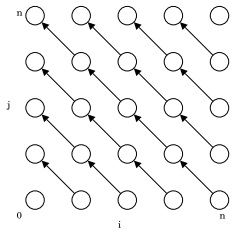
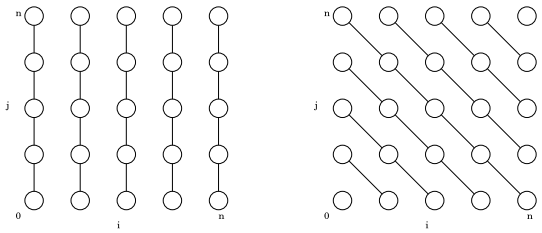


Figura 5.10: Gráfica de dependencia del anidamiento de la figura 5.9.



(a) Partición temporal de los vertices de 5.10

(b) Partición espacial de los vertices de 5.10

Figura 5.11: Particiones

y el espacio. El polítopo resultante se muestra en la figura 5.12. Para el código paralelo resultante tenemos dos opciones, con respecto al orden de los ciclos:

- Si el ciclo exterior es el tiempo y el interior es el espacio, entonces tenemos un programa síncrono con un reloj global, para todos los procesadores, figura 5.13.
- Si los ciclos están dispuestos de manera inversa, tenemos un programa asíncrono, donde cada procesador puede tener su propio reloj, figura 5.14.

En ambos casos el cuerpo del ciclo es el mismo, sólo hay que hacer un cambio de índices para invertir el mapeo $i = e$ y $j = e - t$.

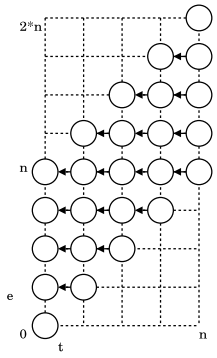


Figura 5.12: Combinación de las particiones espacial y temporal.

```
for(int t = 0; t <= n; t++)
   parallel_for(int e = t; e <= t + n; e++)
      if( t==0 || p-e==0 )
         c[t][e-t] = a[t]*b[e-t];
      else
         c[t][e-t] = c[t-1][e-t+1] + a[t]*b[e-t];
```

Figura 5.13: Ciclo paralelo síncrono.

```
parallel_for(int e = 0; e <= 2*n; e++)
for(int t = max(0,p-n); t <= min(n,p); t++)
      if( t==0 || p-e==0 )
         c[t][e-t] = a[t]*b[e-t];
      else
         c[t][e-t] = c[t-1][e-t+1] + a[t]*b[e-t];
```

Figura 5.14: Ciclo paralelo asíncrono

5.6. Paralelismo especulativo de hilos

En la actualidad los procesadores CMP son cada vez más comunes. Puesto que se han vuelto una alternativa popular a los procesadores secuenciales en las computadoras de escritorio. Y existen muchos indicativos, en particular los esfuerzos de los fabricantes de procesadores por crear procesadores con esta arquitectura, de que su presencia será dominante en los próximos años.

Esta arquitectura permite correr varios hilos al mismo tiempo en cada uno de los núcleos del procesador. Esto resulta ideal para entornos multitarea, pero los programas secuenciales no pueden percibir el beneficio de los núcleos extra que ofrece la arquitectura.

Esto aviva el problema del autoparalelismo que antes estaba prácticamente restringido al supercómputo. Ahora existe una mayor gama de aplicaciones que se requiere paralelizar. Pero las técnicas mencionadas anteriormente no resultan adecuadas para todo tipo de aplicación ya que en muchos casos es difícil exponer el paralelismo de forma estática.

El paralelismo especulativo de hilos (*speculative thread-level parallelism* ó *STP*) puede aprovechar las capacidades de un procesador de este tipo para ejecutar programas secuenciales. Esta técnica consiste en hardware especial que permita ejecutar hilos especulativamente sin que intervenga ninguna clase de análisis previo. La idea es tomar regiones que se sospechen paralelas, sin dependencias, dividirlas en hilos y ejecutarlas de manera concurrente. En caso de encontrar alguna dependencia, el hilo que viole dicha dependencia se vuelve a ejecutar.

Esta técnica no está limitada a la habilidad del programador o del compilador de encontrar el paralelismo dentro de alguna región. Es más, es capaz de alcanzar un nivel de paralelismo mayor a una paralelización estática perfecta, pues explota el paralelismo dinámico y no está restringida al esquema conservador que debe de seguir una técnica estática.

5.6.1. Hilos especulativos

Cuando un multiprocesador encuentra una región posiblemente paralela, crea varios hilos, dependiendo del nivel de paralelismo de la región y el número de

núcleos del multiprocesador, y procede a ejecutarlos especulativamente. Estos hilos se ejecutan de manera concurrente, pero "entregan" secuencialmente, en el orden adecuado. Aquí con el verbo entregar nos referimos a terminar exitosamente su ejecución. En caso de encontrar alguna dependencia se genera una *violación*. El estado del hilo que es afectado por dicha dependencia se descarta y el procesador devuelve al hilo a un estado inicial y comienza nuevamentesu ejecución, a esta operación le llamaremos *rebobinado*. El multiprocesador con STP debe de proveer hardware que facilite estos mecanismos de generación de violaciones y rebobinado de hilos.

Existen dos tipos de dependencia que pueden provocar una violación:

- Dependencia de datos: se detecta una violación de este tipo cuando un hilo tiene que modificar una localidad que un hilo posterior ya utilizó para realizar algún cálculo. En este caso el hilo que operó con el valor caducado es rebobinado. Este tipo de dependencia, "dependencia real", es el único tipo de dependencia de datos que provoca una violación.
- Dependencia de control: una violación de este tipo se genera cuando el flujo de control de un hilo determina que otro hilo subsecuente no debe de ser ejecutado, en cuyo caso dicho hilo es "rebobinado" y descartado.

Dos regiones que suelen ser buenos candidatos para ser ejecutados de manera especulativa son:

- Los ciclos.
- Las llamadas a procedimientos.

5.6.2. Paralelismo especulativo en ciclos

Los ciclos son el objetivo tradicional de la paralelización estática y también un candidato obvio para STP. Cada iteración del ciclo puede convertirse en hilo y correr en paralelo con las demás iteraciones. La única forma de dependencia de control que se comparte entre hilos son las condiciones de terminación. La paralelización de ciclos se puede dar a nivel *múltiple*, cuando se realiza sobre varios ciclos de un mismo anidamiento, o a nivel *simple* sobre sólo uno de dichos ciclos.

Cuando se realiza a nivel simple, hay que decidir cuál de los ciclos del anidamiento es más conveniente paralelizar; hay dos aspectos a tomar en cuenta:

- El grado de paralelismo. Esto está sujeto a la presencia de dependencias entre las diferentes iteraciones. Cuando éstas operan sobre un conjunto de datos disjuntos (un ciclo *doAll*) el grado de paralelismo es igual al número de iteraciones. Si existen dependencias a través de las iteraciones (ciclo *doAcross*) el grado de paralelismo se reduce al número de particiones de datos independientes del conjunto.
- La cobertura del paralelismo: si se paraleliza un ciclo interior entonces el código exterior correrá de forma secuencial. De manera que puede resultar conveniente ejecutar un ciclo *doAcross* exterior por sobre un ciclo *doAll* interior. En general la cobertura se refiere al porcentaje del código que se ejecutará en paralelo.

Los ciclos a ejecutar pueden ser elegidos a mano por el programador, buscando un balance en ambos aspectos. Si hay poca cobertura el desempeño puede verse mermado por la sobrecarga de crear y mantener los hilos. Un grado de paralelismo pequeño repercute en un gran número de violaciones.

5.6.3. Paralelismo especulativo a nivel de procedimiento

Los procedimientos son una herramienta de abstracción que sirven para crear unidades de cómputo poco acopladas entre sí. Esto sugiere la posibilidad de traslapar la ejecución de un procedimiento con el código que normalmente se ejecuta después del regreso de dicho procedimiento.

El paralelismo especulativo a nivel procedimiento consiste en ejecutar concurremente el procedimiento llamado junto con el código que sigue al regreso del mismo. El segundo es el que se ejecuta de forma especulativa. Este método resulta atractivo pues no existen dependencias de control entre el hilo del procedimiento y el hilo especulativo.

Una violación de dependencia de datos ocurre cuando el código que le sigue al regreso lee una localidad de memoria antes de que el procedimiento escriba en

ella. En este caso se utiliza el mismo mecanismo usado en los ciclos: se rebobina el hilo y se ejecuta nuevamente.

Predicción de valores

Frecuentemente el valor regresado por un procedimiento es usado inmediatamente después del regreso de éste; dicha circunstancia puede provocar un gran número de violaciones. Una forma de disminuir las violaciones es mediante la *predicción de valores*. El hilo del código que llama se ejecuta de forma especulativa usando un valor probable del valor de regreso del procedimiento. Una vez que termina el hilo del procedimiento -nótese que el hilo del procedimiento debe terminar primero- se compara el valor de regreso real con el valor predicho; si estos dos valores son diferentes el hilo especulativo es rebobinado y ejecutado nuevamente utilizando el valor real.

La predicción de valores en general permite a un programa correr más rápido que el límite establecido por las dependencias de datos, ya que se anticipa al resultado. Existen varios esquemas de predicción de datos, dentro de los cuales se encuentran:

- La predicción de último valor: ésta toma como valor probable el valor cargado en la localidad durante la última operación de carga.
- El valor de salto o zancada: toma como valor probable el último valor cargado más la diferencia entre los últimos dos.

Para hacer posible la predicción de valores es necesario que el hardware facilite un cache para poder almacenar los valores pertinentes, dependiendo del esquema de predicción que se use.

5.6.4. Complicaciones

Obviamente uno de los pormenores de esta técnica es la sobrecarga inherente a la creación y mantenimiento de los hilos. Por lo tanto es necesario tener hardware

especializado. También es necesario que el mismo hardware maneje las interrupciones provocadas por las violaciones.

Para disminuir la sobrecarga total al ejecutar un programa hay que aumentar el tamaño de los hilos. Mientras más largos sean los hilos el nivel total de sobrecarga disminuirá. Para aumentar el tamaño de los hilos se pueden usar técnicas estáticas como *forrado de funciones* y pegado de ciclos. También hay que tomar en cuenta que tener hilos muy largos aumenta el riesgo de que se suscite una violación.

Otra limitante es el tiempo de procesamiento extra que provoca el rebobinar un hilo y comenzar su ejecución desde el estado inicial. Una forma de disminuir este tiempo es implementando puntos de control a través del flujo del hilo. Si ocurriese una violación el hilo sería devuelto al último punto de control que éste hubiera alcanzado.

Capítulo 6

Conclusiones

Los análisis de flujo de control, flujo de datos y de dependencias han sido un campo de estudio muy difundido. Existen muchas técnicas de optimización que ya han sido implementadas y se encuentran en casi cualquier compilador. El rango de programas y arquitecturas para las cuales estas optimizaciones son útiles es amplio. La paralelización es un subconjunto de las optimizaciones posibles, pero el rango de programas que se benefician de ella es mucho menor al resto; en parte porque hasta hace poco, el accesso a este tipo de arquitecturas estaba restringido a un grupo de usuarios reducido, en su mayoría científicos y dependencias gubernamentales. La paralelización dinámica ha estado restringida aún más, puesto que el conjunto de arquitecturas que soportan esta clase de técnicas es todavía menor.

Sin embargo el problema que orilló al súper cómputo a implementar arquitecturas paralelas, la necesidad de una mayor velocidad, la cual estaba siendo acotada por limitaciones tecnológicas, ha alcanzado a las computadoras de escritorio comunes.

El uso de las arquitecturas multiprocesador será cada vez más común, lo cual acerca las arquitecturas paralelas a un rango mayor de aplicaciones que podrían verse beneficiadas de manera directa[1] por la existencia de unidades adicionales de procesamiento.

[1] En ambientes multitareas el sistema operativo siempre puede beneficiar a estas aplicaciones de manera indirecta

Esta situación es un incentivo para replantear el problema de la paralelización desde otra perspectiva: redefinir conceptos y buscar nuevas técnicas dinámicas estáticas o híbridas que puedan aprovechar esta tendencia.

Los polítopos y arreglos SSA son técnicas estáticas que pertenecen a la vieja escuela, las cuales benefician principalmente a programas que realizan cálculos numéricos y a los ciclos que éstos contienen. Pero TSP es una de las nuevas técnicas que ha tenido auge en los últimos años. Existen muchas publicaciones, a nivel tanto teórico como técnico, acerca de este método, con análisis sobre sus ventajas y desventajas, así como el impacto que tiene en distintos tipos de programas.

Por todo esto creemos que se seguirán desarrollando nuevas técnicas de paralelización que adecúen el software a las nuevas opciones de procesamiento, por lo que es importante continuar el estudio del problema, utilizando diferentes perspectivas que cubran una gama más amplia de posibilidades.

Apéndice A

Implementaciones

El siguiente apéndice recopila las implementaciones de algunos algoritmos relativos a las primeras secciones del trabajo, en particular los capítulos 1 y 2. El objetivo es reforzar los conceptos revisados en dichos capítulos.

Los programas presentados están escritos en el lenguaje de programación *Python*, versión 2.4.2. Python es un lenguaje poderoso con una sintaxis simple. Es interpretado y utiliza tipos dinámicos. Cuenta con estructuras de datos eficientes y de alto nivel. Además sigue el paradigma orientado a objetos. Por estas razones fue el lenguaje escogido para implementar las funciones y las clases que se listan a continuación.

A.1. Análisis de flujo de control

Primero se listarán las estructuras de datos implementadas para el análisis de flujo de control.

A.1.1. Estructuras de datos

En el listado A.1 se muestra la clase `Nodo`, la cual sirve para representar un nodo o bloque dentro de una gráfica de flujo. Cada nodo cuenta con un conjunto de predecesores, un conjunto de sucesores y un conjunto de arcos o aristas dirigidas.

```
1  import sets
2  class Nodo:
3      BLANCO = 'blanco';
4      GRIS = 'gris';
5      NEGRO = 'negro';
6      colores = (BLANCO, GRIS, NEGRO)

8      '''Esta clase representa a un nodo dentro de una
           grafica'''

10     def __init__(self, nombre):
11         '''Crea un nodo sin predecesores ni sucesores'''
12         self.nombre = nombre
13         self.predecesores = sets.Set()
14         self.sucesores = sets.Set()
15         self.color = Nodo.BLANCO
16         self.arcos = Mapa()

18     def agregaPredecesor(self, predecesor):
19         '''Agrega un nodo al conjunto de predecesores'''
20         if self != predecesor:
21             self.predecesores.add(predecesor)

23     def agregaSucesor(self, sucesor):
24         '''Agrega un nodo al conjunto de sucesores'''
25         if self != sucesor:
26             if not sucesor in self.sucesores:
27                 self.sucesores.add(sucesor)
28                 sucesor.agregaPredecesor(self)
29                 self.arcos[sucesor] = Arco(self,sucesor)

31     def remueveSucesor(self, sucesor):
32         '''Remueve un sucesor del conjunto de sucesores'''
33 5       self.sucesores.discard(sucesor)
34         sucesor.remuevePredecesor(self)

36     def remuevePredecesor(self, predecesor):
```

```
37          '''Remueve el predecesor del conjunto de
                predecesores'''
38          self.predecesores.discard(predecesor)

40      def getLlave(self):
41          '''Obtiene la llave del nodo el cual es un objeto
                inmutable y es util para usarlo como llave en
                estructuras que asi lo requieran como los mapas.
                '''
42          return self.nombre

44      def getArco(self,nodo):
45          if arcos.has_key(nodo):
46              return arcos[nodo]
47          else:
48              return None

50      # A continuacion vienen las funciones de conveniencia
51      def __eq__(self,otro):
52          '''Dos Nodos son iguales cuando tienen el mismo
                nombre'''
53          return self.nombre == otro.nombre

55      def __str__(self):
56          '''Forma la cadena que representa a este objeto'''
57          if len(self.sucesores) < 1:
58              return str(self.nombre)
59          else:
60              returnValue = str(self.nombre) + ' sucesores:('
61              for sucesor in iter(self.sucesores):
62                  returnValue = returnValue + str(sucesor) +'
                        ,'
63              returnValue = returnValue[:-1] + ')'

65          return returnValue

67      def __hash__(self):
68          '''Funcion hash para esta clase '''
69          return hash(self.nombre)
```

Listing A.1: Implementación de la clase Nodo.

El listado A.2 muestra la clase `Arco`, la cual es una implementación de un arco. Cuenta con cuatro campos: la `cabeza` que es el nodo al que apunta; la `cola`

que es el nodo de donde sale; un `tipo`, el cual indica el tipo de arco que es; y un boleano, el cual indica si esta arista ya ha sido recorrida.

```
1  class Arco:
2      ARBOL = 'arbol'
3      ATRAS = 'hacia atras'
4      ADELANTE = 'hacia adelante'
5      CRUZADO = 'cruzado'

7      tipos = (ARBOL, ATRAS, ADELANTE, CRUZADO)

9      '''Esta clase abstrae un arco entre dos nodos tiene un
           nodo cola, de donde viene, y otro nodo cabeza, a
           donde apunta.'''
10     def __init__(self,cola,cabeza):
11         self.cola = cola
12         self.cabeza = cabeza
13         self.tipo = None
14         self.recorrido = False
```

Listing A.2: Implementación de la clase Arco.

Por último el listado A.3 muestra las implementaciones de un mapa y una pila, estructuras que son usadas en varios de los algoritmos que se listarán posteriormente.

```
1  class Mapa(dict):
2      '''Esta clase es la abstraccion de un mapa con llaves
           mutables. Extiende a diccionario y se comporta de la
            misma manera solo que las llaves deben de
           implementar un metodo llamado getLlave() que regrese
            un objeto inmutable'''

4      def __setitem__(self,llave,objeto):
5          ''' Actualiza o agrega un valor al mapa'''
6          dict.__setitem__(self,llave.getLlave(),objeto)

8      def __getitem__(self,llave):
9          '''Para obtener un item dado una llave'''
10         return dict.__getitem__(self,llave.getLlave())

12     def has_key(self,llave):
13         return dict.has_key(self,llave.getLlave())

15     def __str__(self):
```

```
16          returnValue = "{"
17          for llave in self.iterkeys():
18              returnValue = returnValue + llave + ":"
19              returnValue = returnValue + str(dict.
                    __getitem__(self,llave))
20          returnValue = returnValue + "}"
21          return returnValue

23  class Pila:
24      '''Esta clase implementa una pila'''
25      def __init__(self):
26          self.elementos = list()
27          self.size = 0;

29      def push(self,elemento):
30          self.elementos.append(elemento)
31          self.size += 1

33      def pop(self):
34          if not self.esVacia():
35              elemento = self.elementos.pop(self.size - 1)
36              self.size -= 1
37              return elemento
38          else:
39              return None

41      def esVacia(self):
42          return self.size == 0

44      def peek(self):
45          elemento = self.elementos[self.size]
46          return elemento
```

Listing A.3: Implementación de las estructuras mapa y pila.

A.1.2. Algoritmos

Exploraciones

Las siguientes clases explotan la herencia para definir primero una exploración genérica, en la clase *Exploracion*, y un método de procesamiento para los nodos de

una gráfica, en la clase *Procesador*, ver listado A.4.

```
1  class Exploracion:
2      '''Esta clase representa a una exploracion abstracta,
           realmente no hace nada'''
3      def __init__(self, procesador):
4          self.procesador = procesador

6      def explora(self, nodo):
7          pass

9  class Procesador:
10     '''Esta clase abstrae todos los procesos que hay que
           hacer a una grafica con cada exploracion. El
           objetivo es poder combinar distintos criterios de
           exploracion con diferentes tecnicas de procesamiento
            para lograr cosas diferentes.'''
11     def procesaAntes(self,nodo):
12         '''Realiza las operaciones correspondientes sobre
               el nodo al comenzar el proceso'''
13         nodo.color = Nodo.GRIS

15     def procesaDespues(self,nodo):
16         '''Realiza las operaciones sobre el nodo
               corespondientes a finalizar el proceso'''
17         nodo.color = Nodo.NEGRO

19     def procesaArco(arco):
20         '''Procesa un arco'''
21         arco.recorrido = True
```

Listing A.4: Exploración genérica y Procesador básico.

El listado A.5 muestra una implementación particular de exploración. Ésta realiza un recorrido a profundidad, *DFS*, en una gráfica y ejecuta diferentes métodos de un objeto de tipo `Procesador` en tres puntos del recorrido:

- `procesaAntes()`: método que procesa un nodo antes de explorar a sus sucesores,
- `procesaDespues()`: procesa un nodo después de explorar a sus sucesores,
- `procesaArco()`: procesa un arco al encontrarlo.

Estos métodos pueden ser redefinidos en clases que hereden de DFS para agregar procesamiento al recorrido.

```
1  class DFS(Exploracion):
2      ''' La exploracion DFS basica '''
3      def __init__(self, procesador):
4          Exploracion.__init__(procesador)

6      def explora(self, nodo):
7          '''Exploracion sobre una grafica tipo DFS'''
8          self.procesador.procesaAntes(nodo)
9          for sucesor in iter(nodo.sucesores):
10             self.procesador.procesaArco(nodo.getArco(
                   sucesor))
11             if sucesor.color == Nodo.BLANCO:
12                 self.explora(sucesor)
13         self.procesador.procesaDespues(nodo)
```

Listing A.5: Implementación de la exploración DFS.

Por último la clase `EncuentraArbolGeneradorDFS` comprende un Procesador más particular y la exploración DFS, puesto que extiende ambas clases y redefine los métodos de procesamiento. Al ejecutar su método `exploracion` ésta obtiene el árbol generador de la gráfica en *post* orden y en *pre* orden, también etiqueta los arcos de la gráfica según el tipo (arco del árbol, arco hacia adelante, arco hacia atrás o arco cruzado).

```
1  class EncuentraArbolGeneradorDFS(Procesador,DFS):
2      ''' Procesa el arbol generador utilizando informacion
           obtenida en post orden y en pre orden'''
3      def __init__(self):
4          Procesador.__init__()
5          DFS.__init__(self)
6          self.i = 1
7          self.j = 1
8          self.pre = Mapa()
9          self.post = Mapa()

11     def procesaAntes(self,nodo):
12         pre[nodo] = i
13         i = i + 1
14         Procesador.procesasAntes(nodo)


17     def procesaDespues(self,nodo):
```

```
18          post[nodo] = j
19          j = j + 1
20          Procesador.procesaDespues(nodo)

22      def procesaArco(arco):
23          if not arco.recorrido:
24              arco.recorrido = True
25              arco.tipo = Arco.ARBOL
26          elif pre[arco.cola] > pre[arco.cabeza]:
27              arco.tipo = Arco.ADELANTE
28          elif not post.has_key(arco.cabeza):
29              arco.tipo = Arco.ATRAS
30          else:
31              arco.tipo = Arco.CRUZADO
```

Listing A.6: Clase que hereda de DFS y Procesador e implementa un algoritmo que encuentra el árbol generador de la gráfica etiquetando sus arcos.

La idea completa de este tipo de algoritmos se muestra en [12].

Bloques básicos

La clase `BuscadorBBExt`, mostrada en el listado A.7, construye los conjuntos de bloques básicos extendidos de una gráfica de flujo. El método `agregaBB()` agrega nodos al bloque básico extendido que recibe como parámetro, hasta que ya no haya más nodos que agregar. El método `encuentraTodosBBE()` encuentra todos los bloques básicos extendidos de la gráfica dada la raíz y los almacena en el campo `todosLosBBE`.

```
2   from estructuras import Nodo
3   from sets import Set

5   class BuscadorBBExt:
6       ''' Esta clase construye todos los bloques basicos
            extendidos de una grafica de flujo dada la raiz de
            esta grafica, y los almacena en su campo todosLosBBE
            .'''

8       def __init__(self,raiz):
9           encuentraTodosBBE(raiz)
```

```
11      def encuentraBBExtendido(self,raiz):
12          bB_Extendido = Set()
13          agregaBB(raiz,bB_Extendido)
14          return bBExtendido

16      def agregaBB(self,raiz, bBExtendido):
17          cambio = True
18          while cambio:
19              cambio = False
20              bBExtendido.add(raiz)
21              for sucesor in iter(raiz.sucesores):
22                  if len(sucesor.predecesores) == 1 and not
                        bBExtendido.contains(sucesor):
23                      agregaBB(sucesor,bBExendido)
24                      cambio = True
25                  elif not raicesBBE.contains(sucesor):
26                      raicesBBE.add(sucesor)
27                      cambio = True

29      def encuentraTodosBBE(self,raiz):
30          self.raicesBBE = Set()
31          self.raicesBBE.add(raiz)
32          self.todosLosBBE = Mapa()

34          while len(self.raicesBBE) > 0:
35              actual = raicesBBE.pop()
36              raicesBBE.remove(actual)
37              if not todosLosBBE.has_key(actual):
38                  todosLosBBE[actual] = encuentraBBExtendido(
                        actual)
```

Listing A.7: Clase que encuentra los bloques básicos extendidos de una gráfica de flujo.

Dominadores

El listado A.8 muestra funciones que encuentra los conjuntos de dominadores para cada nodo de una gráfica.

La función `dominadores()` encuentra el conjunto de dominadores `dominadores[nodo]`

para cada nodo de la gráfica. Y regresa un `Mapa` que asocia a cada nodo con sus dominadores.

La función `dominadoresI()` encuentra los dominadores inmediatos para todos los nodos en un conjunto dado; esta función recibe como parámetro el `Mapa` obtenido en la función mencionada arriba.

```
1  from estructuras import *
2  from sets import Set

4  def dominadores(raiz,n):
5      '''Regresa un mapa con un nodo y sus dominadores. raiz
           : es un Nodo en del cual vamos a partir n: es un Set
            de nodos sobre el cual vamos a calcular los
           dominadores.'''
6      cambio = True
7      dominadores = Mapa()
8      n1 = n.copy()
9      n1.discard(raiz)
10     for nodo in iter(n1):
11         dominadores[nodo] = n
12     while cambio:
13         cambio = False
14         for nodo in iter(n1):
15             t = n1.copy()
16             for pred in iter(nodo.precesesores):
17                 t.intersection_update(dominadores[nodo])
18             t.add(nodo)
19             if t != dominadores[nodo]:
20                 chancge = True
21                 dominadores[nodo] = t
22     return dominadores

24 def dominadoresI(raiz,nodos,dominadores):
25     '''Encuentra los dominadores inmediatos dado el nodo
           raiz, y el mapa con los dominadores de cada nodo.
           raiz: el raiz de la grafica: el conjounto de nodo
           dominadores: el mapa que asocia a cada nodo con su
           conjunto de dominadores.'''
26     temp = Mapa()
27     idom = Mapa()
28     for n in iter(nodos):
29         conjuntoN = Set()
30         conjuntoN.add(n)
31         temp[n] = dominadores[n].diference(conjuntoN)
```

```
32      nMenosRaiz = nodos.copy()
33      nMenosRaiz.discard(raiz)

35      for n in iter(nMenosRaiz):
36          for s in iter(temp[n]):
37              tempMenosS = temp[n].copy()
38              tempMenosS.discard(s)
39              for t in iter(tempMenosS):
40                  if temp[s].contains(t):
41                      temp[n].discard(t)

43      for n in iter(nMenosRaiz):
44          idom[n] = temp[n].pop()
45      return idom
```

Listing A.8: Funciones que encuentran los conjuntos de dominadores y dominadores inmediatos para los nodos de una gráfica.

A.1.3. Análisis de flujo de datos

A continuación se muestran los listados de las clases y funciones que involucran algoritmos de análisis de flujo de datos.

Estructuras de datos

El listado A.9 muestra la implementación de un vector de bits. En ésta se redefinen diferentes métodos para aprovechar la sobrecarga de operadores que nos brinda el leguaje de programación.

```
2   class VectorBits:
3       '''Esta clase implementa a un vector de bits y las
            operaciones que se pueden realizar sobre la misma'''

5       UNO = 1
6       CERO = 0
7       TOPE = 1
8       FONDO = 0
```

```
10      def __init__(self,valores):
11          self.valores = valores

13      def __eq__(self,vector):
14          return self == vector

16      def __len__(self):
17          return len(self.valores)

19      def __and__(self,vector):
20          '''Regresa un nuevo vector que es el resultado de
                reunir este vector con el que se da como
                parametro'''
21          valoresNuevos = list()
22          i = 0
23          for v in iter(self.valores):
24              valoresNuevos.append(v & vector.valores[i])
25              i = i + 1
26          vectorNuevo = VectorBits(valoresNuevos)
27          return vectorNuevo

29      def __or__(self,vector):
30          '''Regresa un nuevo vector que es el resultado de
                reunir este vector con el que se da como
                parametro'''
31          valoresNuevos = list()
32          i = 0
33          for v in iter(self.valores):
34              valoresNuevos.append(v | vector.valores[i])
35              i = i + 1
36          vectorNuevo = VectorBits(valoresNuevos)
37          return vectorNuevo

39      def __eq__(self,otro):
40          indice = 0
41          for i in iter(self.valores):
42              if i != otro.valores[indice]:
43                  return False
44              indice = indice + 1
45          return True

47      def __ne__(self,otro):
48          return not self == otro

50      def __str__(self):
```

```
51          returnValue = '<'
52          for valor in self.valores:
53              returnValue = returnValue + str(valor) + ','
54          returnValue = returnValue[:-1]
55          returnValue = returnValue + '>'
56          return returnValue

58      def __getitem__(self,indice):
59          return self.valores[indice]
```

Listing A.9: Implementación de un vector de bits.

La implementación de una retícula de bits se muestra en el listado A.10.

```
1   class ReticulaBits:
2       '''Implementa a una reticula'''
3       def __init__(self,size):
4           self.size = size
5           tope = list()
6           for i in range(0,self.size):
7               tope.append(VectorBits.TOPE)
8           self.TOPE =  VectorBits(tope)
9           fondo = list()
10          for i in range(0,self.size):
11              fondo.append(VectorBits.FONDO)
12          self.FONDO =  VectorBits(fondo)
```

Listing A.10: Implementación de una retícula de bits.

Funciones de flujo

Los siguientes listados muestran clases que sirven para implementar el comportamiento de una función de flujo.

El listado A.11 muestra la clase `Regla`. Esta clase se encarga de implementar las tranformación del vector de bits correspondiente a la función de flujo.

```
1   class Regla:
2       '''Abstrae las reglas de una funcion'''

4       def __init__(self, *valores):
5           self.valores = valores
```

```
7       def aplica(self,vector):
8           nuevoVector = list()
9           indice = 0
10          for x in self.valores:
11              if x == 'x':
12                  nuevoVector.append(vector[indice])
13              else:
14                  nuevoVector.append(vector[indice] | x)
15              indice = indice + 1
16          return VectorBits(nuevoVector)
```

Listing A.11: Clase Regla.

La función de flujo es implementada en la clase `FuncionFlujo` que se muestra en el listado A.12. La función se construye a partir de un mapa de reglas, el cual asocia a cada nodo en el dominio de la función con una regla. El método `funcion()` regresa una expresión *lambda*. Esta expresión es la función verdadera que recibe un `nodo` y un `vector`, y aplica la regla correspondiente a dicho vector.

```
1   class FuncionFlujo:
2       ''' Abstrae una funcion de flujo.'''
3       def __init__(self, reglas):
4           self.reglas = reglas

6       def funcion(self):
7           ''' Regresa la funcion de flujo la cual recibe un
                bloque y un vector'''
8           return lambda bloque,vector: self.reglas[bloque].
                aplica(vector)
```

Listing A.12: Implementación de la función de flujo.

Análisis iterativo

El listado A.13, muestra la función `analisisIterativo()`, la cual recibe los siguientes parámetros:

- `bloques`: el conjunto de bloques o nodos que conforman la gráfica,
- `entrada`: el nodo de entrada al procedimiento,

- `init`: el vector con la información inicial,
- `funcion`: la función de flujo correspondiente al problema que se quiere resolver,
- `reticula`: la retícula que representa al espacio del problema.

La función `analisisIterativo()` regresa un mapa con los valores de la función *in(B)* para cada bloque de la gráfica. Esta función encuentra una solución MFP para cada nodo de la gráfica.

```
1  def analisisIterativo(bloques, entrada, init, funcion,
       reticula):
2      efecto = None
3      efectoTotal = None
4      dfin = Mapa()

6      dfin[entrada] = init
7      listaDeTrabajo = bloques.copy()
8      listaDeTrabajo.discard(entrada)
9      for bloque in bloques:
10         dfin[bloque] = reticula.FONDO
11     while len(listaDeTrabajo) > 0:
12         bloque = listaDeTrabajo.pop()
13         print "Prosesando" + bloque.getLlave()
14         efectoTotal = reticula.FONDO
15         for p in bloque.predecesores:
16             efecto = funcion(p,dfin[p])
17             print p.getLlave() + ":" +  str(efecto)
18             efectoTotal = efectoTotal | efecto
19             if dfin[bloque] != efectoTotal:
20                 dfin[bloque] = efectoTotal
21                 listaDeTrabajo.add(bloque)
22     return dfin
```

Listing A.13: Función que realiza análisis iterativo sobre un conjunto de nodos.

Prueba

El listado A.14 muestra una función que prueba el análisis iterativo del listado A.13, utilizando la gráfica de flujo mostrada en la figura 2.2 y las funciones de flujo mostradas en el cuadro 3.2.

```
1  from estructuras import *
2  from sets import *
3  from reticulas import *
4  from analisis import *

6  def pruebaAnalisisIterativo():
7      # Primero creamos la grafica de flujo que vamos a
           probar

9      # Iniciamos los nodos
10     entrada = Nodo('entrada')
11     b1 = Nodo('B1')
12     b2 = Nodo('B2')
13     b3 = Nodo('B3')
14     b4 = Nodo('B4')
15     b5 = Nodo('B5')
16     b6 = Nodo('B6')
17     salida = Nodo('salida')

19     # Los agregamos al conjunto de bloques
20     bloques = Set([entrada,b1,b2,b3,b4,b5,b6,salida])

22     # Establecemos las relaciones entre ellos
23     entrada.agregaSucesor(b1)
24     b1.agregaSucesor(b2)
25     b1.agregaSucesor(b3)

27     b2.agregaSucesor(salida)

29     b3.agregaSucesor(b4)

31     b4.agregaSucesor(b6)
32     b4.agregaSucesor(b5)

34     b5.agregaSucesor(salida)
35     b6.agregaSucesor(b4)

37     # Ahora Iniciamos la funcion de flujo
38     mapaFuncion = Mapa()
39     mapaFuncion[entrada] = Regla('x','x','x','x','x','x','x
           ','x')
40     mapaFuncion[b1] = Regla(1,1,1,'x','x',0,0,'x')
41     mapaFuncion[b2] = Regla('x','x','x','x','x','x','x','x'
           )
```

```
42      mapaFuncion[b3] = Regla('x','x','x',1,'x','x','x',0)
43      mapaFuncion[b4] = Regla('x','x','x','x','x','x','x','x'
            )
44      mapaFuncion[b5] = Regla('x','x','x','x','x','x','x','x'
            )
45      mapaFuncion[b6] = Regla('x',0,0,0,1,1,1,1)

47      inicial = VectorBits((0,0,0,0,0,0,0,0))

49      #Inicamos la reticula de 8 bits y la funcion
50      reticula = ReticulaBits(8)
51      ff = FuncionFlujo(mapaFuncion)

53      #Por ultimo ejecutamos el metodo
54      resultados = analisisIterativo(bloques,entrada,inicial
            ,ff.funcion(),reticula)

56      print 'Los resultados son: ' + str(resultados)
```

Listing A.14: Prueba del análisis iterativo.

Después de correr esta prueba se obtienen los siguientes resultados:

```
1 salida:<1,1,1,1,1,1,1,1>
2 B4:<1,1,1,1,1,1,1,1>
3 B5:<1,1,1,1,1,1,1,1>
4 B6:<1,1,1,1,1,1,1,1>
5 B1:<0,0,0,0,0,0,0,0>
6 B2:<1,1,1,0,0,0,0,0>
7 B3:<1,1,1,0,0,0,0,0>
8 entrada:<0,0,0,0,0,0,0,0>
```

Estos resultados son congruentes con los resultados obtenidos en el capítulo 3, ver el cuadro 3.3.

Bibliografía

[1] **Steven S. Muchnick.** *Advanced Compiler Design and Implementation*. San Francisco: Morgan Kaufmann, 1997.

[2] **Keith D. Cooper & Linda Torczon.** *Engineering a Compiler.* Houston: Rice University, 2000.

[3] **José Galaviz Casas.** *Arquitectura de Computadoras.* México, D.F.: Facultad de Ciencias, UNAM, 2002.

[4] **Ken Kennedy.** *A Survey of Data Flow Analysis Techniques.* Yorktown Heights: IBM Thomas J. Watson Research Center, 1977.

[5] **Christian Lengauer, Sergei Gorlatch & Christoph Herrmann.** *The Static Parallelization of Loops and Recursions.* The Journal of Supercomputing archive, Volumen 11, Número 4, 333-353, 1997.

[6] **Kathleen Knobe & Vivek Sarkar.** *Array SSA Form and its Use in Parallelization.* Annual Symposium on Principles of Programming Languages archive, 107-120, 1998.

[7] **Jeffrey T. Oplinger, David L. Heine & Monica S. Lam** *In Search of Speculative Thread-Level Parallelism.* IEEE, 1999.

[8] **Manohar K. Prabhu & Kunle Olukotun.** *Exposing speculative thread parallelism in SPEC2000.* Principles and Practice of Parallel Programming, 142-152, 2005.

[9] **Kunle Olukotun, Lance Hammond & Mark Willey.** *Improving the Performance of Speculatively Parallel Applications on the Hydra CMP.* Standford Univeresity, 1999.

[10] **Pradip Bose.** *Computer Architecture research: Shifting proryties and newer challenges.* IEEE, 2004.

[11] **David Geer.** *Chip Makers Turn to Multicore Processors.* IEEE, 2005.

[12] **Elisa Viso Gurovich.** *Algoritmos genéricos: herencia y polimorfismo en el análisis de algoritmos.* Tesis de Doctorado, UNAM, 2005.

[13] **Wikipedia Contributors.** *Supercomputer.* Wikipedia: The Free Encyclopedia, http://en.wikipedia.org/wiki/supercomputer, 20 de agosto de 2005.

[14] **Wikipedia Contributors.** *Computer Cluster.* Wikipedia: The Free Encyclopedia, http://en.wikipedia.org/wiki/Computer_cluster, 20 de agosto de 2005.