El optimizador de bucles del compilador Open64/ORC

Eduard Santamaria Marta Jiménez Agustín Fernández
Josep Ma. LLabería

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

14 de diciembre de 2004

Índice general

Capítulo 1

Introducción

1.1. Introducción

Open64 y *ORC* (Open Research Compiler) son dos iniciativas de código abierto basadas en el compilador *SGI Pro64*. Open64 está gestionada por miembros de la Universidad de Delaware, y ORC es una extensión del compilador desarrollada por Intel y la Chinese Academy of Science. Para más información consultar las respectivas páginas web [2] y [1].

SGI Pro64 es un conjunto de compiladores optimizadores desarrollados por SGI. Incluye compiladores de C, C++ y Fortran90/95 que siguen los estándares ABI y API de Linux IA-64. Los archivos fuente son de dominio público y se distribuyen bajo los términos de la GNU General Public License. El conjunto de compiladores está disponible para correr sobre plataformas Linux IA-32 e IA-64.

En el technical report "Introducción al compilador Open64/ORC"[14] se describen los componentes y el lenguaje de representación intermedia (WHIRL) utilizados por el compilador. En este documento nos centramos en uno de los componentes del compilador: el optimizador de bucles o LNO (Loop Nest Optimizer). Las principales tareas llevadas a cabo durante la optimización de bucles incluyen (ver figura 1.1):

1. Construcción de anidaciones de bucles casi perfectas, denominadas SNLNs, utilizando las transformaciones fusión y distribución (Etapa 1 - Generación SNLNs).

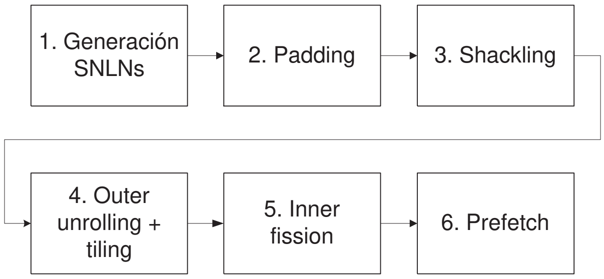
2. Se aumenta el tamaño de algunas estructuras de datos para conseguir una alineación en memoria que reduzca los fallos por conflicto en los accesos a la cache (Etapa 2 - Padding).

3. Transformación del código mediante shackling [11] para mejorar los accesos a la cache de segundo nivel (Etapa 3 - Shackling).

4. Aplicación de tiling, intercambio y outer loop unrolling en los SNLNs internos para mejorar la localidad de los accesos a registros, y a los distintos niveles de cache (Etapa 4 - Outer unrolling + tiling).

5. Distribución de los bucles más internos para reducir la presión sobre los registros (Etapa 5 - Inner fission).

Figura 1.1: Etapas del optimizador de bucles



6. Optimización de los accesos a memoria mediante prebúsqueda de datos (Etapa 6 - Prefetch).

Las transformaciones del optimizador de bucles actúan sobre la representación en WHIRL [14]. Para llevarlas a cabo es necesario conocer una serie de información que se mantiene en distintas estructuras de datos: el gestor de definiciones y usos, el gestor de alias, el gestor de reducciones, los access arrays y los grafos de dependencia de datos. Parte de estas estructuras de datos se reciben desde la etapa de compilación anterior (llevada a cabo por el optimizador global), otras se calculan al inicio de la ejecución del optimizador de bucles. Éste es el caso de los access arrays, el gestor de reducciones y el grafo de dependencias entre arrays.

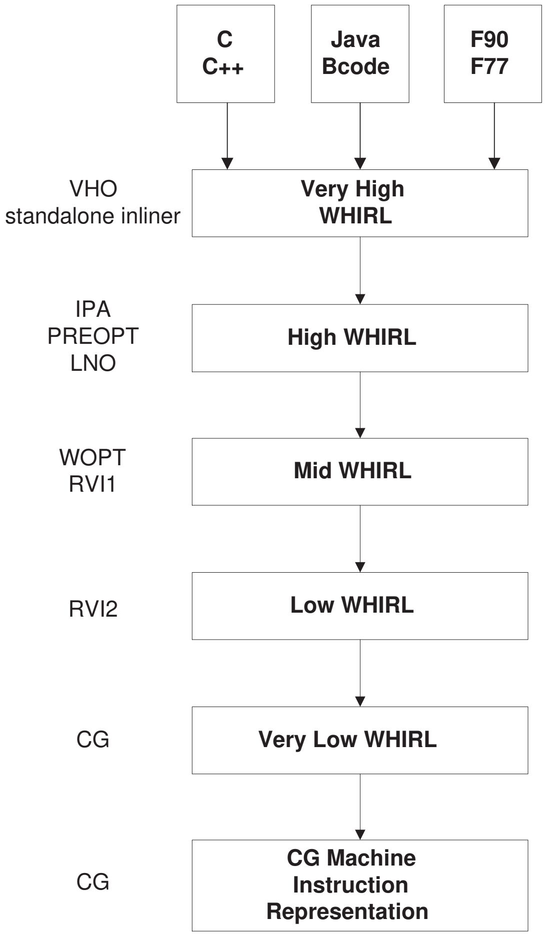
La figura 1.2 muestra el proceso de compilación como una transformación progresiva de la representación intermedia hasta llegar a la generación de código. Nos permite ver (a mano izquierda) como se suceden las distintas etapas de la compilación. Las siglas hacen referencia a transformaciones de muy alto nivel (VHO), análisis interprocedural (IPA), ejecución preliminar del optimizador global (PREOPT), optimizador de bucles (LNO), optimizador global (WOPT), identificación de variables de registro (RVI) y generación de código (CG). La ejecución de estas etapas depende de las opciones de compilación, pero antes de llamar al optimizador de bucles siempre se ejecuta la etapa de PREOPT. Durante esta fase el optimizador global lleva a cabo, entre otras, transformaciones como la eliminación de variables de inducción, la propagación de copias, la eliminación de *dead code* y la normalización de bucles.

1.2. Estructura del documento

Este documento detalla las etapas 1 (generación de SNLNs) y 3 (shackling) del optimizador de bucles. Para validar la legalidad de algunas transformaciones realizadas durante estas y otras etapas el compilador utiliza grafos de dependencias. La estructura y el proceso de construcción de los grafos de dependencias se describen en el capítulo 2. También en este capítulo se presentan la estructura de datos access array, utilizada para representar las referencias a variables de tipo array, y los tests que determinan si puede existir dependencia entre dos accesos a memoria.

Los Singly Nested Loop Nests (SNLNs) son las anidaciones de bucles que la etapa de LNO podrá transformar. En el capítulo 3 se definen sus características

Figura 1.2: Etapas de la compilación



y se detalla como se analizan y transforman las anidaciones de bucles para obtener SNLNs de profundidad máxima.

Por último, el capítulo 4 describe la aplicación de shackling, una optimización que intenta bloquear las estructuras de datos para mejorar la localidad en los accesos a la cache de segundo nivel.

Los contenidos de este documento se ampliarán en otro technical report donde describiremos con detalle la etapa 4.

Capítulo 2

Grafos de dependencias de datos

Los grafos de dependencias de datos describen las relaciones de dependencia entre operaciones de acceso a memoria dentro de una unidad de programa (función). Estas dependencias se producen cuando dos operaciones acceden al mismo dato y condicionan las transformaciones que se van a poder realizar.

El compilador utiliza tres tipos distintos de grafos de dependencias de datos:

1. Grafo de dependencias entre arrays, que almacena las dependencias entre accesos a elementos de tipo array.

2. Grafo de niveles de dependencia, que se utiliza durante la distribución y fusión de bucles.

3. Grafo de dependencias para generación de código.

En este capítulo se tratan los grafos de dependencias entre arrays y los grafos de niveles de dependencia, que son los que se utilizan durante la etapa de LNO.

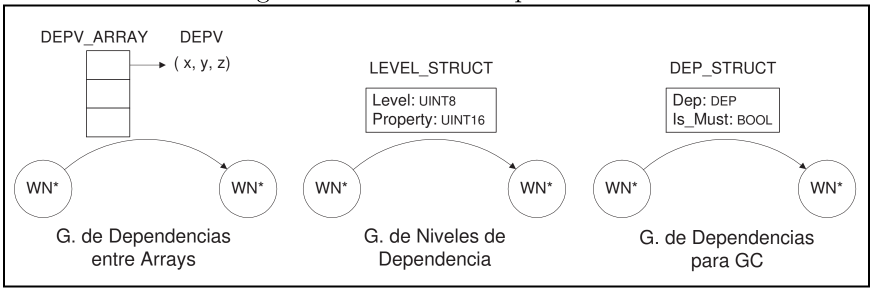
En la sección 2.1 se muestra, en primer lugar, como son las estructuras de datos que se utilizan para almacenar estos grafos. A continuación se describen el grafo de dependencias entre arrays y el grafo de niveles de dependencia, se presentan ejemplos y se muestra la relación entre ambos tipos de grafo.

La sección 2.2 describe como se construye el grafo de dependencias entre arrays. Dentro de esta sección encontramos tres grandes apartados:

2.2.1 En esta sección se describe la estructura de datos access array, utilizada para representar los accesos a variables de tipo array. Es importante porque el cálculo de las dependencias se realiza utilizando la información contenida en los access arrays.

2.2.2 En esta sección se explica como se construye el grafo de dependencias y se describen los tests que se aplican para determinar si puede haber dependencia entre dos accesos. Si los tests no pueden garantizar que los accesos son independientes se crea un sistema de ecuaciones que expresa las condiciones que deben producirse para que exista dependencia. A este sistema

Figura 2.1: Grafos de dependencias



se añaden también ecuaciones que representan los límites de las variables de iteración. Si podemos demostrar que el sistema es inconsistente se garantiza la independencia de los dos accesos.

2.2.3 En esta sección se describe como se analiza la consistencia del sistema de ecuaciones creado en 2.2.2.

2.1. Descripción de los grafos

Los distintos tipos de grafo se diferencian por la granularidad de sus nodos y la información contenida en las aristas. En todos los casos, los nodos del grafo contienen un apuntador a una operación de la representación en WHIRL. A su vez, las operaciones de la representación intermedia también tienen asociado un apuntador que les va a permitir acceder al nodo correspondiente del grafo de dependencias. En el grafo de dependencias entre arrays, los nodos apuntan a operaciones de load y store sobre arrays, mientras que en el grafo de niveles de dependencia, los nodos apuntan a sentencias, que son elementos de mayor granularidad.

Respecto a las aristas, éstas van a contener vectores de dependencia en el grafo de dependencias entre arrays o una etiqueta indicando el nivel de la dependencia en el caso del grafo de niveles de dependencia.

La figura 2.1 muestra de forma esquemática como se guardan los grafos. La estructura de datos que se utiliza es la clase ARRAY_DIRECTED_GRAPH16, que se declara en /be/com/dep_graph.h. El tipo de dato que se utiliza para declarar las aristas es una *union* de tres tipos distintos, que son *DEPV_ARRAY* para el grafo de dependencias entre arrays, *LEVEL_STRUCT* para el de niveles de dependencia y *DEP_STRUCT* para el utilizado en generación de código.

La clase DEPV_ARRAY contiene una lista de vectores de dependencia, cada uno de ellos de tipo DEPV. Cada dimensión de un vector de dependencia guarda un elemento de tipo DEP, que es un entero de dieciséis bits organizado de la siguiente forma:

bit 15: indica si la distancia es constante.

bits 12-14: señalan la dirección de la dependencia.

bits 0-11: indica la distancia, en caso que ésta sea constante, o un límite en caso contrario.

En el tipo *LEVEL_STRUCT*, el campo *Level* indica cuál es el bucle más interno que acarrea una dependencia entre dos sentencias. Aparentemente el campo *Property* no se utiliza.

2.1.1. Grafo de dependencias entre arrays

El grafo de dependencias entre arrays se crea al inicio de la etapa de optimización de bucles y se utiliza para validar la legalidad de distintas transformaciones. Se crea un grafo por función, que se genera una sola vez y se actualiza de forma incremental cuando se modifica la representación intermedia del programa.

Los nodos del grafo de dependencias entre arrays representan operaciones de acceso a un array (load, store). Las aristas reflejan las relaciones de dependencia entre estas operaciones y cada una tiene asociados uno o más vectores de dependencia. Dadas dos referencias a memoria dependientes entre sí, los vectores de dependencia asociados a la arista tendrán una dimensión para cada uno de los bucles que contengan ambas referencias. Cada dimensión del vector de dependencia contendrá una distancia indicando el número de iteraciones, del bucle correspondiente a esa dimensión, que separa los accesos a memoria dependientes entre sí. Si la dependencia se produce dentro de una misma iteración la distancia para ese bucle será 0. Si no es posible calcular esta distancia se indicará la dirección de la dependencia.

Sean Ref_o y Ref_d dos referencias dependientes, cuya dependencia se representa mediante una arista con origen en Ref_o y destino en Ref_d, y sea d_L la dimensión correspondiente al bucle L de uno de sus vectores de dependencia. Si d_L expresa una dirección podrá tomar uno de los siguientes valores:

(=) Ref_d accede al mismo dato que Ref_o en la misma iteración del bucle L. Equivale a distancia 0.

(<) Ref_d accede al mismo dato que Ref_o en alguna iteración posterior del bucle L. El comparador $<$ indica que el valor de la variable de iteración en el acceso de Ref_o es inferior a su valor en el acceso de Ref_d.

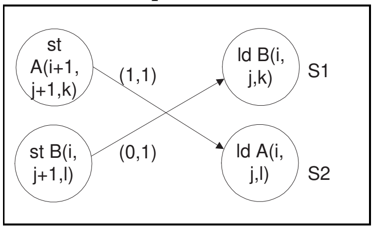
(>) Ref_d accede al mismo dato que Ref_o en alguna iteración anterior del bucle L.

($*$) No es posible determinar la dirección de la dependencia.

A veces se utilizan los símbolos $(+)$ en lugar de $(<)$ y $(-)$ en lugar de $(>)$. Ambas notaciones son equivalentes. La estructura de datos que almacena los vectores de dependencia también soporta una notación resumida y acepta los siguientes valores adicionales: $(\leq)$, $(\geq)$ y por último $(><)$, que indica dependencia en iteraciones distintas sin especificar dirección. Esta notación resumida no se utiliza en el grafo de dependencias, donde se añaden tantos vectores como sean necesarios para cada dependencia.

Veamos un ejemplo con el siguiente fragmento de código:

Figura 2.2: Grafo de dependencias entre arrays (ej.1)




```
do i
    do j
        do k
S1        A(i+1,j+1,k) = B(i,j,k)
        enddo
        do l
S2        B(i,j+1,l) = A(i,j,l)
        enddo
    enddo
enddo
```

Este código contiene las siguientes dependencias:

- El valor generado en S1 ($A(i+1, j+1, k)$) se utiliza en S2 en la siguiente iteración del bucle I y en una iteración posterior del bucle J.
- El valor generado en S2 ($B(i, j+1, l)$) se utiliza en S1 en la siguiente iteración del bucle J.

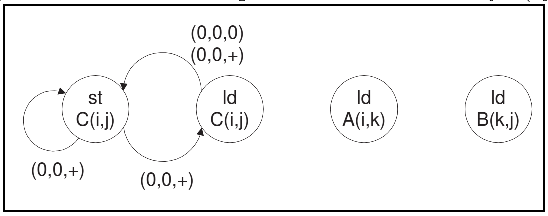
Estas dependencias se reflejan en el grafo de la figura 2.2. El grafo contiene un nodo para cada uno de los accesos a un array, las aristas muestran qué operaciones son dependientes entre sí y los vectores de dependencia indican la distancia en iteraciones, para cada uno de los bucles, que separa las operaciones dependientes.

La figura 2.3 muestra el grafo generado para el siguiente código de multiplicación de matrices:

```
do j
  do i
    do k
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    enddo
  enddo
enddo
```

Observamos que, a diferencia del caso anterior, en este ejemplo aparecen vectores de dependencia donde alguno de sus elementos es una dirección. Es el caso, por ejemplo, de la dependencia que aparece por la ejecución del store sobre C(i,j) en sucesivas iteraciones del bucle K. Podemos ver también como la arista entre el load y el store de C(i,j) tiene asociados dos vectores de dependencia, uno de ellos reflejando la dependencia dentro de la misma iteración de K y otro para la dependencia entre iteraciones diferentes.

Figura 2.3: Grafo de dependencias entre arrays (ej.2)



Durante una sesión de debug, es posible ver los grafos de dependencia en un determinado punto llamando al método *Print* de la clase utilizada para almacenarlos, que es ARRAY_DIRECTED_GRAPH16. La descripción del grafo de la figura 2.3 generada por el compilador es la siguiente:

```
Printing an ARRAY_DIRECTED_GRAPH16 of type DEPV_ARRAY
Vertex 1 for store into Wn = C[-1 + 1*loop_var0 ][-1 + 1*loop_var1 ]
Edge 2 to vertex 2  has DEPV_ARRAY =  (   0  0  +  )
Edge 1 to vertex 1  has DEPV_ARRAY =  (   0  0  +  )
Vertex 2 for load from Wn = C[-1 + 1*loop_var0 ][-1 + 1*loop_var1 ]
Edge 3 to vertex 1  has DEPV_ARRAY =  (   0  0  0  )  (   0  0  +  )
Vertex 3 for load from Wn = A[-1 + 1*loop_var2 ][-1 + 1*loop_var1 ]
Vertex 4 for load from Wn = B[-1 + 1*loop_var0 ][-1 + 1*loop_var2 ]
```

2.1.2. Grafo de niveles de dependencia

Los grafos de niveles de dependencia se generan utilizando información del grafo de dependencias entre arrays. El coste de su creación es mucho menor y por este motivo pueden ser creados cada vez que se necesitan. La información contenida en un grafo de niveles de dependencia no abarca todo el cuerpo de una función, sino que se restringe a una determinada anidación de bucles.

Las siguientes características diferencian el grafo de niveles de dependencia del grafo de dependencias entre arrays:

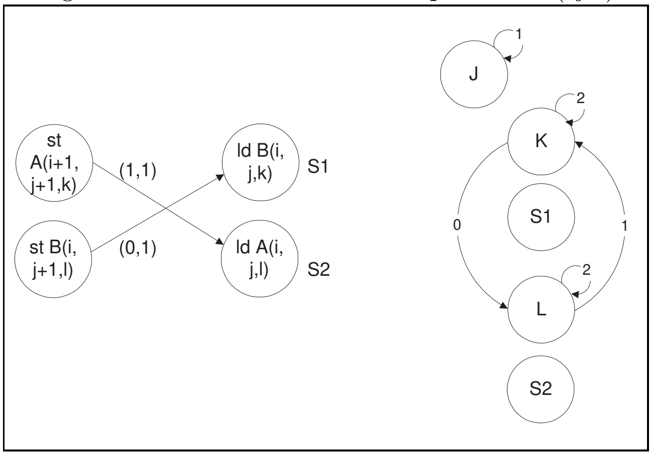
- Los nodos representan sentencias, se trata por tanto de elementos de mayor granularidad.
- Sólo aparecen aristas entre sentencias ubicadas en el mismo nivel de profundidad.
- La información asociada a una arista indica el nivel de la dependencia.
- Contiene dependencias entre escalares.

Podemos definir el nivel de dependencia como:

> El nivel de profundidad del bucle más externo que acarrea una dependencia entre dos sentencias. Para ello se numeran todos los bucles de la anidación asignando el nivel 0 al bucle más externo.

Para mostrar un ejemplo de grafo de niveles de dependencia volveremos al código correspondiente a la figura 2.2. La figura 2.4 incluye también el grafo de

Figura 2.4: Grafo de niveles de dependencia (ej.1)



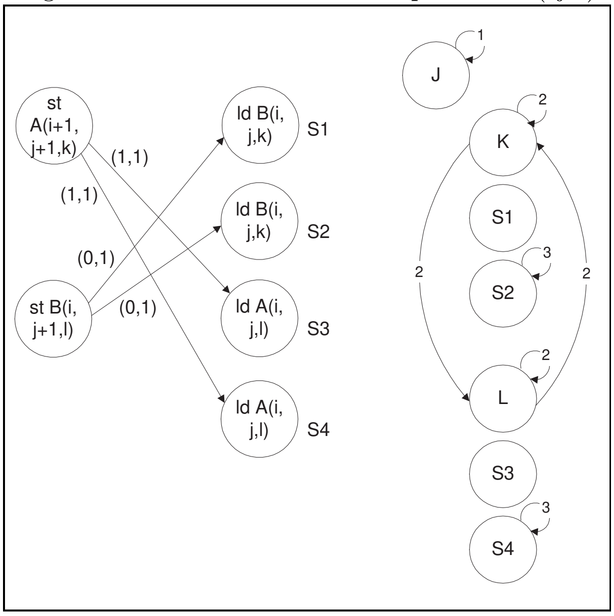
dependencias entre arrays para poder observar la relación entre ambos grafos. Podemos ver como cada sentencia se representa mediante un solo nodo. Vemos también que el origen y destino de las aristas se desplaza hasta encontrar dos sentencias, en este caso los bucles K y L, ubicadas en el mismo nivel. La dependencia detectada entre la escritura de A en S1 y su posterior lectura en S2 es de nivel 0, puesto que es el bucle I el que acarrea la dependencia. Esta dependencia se refleja mediante una arista desde el bucle K al bucle L. La dependencia entre la escritura y la lectura de B es acarreada por el bucle J, siendo su nivel de dependencia 1. Las dependencias que tienen como origen y destino un mismo nodo que representa un bucle, se deben a las variables de iteración.

Al calcular las dependencias entre escalares el compilador comprueba si los escalares participan en reducciones donde sea posible aplicar asociatividad. Si es así no se añaden dependencias al grafo. Durante la generación de SNLs se asume que ningún escalar es privatizable. En etapas posteriores (*Fiss_Gather_Loop*, *Inner_Fission* y *Vintrinsic_Fision_Phase*), cuando un escalar se puede privatizar sólo se incluyen las dependencias cuya sentencia de origen precede a la de destino en orden de programa.

A continuación modificamos el código añadiendo operaciones entre escalares para ver como se incorporan al grafo sus dependencias.

```
DO I
    DO J
        DO K
S1        A(i+1,j+1,k) = B(i,j,k)
S2        x = x + B(i,j,k)
        ENDDO
        DO L
S3        B(i,j+1,l) = A(i,j,l)
S4        x  = x * A(i,j,l)
        ENDDO
    ENDDO
ENDDO
```

Figura 2.5: Grafo de niveles de dependencia (ej.2)



Añadimos al ejemplo las sentencias S2 y S4, escogiendo operadores de modo que no se pueda aplicar asociatividad, ya que en ese caso estas sentencias se considerarían independientes. El grafo correspondiente a este código se muestra en la figura 2.5. En este segundo grafo, los niveles de dependencia han cambiado, ya que se incorporan las dependencias producidas por los accesos a la variable x. Estas dependencias, sin embargo, no son visibles en el grafo de dependencias entre arrays.

2.2. Cálculo del grafo de dependencias entre arrays

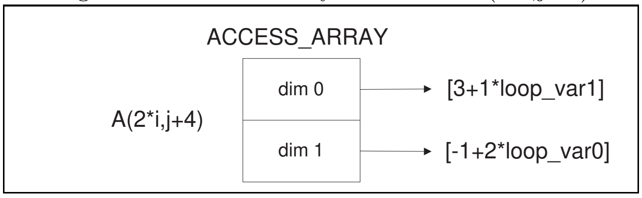
Entre las primeras tareas llevadas a cabo por el optimizador de bucles se encuentran la inicialización de las estructuras de datos que representan las expresiones utilizadas en los accesos a arrays y la construcción del grafo de dependencias.

A continuación mostraremos, en primer lugar, como es la estructura de datos que se utiliza para representar los accesos a array (sección 2.2.1). La información guardada mediante esta representación se utilizará posteriormente en el cálculo de las dependencias.

En la sección 2.2.2 comentaremos como se construye el grafo de dependencias entre arrays y detallaremos los distintos tests que se aplican para decidir si puede haber dependencia entre dos referencias. Si los distintos tests no son determinantes se genera un sistema de ecuaciones que describe las condiciones necesarias para que exista dependencia.

La sección 2.2.3 describe el método utilizado para determinar la consistencia del sistema de ecuaciones generado en el apartado anterior.

Figura 2.6: Access array del acceso A(2*i,j+4)



2.2.1. Representación de los accesos a arrays

El tipo de dato utilizado para representar los accesos a arrays es la clase ACCESS_ARRAY (be/lno/access_vector.h). Mediante el mecanismo de mapeo[1], se asocia a cada nodo de la representación intermedia que represente un acceso a un array un elemento de la clase ACCESS_ARRAY. Un ACCESS_ARRAY es una lista de ACCESS_VECTORs donde cada ACCESS_VECTOR representa el polinomio correspondiente a una de las dimensiones del acceso a array.

En la figura 2.6 vemos un ejemplo de access array, que representa un acceso a una matriz de dos dimensiones de un programa en Fortran. En el ejemplo *loop_var0* hace referencia al índice *i*, y *loop_var1* al índice *j*. A nivel interno el compilador trabaja con almacenamiento por filas, por este motivo se invierte el orden de las dimensiones que conforman el acceso. Además se resta una unidad a cada una de las dimensiones para que el primer elemento de la matriz sea el (0,0), ya que en Fortran el primer elemento de un vector se encuentra en la posición 1.

Para representar las expresiones utilizadas para indexar un acceso a un array se utiliza la clase ACCESS_VECTOR. Un ACCESS_VECTOR contiene:

- Un atributo *offset* que almacena el término independiente del polinomio.
- Lista de coeficientes: contiene los factores que acompañan a los índices de los bucles en los términos lineales.
- Lista de términos lineales: los términos lineales cuya variable no es un índice de un bucle se guardan en esta lista. Cada elemento de la lista está compuesto por un coeficiente y un símbolo.
- Lista de términos no lineales: los términos donde aparezca un producto de varios símbolos se almacenan en esta lista. Cada elemento está formado por un coeficiente y una lista de símbolos.

Por ejemplo, dada la expresión $2 * i + k + j * k + 5$, contenida en los bucles i, j y siendo k una variable cuyo valor se desconoce en tiempo de compilación, el contenido del access vector correspondiente seria:

- *offset*=5 .

[1]mecanismo de mapeo: cada nodo whirl contiene un identificador que se utiliza como un índice a una tabla donde cada columna representa un atributo del nodo. Añadiendo columnas a esta tabla podemos extender la información sobre un nodo. Cuando estos datos ya no son necesarios la columna se puede liberar.

- Lista de coeficientes: {2,0} (2 es el coeficiente de i, y 0 el de j)
- Lista de términos lineales: {[1,k]}
- Lista de términos no lineales: {[1,{k,j}]}

Los ACCESS_VECTORs no permiten representar cualquier expresión. Por ejemplo, no pueden representar términos que contengan potencias. Cuando nos encontramos con expresiones que no podemos representar, se marca el atributo *Too_Messy* como cierto y se considera el valor del resto de campos de la estructura como indeterminado.

Deslinearización

Las expresiones que contienen términos no lineales no pueden ser utilizadas en el cálculo de dependencias[2]. Por este motivo se intenta eliminar los términos no lineales aplicando deslinearización. Este proceso se lleva a cabo cada vez que se genera un access array. Veremos en qué consiste mediante un ejemplo, supongamos que encontramos la siguiente expresión en el access vector A_r, que corresponde a la dimensión r del access array A:

$$[-1 + 2 * i + k + j * k]$$

Siendo i, j índices de bucle y k un símbolo no índice, procederemos del siguiente modo para deslinearizar A_r:

1. En primer lugar localizaremos dentro de A_r un símbolo no índice que aparezca en todos los términos no lineales de A_r. En el ejemplo encontramos la k.

2. A continuación debemos comprobar que dicho símbolo cumple la siguiente condición:

   $$0 <= tmp < k,$$

   donde tmp está compuesto por todos los términos de A_r eliminando aquellos en los que aparece k. Es decir,

   $$0 <= (-1 + 2 * i) < k,$$

3. Finalmente modificamos A creando una nueva dimensión. Desplazamos todos los A_s tales que $s > r$ una posición a la derecha. En A_r dejamos todos los términos donde aparece k sacando factor común, e inicializamos el atributo *Delinearized_Symbol* indicando que el símbolo deslinearizado es k:

   $$A_r = 1 + j; \; Delinearized\_Symbol = k$$

   En A_{r+1} guardamos todos los términos donde no aparece k.

[2]Salvo cuando se trata de términos no lineales formados por símbolos invariantes dentro de los bucles que se están considerando y, además, estos términos son idénticos en ambas referencias, de modo que se anulan en las ecuaciones generadas para el cálculo de dependencias.

Figura 2.7: Non_Const_Loops (ejemplo 1)

```
L1: do i
L2:    do j
          k = ...
          A( 2*k, j )
```

$\text{Non\_Const\_Loops}(A_1) = 2$

Figura 2.8: Non_Const_Loops (ejemplo 2)

```
L1: do i
L2:    do j
          A( i*r, j )
```

$\text{Non\_Const\_Loops}(A_1) = 1$

$$A_{r+1} = -1 + 2 * i$$

Después de este proceso el resultado final es:

$$[1 + j][-1 + 2 * i]$$

Se está creando una nueva dimensión de k elementos, y observamos que la condición del punto 2 garantiza que A_{r+1} no se sale de rango al seleccionar un elemento de A_r. Este proceso se puede aplicar repetidamente para intentar eliminar el mayor número de términos no lineales posible.

El atributo Non_Const_Loops

El atributo Non_Const_Loops se calcula después de haber aplicado deslinearización. El valor de este atributo indica en qué bucles se modifican los símbolos no invariantes que puedan aparecer dentro de un access array. Esta información se debe tener en cuenta durante el cálculo de dependencias.

Si algún término de un access vector contiene un símbolo que no es invariante dentro de los bucles que contienen el acceso, el atributo Non_Const_Loops indica cual es el bucle más interno donde se modifica el valor de la variable. En el ejemplo de la figura 2.7 el valor de k se modifica dentro de L2, por tanto el valor del atributo Non_Const_Loops del access vector donde aparece k es igual a 2. Si k no se modificara en ninguno de los bucles Non_Const_Loops valdría 0.

Adicionalmente, si en alguno de los términos de un access vector aparece una variable de iteración dentro de un término no lineal, Non_Const_Loops indica a qué bucle corresponde dicha variable de iteración. Podemos verlo en el ejemplo de la figura 2.8.

Cuando en un access vector hay más de un símbolo que cumple alguna de las condiciones anteriores Non_Const_Loops toma el valor máximo (figura 2.9). Un access vector con el valor de Non_Const_Loops distinto de 0 no podrá ser utilizado para generar ecuaciones cuando el cálculo de dependencias incluya bucles marcados como Non_Const_Loops.

Figura 2.9: Non_Const_Loops (ejemplo 3)

```
L1: do i
L2:    do j                    Non_Const_Loops(A1) = 2
          k = ...
          A( i*r+2*k, j )
```

2.2.2. Cálculo de dependencias

El primer paso para la construcción del grafo de dependencias entre arrays consiste en recorrer la representación intermedia del programa y localizar las distintas anidaciones de bucles (ARRAY_DIRECTED_GRAPH16::Find_Region() definida en be/com/dep_graph.cxx).
Una vez encontrado un bucle válido se recorre el cuerpo del bucle y se generan unas listas, de modo similar a como se describe en la sección 3.3.3, con todas las referencias a memoria y llamadas que éste contiene. Al mismo tiempo se añaden en el grafo los vértices correspondientes. Finalmente se van tomando por parejas las referencias encontradas y se analizan para determinar si puede haber dependencia. Esta tarea la realizan las funciones Build_Region() y Add_Edge() de la clase ARRAY_DIRECTED_GRAPH16 (be/com/dep_graph.cxx).

El número de dimensiones del vector de dependencias (*dv_dim*), se calcula como el número de bucles que las dos referencias tienen en común menos el número de bucles externos no válidos. Un bucle puede ser no válido si contiene gotos/returns que abandonan el bucle o llamadas sin información del análisis interprocedural (ver definición de DO_LOOP_INFO en be/lno/lnopt_main.h).

Para que pueda haber dependencia entre dos accesos a un array hay que determinar si es posible que estos dos accesos hagan referencia al mismo elemento. En el ejemplo

```
DO I=1,10
  X(I) = X(I+1)
ENDDO
```

para determinar si hay dependencia entre la lectura de X(I+1) y la escritura de X(I) hay que resolver la siguiente ecuación:

$$I_1 = I_2 + 1$$

Puesto que el valor de I_1 y I_2 viene delimitado por los límites del bucle se añaden las siguientes ecuaciones:

$$1 \leq I_1 \leq 10$$
$$1 \leq I_2 \leq 10$$

En un caso tan simple como este el compilador va a ser capaz de devolver un vector de dependencias con la distancia exacta de esta anti-dependencia. En el caso general pueden aparecer sistemas de ecuaciones más complejos. Si el compilador no es capaz de probar que el sistema es inconsistente habrá que suponer dependencia.

Una vez calculados los vectores de dependencia entre dos referencias, éstos se clasifican en dos listas. La primera de ellas contiene los que expresan dependencias positivas, aquellos cuyo primer elemento distinto de '=' es '<'. La otra lista contiene las dependencias negativas. Antes de añadir un elemento a esta segunda lista se invierte el signo de cada una de sus dimensiones[3]. Una vez hecha esta descomposición, si la lista de dependencias positivas no está vacía, se crea una arista en el grafo de dependencias entre la primera y la segunda referencia. Si la lista de dependencias negativas no está vacía se crea otra arista, pero de sentido contrario.

Dependencia entre accesos a array

La figura 2.10 muestra cuáles son los pasos seguidos para determinar si dos accesos a array pueden ser dependientes (DEPV_COMPUTE::Compute() en be/com/dep.cxx). Podríamos clasificar estos pasos en tres grupos:

1. En primer lugar encontramos *Base_Test* y *delin_ok?*, que verifican que las dos referencias cumplen unos requisitos mínimos para intentar realizar el cálculo de dependencias.
2. En segundo lugar encontramos las funciones *Same_Monotonic*, *Simple_Gcd* y la comprobación $num\_bad\_outer \geq dv\_dim$, que tratan algunos casos particulares ante la presencia de términos no lineales o términos que contienen variables cuyo valor se modifica dentro de los bucles.
3. Por último, *Trivial_Test* selecciona los términos válidos para generar ecuaciones y las funciones *Find_Init_Distance* y *Compute_Dep_Vectors* llevan a cabo el cálculo de dependencias.

A continuación explicamos con detalle cada uno de estos tests. Todas las funciones referenciadas en el resto de apartados de la sección 2.2.2 se definen en el archivo be/lno/dep.cxx.

Base_Test

La función Base_Test() comprueba si es posible que haya solapamiento entre las estructuras de datos accedidas por las dos referencias (ver figura 2.11). Si las dos referencias acceden a variables disjuntas en memoria los accesos son independientes. Si hay un solapamiento parcial se asume dependencia. Si el solape es total se está accediendo a los mismos datos y hace falta comprobar que ambas referencias tengan la misma visión de éstos y que, por tanto, el análisis de dependencias, basándonos en las expresiones que indexan estos accesos, sea factible. Para ello el compilador comprueba que ambos arrays tengan el mismo número de dimensiones y que el número de elementos en cada dimensión sea el mismo en ambos vectores.

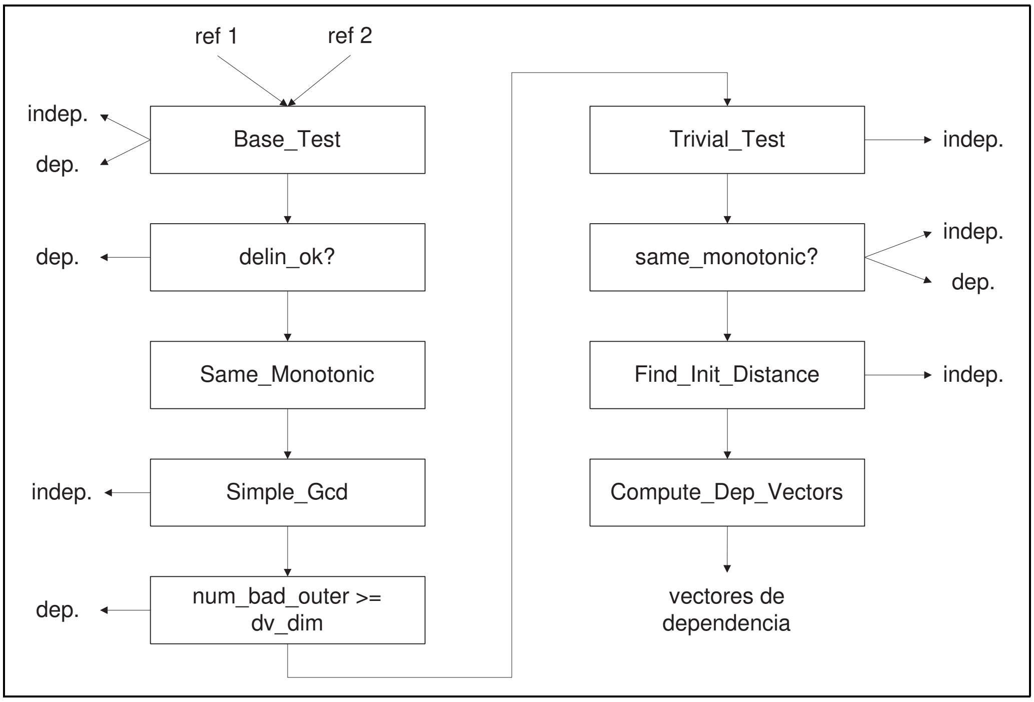
delin_ok?

Comprueba que si alguna dimensión de A o B ha sido deslinearizada, esta deslinearización se haya hecho de igual manera en ambas referencias. Si no es así se asume dependencia.

Dados dos access arrays A y B, $delin\_ok = CIERTO$ si y sólo si

[3]No es posible que existan dependencias con signo negativo en la primera dimensión del vector de dependencia. Una dependencia de flujo (o antidependencia) cuyo vector de dependencia no es posible es equivalente a una antidependencia (o dependencia de flujo) cuyo vector de dependencia sí se pueda dar. Ver [4].

Figura 2.10: Cálculo de dependencias



- Número de dimensiones de A = Número de dimensiones de B y
- $\forall i$ $Delinearized\_Symol(A_i) = Delinearized\_Symol(B_i)$

Si no se cumple alguna de las condiciones anteriores, las dos referencias se consideran dependientes.

Same_Monotonic

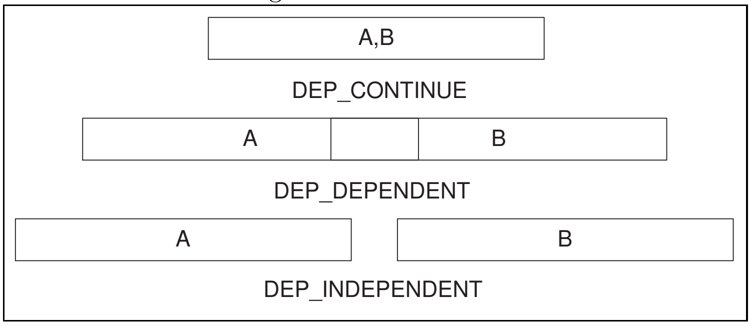
Dados dos access arrays A y B a los que no se ha aplicado deslinearización, se busca una dimensión i tal que:

- $A_i = B_i$,
- $common\_nest - Non\_Const\_Loops(A_i) < dv\_dim$,
  donde $common\_nest$ es el número de bucles que A y B tienen en común y dv_dim es el número de dimensiones del vector de dependencia, y
- todos los términos de A_i, B_i son lineales y no dependen de variables de iteración.

Estas condiciones implican que A_i y B_i contienen algún símbolo cuyo valor se modifica dentro de los bucles que se van a considerar en el cálculo de dependencias. Si estos A_i, B_i cumplen además que:

- Dado un término cualquiera de A_i, B_i, si contiene una variable k, la definición de k es única y es la misma para A y para B.

Figura 2.11: Base_Test



- Esta definición está ubicada en el bucle más interno de los bucles que A y B tienen en común.
- La definición de k sigue el patrón $k = k + ct$, siendo ct un valor literal, y al uso de k en esta asignación sólo le llega esta misma definición o una definición externa a los bucles que se utilizarán en el cálculo de dependencias.
- Todos los símbolos que aparecen en los términos de la dimensión i se actualizan utilizando el mismo operador (que puede ser suma o resta).

Entonces el test de igual monotonicidad para las referencias A y B devuelve cierto.

Si encontramos dos referencias que cumplen este test, hay dos posibilidades: que las dos referencias que estamos analizando sean la misma, en cuyo caso podemos deducir que no hay dependencia, o bien que sean referencias distintas dependientes dentro de una misma iteración del bucle interno. El resultado del test de monotonicidad se almacena hasta que en *same_monotonic?* se decide si hay dependencia o no.

Este test va a ser aplicable en muy pocas ocasiones puesto que el optimizador global, que se ejecuta durante la fase de PREOPT, antes de pasar al optimizador de bucles, intenta eliminar las variables de inducción. De modo que es improbable que el compilador vaya a encontrar una asignación del tipo $k = k + ct$ dentro de un bucle.

En la figura 2.12 se presenta un caso donde a priori este test podría devolver cierto, pero que no es aplicable después de que el bucle haya sido normalizado y tras la eliminación de variables de inducción.

Simple_Gcd

Dadas dos funciones afines f(i), g(i) tales que:

$$f(i) = a_0 + a_1 * i, \text{ y}$$
$$g(i) = b_0 + b_1 * i,$$

la ecuación $f(i) = g(i)$, o lo que es lo mismo,

$$a_1 * x - b_1 * y = b_0 - a_0$$

Figura 2.12: Same_Monotonic

```
k=0
do j=r,n,step
  k=k+1
  a(k)=a(k)+j
enddo
```

después de normalizar y eliminar variables de inducción

```
t1=((step+n-r) div step)-1
do t2=0,t1,1
  a(t2+1)=a(t2+1)+r+step*t2
enddo
```

tiene solución entera si y sólo si

$$gcd(a_1, b_1)|(b_0 - a_0),$$

es decir, si y sólo si el máximo común divisor de a_1, b_1 es divisor de $b_0 - a_o$. Dada una ecuación con más incógnitas, por ejemplo

$$a_1x_1 + a_2x_2 + a_3x_3 + \cdots + a_nx_n = b$$

existe solución entera sólo cuando

$$gcd(a_1, a_2, a_3, \cdots, a_n)|b$$

Inicialmente este test se aplica sólo a aquellas dimensiones de los access arrays donde aparecen símbolos no invariantes. Más adelante, al normalizar las ecuaciones que intervienen en el cálculo de dependencias, ya se aplicará el test gcd al resto de dimensiones.

En primer lugar se recorre cada dimensión de los access arrays. Para cada dimensión que cumpla

$$Non\_Const\_Loops(A_i) \geq (common\_nest - dv\_dim + 1) \vee$$
$$Non\_Const\_Loops(B_i) \geq (common\_nest - dv\_dim + 1)$$

se comprueba que las expresiones correspondientes a A_i y B_i estén formadas por términos lineales y se aplica el test GCD a la ecuación $A_i = B_i$.

num_bad_outer≥dv_dim

Dados dos accesos A,B a un array de n dimensiones, *num_bad_outer* se calcula como[4]

$$num\_bad\_outer = \max(\min(NCL(A_0), \ldots, NCL(A_{n-1})),$$
$$\min(NCL(B_0), \ldots, NCL(B_{n-1}))) - (common\_nest - dv\_dim)$$

donde NCL es la abreviatura de Non_Const_Loops. Si $num\_bad\_outer \geq dv\_dim$ entonces se asume que las dos referencias son dependientes.

[4]En realidad en el código del compilador no se utiliza la función máximo y sólo se utiliza el mínimo de los B_i. Pensamos que se trata de un error que si bien puede producir la generación de vectores de dependencia menos precisos no va a provocar la generación de código erróneo.

Non_Const_Loops(A_i) distinto de cero indica que la dimensión A_i de la referencia A contiene algún término no lineal (un producto de dos o más símbolos) o bien contiene una variable cuyo valor se modifica dentro de algún bucle. Como solamente nos interesan los bucles para los cuales vamos a calcular dependencias se restan los bucles más externos ($common\_nest - dv\_dim$). Si después de tomar el mínimo de Non_Const_Loops(A_i) para todo i, una vez restados los bucles externos, la cifra obtenida es mayor o igual que dv_dim, esto significa que no vamos a poder generar ninguna ecuación con los índices de A. Lo mismo se puede decir de B.

Si *num_bad_outer* es mayor que cero todas las dimensiones de alguno de los accesos contienen al menos un término que invalida esa dimensión para ser utilizada en la generación de ecuaciones. Sin embargo, si *num_bad_outer* es menor que *dv_dim*, es posible que si nos restringimos a los bucles más internos alguno de los símbolos que se modifican pase a ser invariante y esto permita que sí se pueda generar alguna ecuación.

En el siguiente ejemplo *num_bad_outer* es igual a dos. Como no va a ser posible generar ecuaciones se asume dependencia, siendo (+,*) y (0,*) los vectores de dependencia entre los accesos a la matriz A

```
DO I=1,10
  DO J=1,10
    IF (MOD(J,2) .EQ. 0) THEN
      K = K+1
    ENDIF
    A(I+K, J+K) = I+J
  ENDDO
ENDDO
```

En este otro ejemplo *num_bad_outer* es igual a uno, y K es invariante dentro del bucle más interno. Si suponemos el caso peor para las dependencias entre iteraciones del bucle I y nos limitamos a estudiar las dependencias dentro de una misma iteración de I podemos generar ecuaciones.

```
DO I=1,10
  IF (MOD(J,2) .EQ. 0) THEN
    K = K+1
  ENDIF
  DO J=1,10
    A(I+K, J+K) = I+J
  ENDDO
ENDDO
```

Las igualdades que se generan son:

$$-1 + I + K = -1 + I + K$$
$$-1 + J_1 + K = -1 + J_2 + K$$

De la segunda ecuación se deduce que la única dependencia posible tendría que tener distancia cero. Puesto que las dos referencias que estamos comparando son la misma, esta dependencia se descarta. En este caso el vector de dependencia entre los accesos a A es (+,*).

Trivial_Test

La función Trivial_Test() selecciona los access vectors válidos para generar ecuaciones e inicializa el vector *non_trivial_dim*. *non_trivial_dim*[*i*]=CIERTO significa que la dimensión i de las referencias A y B es válida para generar una ecuación.

Dados dos accesos A y B, *non_trivial_dim*[*i*] será CIERTO si se cumple:

- $common\_nest - Non\_Const\_Loops(A_i) \geq dv\_dim - num\_bad\_outer$
- $common\_nest - Non\_Const\_Loops(B_i) \geq dv\_dim - num\_bad\_outer$
- términos no lineales de A_i = términos no lineales de B_i[5]
- A_i o B_i contienen, al menos, un término lineal no constante[6]

Si se cumplen todas las condiciones anteriores menos la última ambos términos sólo pueden contener términos no lineales idénticos (e invariantes) y términos constantes. Si los valores constantes no coinciden entonces los dos accesos son independientes. En el siguiente ejemplo, puesto que $R*K \neq R*K+2$, podemos deducir independencia.

```
DO I
  DO J
    V(R*K,J) = ...
    ... = V(R*K+2,J)
  ENDDO
ENDDO
```

Generación del sistema de ecuaciones

Una vez identificadas las dimensiones que se van a utilizar el siguiente paso es generar el sistema de ecuaciones. Para cada dimensión marcada como no trivial en *non_trivial_dim* se generará una ecuación $A_i = B_i$, que se guardará con el siguiente formato:

$$(b_1 - a_1)x_1 + (b_2 - a_2)x_2 + \cdots + (b_n - a_n)x_n = b_0 - a_0.$$

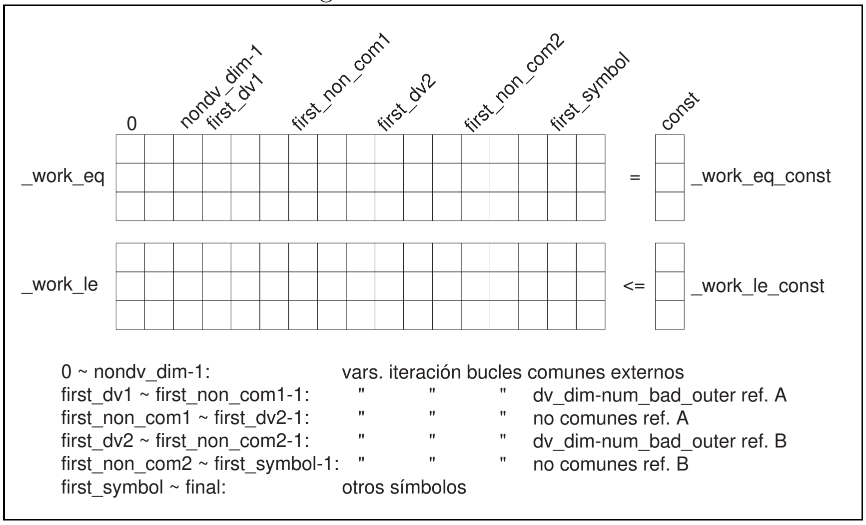
Estas ecuaciones se guardan en una matriz (ver la figura 2.13) donde cada columna representa una variable y cada fila representa una ecuación. El término independiente de cada ecuación se guarda en un vector a parte. Si para todo i, *non_trivial_dim*[*i*] $= 0$, no se puede generar ninguna ecuación y se asume dependencia. A las igualdades generadas les añadiremos las desigualdades establecidas por los límites de los bucles.

DEPV_COMPUTE::Copy_Equals_To_Work() inicializa las estructuras de datos que contienen las igualdades. A priori se desconoce el número de símbolos (que no sean variables de iteración) que van a aparecer en las ecuaciones finales. Cada vez que se encuentra un nuevo símbolo se guarda en una pila, de este modo se controla qué símbolos ya han aparecido al menos una vez y que, por tanto, ya tienen una columna asignada en la matriz que almacena las ecuaciones. La

[5] Las condiciones anteriores garantizan que los términos no lineales son invariantes. Si éstos son idénticos en ambas expresiones se anularán al generar la ecuación.

[6] Se entiende por constante un valor conocido en tiempo de compilación.

Figura 2.13: Ecuaciones



posición del símbolo dentro de la pila determina su ubicación en la matriz. Esta pila de símbolos se pasará también a la función encargada de inicializar las desigualdades (DEPV_COMPUTE::Copy_Bounds_To_Work()).

Una vez generado el sistema de ecuaciones se puede empezar con el cálculo de vectores de dependencia. Básicamente el algoritmo que se va aplicar es el descrito por Burke y Citron en [4]. El vector de dependencias inicial que se utiliza al aplicar este algoritmo es el calculado por la función Find_Init_Distance().

Find_Init_Distance

Esta función calcula las distancias constantes que aparecerán en los vectores de dependencia. El vector de dependencias devuelto por esta función se utiliza como valor inicial en el algoritmo de Burke-Cytron. Resuelve las ecuaciones que se obtienen al igualar expresiones que representan rectas de idéntica pendiente (ver figura 2.14).

Dichas ecuaciones serán del tipo $a \cdot i_1 + b \cdot i_2 = c$ donde:

- $a = -b$,
- i_1, i_2 representan instancias distintas de la variable de iteración i.

La distancia que separa ambas referencias $(i_2 - i_1)$ es $-c/a$. Esta distancia debe ser un valor entero, en caso contrario las dos referencias son independientes (test gcd). Al resolver una ecuación de este tipo se inicializa la posición correspondiente al bucle i en el vector de dependencias. Si de otra ecuación se deduce un valor distinto para esta dimensión las dos referencias son independientes ya que no va a ser posible satisfacer las dos ecuaciones simultáneamente.

Compute_Dep_Vectors

Utilizando como punto de partida el vector de dependencia calculado por la función Find_Init_Distance() se aplica el algoritmo de Burke-Cytron [4]. Este

Figura 2.14: Rectas paralelas

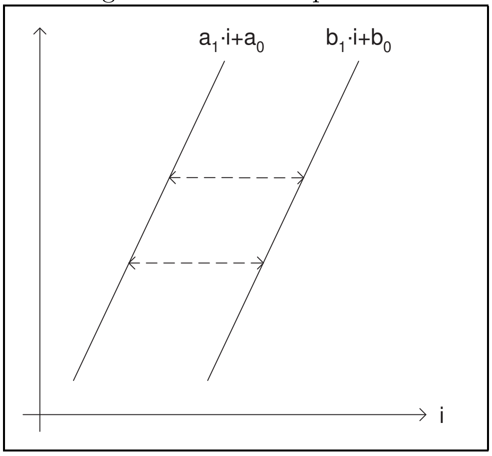
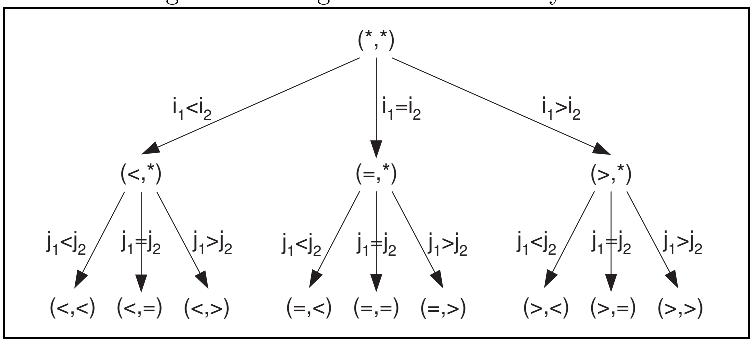


Figura 2.15: Algoritmo de Burke-Cytron



algoritmo consiste, a grandes trazos, en tomar un vector de dependencia inicial e ir refinando cada una de las dimensiones añadiendo restricciones al sistema de ecuaciones. Cada vez que se añade una ecuación se comprueba la consistencia del sistema. Esta exploración de los distintos valores que puede tomar el vector de dependencia se hace de forma jerárquica (ver figura 2.15), de modo que si un valor no es válido no hace falta explorar el subárbol situado por debajo. Al final de este proceso se devuelve una lista con todos los vectores de dependencia válidos.

El algoritmo 1 muestra como se implementa esta búsqueda de los vectores de dependencia. Se trata de una versión simplificada. En la implementación del compilador se añade una comprobación para evitar el cálculo para el vector (=, =, ..., =) cuando estamos buscando dependencias entre ejecuciones distintas de un mismo store. Además aparecen algunos parámetros adicionales para contemplar los casos en que alguna variable de iteración no aparece en las ecuaciones, ya que en ese caso no tiene sentido añadir nuevas restricciones sobre esa variable. También se tiene en cuenta que *num_bad_outer* puede ser distinto de cero, en cuyo caso nos limitaremos a calcular dependencias dentro de una misma iteración de los bucles externos.

Algoritmo 1: Compute_Dep_Vectors
Entrada: Sistema de ecuaciones (*soe*), vector de dependencia (*input_dependence*) y lista de vectores de dependencia (*result*).
Salida: La lista de vectores de dependencia (*result*).

i = posición del primer '*' dentro de *input_dependence*
if $i = -1$
    Añadir *input_dependence* a *result*
    Devolver *result*

Añadir ecuación $x_1 < x_2$ al soe (donde x es la variable de iteración del bucle i)
input_dependence$[i] = ' <'$
if soe.EsConsistente()
    Compute_Dep_Vectors(*soe*,*input_dependence*,*result*)
Eliminar ecuación $x_1 < x_2$ del soe
input_dependence$[i]$ = '*'

Añadir ecuación $x_1 = x_2$ al soe
input_dependence$[i]$ = '='
if soe.EsConsistente()
    Compute_Dep_Vectors(*soe*,*input_dependence*,*result*)
Eliminar ecuación $x_1 = x_2$ del soe
input_dependence$[i]$ = '*'

Añadir ecuación $x_1 > x_2$ al soe
input_dependence$[i] = ' >'$
if soe.EsConsistente()
    Compute_Dep_Vectors(*soe*,*input_dependence*,*result*)
Eliminar ecuación $x_1 > x_2$ del soe
input_dependence$[i]$ = '*'

Lex_Pos_Decompose

Una vez calculados los vectores de dependencia y antes de añadir las aristas respectivas al grafo de dependencias, los vectores de dependencia son clasificados. La función Lex_Pos_Decompose() distribuye los vectores de dependencia entre una lista de dependencias positivas y una de negativas, ver el algoritmo 2. Las variables *keep_pos_equals* y *keep_neg_equals* se inicializan de modo que los vectores que indican dependencia dentro de una misma iteración, que tienen la forma ('=','=',...,'='), se asocien a la arista con origen en la referencia situada lexicográficamente en primer lugar. Si se están analizando las dependencias entre dos ejecuciones distintas de una misma referencia estos vectores de dependencia no se añaden a ninguna de las dos listas.

Algoritmo 2: Lex_Pos_Decompose
Entrada: *depv*: vector de dependencia; *pos*: lista de dependencias positivas; *neg*: lista de dependencias negativas; *dv_dim*: número de dimensiones de los vectores de dependencia; *first_dim*: dimensión en curso; *keep_pos_equals*: en caso de igual en todas las dimensiones añadir a pos; *keep_neg_equals*: en caso de igual en todas las dimensiones añadir a neg.
Salida: *pos*, *neg*.

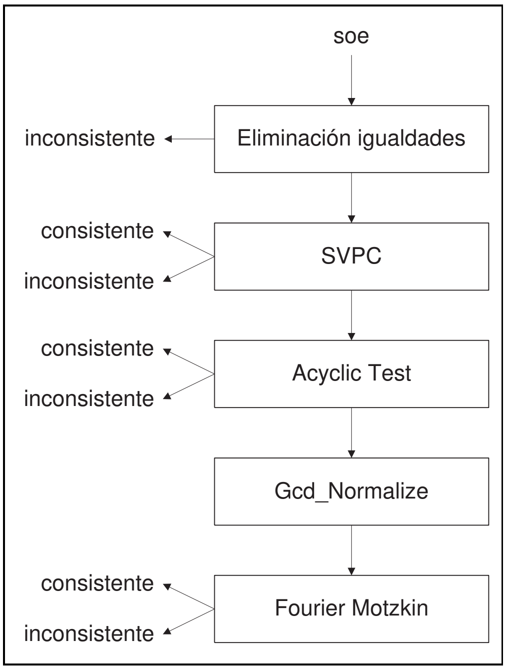
```
if first_dim = dv_dim
    if keep_pos_equals
        Añadir depv a pos
    if keep_neg_equals
        Añadir depv a neg

if depv[first_dim] = '<'
    Añadir depv a pos
else if depv[first_dim] = '>'
    foreach dimensión i mayor o igual que first_dim
        Invertir signo de depv[i]
    Añadir depv a neg
else if depv[first_dim] = '*'
    depv[first_dim] = '<'
    Añadir depv a pos
    depv_tmp = depv
    foreach dimensión i mayor que first_dim
        Invertir signo de depv_tmp[i]
    Añadir depv_tmp a neg
    depv[first_dim] = '='
    Lex_Pos_Decompose( depv, pos, neg, dv_dim,
    first_dim+1, keep_pos_equals, keep_neg_equals )
else if depv[first_dim] = '='
    Lex_Pos_Decompose( depv, pos, neg, dv_dim,
    first_dim+1, keep_pos_equals, keep_neg_equals )
```

2.2.3. Consistencia del sistema de ecuaciones

Para determinar la consistencia de un sistema de ecuaciones se utilizan las estructuras de datos i funciones definidas en be/lno/soe.h y be/lno/soe.cxx. El sistema de ecuaciones se compone de un conjunto de igualdades $A_{eq} \cdot x = b_{eq}$ más un conjunto de desigualdades $A_{le} \cdot x \leq b_{le}$. Estas ecuaciones se guardan mediante dos matrices de enteros y dos vectores para los términos independientes. Adicionalmente existe una área de trabajo que sólo va a contener desigualdades. El primer paso, antes de empezar a realizar los tests para determinar la consistencia del sistema, es copiar todas las desigualdades en el área de trabajo. Hecho esto se llevan a cabo los pasos que muestra la figura 2.16:

- Mediante el algoritmo descrito en [13] se eliminan las igualdades.
- Se aplica el test SVPC (Single Variable Per Constraint) utilizando aquellas ecuaciones donde sólo aparece una variable.

Figura 2.16: Consistencia del sistema de ecuaciones



- Se aplica el Acyclic Test, descrito en [12].
- Finalmente, si los tests anteriores no han sido concluyentes, se normalizan las ecuaciones y recurrimos al algoritmo de Fourier Motzkin.

Eliminación de las igualdades

La eliminación de las igualdades se lleva a cabo utilizando el algoritmo descrito por Pugh en [13]. Podemos expresar cada igualdad como un sumatorio de términos $\sum_{i \in V} a_i x_i = 0$. Si se encuentra algún término $j \neq 0$ tal que $|a_j| = 1$ se elimina la ecuación aislando x_j y sustituyendo en el resto de ecuaciones.

Para eliminar una ecuación que no contiene ningún coeficiente igual a 1 se sigue un proceso que va a permitir la sustitución de esta ecuación por otra cuyos coeficientes en valor absoluto serán menores. Este proceso se repite hasta que se obtiene un término con un coeficiente cuyo valor absoluto sea igual a 1. Después de aplicar el algoritmo de Pugh el sistema de ecuaciones resultante tiene solución si y sólo si el sistema original tenía solución.

Dada un ecuación $\sum_{i \in V} a_i x_i = 0$ que no contiene ningún término j tal que $|a_j| = 1$ y $j \neq 0$, y sea k ($k \neq 0$) el índice del término con el coeficiente de valor absoluto menor, y sea $m = |a_k| + 1$, definimos la operación $\widehat{mod}$ como:

$$a \ \widehat{mod} \ b = a - b\lfloor a/b + 1/2 \rfloor$$

Creamos una nueva variable σ y añadimos la siguiente ecuación al sistema:

$$m\sigma = \sum_{i \in V} (a_i \ \widehat{mod} \ m) x_i$$

Nótese que $a_k \widehat{mod}\ m = -sign(a_k)$. Aislamos x_k en la ecuación anterior:

$$x_k = -sign(a_k)m\sigma + \sum_{i \in V-\{k\}} sign(a_k)(a_i \widehat{mod}\ m)x_i$$

y sustituimos x_k en todas las ecuaciones del sistema. La ecuación original se convierte en:

$$-|a_k|m\sigma + \sum_{i \in V-\{k\}} (a_i + |a_k|(a_i \widehat{mod}\ m))x_i = 0$$

Puesto que $|a_k|$ es igual a $m-1$, reescribimos la ecuación como:

$$-|a_k|m\sigma + \sum_{i \in V-\{k\}} (a_i - (a_i \widehat{mod}\ m) + m(a_i \widehat{mod}\ m))x_i = 0$$

En esta ecuación todos los términos son divisibles por m. Podemos normalizar obteniendo:

$$-|a_k|\sigma + \sum_{i \in V-\{k\}} (\lfloor (a_i/m + 1/2) \rfloor + (a_i \widehat{mod}\ m))x_i = 0$$

De este modo hemos transformado la ecuación original en otra donde aparece una nueva variable σ, cuyo coeficiente tiene el mismo valor absoluto que el coeficiente de x_k, y donde el valor absoluto del resto de coeficientes se ve reducido. Aplicando repetidamente esta transformación llegaremos a obtener un término con coeficiente igual a 1 que utilizaremos para aislar la variable correspondiente y eliminar la ecuación del sistema.

En la siguiente ecuación:

$$7x + 12y + 31z - 17 = 0$$

el coeficiente de valor absoluto menor acompaña a la variable x y, por tanto, $m = |7| + 1 = 8$. Generamos la ecuación:

$$8\sigma = (7\widehat{mod}\ 8)x + (12\widehat{mod}\ 8)y + (31\widehat{mod}\ 8)z + (-17\widehat{mod}\ 8)$$

Aplicamos la definición del operador $\widehat{mod}$:

$$8\sigma = -x - 4y - z - 1$$

Aislamos la x:

$$x = -4y - z - 1 - 8\sigma$$

y sustituimos en la ecuación original (y resto de ecuaciones del sistema):

$$7(-4y - z - 1 - 8\sigma) + 12y + 31z - 17 = 0$$

agrupamos los términos:

$$-16y + 24z - 56\sigma - 24 = 0$$

y normalizamos dividiendo por m. Como resultado se obtiene la siguiente ecuación, donde el valor absoluto de todos los coeficientes (menos el de σ) se ha visto reducido:

$$-2y + 3z - 7\sigma - 3 = 0$$

Si repetimos el mismo proceso partiendo de la ecuación $-2y + 3z - 7\sigma - 3 = 0$, entonces $m = |-2| + 1$. Reescribiendo la ecuación y aislando la y, obtendremos que $y = 3\tau + \sigma$. Sustituyendo la y en la ecuación inicial y simplificando el resultado es:

$$-3\sigma - 2\tau + z = 1$$

En esta ecuación podemos aislar la z, que se sustituiría en el resto de ecuaciones del sistema, eliminando una ecuación y reduciendo el número de variables.

SVPC (Single Variable Per Constraint)

Este test consiste en intentar encontrar una cota máxima y una cota mínima para cada variable. Si en algún momento se detecta que el valor máximo está por debajo del mínimo el sistema es inconsistente.

Para calcular el valor máximo y mínimo de cada variable, en primer lugar se inicializan dos vectores de enteros (*_lower_bound* y *_upper_bound*) con los valores mínimos y máximos que se pueden representar. Hecho esto se recorren las ecuaciones del sistema. Cada vez que encontramos una ecuación que contiene sólo una variable se actualiza la cota correspondiente. Si las cotas son incompatibles el sistema es inconsistente. Las cotas calculadas durante este test quedan guardadas para poder ser utilizadas más adelante.

Si todas las ecuaciones contienen sólo una variable y después de aplicar este test no se ha podido determinar su inconsistencia, podemos concluir que el sistema de ecuaciones es consistente.

Acyclic Test

El objetivo de este test es eliminar variables hasta conseguir un sistema donde cada ecuación contenga sólo una variable y que, por tanto, se pueda resolver aplicando el test anterior. Las variables que se van a poder eliminar son aquellas cuyo valor sólo está acotado en una dirección. Ver [12].

Dado un conjunto de ecuaciones con más de una variable, para poder eliminar x_i, primero se debe comprobar que x_i sólo está acotada en una dirección. Para ello se reescriben las ecuaciones aislando x_i:

$$a_{1,i}x_i \leq f_1(x)$$

$$a_{2,i}x_i \leq f_2(x)$$

$$\ldots$$

$$a_{n,i}x_i \leq f_n(x)$$

Si hecho esto todos los coeficientes $a_{k,i}$ son mayores que cero, la variable x_i sólo está acotada por arriba, ya que en todas las ecuaciones el valor de x_i es menor que una cierta función del resto de variables. A continuación se sustituye x_i por la cota inferior de x_i calculada en el test SVPC. El sistema resultante sólo va a ser consistente si el sistema original también lo era. Si no existe una cota inferior para x_i, entonces siempre será posible encontrar un valor para x_i menor que la cota superior más restrictiva. En este caso eliminaremos las ecuaciones donde aparece x_i porque seguro que estas ecuaciones no van a hacer que el sistema sea inconsistente. Podemos aplicar un razonamiento similar cuando todos los $a_{k,i}$ son menores que cero. En ese caso sustituiremos x_i por su cota superior.

Si después de eliminar una de las variables alguna de las ecuaciones pasa a tener una sola variable se aplica el test SVPC y se actualiza el valor almacenado en *_lower_bound* o *_upper_bound* según corresponda.

El algoritmo 3 muestra el algoritmo que aplica este test.

Algoritmo 3: Acyclic Test
Entrada: sistema de ecuaciones
Salida: sistema de ecuaciones, resultado del test

```
while hay progreso
    // Se ha eliminado alguna variable y quedan
    // variables por eliminar
    foreach variable x_i del sistema
        if x_i no ha sido eliminada
            if x_i está acotada en una dirección
                // Todos los coeficientes de la variable i
                // tienen el mismo signo
                foreach ecuación del sistema
                    Sustituir x_i por el valor correspondiente
                    Comprobar si la ecuación es SVPC
```

Fourier-Motzkin

Si los tests anteriores no han tenido éxito se va a aplicar el algoritmo de Fourier-Motzkin para determinar la consistencia del sistema. Básicamente el algoritmo consiste en ir eliminando una a una todas las variables del sistema hasta encontrar una ecuación sin variables que es inconsistente o hasta que sólo queda una variable [3]. Llegado este punto se aplica el test SVPC.

Por ejemplo, dado un sistema donde aparecen las siguientes ecuaciones:

$$a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = a_{1,0}.$$

$$a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = a_{2,0}.$$

$$a_{3,1}x_1 + a_{3,2}x_2 + \cdots + a_{3,n}x_n = a_{3,0}.$$

Supongamos que queremos eliminar la variable x_i del sistema y que $a_{1,i}, a_{3,i} < 0 < a_{2,i}$. Para eliminar x_i lo primero que haríamos es aislar la variable en cada una de las ecuaciones, obteniendo:

$$x_i \geq f_1(x)/a_{1,i}$$

$$x_i \leq f_2(x)/a_{2,i}$$

$$x_i \geq f_3(x)/a_{3,i}$$

Por tanto:

$$f_2(x)/a_{2,i} \geq x_i \geq f_1(x)/a_{1,i}$$

$$f_2(x)/a_{2,i} \geq x_i \geq f_3(x)/a_{3,i}$$

Eliminamos x_i generando las ecuaciones

$$f_2(x)\frac{a_{1,i}}{mcd(a_{1,i},a_{2,i})} \leq f_1(x)\frac{a_{2,i}}{mcd(a_{1,i},a_{2,i})}$$

$$f_2(x)\frac{a_{3,i}}{mcd(a_{3,i},a_{2,i})} \leq f_3(x)\frac{a_{2,i}}{mcd(a_{3,i},a_{2,i})}$$

En este caso se ha reducido el número de ecuaciones del sistema. En general, se deben cruzar todas las ecuaciones con coeficiente de x_i positivo con todas las ecuaciones con coeficiente negativo, y eso se debe hacer para cada una de las variables. Se trata, por tanto, de un algoritmo cuyo coste puede ser muy elevado. El algoritmo 4 muestra las líneas generales de la implementación.

Algoritmo 4: Fourier-Motzkin
Entrada: sistema de ecuaciones
Salida: resultado del test

foreach variable x_i del sistema
    while queda alguna ecuación por tratar
        k = índice de la ecuación en curso
        if $a_{k,i} > 0$
            foreach ecuación l tal que $x_{l,i} < 0$
                añadir una nueva ecuación sin x_i
            Eliminar la ecuación k
    foreach ecuación l tal que $x_{l,i} < 0$
        Eliminar la ecuación l
    Eliminar ecuaciones redundantes

Capítulo 3

Obtención de SNLNs

En este capítulo se describe con detalle la implementación de la primera etapa del optimizador de bucles, consistente en la identificación de aquellas anidaciones de bucles que se van a tratar, denominadas SNLNs. Durante esta etapa se aplicará fusión y distribución con el objetivo de conseguir anidaciones de profundidad máxima.

En la sección 3.1 se define el término SNLN y se presenta el algoritmo que implementa esta primera etapa del optimizador de bucles.

En la sección 3.2 vemos en qué consiste la reordenación topológica, transformación que se lleva a cabo para facilitar la aplicación de transformaciones posteriores.

Las secciones 3.3 y 3.4 detallan, respectivamente, la aplicación de fusión y distribución, que son las transformaciones utilizadas para generar SNLNs de profundidad máxima. A continuación, en la sección 3.5, se muestra un ejemplo de su aplicación.

Por último, en la sección 3.6, vemos como se aplica fusión a los bucles externos con el objetivo de mejorar la localidad de los accesos a memoria.

3.1. Definiciones y algoritmo

Dos bucles están perfectamente anidados cuando uno de ellos (bucle externo) contiene al otro (bucle interno) y el bucle externo no contiene más sentencias que las que pertenecen al bucle interno. Podemos extender esta definición a más bucles añadiendo niveles de profundidad, de modo que cualquier pareja de bucles consecutivos se mantenga perfectamente anidada.

Si el bucle externo contiene sentencias que no pertenecen al bucle interno, nos referiremos a este conjunto de instrucciones como el código imperfecto.

Definiciones (ver [15]):

SNLN (Singly Nested Loop Nest) Conjunto de bucles perfectamente anidados, o bien no perfectamente anidados cuyo código imperfecto no contiene ningún bucle u otro tipo de control de flujo complejo. En el código imperfecto sólo se permiten llamadas, condicionales y sentencias de asignación. También consideraremos SNLN cualquier bucle tomado individualmente, esté o no esté anidado.

Figura 3.1: SNLNs maximales

```
DO i=1,N
  DO j=1,N
    DO k=1,N
      IF ()...
      DO l=1,N
        S1
      ENDDO
    ENDDO
    DO m=1,N
      S2
    ENDDO
  ENDDO
ENDDO
```

SNLNs maximales Son los que se obtienen al agrupar SNLNs de tal forma que no sea posible combinar dos SNLNs para formar un SNLN de mayor profundidad.

En el código de la figura 3.1 podemos observar el SNLN maximal formado por los bucles i y j, el formado por k y l y finalmente el bucle m. Los bucles i, j, k y l son cada uno de ellos un SNLN, pero no son SNLNs maximales.

La definición que se deriva de la implementación del compilador es más restrictiva. Para poder formar parte de un SNLN un bucle no debe contener llamadas ni *gotos*, independientemente de que éstos se encuentren en el código imperfecto o no. Adicionalmente todos los bucles de un SNLN deben cumplir la propiedad Do_Loop_Is_Good. Esta propiedad puede devolver falso cuando en el cuerpo del bucle existe alguna llamada o referencia a memoria que no dispone de su correspondiente nodo en el grafo de dependencias entre arrays[1]. Do_Loop_Is_Good también puede devolver falso cuando el compilador detecta que el bucle contiene gotos/returns que abandonan el bucle o llamadas sin información del análisis interprocedural. Estas propiedades, tanto la presencia de llamadas y/o *gotos* como Do_Loop_Is_Good, se propagan hacia arriba de forma que, si un bucle no satisface alguna de ellas, automáticamente dejan de satisfacerlas todos aquellos bucles que lo contienen.

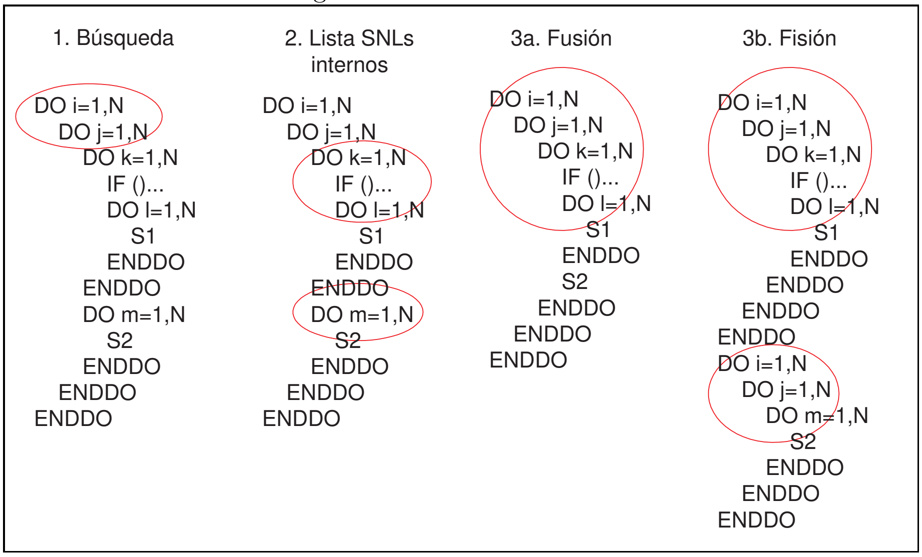
El algoritmo que presentamos a continuación explora una anidación de bucles para detectar/construir SNLNs de profundidad máxima. Podemos distinguir tres partes:

1. Se desciende por la anidación hasta que no hay más bucles, hay más de uno al mismo nivel de profundidad, o se encuentra un bucle dentro de una cláusula *if*. Los bucles encontrados conforman el SNLN en curso.

2. Se genera una lista con los SNLNs contenidos en el bucle más interno del SNLN en curso. Todos los SNLNs que aparecen en la lista se encuentran al mismo nivel y son los visibles desde el SNLN en curso. Este proceso se

[1]Esto puede producirse cuando el compilador no dispone de suficiente memoria para construir el grafo completo, cosa que no impide continuar con la compilación.

Figura 3.2: Obtención SNLNs



lleva a cabo mediante llamadas recursivas cuando se detecta un DO en el cuerpo del bucle.

3. Se intenta combinar el SNLN en curso con los de la lista generada en el paso 2 para obtener SNLNs de mayor profundidad. Para ello aplicaremos fusión y/o distribución a los elementos de la lista de SNLNs.

La figura 3.2 nos muestra el resultado de aplicar este algoritmo al ejemplo anterior. Se empieza por el SNLN más externo, a continuación se obtiene la lista de SNLNs que éste contiene y, finalmente, se muestra el resultado después de aplicar fusión o fisión. La selección del tipo de transformación que se va a aplicar depende de las opciones y directivas de compilación y de las características del código.

A continuación detallamos cada uno de estos tres pasos. Aunque los tratemos por separado, todos ellos forman parte de una misma función que se denomina Fiz_Fuse(). En el punto 2 se van a realizar llamadas recursivas a esta función. Dentro de los archivos fuente del compilador, podemos encontrar la implementación de Fiz_Fuse() en be/lno/fiz_fuse.cxx. Apuntar también que, en adelante, utilizaremos las siglas SNLN y SNL indistintamente.

3.1.1. Búsqueda de un SNLN

El algoritmo 5 muestra como descendemos por una anidación de bucles para detectar un SNL. Se considera que un bucle es válido si no contiene llamadas, no contiene *gotos* y cumple la propiedad Do_Loop_Is_Good. Cada vez que encontramos un bucle válido se reordenan sus sentencias como se describe en la sección 3.2.

Algoritmo 5: Búsqueda de un SNL
Entrada: Un bucle. Lista $snls$, aunque de momento no la utilizamos (ver 3.1.2).
Salida: Una lista con los bucles que forman parte del SNL (*bucles_snl*), una referencia al bucle más externo del SNL (*bucle*) y una referencia al último bucle visitado ($wn1$).
(1) $wn1 = bucle$ (bucle externo de la pareja en curso)
(2) $wn = bucle$ (bucle interno de la pareja en curso)
(3) **while** $wn \neq NULL$
(4) **if** $wn1$ no es válido **and** wn es válido
(5) $bucle = wn$
(6) **if** wn es válido
(7) Añadir wn a *bucles_snl*
(8) $wn1 = wn$
(9) wn = Obtener_Siguiente_Bucle(wn)
(10) **if** $wn1$ es válido **and** $wn1$ contiene más bucles
(11) Reordenar sentencias de $wn1$

3.1.2. Obtención de SNLNs internos

Supongamos que en el paso anterior hemos conseguido encontrar un SNLN, llamémosle A. Ahora vamos a generar una lista ($info$) con los SNLNs contenidos en A. Los SNLNs que vamos a añadir a $info$ estarán todos al mismo nivel y serán todos visibles desde A. Esto quiere decir que el bucle más externo de cualquier SNLN contenido en $info$ será hijo directo de $wn1$, el bucle más interno de A.

Para generar esta lista se recorre el cuerpo de $wn1$, actuando del siguiente modo en función del tipo de sentencia encontrada:

DO Llamada recursiva a Fiz_Fuse(). Devuelve una lista con cero, uno, o más SNLNs que se añaden a $info$. La lista estará vacía cuando no se encuentre ningún bucle válido, y puede contener más de un SNLN si en la llamada recursiva se ha fisionado el DO.

IF, REGION, DO_WHILE, WHILE_DO Llamada a la función recursiva If_While_Region_Fiz_Fuse() definida en be/lno/fiz_fuse.cxx. Devuelve un SNLN ficticio que se añade a $info$. El SNLN devuelto se marca como *Non_SNL* porque un SNLN sólo puede estar formado por bucles DO.

OTRAS SENTENCIAS Se ignoran.

En ambos casos, es posible que por debajo del SNLN devuelto haya otros SNLNs no visibles desde A. Éstos se añaden a una lista, llamada $snls$, que se pasa por referencia. Ver algoritmo 6.

Si en el primer paso (sección 3.1.1) no hemos encontrado ningún bucle válido, generaremos la lista $info$ igualmente, añadiremos todos los SNLNs encontrados a $snls$ y devolveremos una lista vacía.

Otra posibilidad es que al profundizar en una anidación de bucles, los primeros bucles encontrados no sean válidos, pero que sí se pueda obtener un SNLN con el resto. No hay un tratamiento especial para este caso y podríamos preguntarnos qué pasará cuando al volver de una llamada recursiva se devuelva

Algoritmo 6: Obtención lista SNLNs internos
Entrada: Las variables *bucles_snl*, *bucle* y $wn1$ obtenidas en la búsqueda (algoritmo 5) y la lista *snls*.
Salida: $info$ es la lista con los SNLNs contenidos en $wn1$. *snls* es la lista donde se acumulan los SNLNs encontrados.
(1) wn = Primera_Sentencia($wn1$)
(2) **while** $wn \neq NULL$
(3)   *siguiente* = Siguiente_Sentencia(wn)
(4)   **if** wn es un bucle DO
(5)   Añadir a $info$ el resultado de Fiz_Fuse(wn, *snls*)
(6)   **if** wn es un IF, REGION, WHILE_DO o DO_WHILE
(7)   Añadir a $info$ el resultado de If_While_Region_Fiz_Fuse(wn, *snls*)
(8) wn = *siguiente*
(9) **if** *bucle* no es válido
(10)   Añadir a *snls* el contenido de $info$
(11)   Devolver una lista vacía
(12) **if** $info$ está vacía
(13)   Devolver el SNLN en curso

este SNLN como resultado. Si esto ocurre es posible que $info$ contenga SNLNs ubicados en distintos niveles de profundidad. Esto no va a ser problema, porque si un bucle no es válido, ninguno de los que lo contienen lo es. Como los SNLNs de $info$ no estarán contenidos dentro de un SNLN, el siguiente paso una vez elaborada la lista será añadir todos sus elementos a *snls* y devolver una lista vacía, sin que exista la posibilidad de que se intente combinar SNLNs situados en distintos niveles de profundidad.

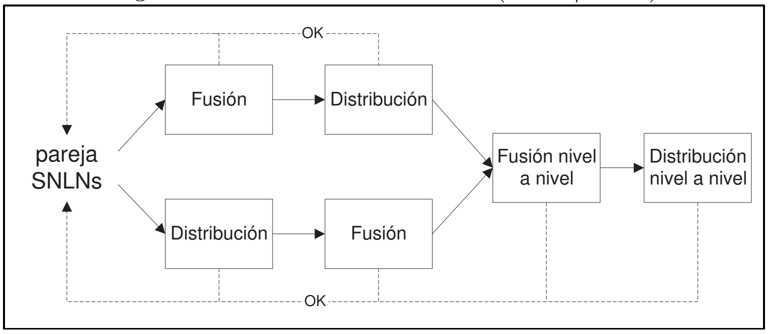
3.1.3. Combinación de SNLNs (fusión/fisión)

Finalmente se intenta combinar los SNLNs encontrados en el paso anterior para formar SNLNs de mayor profundidad. Se toma una pareja de SNLNs, se decide qué transformación se va a intentar en primer lugar, y se intenta aplicar fusión o distribución en el orden que muestra la figura 3.3. En cuanto una de las transformaciones tiene éxito, se obtiene la siguiente pareja de SNLNs (teniendo en cuenta que el código se habrá modificado) y se repite el proceso.

A veces, para poder fusionar, será necesario extraer iteraciones de alguno de los bucles. Al fusionar varios niveles sólo se permite extraer iteraciones en los bucles del nivel más interno. La fusión nivel a nivel permite extraer iteraciones también en el nivel más externo. La distribución nivel a nivel permite fisión parcial, es decir, que no sea obligatorio fisionar todos los niveles del SNL.

El algoritmo 7 ofrece una visión más detallada de este proceso. En él aparecen las variables $wn1$ y $wn2$, a las cuales nos referimos unas veces como bucles y otras veces como SNLNs. Efectivamente ambas variables son referencias a bucles, pero también podemos identificar un SNLN mediante su bucle más externo.

Figura 3.3: Combinación SNLNs (fusión/fisión)



Algoritmo 7: Combinación SNLNs (fusión/fisión)
Entrada: La variable *bucles_snl* y las listas de SNLNs $info$ y $snls$.
Salida: *rval* es una lista de SNLNs con los SNLNs obtenidos al final de esta etapa. La variable $snls$.

(1) **while** quedan snlns de $info$ por tratar
(2) $< wn1, wn2 >=$ Obtener_Pareja_Snls(...)
(3) **if** intentar primero fusión
(4) **if** Mover_Adyacentes($wn1$, $wn2$)
(5) fusion_status = Fusión($wn1$, $wn2$, *min_snl_level*, ...)
(6) **else**
(7) fusion_status = fracaso
(8) **if** fusión con éxito
(9) ir a la siguiente iteración
(10) **if** intentar fisión
(11) fision_status = Fisión(*wn_padre*, $wn1$, $wn2$, *bucles_snl*.Ultimo_Indice()+1)
(12) **if** fisión con éxito
(13) actualizar *rval* y *snls*
(14) ir a la siguiente iteración
(15) **if** intentar fusión **and** fusión no fallida
(16) (repetición líneas 4-9)
(17) **if** intentar fusión **and** *min_snl_level* > 1 **and** $fusion\_status = probar\_nivel\_a\_nivel$
(18) **foreach** nivel a fusionar
(19) intentar fusión
(20) **if** niveles fusionados < *min_snl_level*
(21) fusión fallida
(22) **else**
(23) ir a la siguiente iteración
(24) **if** intentar fisión **and** *bucles_snl*.Ultimo_Indice()+1>1
(25) **foreach** nivel a fisionar
(26) intentar fisión
(27) **if** fisión total o parcial con éxito
(28) actualizar *rval* y *snls*
(29) ir a la siguiente iteración
(30) actualizar rval, snls
(31) devolver rval

La función *Mover_Adyacentes()* intenta situar los bucles $wn1$ y $wn2$ uno inmediatamente a continuación del otro. Para ello se intenta desplazar, si las dependencias lo permiten, las sentencias que se interponen entre ambos a una posición anterior a $wn1$ o posterior a $wn2$.

min_snl_level es la profundidad del SNLN de menor profundidad e indica el límite en el número de niveles que vamos a poder fusionar. En cuanto a la fisión *wn_padre* indica el bucle que contiene $wn1$ y $wn2$. En este caso el límite en el número de niveles a fisionar viene determinado por la profundidad del SNLN que contiene a $wn1$ y $wn2$.

Las funciones de fusión y fisión permiten especificar el número de niveles a los que vamos a intentar aplicar la transformación y pueden devolver alguno de los siguientes resultados:

éxito	Se ha podido aplicar la transformación en todos los niveles.
fracaso	No se cumple alguna condición necesaria para poder llevar a cabo la transformación.
probar_nivel_a_nivel	Se debe intentar nivel a nivel.

Como ya se ha comentado, la fisión nivel a nivel permite una aplicación parcial de la transformación. Respecto a la fusión, ésta debe ser completa y su aplicación nivel a nivel sólo sirve para permitir peeling en los bucles del nivel más externo. En este caso podemos extraer tantas iteraciones de los bucles externos como sea necesario. No ocurre igual en el nivel más interno, donde siempre existe un límite prefijado en el número de iteraciones que se pueden extraer.

La siguiente sección describe cómo se decide cuándo se debe intentar fusión y/o fisión y cuál de las dos transformaciones tiene prioridad.

Selección de la transformación

En primer lugar vamos a definir SNL interno:

> Un SNL es interno cuando no contiene bucles que no forman parte del SNL.

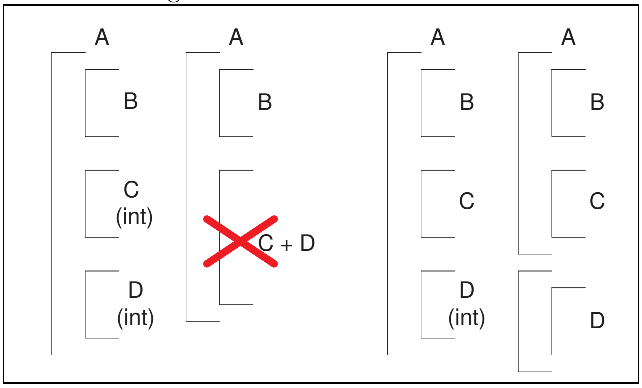
Al combinar los SNLs para obtener anidaciones de mayor profundidad se intenta que los nuevos SNLs, generados mediante la aplicación de fusión y fisión sean, además, internos. Esta política condiciona la selección de la transformación a aplicar, que también depende de las opciones y directivas de compilación.

La figura 3.4 muestra un SNL A en el que B y C no han podido ser combinados. Aunque C y D son SNLs internos se desestima su fusión porque no se pueden combinar con A para generar un SNL interno de mayor profundidad. Sí es posible, en cambio, que se utilice distribución, que va a ser aplicable siempre que una de las dos anidaciones resultantes pueda dar lugar a un SNL interno.

En la implementación, y siguiendo con el ejemplo, aunque está claro que antes de aplicar transformaciones el SNL A no es interno, inicialmente se marca como interno. En el momento en que el compilador observa que los bucles de A no podrán formar parte de un SNL interno se modifica el valor de este atributo. Hay dos posibles casos que nos van a llevar a marcar el SNL A como no interno:

1. No es posible fusionar ni fisionar dos SNLs de la lista de SNLs contenidos en A.

Figura 3.4: Combinación SNLs



2. Se produce una fisión parcial.

Tampoco podemos marcar como interno un SNL producto de la fusión o distribución de SNLs no internos. Esta situación se corrige a posteriori, justo antes de añadirlo al resultado.

Dados los SNLs:

> B, cuyo bucle más externo es $wn1$,
>
> C, que sigue inmediatamente a B y cuyo bucle más externo es $wn2$, y
>
> A, cuyo bucle más interno es $parent\_wn$ y contiene $wn1$ y $wn2$ (ver figura 3.4),

para que el compilador intente aplicar fusión se deben dar las siguientes condiciones:

1. Se permite la fusión. Es decir, ésta no ha sido desactivada mediante una opción de compilación (flag *fusion* dentro del grupo de opciones de LNO).

2. B y C son ambos internos y también lo es A, o bien B y C son ambos no internos.

3. $wn1$ y $wn2$ no están afectados por directivas de compilación que prohiban la fusión.

4. Los SNLs B y C tienen profundidad mayor que cero ($min\_snl\_level > 0$).

5. Los bucles $wn1$ y $wn2$ no contienen directivas de paralelización.

6. Si los bucles contienen directivas que inhiben la paralelización, éstas deben aparecer en ambos o en ninguno de los dos.

7. Finalmente, sólo se permite fusionar un bucle paralelizable con uno no paralelizable si ambos están contenidos en un bucle paralelizable. El cálculo de si un bucle es paralelizable se realiza cuando se aplica auto-paralelización.

En cuanto a la fisión, éstas son las condiciones que se deben cumplir:

1. No se ha desactivado la fisión mediante una opción de compilación.
2. *parent_wn* no está afectado por una directiva que prohiba su fisión.
3. *parent_wn* no está afectado por directivas de paralelización.
4. B y C son de tipo interno o no interno. Los valores posibles para el tipo de un SNL son: interno, no interno, no SNL e inválido.
5. El valor de la variable $last\_fissioned\_level$[2] es distinto de 0 y B es interno, o bien C es interno, sin importar el valor de $last\_fissioned\_level$.
   Antes de entrar a tratar los SNLs de la lista se inicializa la variable $last\_fissioned\_level$ con el número de bucles de A. Esta variable se pone a cero cuando al llegar a la fisión nivel a nivel (ningún intento de transformación previo ha tenido éxito) ésta no se lleva a cabo, ya sea porque fracasa (no se consigue ni tan solo parcialmente) o porque se había decidido no intentar fisión.

Finalmente, si se determina que ambas transformaciones pueden ser aplicadas, preferiremos la fusión cuando se da alguna de estas condiciones:

- Estáticamente se da prioridad a la fusión.
- Fisión y fusión tienen la misma prioridad, y B y C tienen igual profundidad.

A priori, las condiciones “A no interno” y “$last\_fissioned\_level = 0$” podrían parecer equivalentes. Esto no es así porque tras una fisión parcial marcaríamos A como no interno y $last\_fissioned\_level$ sería distinto de 0.

3.2. Reordenación topológica

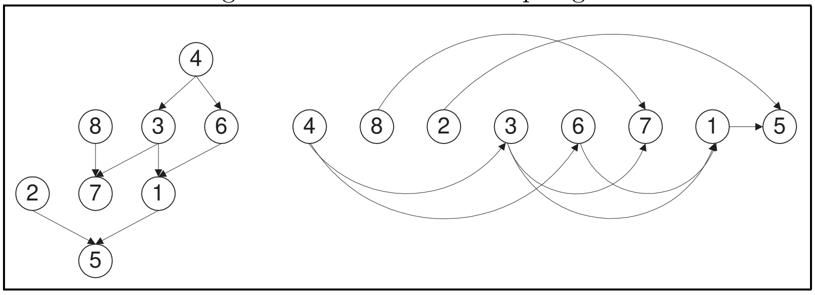
Durante la etapa de búsqueda de SNLNs, a la vez que se profundiza en una anidación de bucles, se reordenan las sentencias que éstos contienen. Esta reordenación nos será de ayuda en etapas posteriores, como por ejemplo, cuando queramos fisionar alguno de los bucles. Para explicar como se lleva a cabo necesitamos definir primero el concepto *Strongly Connected Component*:

Strongly Connected Component (SCC) es un subconjunto, S, de nodos de un grafo dirigido, tal que, cualquier nodo de S se puede alcanzar desde cualquier otro nodo de S. Además S debe ser maximal, es decir, no puede ser ningún subconjunto de un conjunto que cumpla la condición anterior.

Aplicando esta definición se genera un grafo dirigido acíclico a partir del grafo de niveles de dependencia (ver capítulo 2). En este nuevo grafo aparece un solo nodo para cada SCC. Hecho esto se procede a la reordenación topológica de los nodos del nuevo grafo, llamémosle G=(V,E), que consiste en generar una ordenación lineal de los vértices de V tal que para toda arista (i,j) de E, el vértice i aparezca antes que j. La figura 3.5 muestra un ejemplo. Una vez obtenida esta lista, se reconstruye el cuerpo del bucle colocando las sentencias en este orden, dentro de cada SCC se mantiene el orden de programa original. Esta reordenación se lleva a cabo como sigue:

[2] Para simplificar, la variable *last_fissioned_level* no aparece en el algoritmo 7.

Figura 3.5: Reordenación topológica



1. Generar grafo de niveles de dependencia (*sdg*).
2. Crear una copia del grafo de dependencias sobre *dep_g_p*. Este nuevo objeto, instancia de la clase SCC_DIRECTED_GRAPH16, ofrece operaciones para manipular SCCs.
3. Obtener SCCs presentes en *dep_g_p*.
4. Agrupar SCCs por niveles.
5. Para cada nivel

   Para cada SCC del nivel en curso

      Generar lista de sentencias correspondiente al SCC. La función devuelve una lista donde cada elemento representa un SCC.

La figura 3.6 muestra un ejemplo donde se reordenan las sentencias del bucle *J*. Éste contiene dos bucles, cuyas dependencias se muestran en el grafo adjunto. Después de aplicar condensación acíclica, que en este caso no reduce el número de nodos, se realiza la reordenación. El resultado final es que los bucles *L* y *K* intercambian su posición.

Nótese que el algoritmo 8, que aplica esta reordenación, es llamado en el algoritmo 5 cada vez que se encuentra un nuevo bucle. Antes de reordenar, el primer paso es generar un nuevo grafo de dependencias ignorando las dependencias acarreadas por aquellos bucles que ya hemos visitado. Esto va a hacer que cuando aparezcan ciclos con dependencias de distinto nivel, las sentencias se reordenen de modo que las dependencias acarreadas por los bucles más internos tengan origen en una sentencia que preceda al destino en orden de programa. Esta ordenación es deseable para facilitar la fusión parcial (ver sección 3.4).

3.3. Fusión

El código que implementa la fusión contempla la posibilidad que el incremento del bucle sea distinto de uno y hay código destinado a tratar los casos en que el incremento sea negativo. Probablemente esto se deba a motivos históricos, ya que los bucles que recibe el LNO están normalizados y por tanto, gran parte de este código está tratando un caso demasiado general mientras que otras partes jamás se ejecutarán.

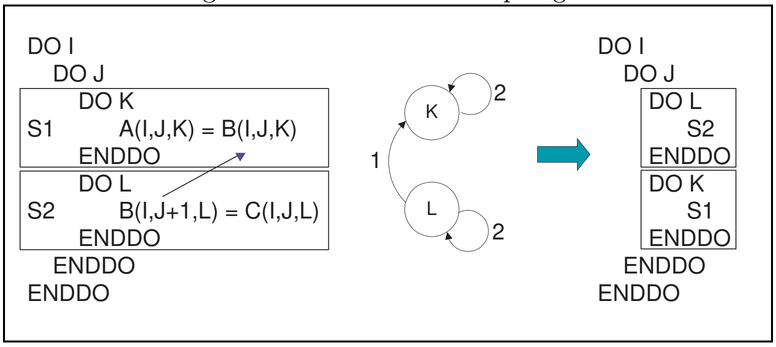
Algoritmo 8: Reordenación topológica
Entrada: Bucle a reordenar (*in_loop*) y el grafo de dependencias entre arrays (*adg*).
Salida: Lista de listas (una por SCC) con las instrucciones del bucle reordenadas.

sdg = Crear_Grafo_Niveles_Dependencia(in_loop, adg)
foreach sentencia s en in_loop
    Añadir vértice a dep_g_p
foreach sentencia s en in_loop
    foreach arista de salida a de s en sdg
        $dest$ = Destino(a)
        Añadir arista $s \rightarrow dest$ a dep_g_p
ac_g = Condensacion_Aciclica(dep_g_p)
foreach SCC scc en ac_g
    l = Nivel(scc)
    Añadir scc a $scc\_group[l]$
$i = 0$
foreach nivel l de ac_g
    foreach SCC scc en $scc\_group[l]$
        foreach sentencia s en scc
            Añadir s a $sentencias[i]$
        $i = i + 1$

Llamemos A y B a las dos anidaciones de bucles que queremos fusionar. Nos referiremos a cada uno de los bucles utilizando la letra que indica la anidación seguida de un subíndice. Si A está formada por n bucles, los numeraremos de 0 a $n-1$, siendo A_0 el bucle más externo y A_{n-1} el más interno. $fusion\_level$ indica el número de niveles a fusionar. Cuando se aplica fusión para obtener SNLNs maximales $fusion\_level = \min(m, n)$, siendo m y n el número de bucles que forman A y B respectivamente. Los pasos seguidos son:

1. Comprobaciones sobre A_0 y B_0:
   - Las variables índice de los bucles son del mismo tipo.
   - No hay directivas que prohiben la fusión.
   - Los bucles son válidos (Do_Loop_Is_Good) y no tienen llamadas ni *gotos*.
   - No hay sentencias entre A_0 y B_0.
2. Estandarizar los límites superiores de los bucles (mediante una llamada a Upper_Bound_Standardize() definida en be/lno/lnoutils.cxx):
   - En la expresión que indica la condición de finalización, aislar el índice del bucle a la izquierda.
   - Si el operador relacional es distinto de $<$ y de $\leq$, se aborta la fusión.
   - Si el operador relacional es $<$, lo convertimos en $\leq$ restando uno a la parte derecha.
3. Si la variable índice está viva al final del bucle:

Figura 3.6: Reordenación topológica



Generar una sentencia de asignación con el valor final de la variable igual a máx(límite inferior, límite superior). Se implementa en Finalize_Index_Variable().

Actualizar las estructuras de datos que almacenan la información de definición y uso, sustituyendo todos los usos de la variable índice ubicados fuera del bucle por usos de la definición que acabamos de generar.

Renombrar la variable índice dentro del cuerpo del bucle (función scalar_rename()).

En el caso de A_0 la asignación del valor final se colocará antes del bucle, en el caso de B_0, a continuación.

4. Comprobar el signo de las ocurrencias de la variable índice en ambos bucles:

   Si en uno de los bucles todas las ocurrencias están acompañadas por un signo negativo, y en el otro bucle esto no es así, sustituiremos la variable índice, supongamos que es i, en uno de los bucles por la expresión $limite\_inferior + limite\_superior - i$.
   En primer lugar se intenta realizar esta sustitución en el bucle donde se utiliza el índice con signo negativo. Si el cambio no es legal, se intenta en el otro bucle. La inicialización, la condición de finalización y el incremento del bucle no se modifican. El objetivo de esta transformación puede ser intentar que los elementos de los arrays se accedan en el mismo orden en ambos bucles. De todos modos, si en un bucle aparecen ocurrencias del índice afectadas por signos distintos, presumiblemente realizar este cambio no va a mejorar la situación. Además tampoco se analiza a qué arrays se accede ni para qué, por tanto, es posible que no exista ninguna dependencia entre ambos bucles.

5. Para cada pareja A_i, B_i, desde i=1 hasta $fusion\_level$-1:

   Comprobar que no haya directivas que prohiban la fusión.

   Repetir punto 4.

   Repetir punto 2.

Repetir punto 3.

Mover sentencias posteriores a A_i delante de A_i. Si no es posible, devolver *probar_nivel_a_nivel*[3].

Mover sentencias anteriores a B_i detrás de B_i. Si no es posible, devolver *probar_nivel_a_nivel*.

6. Determinación de los parámetros con los que se realizará la fusión:

   prolog_out Número de iteraciones, extraídas de A_{n-1}, que conformarán el prólogo.

   epilog_out Número de iteraciones del epílogo. Pueden proceder a A_{n-1} o B_{n-1}.

   epilog_loop_out Indica de qué bucle se obtienen las iteraciones del epílogo.

   offset Vector que contiene, para cada pareja de bucles (A_i, B_i), el desplazamiento que debemos aplicar a la variable de iteración en las sentencias correspondientes al bucle B_i.

7. Actualizar código.

La función Fuse() (fusion.cxx) implementa los pasos que acabamos de listar. Esta función recibe como parámetros los bucles más externos de las anidaciones de bucles, el número de niveles a fusionar, el número de iteraciones que se pueden extraer (*peeling*) de los bucles más internos y un booleano indicando si dichas iteraciones se deben desenrollar.

3.3.1. Cálculo de los parámetros de la fusión

Para referirnos a cada uno de los bucles que forman las anidaciones que queremos fusionar utilizaremos la misma nomenclatura que en el apartado anterior.

Si fusionamos varios niveles a la vez, sólo se permite *peeling* en el nivel más interno. Vamos a calcular un *offset* para cada nivel, pero solamente calcularemos el número de iteraciones del prólogo y el epílogo para el nivel más interno.

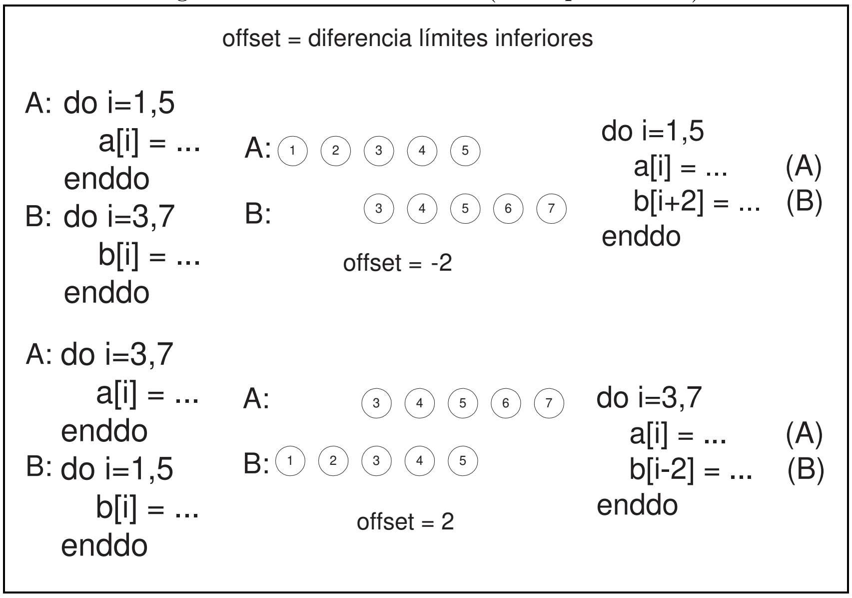
De forma resumida, el modo como se calculan estos parámetros es el siguiente:

offsets Antes de poder determinar los *offsets* debemos calcular, para cada nivel:

1. Diferencia entre los límites inferiores de A_i y B_i (lb_diff_i).
2. La distancia de las dependencias desde sentencias de B_i a sentencias de A_i, que se calcula como si las sentencias de A_i y B_i formaran parte de un mismo bucle. Sólo nos interesa la distancia máxima, que guardamos en $dist_i$.

[3]No se podrá aplicar fusión en este nivel de profundidad pero podría plantearse una aplicación parcial de la transformación y, por este motivo, se devuelve *probar_nivel_a_nivel*. De todos modos, en el contexto de la generación de SNLNs la fusión debe incluir todos los bucles del SNL.

Figura 3.7: Fusión de bucles (sin dependencias)

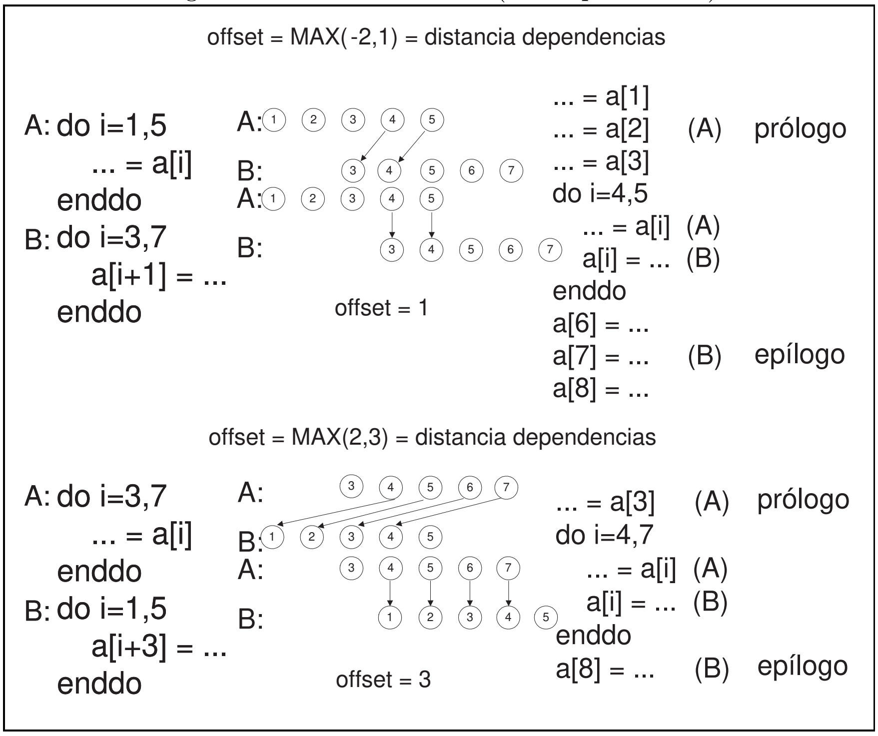


Conocidos estos valores, $offset_i = MAX(lb\_diff_i, dist_i)$. Si es necesario, $offset_i$ se va a corregir para que la distancia entre el límite inferior del epílogo y el límite inferior del bucle principal sea múltiple del *step*. Una vez calculados los *offsets*, se comprueba si hay que hacer *peeling* en algún nivel intermedio. Si esto es así se aborta la fusión.

prólogo y epílogo Si no existen dependencias, la distancia que separa los límites inferiores de $A_{interno}$ y $B_{interno}$ se salva con el *offset*, que se aplica siempre al segundo bucle. A partir de ahí, si alguno de los dos bucles tiene más iteraciones que el otro, las iteraciones sobrantes se ejecutarán en el epílogo. La figura 3.7 muestra dos ejemplos de fusión de bucles cuando no existen dependencias. A la izquierda podemos ver el código original. En el centro, un pequeño diagrama donde cada burbuja representa una iteración de uno de los bucles. Junto al diagrama aparece el valor del *offset* y, a mano derecha, podemos ver el resultado de la aplicación de fusión. Los bucles de la figura tienen todos el mismo número de iteraciones y no es necesario ningún epílogo.

Si el valor del offset viene determinado por una dependencia ($dist_{interno} > lb\_diff_{interno}$), entonces es posible que se tengan que ejecutar algunas de las iteraciones de $A_{interno}$ en el prólogo. El número de iteraciones del epílogo y el bucle de procedencia de éstas dependerá del número de iteraciones restantes de A y del número de iteraciones de B. En la figura 3.8 podemos ver dos ejemplos donde el valor del *offset* viene determinado por la distancia de las dependencias. En esta figura añadimos un segundo diagrama para cada ejemplo mostrando como deben alinearse las iteraciones de los bucles para respetar las dependencias. Observamos que en ambos casos se generarán prólogo y epílogo.

Figura 3.8: Fusión de bucles (con dependencias)



Nótese que el prólogo sólo puede contener iteraciones de A, cosa que es de esperar si tenemos en cuenta el sentido que pueden tomar las dependencias antes y después de la fusión.

Más detalladamente, los pasos que se llevan a cabo son:

1. Comprobaciones sobre A_0 y B_0:

   A_0 y B_0 son bucles adyacentes.

   No se utilizan las funcionas MAX o MIN para especificar los límites superiores e inferiores de los bucles.

   Los bucles son válidos (Do_Loop_Is_Good) y no tienen llamadas ni *gotos*.

2. Para cada pareja A_i, B_i, desde i=1 hasta $fusion\_level$-1:

   Los bucles están simplemente anidados. Esto es, bucle X_i es el único bucle contenido en X_{i-1} y no está dentro de una cláusula *if*.

   No se utilizan las funciones MAX o MIN para especificar los límites superiores e inferiores de los bucles.

3. Para cada pareja A_i, B_i, desde i=0 hasta $fusion\_level$-1:

Ambos bucles tienen el mismo incremento (diferencia del *step* = 0). Al estar los bucles normalizados esta condición siempre se cumplirá, puesto que el incremento de ambos será uno.

Calcular diferencia entre límites inferiores[4] ($lb\_diff = L1 - L2$) y comprobar que es constante.

Calcular diferencia entre límites superiores ($ub\_diff = U1 - U2$) y comprobar que es constante.

4. Análisis de dependencias (ver sección 3.3.3):

   Se comprueba que no existan dependencias entre escalares que impidan la fusión. Para los accesos a arrays, se calcula la distancia máxima de dependencia para cada nivel de profundidad ($dist$).

5. Para cada pareja A_i, B_i, desde i=0 hasta $fusion\_level$-1, cálculo del *offset*:

   $offset_i = MAX(dist_i, lb\_diff_i)$

   Corregimos el *offset* para que la distancia entre el límite inferior del prólogo y el límite inferior del bucle principal sea múltiple del *step*. Sea $r = (offset_i - lb\_diff_i)$ mod $step_i$, si $r \neq 0$, entonces $offset_i = offset_i + r$.

   Si $offset_i > dist_i$ se garantiza que también se respentan las dependencias originales para todo j tal que $j > i$ (bucles más internos). Modificamos el vector $dist$ para reflejar este hecho.

   Si $offset_i \neq lb\_diff_i$ ó $lb\_diff_i \neq ub\_diff_i$ será necesario aplicar peeling.

   Si el número de iteraciones a extraer supera el umbral permitido abortamos la fusión.

   Si estamos fusionando varios niveles a la vez y es necesario hacer peeling en el nivel más externo devolveremos *probar_nivel_a_nivel*, no sin antes comprobar el resto de niveles.

   Si es necesario hacer *peeling* en un nivel intermedio abortamos la fusión.

   Finalmente corregimos el valor de lb_diff_j y ub_diff_j para todo j tal que $j > i$ del siguiente modo:

   $lb\_diff_j = lb\_diff_j + offset_i * coeff_j(i)$

   donde $coeff_j(i)$ corresponde al coeficiente que acompaña al índice i en los límites de los bucles del nivel j. Este coeficiente tiene que ser

[4]Los límites inferior y superior se representan mediante una desigualdad. Por ejemplo en "DO i=1,10", el límite inferior se expresa $i \geq 1$ y el límite superior $i \leq 10$. Al guardar estos límites, éstos deben seguir el formato ... + $Ax \leq b$, donde x es la variable índice del bucle, A es el coeficiente que la acompaña y b es un valor constante. Por tanto el límite inferior del ejemplo se convierte en $-1i \leq -1$. Al calcular la diferencia entre límites inferiores y superiores se contempla la posibilidad que A tome un valor distinto de ± 1. Por este motivo en el código se divide el resultado de la resta por A. Debemos tener en cuenta que, puesto que la diferencia tiene que ser constante (en caso contrario se desestima la fusión), después de restar, todos los coeficientes de las distintas variables van a ser 0. Esta condición también nos garantiza que A va a tener el mismo valor en ambos bucles. Normalmente esta división sólo corregirá el signo del resultado.

el mismo en ambos bucles porque si al calcular la diferencia no se anulan se aborta la fusión. Si la variable i no aparece en la definición de los límites de los bucles del nivel j, su coeficiente será 0 y, en consecuencia, el *offset* del bucle i no tendrá ninguna incidencia en el cálculo de lb_diff_j y ub_diff_j.

6. Cálculo del prólogo y el epílogo del bucle más interno:
La figura 3.9 muestra como van a quedar el prólogo, el bucle principal y el epílogo después de aplicar fusión. Aunque el prólogo y el epílogo aparecen como bucles, en la práctica éstos se van a desenrollar.
Para obtener el número de iteraciones del prólogo y el epílogo se realiza el siguiente cálculo:

$$iteraciones = \frac{limite\_superior - limite\_inferior}{step} + 1$$

A partir de aquí se deben contemplar las distintas posibilidades que aparecen en la figura 3.9. Por ejemplo, veamos como se calcula el número de iteraciones del prólogo:

Número de iteraciones en el prólogo:

$$prologo = \frac{L2 + offset - step - L1}{step} + 1$$

Teniendo en cuenta que $-L1 + L2 = -(L1 - L2) = -lb\_diff$, reescribimos la expresión anterior como

$$prologo = \frac{-lb\_diff + offset - step}{step} + 1$$

Si el número de iteraciones en el prólogo es superior al número de iteraciones en el primer bucle, entonces asignaremos al prólogo el número de iteraciones en el primer bucle:

$$prologo = \frac{U1' - L1}{step} + 1$$

Si no es posible calcular el número de iteraciones de los bucles la comprobación anterior no va a ser factible, existiendo la posibilidad que el número de iteraciones en el prólogo sea superior al número de iteraciones del primer bucle. Esta situación no va a provocar la generación de código incorrecto porque cada iteración que se ubique en el prólogo estará protegida por una cláusula *if* que garantizará que el valor del índice para esa iteración no supere el límite superior del primer bucle.
Por último, si $L2 + offset - step < L1$, el número de iteraciones en el prólogo será 0.

De forma parecida podemos calcular el epílogo.

3.3.2. Fusión y generación del prólogo y el epílogo

Conocidos los *offsets*, el prólogo y el epílogo, se lleva a cabo la fusión transformando la representación en WHIRL del programa.

Figura 3.9: Cálculo parámetros fusión

prólogo	DO $i = L1, MIN(U1', L2 + offset - step), step$
principal	DO $i = L2 + offset, MIN(U1', U2' + offset), step$
epílogo	si $U1' > U2' + offset$ DO $i = MAX(L1, U2' + offset + step), U1', step$ si $U1' \leq U2' + offset$ DO $i = MAX(L2 + offset, U1' + step), U2' + offset, step$

donde:

Lx Límite inferior bucle x.

Ux' Límite superior bucle x, ajustado para que se mantenga alineado con el incremento del bucle.

step Incremento del bucle.

offset Desplazamiento que aplicamos al índice del segundo bucle.

1. En un primer paso, para cada nivel, se sustituyen las ocurrencias de los índices de ambos bucles por una nueva variable común. Se desplaza el segundo bucle aplicando el *offset* a sus límites y se modifican los usos del índice para compensar este desplazamiento.

2. A continuación se lleva a cabo la fusión en todos los niveles excepto en el más interno, que se trata separadamente. Recorriendo los niveles desde dentro hacia fuera, para cada nivel, movemos las sentencias del segundo bucle al cuerpo del primero.

3. Tratamos el nivel más interno:

   a) Extraemos del primer bucle las iteraciones que conformarán el prólogo (ver figura 3.10).

      Modificamos el límite inferior del bucle.

      Extraemos cada una de las iteraciones del prólogo, empezando por la primera. Cada una de ellas se protege con un condicional, cuya condición se intenta evaluar en tiempo de compilación. Si esta evaluación es posible se elimina el condicional.

   b) Extraemos del primer o segundo bucle (según corresponda) las iteraciones que conformarán el epílogo (ver figura 3.11).

      Se genera la expresión en WHIRL correspondiente a $(n - m)$ mod *step*.

      Se modifica el límite superior del bucle restando $(n-m)$ mod *step* en la parte derecha de la condición de finalización.

      Sacamos las iteraciones del epílogo, empezando por la última, protegiendo cada iteración con un condicional. Se intenta evaluar la condición y eliminar el condicional.

Figura 3.10: Extracción de iteraciones en el prólogo

```
DO i=m,n,step
   S(i)
ENDDO
```

→

```
IF (m<=n)
   S(m)
ENDIF
IF (m+step<=n)
   S(m+step)
ENDIF
...
IF (m+step*(iter_count-1) <=n)
   S(m+step*(iter_count-1))
ENDIF
DO i=m+step*iter_count, n, step
   S(i)
ENDDO
```

Figura 3.11: Extracción de iteraciones en el epílogo

```
DO i=m,n,step
   S(i)
ENDDO
```

→

```
DO i=m, n-step*iter_count-(n-m)%step, step
   S(i)
ENDDO
IF (m<=n-step*(iter_count-1)-(n-m)%step)
   S(n-step*(iter_count-1)-(n-m)%step)
ENDIF
IF (m<=n-step*(iter_count-2)-(n-m)%step)
   S(n-step*(iter_count-2)-(n-m)%step)
ENDIF
...
IF (m<=n-(n-m)%step)
   S(n-(n-m)%step)
ENDIF
```

Se modifica el límite superior del bucle sumando *iter_count*[5] en la parte izquierda de la condición de finalización.

c) Movemos las sentencias del cuerpo del segundo bucle al cuerpo del primero.

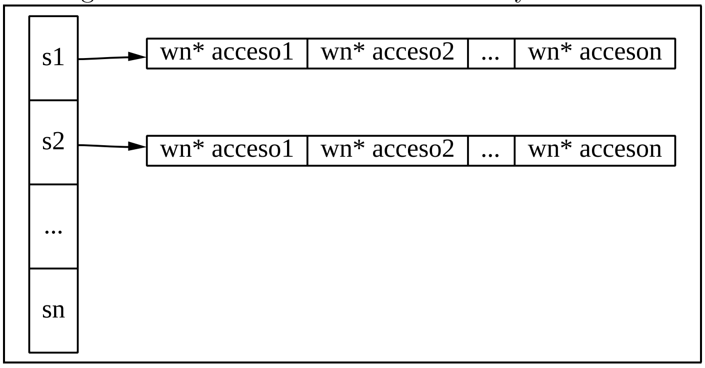
d) Si el cuerpo del bucle resultante de la fusión nunca se va a ejecutar lo eliminamos.

3.3.3. Análisis de dependencias

Para que la fusión sea legal se debe garantizar que después de su aplicación se mantengan las dependencias. Dados dos bucles al mismo nivel, cualquier dependencia entre ellos, salvo aquellas acarreadas por un bucle externo, debe ir desde el primero hacia el segundo. Si después de aplicar fusión aparecen dependencias

[5]Esto es correcto porque los bucles están normalizados. Si no fuera así se debería sumar $step * iter\_count$. Resulta curioso que en este punto se realice esta suposición cuando en el resto del código se trata un caso más general.

Figura 3.12: Formato listas de lecturas y escrituras



que van desde instrucciones procedentes del segundo bucle a instrucciones del primero, es posible que se esté modificando la semántica del programa y, en consecuencia, se debe abortar la fusión.

Dependencias entre escalares

En primer lugar se analizan las dependencias entre escalares. Para ello clasificaremos las referencias a variables de este tipo en tres listas:

scalar_writes Cada elemento representa un símbolo y contiene su lista de accesos de escritura.

scalar_reads Cada elemento representa un símbolo y contiene su lista de accesos de lectura.

params Parámetros encontrados que se pasan por referencia.

La figura 3.12 muestra el aspecto que tienen la lista de lecturas y escrituras. En la figura, la lista $(s1, s2, \ldots, sn)$ representa los distintos símbolos sobre los que se escribe o se lee dependiendo de la lista que se esté elaborando. Para cada símbolo se genera una lista de referencias, donde cada referencia se guarda como un apuntador al nodo de la representación en whirl que la representa.

Nombramos también la lista *params* para mantenernos fieles al código del compilador, pero puesto que un bucle con llamadas no puede formar parte de un SNL, parece que no sería necesaria.

Una vez elaboradas estas listas, se cruzan los elementos de las de un SNL con los elementos de las del otro y se comprueba si existe alguna dependencia que impida la fusión:

1. Para cada uno de los SNLs a fusionar, generar listas de referencias a memoria.

2. scalar_writes1 $\leftrightarrow$ scalar_writes2 (cruzamos las escrituras del primer SNL con las del segundo)
   Para cada pareja (s1, s2) posible tomando un elemento de cada lista:

   Si acceden a direcciones distintas, continuar con la siguiente pareja.

Si es posible que s1 y s2 representen la misma dirección, para cada pareja de referencias:

Si no hay reducción [6] o el tipo de reducción de las dos referencias es distinto, acabamos de encontrar una dependencia que impide la fusión.

3. scalar_writes1 ↔ scalar_reads2
   Para cada pareja (s1, s2) posible tomando un elemento de cada lista:

   Proceder como en el caso anterior.

4. scalar_writes1 ↔ params2
   Para cada pareja (s1, param2) posible tomando un elemento de cada lista:

   Si existe la posibilidad que accedan a la misma dirección, abortar fusión.

5. Tratar el resto de combinaciones posibles de la misma manera:

   scalar_reads1 ↔ params2,

   scalar_writes2 ↔ scalar_reads1,

   scalar_writes2 ↔ params1 y

   scalar_readS2 ↔ params1

Dependencias entre arrays

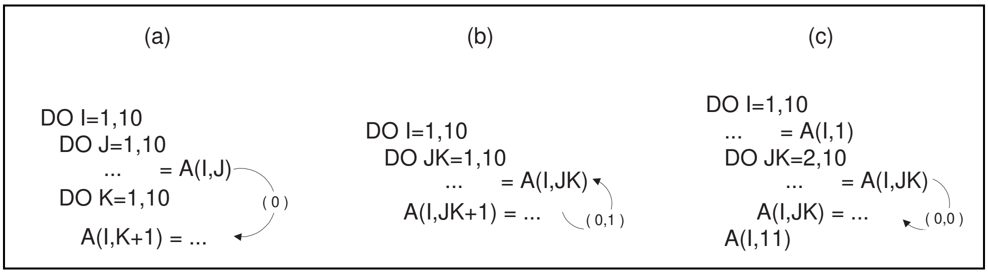
El siguiente paso consiste en estudiar las dependencias entre accesos a arrays. El objetivo de este análisis es determinar si la fusión va a introducir dependencias que modifiquen la semántica del programa. Para ello se van a calcular las dependencias entre las sentencias de los distintos bucles suponiendo que ya se ha llevado a cabo la fusión.

En el fragmento de programa de la figura 3.13 (a) aparece una dependencia entre la lectura de A en el bucle J y su asignación en el bucle K. Estos dos accesos tienen un solo bucle en común (bucle I) y por tanto el vector de dependencias tiene una sola dimensión. Puesto que la dependencia se produce dentro de una misma iteración del bucle I la distancia especificada por el vector de dependencias es 0. En la figura 3.13 (b) podemos ver lo que ocurriría si fusionáramos los bucles tal cual. Al aplicar esta transformación se está modificando el orden de los accesos a la variable A y, en este caso, esto provoca un cambio de sentido de la dependencia dando lugar a un código incorrecto. El vector de dependencias correspondiente es (0,1), indicando que la dependencia se produce dentro de una misma iteración del bucle I y en iteraciones consecutivas del nuevo bucle JK. En la figura 3.13 (c) vemos como podemos corregir este problema mediante la aplicación de un desplazamiento (offset) y la extracción de iteraciones (peeling).

Para calcular el desplazamiento que se debe aplicar es necesario conocer las dependencias producidas por la fusión. Estas dependencias se calculan fijando los índices de los bucles externos, de forma que en los vectores de dependencias sólo aparezcan aquellas dependencias acarreadas por los bucles que se van a

[6] El gestor de reducciones sólo se crea si se va a permitir aplicar asociatividad. Por defecto cuando se compila con -O3 sí se permite reorganizar las operaciones aplicando esta propiedad.

Figura 3.13: Dependencias fusión



fusionar. Volviendo a la figura 3.13 (b) al realizar el análisis de dependencias se calcularía un vector de una sola dimensión correspondiente al bucle JK. El cálculo de las dependencias se realizaría basándonos en las siguientes ecuaciones.

$$I = I$$
$$JK_2 = JK_1 + 1$$

En la primera ecuación no aparecen subíndices porque hemos fijado el valor de la variable I. Esta ecuación no nos aporta ninguna información. Resolviendo la segunda ecuación obtenemos la distancia de las dependencias acarreadas por el bucle JK. El vector de dependencias que se obtiene es (1). Si aparecieran más dependencias sólo nos interesaría la más restrictiva y sería suficiente quedarnos con el valor máximo. Si hubiera más bucles a fusionar los vectores de dependencias tendrían mayor número de dimensiones y tendríamos que guardar un máximo por cada dimensión.

En la figura 3.14 vemos otro ejemplo. En este caso el cálculo de dependencias se realizaría partiendo de las siguientes ecuaciones:

$$I = I + 1$$
$$JK_2 = JK_1 + 1$$

Como en el caso anterior hemos fijado el valor de los índices de los bucles externos (la variable I). Vemos que el sistema de ecuaciones resultante es inconsistente puesto que no es posible satisfacer la primera ecuación. En este caso el cálculo de dependencias nos indicaría que, dentro de una misma iteración de I, las dos referencias de la matriz A son independientes. Por consiguiente, no hace falta aplicar ningún desplazamiento para que se respeten las dependencias originales.

A continuación se describe el algoritmo para calcular las distancias máximas de las dependencias que aparecen al fusionar. Como en el caso del análisis de dependencias entre escalares, empezaremos generando algunas listas con las referencias a memoria:

writes Cada elemento representa un array, identificado por su dirección base[7], y contiene su lista de accesos de escritura.

[7]En realidad se comparan punteros de entradas en la tabla de símbolos. Para obtener la dirección base del array se consulta el campo *base_idx* de su entrada en la tabla de símbolos. Si este campo contiene un apuntador a una entrada distinta el símbolo que estamos consultando es un alias del símbolo apuntado. Se recorre la cadena de apuntadores hasta llegar a un símbolo que apunte a sí mismo. El apuntador a este símbolo es lo que se devuelve como dirección base.

Figura 3.14: Dependencias fusión

Si el acceso al array se realiza mediante un apuntador se analizan las definiciones que alcanzan este uso del apuntador para intentar determinar su dirección base. Si no es posible, se le asigna la dirección base 0. Cuando se calculan las dependencias se considera que dos referencias de dirección base indeterminada acceden al mismo elemento.

reads Cada elemento representa un array y contiene su lista de accesos de lectura. Al igual que en el caso anterior, asignamos 0 a aquellos elementos cuya dirección base desconocemos.

1. Crear dos vectores de enteros (*max_distance*[] y *max_distance_tmp*[]) con tantos elementos como niveles de profundidad se vayan a fusionar.

2. Inicializar todas las entradas de *max_distance*[] a MIN_INT32.

3. Para cada uno de los SNL a fusionar, generar listas de referencias.

4. writes2 $\leftrightarrow$ writes1
   Inicializar todas las entradas de *max_distance_tmp*[] a MIN_INT32.
   Para cada pareja (v2, v1) posible tomando un elemento de cada lista:

   Si representan arrays distintos, continuar con la siguiente pareja.

   Si representan el mismo array, para cada pareja de referencias:

   Si las dos referencias corresponden al mismo tipo de reducción continuar con la siguiente pareja de referencias.

   En caso contrario, obtener los vectores de dependencias entre estas dos referencias. Calcular las distancias máximas de las dependencias y actualizar *max_distance_tmp*[] con los valores máximos.

   Asignar a cada entrada de *max_distance*[] el máximo entre su valor actual y el valor correspondiente en *max_distance_tmp*[].

5. Tratar de la misma manera las combinaciones:

   writes2 $\leftrightarrow$ reads1 y

   reads2 $\leftrightarrow$ writes1

Distancia máxima de dependencia entre dos referencias a memoria

Una entrada de un vector de dependencias puede contener una distancia, puede indicar una dirección o simplemente informar que existe una dependencia sin dar más detalles. Si existen dependencias entre dos referencias a un elemento de un array:

Para cada vector de dependencias:

Para cada dimensión i del vector de dependencias:

- Si la dimensión i del vector de dependencias no representa una distancia y corresponde al nivel más externo ($i = 0$), no es posible fusionar.
- Si no representa una distancia y corresponde a un nivel interno, entonces:

  Empezando por $i - 1$ hasta 0, recorremos *max_distance_tmp*[] hasta encontrar una entrada j tal que *max_distance_tmp*[j] sea menor que 0.
  Si la encontramos, seguro que en ese nivel no hay ninguna restricción respecto a la distancia. Asignamos el step del nivel j a la entrada *max_distance_tmp*[j].
  Si no aparece incrementamos *max_distance_tmp*[$i - 1$] con el valor del step del nivel $i - 1$.
  De este modo, incrementando la distancia de un nivel más externo, garantizamos que podemos ignorar la dependencia cuya distancia desconocemos.
- Si representa una distancia, actualizamos la distancia máxima calculada hasta el momento con la mayor de ambas.

3.4. Fisión

La fisión es la otra transformación posible para obtener SNLNs más profundos. En be/lno/fission.cxx se encuentran los procedimientos que la implementan. A continuación enumeramos los pasos que se realizan tomando como referencia el ejemplo de la figura 3.15:

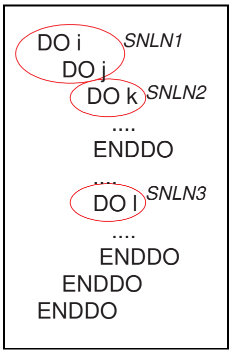
1. Generamos dos listas con las sentencias del bucle más interno de *SNLN1*:

   lista 1: Desde la primera sentencia hasta *SNLN2*.

   lista 2: Desde *SNLN3* hasta el final.

   Observamos que en medio puede haber otras instrucciones que no se asignan a ninguna de las dos listas.
2. Se construye el grafo de niveles de dependencia.
3. Analizamos el cuerpo del bucle i en búsqueda de alguna dependencia que impida la fisión, tal como se indica en la sección 3.4.1.
4. Aplicamos el algoritmo de distribución pasándole el bucle j y las listas que acabamos de generar (sección 3.4.2).

Figura 3.15: SNLNs a distribuir



5. Si el algoritmo de distribución ha devuelto más de dos bucles, agrupamos desde el segundo hasta el final en uno solo.

6. Actualizamos el código.

3.4.1. Análisis de dependencias

Durante la fase de búsqueda de SNLNs se lleva a cabo una reordenación de las sentencias de los bucles para que, si entre dos sentencias existe una dependencia, el origen de la dependencia preceda al destino. En el caso en que las dependencias formen un ciclo, se reordenan aplicando el criterio anterior a la dependencia del ciclo acarreada por un bucle más interno. Con este paso previo,la presencia de una dependencia de sentido opuesto al orden de programa garantiza la existencia de un ciclo, que se debe preservar al llevar a cabo la fisión.

Aplicar distribución puede suponer la fisión de varios bucles imbricados. El bucle más interno se va a tratar en Form_Loops (ver apartado 3.4.2), al resto de bucles se les aplica el siguiente algoritmo para comprobar la legalidad de la fisión. Tratándolos desde dentro hacia fuera, para cada bucle:

1. Se elabora una lista con todas las sentencias desde el inicio del bucle hasta el bucle interno que se va a fisionar (incluido este último).

2. Empezando por el bucle interno, que es la última sentencia visitada en el punto anterior, y para cada sentencia que sigue hasta el final del bucle que estamos tratando:

   a) Obtener el vértice correspondiente a la sentencia dentro del grafo de niveles de dependencia.

   b) Obtener cada una de las aristas (dependencias) de salida de este vértice.

   c) Si alguno de los vértices destino corresponde a una de las sentencias contenidas en la lista del punto 1 acabamos de detectar un ciclo que impide la fisión. Existe un caso especial, que es la sentencia que representa al bucle interno, que es el único elemento de la lista del punto 1 que se va a tratar en el punto 2. Ésta sentencia puede tener una arista hacia sí misma sin que ello impida la fisión.

Como ya se ha comentado, esta comprobación se aplica a todos los bucles del SNLN que pretendemos fisionar excepto al más interno, que se trata dentro del algoritmo de distribución. Siguiendo con el ejemplo de la figura 3.15, aplicaríamos esta comprobación sólo al bucle i, siendo j el bucle al que nos referimos como bucle interno. Si i estuviera contenido en un bucle más externo, digamos h, repetiríamos el proceso con h, pasando i a ser el bucle interno.

3.4.2. Algoritmo de distribución

Form_Loops() es la función del compilador que implementa el algoritmo de distribución, y podemos encontrarla en be/lno/fission.cxx. Esta función se utiliza tanto para reordenar la sentencias del cuerpo de un bucle (ver sección 3.2) como para distribuir éstas en distintos bucles. La tarea desempeñada depende de los parámetros de entrada. En lo que a distribución se refiere, Form_Loops va a intentar generar tantos bucles como sea posible, es decir, uno para cada SCC. Durante este proceso no se reordenan las sentencias del bucle, se confía en que se ha producido una reordenación previa y se hace uso de las dependencias hacia atrás para detectar SCCs. Veamos como se realiza:

1. Hacemos una copia del grafo de niveles de dependencia sobre *dep_g_p*, eliminando aquellas dependencias que no nos van a afectar porque son acarreadas por algún bucle externo que no pretendemos fisionar.

2. Añadimos a *dep_g_p* falsas dependencias entre las sentencias que queremos destinar al mismo bucle, convirtiéndolas en un SCC.

3. Creamos una lista de instrucciones para cada SCC basándonos en las dependencias hacia atrás para determinar su alcance. Cada una de estas listas corresponde a un bucle.

El algoritmo 9 muestra estos pasos con más detalle.

Algoritmo 9: Distribución de bucles
Entrada: Bucle a fisionar (*in_loop*), número de bucles anidados que se van a fisionar (*fission_level*), dos listas de sentencias (*stl_1*, *stl_2*) y el grafo niveles de dependencia (*sdg*).
Salida: Lista de listas de instrucciones. Cada lista corresponde a un bucle y contiene las instrucciones que le han sido asignadas después de llevar a cabo la distribución.

```
foreach sentencia s en in_loop
    Añadir vértice a dep_g_p
foreach sentencia s en in_loop
    foreach arista de salida a de s en sdg
        if Nivel(a) > Profundidad(in_loop) + 1 - fission_level
            dest = Destino(a)
            Añadir arista s → dest a dep_g_p
foreach pareja de sentencias consecutivas (s1, s2) en stl_1
    Añadir arista s1 → s2 a dep_g_p
    Añadir arista s2 → s1 a dep_g_p
foreach pareja de sentencias consecutivas (s1, s2) en stl_2
    Añadir arista s1 → s2 a dep_g_p
    Añadir arista s2 → s1 a dep_g_p
i = 0
begin_stmt = primera sentencia de in_loop
while begin_stmt ≠ NULL
    v = begin_stmt
    end_v = begin_stmt
    while vértice en curso v ≤ vértice final end_v
        foreach arista de entrada a de v en dep_g_p
            if Origen(a) > end_v
                end_v = Origen(a)
        Añadir v a loop[i]
        v = siguiente sentencia
    i = i + 1
    begin_stmt = v
```

3.5. Ejemplo

Hasta el momento hemos descrito la primera etapa de la fase de optimización de bucles, que consiste en la identificación de SNLs y su posterior transformación, mediante fusión y distribución, para obtener SNLs de mayor profundidad. Este proceso es llevado a cabo por una función recursiva en la que podemos distinguir tres partes:

1. En primer lugar se profundiza en una anidación de bucles para obtener un SNL. Cada vez que encontramos un nuevo bucle se aplica una reordenación topológica.
   En el ejemplo de la figura 3.16 durante este paso se detecta el SNL *S1*

Figura 3.16: Pasos 1 y 2

```
DO J=1,N                    S1={J}
   DO I=1, N
      C1(I,J)=0
      C2(I,J)=0
   ENDDO
   DO L=1,N
      DO K=1,N
         C1(L,J)=C1(L,J)+A(L,K)*B(K,J)
      ENDDO
   ENDDO
   DO P=1,N
      DO Q=1,N
         C2(P,J)=C2(P,J)+A(P,Q)*D(Q,J)
      ENDDO
   ENDDO
ENDDO
```

```
DO J=1,N
DO I=1, N                   S2={I}
   C1(I,J)=0
   C2(I,J)=0
ENDDO
DO L=1,N                    S3={L,K}
   DO K=1,N
      C1(L,J)=C1(L,J)+A(L,K)*B(K,J)
   ENDDO
ENDDO
DO P=1,N                    S4={P,Q}
   DO Q=1,N
      C2(P,J)=C2(P,J)+A(P,Q)*D(Q,J)
   ENDDO
ENDDO
ENDDO
```

formado por el bucle *J*.

2. A continuación se analiza el cuerpo del bucle más interno del SNL encontrado y se genera una lista con los SNLNs que éste contiene. Los elementos de esta lista se obtienen mediante llamadas recursivas.
   En la figura 3.16 esta lista contiene los SNLs *S2*, *S3* y *S4*.

3. Finalmente aplicamos fusión y/o fisión para obtener SNLs de mayor profundidad. En este punto será necesario analizar las dependencias para asegurar la legalidad de la transformación.
   Siguiendo con el ejemplo, en primer lugar se distribuyen los SNLs *S2* y *S3* (ver figura 3.17). En este caso se prefiere la fisión por delante de la fusión porque *S2* y *S3* tienen distinta profundidad. Hecho esto tenemos por un lado el nuevo SNL *S12* y por el otro lado el resto de bucles, que debemos seguir tratando.
   Para terminar, se fusionan los SNLs *S3* y *S4*, que junto al bucle *J* forman un nuevo SNL de profundidad 3.

3.6. Fusión de bucles externos

Una vez generados los SNLNs de profundidad máxima y antes de empezar a optimizar el código, se lleva a cabo una última transformación: la fusión de bucles externos. La fusión de bucles externos recorre el cuerpo de una función y decide si se debe aplicar fusión cuando detecta SNLs adyacentes. El objetivo de esta transformación es mejorar la localidad de los accesos a arrays.

3.6.1. Recorrido de la función

Outer_Loop_Fusion_Walk(), definida en /be/lno/outer.cxx, realiza un recorrido en profundidad de la función que estamos tratando. Cuando localiza un bucle existen dos posibilidades:

1. Se trata del bucle más externo de un SNL. Hay que comprobar si a continuación aparece otro SNL y decidir si se va a intentar fusionarlos. No

Figura 3.17: Paso 3a

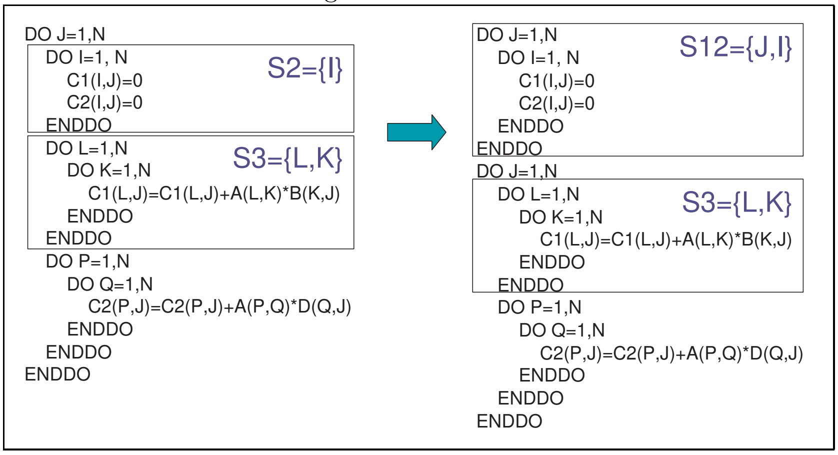
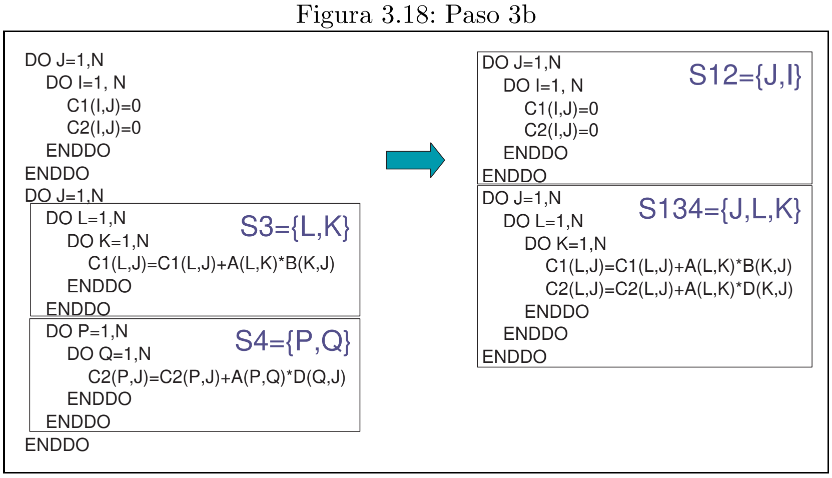


Figura 3.18: Paso 3b



vamos a profundizar en los SNLs encontrados porque éstos ya han sido tratados en la etapa anterior (ver algoritmo de la sección 3.1).

2. El bucle no forma parte de ningún SNL. En este caso se procede a buscar SNLs adyacentes en el cuerpo del bucle mediante llamadas recursivas.

Durante la etapa de fusión de bucles externos hay una lista de SNLs que se va pasando de una función a otra y que hay que mantener actualizada. Cada entrada de la lista contiene información relativa al SNL como cuál es su bucle más externo, la profundidad del SNL y el tipo de SNL (interno, no interno, inválido).

Otro aspecto que se tiene en cuenta al realizar el recorrido es que después de fusionar pueden aparecer nuevos bucles debido a la aplicación de peeling. Estos nuevos bucles también deben ser tratados.

3.6.2. Fusión de bucles externos

Una vez encontrados dos SNLs adyacentes el compilador chequea una serie de condiciones y decide si va a intentar fusionarlos. Concretamente se van a realizar las siguientes comprobaciones:

1. No hay directivas que impidan la fusión.
2. Los límites de los bucles no son *Too_Messy*. Dicho de otro modo, se pueden representar mediante un access array (ver sección 2.2.1).
3. Los bucles son válidos (Do_Loop_Is_Good) y no contienen llamadas ni *gotos*.
4. Si los SNLs a fusionar tienen distintos niveles de profundidad, debemos verificar que LNO_Fusion=2. El valor de esta variable se puede modificar mediante el flag de compilación *-fusion*, siendo 1 su valor por defecto.
5. Los SNLs a fusionar deben ser ambos internos o ambos no internos. No se intenta fusionar SNLs de distinto tipo a no ser que LNO_Fusion sea igual a 2. Ver la definición de SNL interno en la sección 3.1.3.
6. No se intentará fusionar bucles paralelos con bucles que se deban ejecutar en serie.

El resto de comprobaciones analizan los accesos a arrays:

7. Si alguno de los SNLs no contiene referencias a arrays se desestima la fusión.
8. El número de referencias total en ambos SNLs no debe superar un límite, que viene determinado por la variable *OLF_size_upperbound*. El valor de esta variable está prefijado a 100 en be/lno/outer.cxx.
9. Si Stride_One_Level(SNL1) $\neq$ Stride_One_Level(SNL2) y Fusion_Level<2 se aborta la fusión. Ver descripción de Stride_One_Level al final de esta sección.
10. Para realizar la última de las comprobaciones se genera un diccionario de mapeos, que contiene entradas del tipo <nombre de la variable, bit asignado>. Mediante esta estructura de datos se relaciona cada uno de los arrays instanciados en los SNLs con una posición dentro de un mapa de bits. El mapa de bits *bv1* representa el conjunto de los arrays instanciados en el primer SNL, *bv2* representa los arrays instanciados en el segundo SNL. Con la ayuda de estas estructuras de datos realizaremos la última de las comprobaciones:

    Si (LNO_Fusion < 2) y
    (nombres en SNL1+nombres en SNL2 > 15) y
    (2*(nombres en común) < (nombres sólo en SNL2)) y
    (2*(nombres en común) < (nombres sólo en SNL1))
    entonces se aborta la fusión.

    El 15 que aparece en las condiciones anteriores se obtiene de la variable *OLF_size_lowerbound*.

Una vez realizadas todas la comprobaciones, el siguiente paso es llamar a la función Fuse() (definida en /be/lno/fusion.cxx) para que lleve a cabo la fusión tal y como se describe en la sección 3.3. Los parámetros que se pasan son:

- Los SNLs a fusionar.
- Número de niveles a fusionar = MIN(profundidad SNL1, profundidad SNL2).
- Número de iteraciones que se pueden extraer al aplicar peeling en los bucles más internos. Este valor se determina del siguiente modo:

  Si profundidad SNL1≠profundidad SNL2, o alguno de los SNLs no es interno, entonces *peeling_limit=0.*

  si no *peeling_limit*=MIN(máximo permitido[8], iteraciones estimadas bucle externo de SNL1, iteraciones estimadas bucle externo de SNL2)
- *peeling_unrolled*=CIERTO, indicando que las iteraciones que se extraigan de los bucles internos deben ser desenrolladas.

Si la fusión no tiene éxito y el resultado devuelto es *probar_nivel_a_nivel* se intenta realizar la fusión nivel a nivel. La diferencia entre la fusión "normal" y la fusión nivel a nivel es que en esta última se va a permitir hacer *peeling* en los bucles externos de los SNLs (sin desenrollar las iteraciones extraídas). Además, si LNO_Fusion=2 también se va a permitir fusión parcial.

Los parámetros que se pasan a la función Fuse_Level_By_Level() son:

- Todos los anteriores (calculados de igual modo), excepto *peeling_unrolled.*
- *allow_partial_fusion*, que valdrá cierto si LNO_Fusion=2.
- *allow_outer_peeling*, que valdrá cierto, permitiendo la extracción de iteraciones de los bucles más externos.
- La lista de SNLs de la función que estamos tratando.

Stride_One_Level

La función Stride_One_Level() recibe las listas de referencias a arrays contenidas en un SNL, el nivel del bucle más externo y la profundidad del SNL[9]. A partir de ahí se recorren todas las referencias y se contabilizan todas las ocurrencias de variables de iteración del SNL que cumplan:

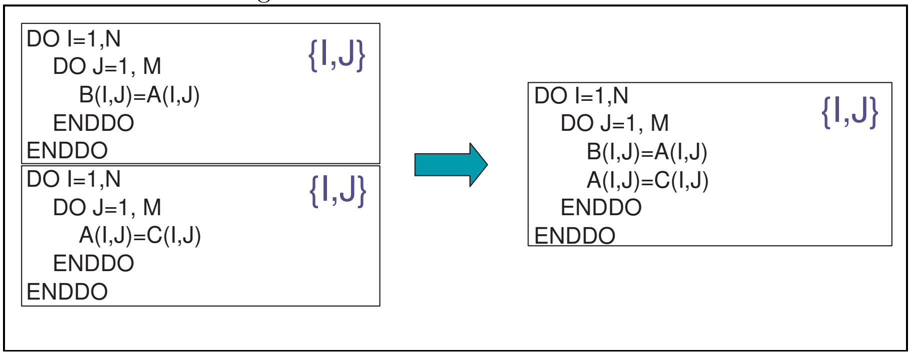
1. La variable es instanciada en la última dimensión del access array[10] que representa el acceso al array.

[8]El número máximo de iteraciones que se pueden extraer se especifica mediante el flag de compilación *-fusion_peeling_limit.* Su valor por defecto es 5.

[9]Pensamos que en realidad la función debería recibir (o calcular a partir de los parámetros disponibles) el nivel del bucle más interno del SNL en lugar de la profundidad del SNL. Con la codificación actual de la función, el bucle que itera por los distintos niveles del SNL ($for(i = outer\_level; y \leq level; i = i + 1)$ tratará los bucles que uno espera sólo cuando outer_level sea igual a 0 y el SNL sea interno.

[10]La última dimensión del access array corresponde a la primera dimensión en la especificación del acceso de un programa Fortran. Esto es debido a que internamente el compilador trabaja con almacenamiento por filas.

Figura 3.19: Fusión de bucles externos



2. El coeficiente que acompaña la variable vale 1 o -1.

3. No hay otras variables de iteración del SNL que aparezcan en la dimensión del acceso que estamos analizando.

Si el access array de la referencia o el access vector correspondiente a la última dimensión son Too_Messy se ignora la referencia. Si el access_vector contiene términos no lineales también se ignora la referencia.

Cada vez que se detecta una ocurrencia que cumpla las condiciones anteriores se incrementa un contador asociado a la variable. Una vez analizadas todas las referencias, se devuelve el nivel del bucle cuya variable de iteración registra un mayor número de ocurrencias. Si todos los contadores están a 0 se devuelve -1.

En el siguiente ejemplo Stride_One_Level vale 0, que es el nivel que corresponde al bucle cuya variable de iteración es i.

```
DO I=1,N
  DO J=1,M
    A(I,J) = B(I,J)
  ENDDO
ENDDO
```

Capítulo 4

Shackling

La primera optimización llevada a cabo una vez generados los SNLs de profundidad máxima es la aplicación de *padding*, que aumenta el tamaño de algunas estructuras de datos con la finalidad de conseguir una alineación de éstas en memoria que reduzca los fallos por conflicto en la cache. A continuación se aplica el *shackling*[9, 10, 11], optimización a la que dedicamos este capítulo, cuyo objetivo es bloquear los accesos a memoria para explotar la localidad espacial y temporal. A diferencia de otras optimizaciones, el shackling realiza el bloqueo partiendo directamente de los datos y puede aplicarse ante la presencia de bucles no perfectamente anidados sin necesidad de realizar transformaciones previas.

Al aplicar esta transformación se particionan las estructuras de datos accedidas en bloques y se establece una ordenación para el recorrido de estos bloques. A continuación se modifica el código original de modo que, al recorrer los distintos bloques, sólo se ejecuten aquellas instancias de cada sentencia que accedan al bloque en curso.

Con esta optimización el compilador sólo intenta bloquear los accesos para la cache de nivel 2 (ver [11]). Posteriormente, durante la aplicación del tiling, se bloquean también los accesos para el resto de niveles de la jerarquía de memoria.

En la sección 4.1 vamos a explicar con más detalle en qué consiste esta optimización.

La sección 4.2 muestra a qué tipo de anidaciones se intentará aplicar shackling y la sección 4.3 presenta la estructura básica del algoritmo que lo implementa.

Como veremos, existen dos variantes del algoritmo, la elección de la variante que se aplicará depende de como son las escrituras en el código que deseamos transformar. Si todas las sentencias escriben sobre el mismo array estamos en el caso simple, que se explica en la sección 4.4. El caso contrario, que denominaremos no simple, se describe en la sección 4.5.

La aplicación del shackling va a modificar el código introduciendo numerosos condicionales. Éstos podrían mermar considerablemente el rendimiento del programa y por este motivo se intenta eliminarlos. La sección 4.6 muestra como se lleva a cabo este proceso.

Por último, también con el objetivo de mejorar el rendimiento, se transforma el código para hacer desaparecer la función MIN de los límites superiores de los bucles. Este paso se describe en la sección 4.7.

A lo largo de todo el capítulo indicaremos también algunas diferencias entre la implementación del shackling en este compilador y lo propuesto en la tesis de I. Kodukula [9]. Asimismo comentaremos algunos errores que hemos podido detectar en la implementación, especialmente en lo referente a la validación de la legalidad de la transformación.

La aplicación del shackling se puede desactivar utilizando el flag de compilación shackle=0 dentro del grupo de opciones de LNO. El punto de entrada a esta etapa de la optimización de bucles es la función SHACKLE_Phase(). Salvo que se especifique lo contrario, todas las funciones a las que nos referimos en este capítulo se definen en be/lno/shackle.cxx.

4.1. Definición de Data Shackle

El mecanismo utilizado para reordenar la ejecución de las sentencias se denomina Data Shackle. La definición de este mecanismo consta de tres partes:

1. Se escoge una estructura de datos y se divide en bloques.
2. Se determina la secuencia que se seguirá para recorrer estos bloques.
3. Se determina qué instancias de cada sentencia se ejecutarán cuando se visite un bloque.

Esta optimización es aplicable tanto a anidaciones de bucles perfectas como a anidaciones imperfectas cuyo código imperfecto contiene otros bucles.

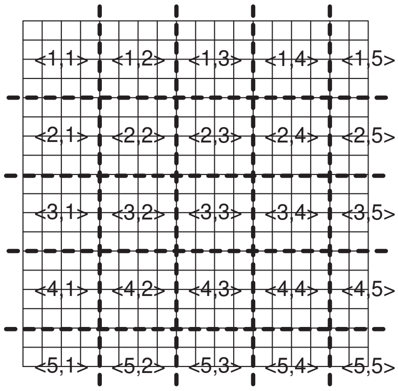
Podemos interpretar el término Data Shackle[1]como un mecanismo que "ata" las instancias de cada sentencia a un determinado bloque de datos. Para cada bloque de datos se examina el código del programa ejecutando solamente aquellas instancias de las sentencias que acceden al bloque en curso. Esta noción quedará más clara con el siguiente ejemplo, donde aplicamos shackling a una multiplicación de matrices:

```
do I = 1,N
  do J = 1,N
    do K = 1,N
      C[I,J] = C[I,J] + A[I,K] * B[K,J]
```

1. Dividimos la matriz C en bloques mediante conjuntos de planos paralelos y equidistantes entre si. Ver figura 4.1.
2. Establecemos un orden en el recorrido de los bloques. El código que se va a generar planificará los bloques mediante una enumeración en orden lexicográfico de sus coordenadas. Ver numeración en la figura 4.1.
3. Finalmente debemos especificar qué instancias de las sentencias se deben ejecutar cuando un bloque se convierte en el bloque en curso. En cada sentencia escogemos una referencia R a la matriz bloqueada. En el ejemplo hay una sentencia y escogeremos la escritura sobre C. Cuando visitemos un bloque de la matriz vamos a ejecutar todas las sentencias del programa cuya referencia R acceda a datos contenidos en ese bloque . De momento podemos suponer que todas las sentencias contendrán al menos un acceso

[1]Literalmente el término *shackles* significa grilletes, y en un sentido más amplio ataduras.

Figura 4.1: Bloqueo de la matriz C



a la matriz bloqueada aunque, como veremos más adelante, esto no va a ser necesario.

El resultado de aplicar shackling al código de multiplicación de matrices utilizando bloques de 25x25 es el siguiente:

```
(1) do b1 = 1,⌈(N/25)⌉
(2)   do b2 = 1,⌈(N/25)⌉
(3)     do I = 1,N
(4)       do J = 1,N
(5)         do K = 1,N
(6)           if ((b1-1)*25 < I ≤ b1*25) &&
                 ((b2-1)*25 < J ≤ b2*25)
(7)             C[I,J] = C[I,J] + A[I,K] * B[K,J]
```

Observamos que se han añadido dos nuevos bucles al programa original en las líneas 1 y 2. Estos bucles realizan el recorrido de los bloques de C. Aparece, además, un condicional en la línea 6 que garantiza que sólo se van a ejecutar aquellos productos que escriban en el bloque en curso.

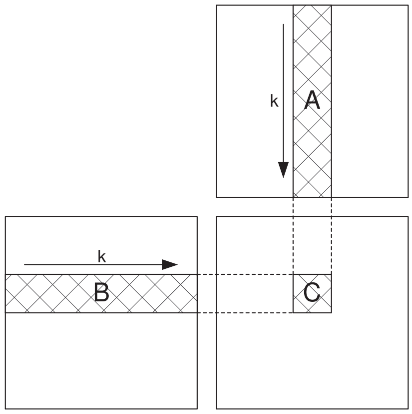
Este código se puede optimizar eliminando el condicional del cuerpo del bucle más interno:

```
do t1 = 1,⌈(N/25)⌉
  do t2 = 1,⌈(N/25)⌉
    do It = (t1-1)*25+1, min(t1*25,N)
      do Jt = (t2-1)*25+1, min(t2*25,N)
        do K = 1,N
          C[It,Jt] = C[It,Jt] + A[It,K] * B[K,Jt]
```

La figura 4.2 muestra gráficamente el resultado del bloqueo aplicado a la multiplicación de matrices. Las partes sobreimpresionadas de cada matriz representan los elementos que se van a acceder durante una iteración de los bucles t1 y t2.

Al llevar a cabo una implementación de esta transformación es necesario concretar una serie de cuestiones:

Figura 4.2: Bloqueo en la multiplicación de matrices



1. Cuál es la unidad de transformación? Es decir, teniendo en cuenta que el shackling se puede aplicar a un conjunto arbitrario de bucles, como determinamos a qué porciones del código se aplica la optimización?

   Para que un conjunto de bucles sea candidato para la aplicación de shackling debe estar formado por un bucle externo (no contenido en otros bucles) que tenga más de un bucle como hijos directos (al mismo nivel de profundidad). Nótese que en el código del ejemplo anterior no se aplicaría shackling.

   Por tanto, LNO sólo aplica shackling a códigos imperfectos (no perfectamente anidados). Los códigos perfectamente anidados se tratan en la etapa 4 (outer unrolling + tiling) del optimizador de bucles.

2. Cuál es la orientación de los planos de corte?

   Los planos de corte son siempre paralelos a los ejes de coordenadas de los datos.

3. Cuál es la separación entre los planos de corte (tamaño de bloque)?

   El tamaño de bloque depende de los accesos a array que aparecen en el código. El objetivo es conseguir que, una vez transformado el programa, la cantidad de datos accedidos durante cada ejecución del bucle externo del código original no supere el tamaño efectivo de la cache. El tamaño efectivo de la cache es un parámetro conocido tal que, si la cantidad de datos en cache no supera ese valor, podemos considerar los fallos por conflicto irrelevantes.

4. Cuál es el orden en el recorrido de los bloques?

   Los bloques se recorren enumerando sus coordenadas en orden lexicográfico.

5. Cómo se escogen los accesos (data-centric references) que utilizaremos para determinar qué instancias de una sentencia se ejecutan para un determinado bloque?

Figura 4.3: Unidad de transformación

(a) no shackling	(b) no shackling	(c) shackling
DO I DO J DO K ... DO L ...	DO I DO J DO K ...	DO I DO J ... DO K ...

Dada una sentencia, se debería calcular la matriz de acceso (data access matrix) para cada referencia y escoger la referencia cuya matriz de acceso tenga un rango mayor. Si existen varios candidatos se da preferencia al destino de la asignación. Hecho esto uno podría aplicar composición de data shackles (ver [9]) hasta asegurar que se bloquean todas las dimensiones de cada acceso. En realidad, únicamente se toman como data-centric references los destinos de las asignaciones y la aplicación de composición es muy limitada. Encontramos aquí una diferencia entre lo propuesto en [9] y lo implementado en el compilador, que como veremos no es la única.

6. Cómo se van a eliminar los condicionales que aparecen al aplicar la transformación?

   Para eliminar aquellos IFs cuya condición no puede ser calculada en tiempo de compilación utilizaremos la transformación conocida como Index Set Splitting. Como paso previo a esta transformación, a veces será necesario añadir nuevos IFs, que posteriormente son eliminados aplicando el mismo algoritmo de forma iterativa.

4.2. Selección de bucles

El primer paso de esta etapa de la optimización de bucles consiste en seleccionar las anidaciones de bucles donde se intentará aplicar shackling. Para ello, se recorren los SNLs contenidos en la función en curso. Dado un SNL, siendo *loop* su bucle más externo, se realizan las siguientes comprobaciones:

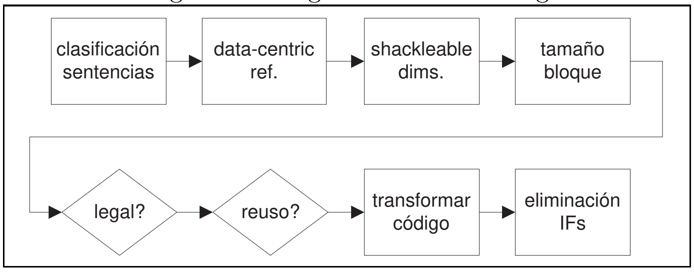
1. *loop* no está contenido dentro de un bucle.

2. *loop* contiene más de un bucle. Estos bucles tienen que ser hijos directos de *loop*. Esta condición, junto con la anterior, implica *loop* es el único bucle del SNL al que pertenece.

3. Comprueba que *loop* no esté afectado por directivas que impidan aplicar shackling.

La figura 4.3 muestra algunos ejemplos de anidaciones indicando en cada caso si el compilador intentaría aplicar shackling.

Figura 4.4: Algoritmo de shackling



4.3. Algoritmo

La figura 4.4 da una visión general de los pasos llevados a cabo al aplicar shackling. Éstos son:

clasificación sentencias: En este punto se clasifican las sentencias que contiene la anidación en clases de equivalencia. Exceptuando las del grupo cero, todas las sentencias que pertenezcan a una misma clase de equivalencia se bloquearán del mismo modo.

data-centric reference: Se selecciona la data-centric reference para cada sentencia. Si bien es posible aplicar composición y aplicar el shackling teniendo en cuenta más de un acceso por sentencia, en la práctica las data-centric references serán el destino de las escrituras.

shackleable dims: Estudiamos cada una de las matrices accesibles desde el código que estamos tratando. No se podrá aplicar el bloqueo a aquellas dimensiones que en algún momento se indexen, por ejemplo, mediante un acceso a otra matriz o mediante alguna variable que no sea índice de un bucle.

tamaño bloque: Se calcula el tamaño de los bloques.

legal? Se comprueba la legalidad de la transformación. Para que el shackling sea legal se deben respetar todas las dependencias del código original.

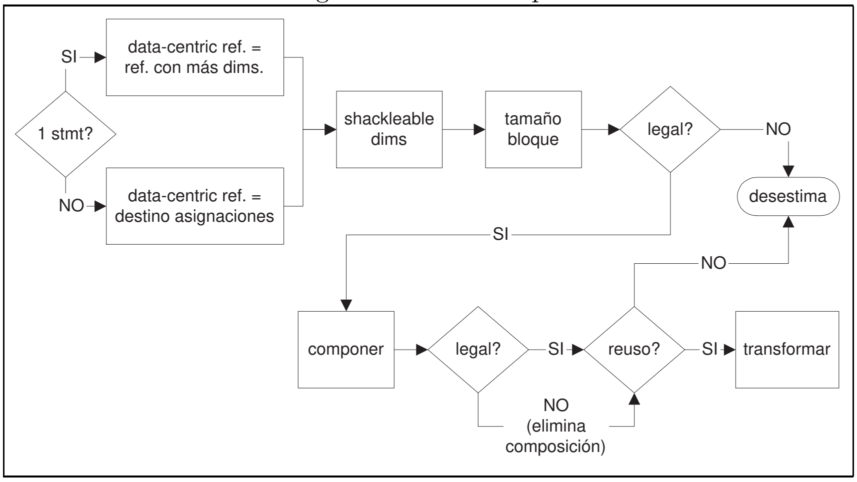
reuso? Sólo se va a aplicar la transformación si existe reuso de los datos.

transformar código: Se modifica el código original añadiendo los bucles que recorren los bloques y protegiendo cada una de las sentencias con condicionales.

eliminación IFs: Se eliminan los condicionales introducidos aplicando Index Set Splitting y eliminando los condicionales cuyo resultado se puede conocer en tiempo de compilación.

Si bien los pasos seguidos son siempre los que acabamos de describir (aunque no necesariamente en el mismo orden), la implementación del algoritmo de shackling presenta dos variantes distintas. La primera de ellas corresponde al caso simple, que se caracteriza por que todas las sentencias escriben sobre el

Figura 4.5: Caso simple



mismo array. El caso no simple se produce cuando aparecen escrituras sobre variables distintas o sobre algún escalar.

Una vez decidido que se va a intentar shackling para optimizar cierta porción de código, el primer paso consiste en determinar qué caso se va a tratar. Para ello se genera una lista con todas las sentencias que aparecen en el cuerpo de los bucles que deseamos transformar (función gather_stmts_in_func()). Esta lista se utiliza para comprobar si todas las referencias de escritura tienen como destino el mismo array. Al mismo tiempo, si se observa que el access array de alguna de las escrituras contiene alguna expresión no afín (el atributo Too_Messy está activado), se abandona el shackling. También se abandona si alguna de las sentencias de la lista no es de tipo asignación.

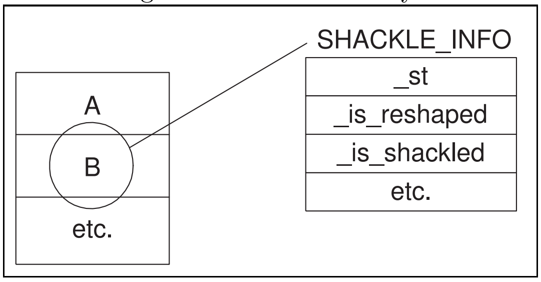
El algoritmo para el caso simple se explica en la sección 4.4. El caso no simple es tratado en la sección 4.5. En aquellos puntos donde el tratamiento para ambos casos coincide, éste se explica en la sección 4.4. En el caso no simple encontraremos una referencia al apartado correspondiente de la sección anterior. La eliminación de condicionales se detalla aparte en la sección 4.6.

4.4. Caso simple

La figura 4.5 muestra un diagrama del caso simple. Comparando con la figura 4.4 vemos que no aparece la clasificación de sentencias. En el caso simple este paso resulta trivial y se ha omitido por razones de espacio. Así pues, nos encontramos en primer lugar con la selección de la data-centric reference, que depende del número de sentencias que haya en el código. Observamos también que aparece una fase de composición de data-shackles, pero sólo se puede activar mediante un flag de compilación no documentado (ver sección 4.4.6). Cuando este flag no está activado y la transformación es legal se pasa directamente a la comprobación del reuso.

A continuación explicamos como se implementan los pasos del shackling para el caso simple.

Figura 4.6: Lista de arrays



4.4.1. Clasificación de sentencias

En el caso simple, que se caracteriza por que todas las escrituras se realizan sobre el mismo array, existe una sola clase de equivalencia (el grupo cero) que contiene todas las sentencias del código que se intentará bloquear.

Como veremos esta clasificación, que en el caso simple resulta trivial, cobra importancia en el caso no simple, donde es parte fundamental del tratamiento de los escalares.

4.4.2. Data-centric reference

En este paso seleccionaremos una data-centric reference para cada una de las sentencias. Esta referencia se asocia a la sentencia mediante el mecanismo de mapeo (usando el identificador *shackle_shackle_map*) y será la utilizada para determinar qué instancias de la sentencia se ejecutarán para el bloque en curso.

Si el código que estamos tratando contiene una sola sentencia, se escoge como data-centric reference la referencia que tenga un mayor número de dimensiones.

Cuando hay más de una sentencia, la data-centric reference de cada una de las sentencias es el destino de la asignación.

4.4.3. Shackleable dims

Se crea una lista que contiene una entrada para cada una de las variables de tipo array presentes en las tablas de símbolos global y local (ver figura 4.6). Podemos encontrar más información acerca de las tablas de símbolos en [14].

Cada elemento de la lista contiene, entre otros, los siguientes campos:

_st: Entrada del array en la tabla de símbolos.

_is_reshaped: Indica si se ha modificado la definición del array (número de dimensiones, número de elementos por dimensión).

_is_shackled: Vector de booleanos, para cada dimensión del array indica si ésta puede ser bloqueada (shackleable dimension).

_shackle_sizes: Vector de enteros, indica el tamaño de bloque que corresponde a cada dimensión del array.

_array_bounds: Información acerca de las dimensiones del array.

Recorremos los elementos de la lista e inicializamos el campo _is_shackled. Una dimensión de un array se puede bloquear cuando en todas las referencias a ese array:

1. No aparecen otros arrays en las expresiones que indexan esa dimensión.
2. Las variables que aparecen en las expresiones de indexación son todas índices de bucle.

Cuando se detecta una referencia cuyo número de dimensiones o número de elementos por dimensión no coincide con lo especificado en la tabla de símbolos, se marca la entrada correspondiente de la lista de arrays poniendo a cierto el campo _is_reshaped. En este caso, ninguna de sus dimensiones se podrá bloquear.

El algoritmo 10 muestra como se lleva a cabo la inicialización del atributo _is_shackled para cada una de las dimensiones de los distintos arrays.

Algoritmo 10: Inicialización del campo _is_shackled
Entrada: Lista de arrays (*shackle_info*). Subárbol WHIRL que se está tratando (*main_snl*).
Salida: Lista de arrays *shackle_info* con el campo *_is_shackled* inicializado.

foreach entrada *sh* de *shackle_info*
  foreach dimensión i del array
    $sh.\_is\_Shackled[i] = TRUE$
    foreach ocurrencia del array en *main_snl*
      Comprobamos si la expresión utilizada para indexar la dimensión i cumple los requisitos.
      if no cumple
        $sh.\_is\_Shackled[i] = FALSE$
  if *sh._is_reshaped*
    foreach dimensión i del array
      $sh.\_is\_Shackled[i] = FALSE$

4.4.4. Tamaño de bloque

Una vez se ha determinado qué array se va a bloquear y conocidas las dimensiones que vamos a poder bloquear, el siguiente paso consiste en calcular el tamaño de bloque.

El shackling intenta bloquear los accesos a nivel de L2, el resto de niveles de memoria se bloquearán mediante otras optimizaciones en etapas posteriores de la compilación (ver artículo [11]). Para realizar el bloqueo se parte de la hipótesis que si la cantidad de datos accedida por los bucles no supera un determinado tamaño efectivo podemos ignorar los fallos por conflicto. Este tamaño efectivo es un parámetro conocido en tiempo de compilación.

El mecanismo propuesto en [9] para el cálculo del tamaño de bloque se basa en una simplificación del modelo utilizado en otras partes del compilador (durante la aplicación de tiling). Básicamente se propone un método para calcular de forma paramétrica la cantidad de datos accedidos en la anidación. Este dato recibe el nombre de *footprint*. Conocido el *footprint*, se calcula el tamaño de bloque para que la cantidad de datos accedidos no supere el tamaño efectivo de la cache. Se supone además que todas las dimensiones bloqueadas tendrán el mismo tamaño de bloque. Descrito muy por encima, el cálculo del *footprint* consta de tres pasos:

1. Identificación, para cada sentencia, de los accesos que más contribuyen a la cantidad de datos accedida.

2. Clasificación, para cada sentencia, de los accesos del punto anterior en clases de equivalencia. A continuación, se intenta agrupar las clases de equivalencia de aquellas sentencias cuya *data-centric reference* es idéntica.

3. Se calcula el *footprint* de cada grupo. El *footprint* de un grupo se calcula como el *footprint* de una de las referencias del grupo tomada al azar. Esta referencia recibe el nombre de referencia significativa.

El cálculo del tamaño de bloque es uno de los puntos donde lo propuesto en [9] no coincide con la implementación en el compilador. La secuencia de pasos que acabamos de enumerar, se reduce a lo siguiente:

Se consideran referencias significativas aquellas referencias que cumplen:

1. Está situada en una sentencia de profundidad máxima.

2. Una de las dimensiones del access array no se puede obtener por combinación lineal de los access vectors de otra referencia significativa. El compilador no comprueba si los accesos que se comparan son de la misma variable.

Para obtener la lista de referencias significativas, se recorre el subárbol WHIRL de los bucles que estamos tratando y, cada vez que encontramos un acceso a un array dentro de una sentencia que está a profundidad máxima, se analiza el acceso y se determina si éste debe añadirse a la lista. Sea $refq$ la lista de referencias significativas (inicialmente vacía) y ar_main el access array del acceso que queremos comprobar, aplicamos el algoritmo 11.

Algoritmo 11: Referencias significativas
Entrada: La lista de referencias significativas $refq$. El access array de la referencia que queremos comprobar ar_main
Salida: La lista de referencias significativas actualizada.
(1) $significativa$ = TRUE
(2) **foreach** access array s de $refq$
(3)   **foreach** dimensión q de ar_main
(4)   **if** q combinación lineal de elementos de s
(5) $signigicativa$ = FALSE

Una vez obtenida la lista de referencias significativas se aplica el siguiente cálculo:

$$allowed\_size = \frac{\text{tamaño efectivo}}{\text{número de referencias significativas}}$$

Y a continuación calculamos el tamaño de bloque en cada dimensión de manera que

$$vol * x^n = allowed\_size$$

donde *vol* es el tamaño de un elemento del array multiplicado por el tamaño de las dimensiones que no podemos bloquear. El tamaño de bloque para las dimensiones que sí se van a bloquear se obtiene mediante el siguiente cálculo:

$$x = \sqrt[n]{\frac{allowed\_size}{vol}}$$

La función que asigna a cada dimensión de las variables el tamaño de bloque es Appropriate_Shackle_Size_Set(). Dentro de esta función se realizan los siguientes pasos:

1. Se obtiene la lista de referencias significativas mediante una llamada a Gather_Deepest_References().
2. Se toma una sentencia del grupo 0.
3. Se obtiene la lista de data-centric references de la sentencia, que sólo tendrá un elemento.
4. Para cada una de las data-centric references (sólo habrá una), se obtiene el identificador de la variable correspondiente, que se utiliza para encontrar su entrada en la lista de arrays generada en el apartado 4.4.3. Con una llamada a Set_Shackle_Size_Info() se actualizan los campos que indican el tamaño de bloque en dicha entrada.

Las funciones Appropriate_Shackle_Size_Set(), Gather_Deepest_References() y Set_Shackle_Size_Info() se definen en /be/lno/shackle_mem.cxx.

4.4.5. Legalidad de la transformación

Para asegurar la legalidad del shackling es necesario comprobar que se respetan las dependencias originales. Si existe una dependencia en el programa original hay que asegurar que se mantiene el sentido de la dependencia en el programa transformado. Esto debe ser así tanto para las dependencias entre escalares como dependencias entre arrays y para todos los tipos de dependencia (dependencia de flujo, antidependencia, dependencia de salida). La función donde se comprueba la legalidad de la transformación es _xis_legal_shackle().

Hemos podido comprobar que el test de legalidad del shackling implementado en el compilador es, cuando menos, incompleto. En esta sección veremos que es lo que sí hace y comentaremos lo que no hace con algún ejemplo de transformación errónea.

Dependencias entre escalares

Cuando estamos en el caso simple, todas las escrituras son sobre la misma variable, que además debe ser de tipo array. Si no existen escrituras sobre escalares no puede haber dependencias entre escalares.

Dependencias entre accesos a array

Para una descripción más formal del análisis de dependencias entre accesos a array consultar [9]. A continuación mostramos mediante un ejemplo como se comprueba que la aplicación del shackling no viola las dependencias.

El algoritmo 12 muestra como se realiza el recorrido de las referencias. Se toma una sentencia y se recorre su lista de referencias. Para cada referencia (ref), se generan todas las parejas posibles formadas por ref y una escritura ($ref2$). Cada una de estas parejas se pasa a la función _xdependence_is_preserved() para comprobar si existe dependencia y, caso de que existiera, si ésta se respeta al aplicar la transformación.

Algoritmo 12: Recorrido de las referencias
Entrada: Lista de sentencias (*stmts*). Cada sentencia tiene asociada la lista de referencias que contiene.
Salida: Devuelve TRUE si la transformación es legal, FALSE en caso contrario.

(1) **foreach** sentencia $step1$
(2)   **foreach** referencia ref de $step1$
(3)   **foreach** sentencia $step2$
(4)   **if** $step2 = step1$
(5) $read\_stmt\_prior$ = TRUE;
(6) $ref2$ = Destino escritura de $step2$
(7) $result$ = _xdependence_is_preserved (ref, ref2, read_stmt_prior, ...)
(8)   **if** $result$ = FALSE
(9)   Devolver FALSE
(10) Devolver TRUE

En la línea 5 se comprueba la posición relativa de las sentencias que estamos procesando. Este dato se almacena en el flag *read_stmt_prior*, que indica si la referencia ref se accede antes que la escritura $ref2$. Observamos que este cálculo presupone que estamos recorriendo las sentencias en orden. En este punto del programa se ha detectado un error.

La función que devuelve la lista de sentencias (gather_stmts_in_func()) realiza un recorrido en anchura de la representación intermedia del programa. Esto quiere decir que las sentencias que se encuentren en un nivel de profundidad inferior en el árbol de la representación intermedia (más cerca del nodo raíz) se añadirán antes a la lista de sentencias. La obtención de las sentencias mediante este procedimiento no garantiza que éstas aparezcan en orden de programa y, por tanto, el valor asignado a *read_stmt_prior* puede ser erróneo.

El código que vamos a utilizar como ejemplo de comprobación de las dependencias corresponde a la factorización de Cholesky, concretamente nos centraremos en las dependencias entre los accesos marcados en negrita.

```
DO K = 1, 2000
   A(K,K) = SQRT(A(K,K))
   DO I = K+1, 2000
      A(I,K) = A(I,K)/A(K,K)
   ENDDO
   DO J = K+1, 2000
      DO I = J, 2000
         A(I,J) = A(I,J)-A(I,K)*A(J,K)
      ENDDO
   ENDDO
```

```
ENDDO
```

Después de aplicar shackling el programa tendría el siguiente aspecto:

```
DO SA1 = 0, 9, 1
  DO SA2 = 0, 9, 1
    DO K = 1, 2000, 1
      ...
      DO J0 = (K + 1), 2000, 1
        DO I0 = J0, 2000, 1
          IF(J0 .LE.((SA1 * 218) + 218)) THEN
            IF(J0 .GT.(SA1*218)) THEN
              IF(I0 .LE.((SA2 * 218) + 218)) THEN
                IF(I0 .GT.SA2 * 218)) THEN
                  A(I0,J0) = A(I0,J0)-A(I0,K)*A(J0,K)
                ENDIF
              ENDIF
            ENDIF
          ENDIF
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Vamos a analizar las dependencias cuando:
$step1 = step2 =$ "`A(I,J) = A(I,J)-A(I,K)*A(J,K)`"
$ref =$ lectura de A(I,J) y
$ref2 =$ escritura sobre A(I,J)

Utilizaremos el subíndice 1 para referirnos a las variables de iteración en ref ($A(I_1, J_1)$) y el 2 para referirnos a las variables de iteración en $ref2$ ($A(I_2, J_2)$).

Para realizar el análisis se crean varios sistemas de ecuaciones:

1. Acceso a la misma posición de memoria: Para que exista dependencia entre dos referencias, éstas deben acceder a la misma posición de memoria.

2. Límites de los bucles: Limitan el rango de valores que pueden tomar la variables de iteración.

3. Dependencia de flujo: Ecuaciones que expresan que la escritura precede a la lectura del elemento.

4. Planificación de bloques: Ecuaciones que expresan que la lectura se realiza cuando se visita un bloque que se planifica antes que el bloque donde se va a realizar la escritura.

5. Restricciones de los condicionales: Las sentencias se encuentran dentro de condicionales. Estos condicionales también limitan el rango de valores que pueden tomar las variables.

Una vez generados todos los sistemas, éstos se combinan y se procede a evaluar su consistencia. Hay dos posibilidades:

- El sistema es inconsistente. No existe dependencia de flujo o no es posible que el bloque correspondiente a la lectura se visite antes que el bloque correspondiente a la escritura.

- El sistema es consistente. Existe dependencia de flujo y además es posible que se ejecute la lectura antes que la escritura. Dicho de otro modo, los sistemas generados en el punto 3 y en el punto 4, junto con el resto, pueden ser satisfechos de forma simultánea. La transformación no es legal.

A continuación mostramos como se construyen las ecuaciones para el ejemplo que nos ocupa.

Acceso a la misma posición de memoria

Para que las referencias accedan a la misma posición de memoria se debe cumplir que $I_1 = I_2$ y $J_1 = J_2$. En la siguiente tabla se muestran las variables de iteración que vamos a considerar y sus correspondientes coeficientes (en blanco es igual a cero).

K_1	J_1	I_1	K_2	J_2	I_2	$SA1_1$	$SA2_1$	$SA1_2$	$SA2_2$	$=$	ct.
	1			-1							0
		1			-1						0

Límites de los bucles

Añadimos a las ecuaciones anteriores los límites de los bucles:

K_1	J_1	I_1	K_2	J_2	I_2	$SA1_1$	$SA2_1$	$SA1_2$	$SA2_2$	$\leq$	ct.
	1	-1									0
1	-1										-1
-1											-1
		1									2000
	1										2000
1											2000
				1	-1						0
			1	-1							-1
			-1								-1
					1						2000
				1							2000
			1								2000

Dependencia de flujo

A continuación se generan los sistemas de ecuaciones que expresan una dependencia de flujo (read after write) o bien una dependencia de salida, en caso de que ref sea una escritura, desde $ref2$ hacia ref. Se generan tantos sistemas de ecuaciones como bucles en común tienen las dos referencias. Si alguno de estos sistemas, junto con las ecuaciones de los apartados anteriores tiene solución, existe una dependencia de flujo (o dependencia de salida).

Al generar estas ecuaciones se tiene en cuenta la posición relativa de las referencias en el programa (flag $read\_stmt\_prior$). En el ejemplo no puede haber una dependencia de flujo dentro de una misma iteración del bucle más interno porque la lectura se ejecuta antes que la escritura. Las ecuaciones son:

$$\begin{array}{lclcl} K_2 < K_1 & \text{o} & K_1 = K_2 & \text{o} & K_1 = K_2 \\ & & J_2 < J_1 & & J_1 = J_2 \\ & & & & I_2 < I_1 \end{array}$$

K_1	J_1	I_1	K_2	J_2	I_2	$SA1_1$	$SA2_1$	$SA1_2$	$SA2_2$		ct.
-1			1							$\leq$	-1
1			-1							$=$	0
	-1			1						$\leq$	-1
1			-1							$=$	0
	1			-1						$=$	0
		-1			1					$\leq$	-1

Planificación de los bloques

Una vez generadas las ecuaciones que expresan una dependencia de flujo entre los dos accesos, generamos los sistemas de ecuaciones que expresan la situación en que el bloque de la lectura dependiente (subíndice uno) se planifica antes que la escritura (subíndice dos). Estos sistemas de ecuaciones se combinan con los generados hasta el momento, y se les añaden también las restricciones de los condicionales introducidos al aplicar shackling (ver siguiente apartado).

Las ecuaciones que se generan en este punto son:

$$\begin{array}{lcl} SA1_1 < SA1_2 & \text{o} & SA1_1 = SA1_2 \\ & & SA2_1 < SA2_2 \end{array}$$

K_1	J_1	I_1	K_2	J_2	I_2	$SA1_1$	$SA2_1$	$SA1_2$	$SA2_2$		ct.
						1		-1		$\leq$	-1
						1		-1		$=$	0
							1		-1	$\leq$	-1

Restricciones de los condicionales

Para terminar se añaden las restricciones de los condicionales. Después de completar estas ecuaciones se comprueba la consistencia de los sistemas y se determina si la aplicación del shackling es legal.

En nuestro ejemplo las ecuaciones son:

K_1	J_1	I_1	K_2	J_2	I_2	$SA1_1$	$SA2_1$	$SA1_2$	$SA2_2$	$\leq$	ct.
	-1					218					217
	1					-218					0
		-1					218				217
		1					-218				0
				-1				218			217
				1				-218			0
					-1				218		217
					1				-218		0

Estas ecuaciones no son exactamente las que se deducirían observando el código transformado. Esto se debe a que se generan suponiendo que los bucles SA1 y SA2 empiezan por uno, cuando en realidad empiezan por cero. Esto no va a afectar el resultado final.

Casos no tratados

Si existe dependencia en un determinado sentido y cambiamos el orden de ejecución de los accesos involucrados en dicha dependencia, estamos aplicando una transformación que no es legal. En esta sección hemos visto como se aplicaba este razonamiento a las dependencias de flujo y a algunas dependencias de salida (cuando ref es una escritura). Pero para que el análisis de dependencias sea completo debe cubrir también las antidependencias y el resto de dependencias de salida posibles. La implementación actual del shackling sólo contempla las dependencias de flujo y parte de las dependencias de salida.

Además, teniendo en cuenta que antes de aplicar shackling ya se ha generado el grafo de dependencias entre arrays, seguramente sería más efectivo tomar la información sobre las dependencias existentes y el tipo de cada una de ese grafo, en lugar de recalcularlas.

El siguiente código muestra un caso donde el compilador aplicará shackling de forma errónea.

```
    DO J=1,N
      DO I=1,N
S1      t = t + A(I+10,I+10)
      ENDDO
      DO I=1,N
S2      A(I+10,J+10) = A(I+1,J)
S3      A(I,J) = A(I,J) + B(I,J)
      ENDDO
    ENDDO
```

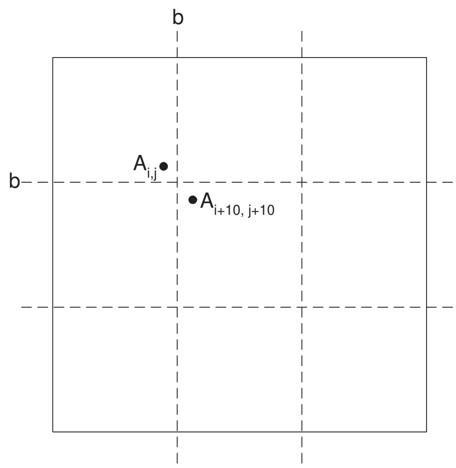
En este código existe una antidependencia entre la lectura de A(I+1,J) en S2 y la escritura de A(I,J) en S3. Al aplicar shackling se divide la matriz A en bloques (digamos de tamaño b $\times$ b) y sólo se ejecutan aquellas sentencias que escriben en el bloque en curso. Supongamos que estamos ejecutando las iteraciones que corresponden al primer bloque (ver figura 4.7), llega un momento en que todavía se cumple que I,J$\leq$b, pero I+10,J+10>b. Cuando esto ocurre se ejecutan las instancias de S3, pero no de S2 que escribiría fuera del bloque en curso. Esta modificación en el orden de ejecución de las instancias de las sentencias provocará que en algunos casos no se respete la antidependencia.

Ante este código normalmente primero se aplicaría fusión de los bucles I. Como en las anidaciones perfectas no se aplica shackling hay que desactivar esta transformación para comprobar el efecto de la aplicación del shackling. Para evitar que se fusionen los bucles se debe añadir la opción de compilación: -LNO:fusion=0.

4.4.6. Composición de Data Shackles

A continuación comentamos mediante un ejemplo en qué consiste la composición de data shackles. Como veremos la composición sólo se aplica cuando se

Figura 4.7: Bloques de tamaño b × b



utiliza un determinado flag del compilador que, en principio, sólo conocen los desarrolladores. Por este motivo no profundizaremos en los detalles.

En el ejemplo de la multiplicación de matrices de la página 64, hemos visto como bloquear los accesos a la matriz C mediante la aplicación de shackling. La figura 4.2 muestra gráficamente dicho bloqueo. Podemos definir un nuevo data shackle particionando la matriz A en bloques de dos dimensiones y tomando la referencia A(I,K) como la data-centric reference. Combinando estos dos shackles podemos definir un shackle compuesto con el siguiente orden de enumeración: en primer lugar recorremos los bloques de C y, para cada bloque de C, visitamos todos y cada a uno de los bloques de A. Sólo se ejecutarán aquellas instancias de la sentencia $C[I,J] = C[I,J] + A[I,K] * B[K,J]$ cuyas referencias C(I,J) y A(I,K) caigan simultáneamente dentro de los respectivos bloques en curso de C y de A.

La composición de shackles sólo se aplica cuando se utiliza un flag no visible para el usuario final. En concreto, la opción de compilación es -Wb,-ttLN2:0x00200000, donde -Wb indica que el flag que sigue va a pasarse directamente a la etapa de compilación identificada por la letra b, que corresponde al back-end. LN2 indica a qué etapa, dentro del back-end, afecta el flag. Por defecto este flag está desactivado y no se aplica composición.

El algoritmo 13 muestra como se aplica la composición. La elección de la sentencia *step* parece algo arbitraria. Esto aporta confusión, ya que el bucle que recorre las referencias de *step* puede tener un número de iteraciones diferente dependiendo de la sentencia escogida. Además, en cada iteración del bucle externo, las referencias significativas para el resto de sentencias siempre serán las mismas. Remarcar también que sólo se escogen dos referencias para cada sentencia (la que ya teníamos y la que se añade), cosa que no se corresponde exactamente con los criterios expuestos en [9] en lo que a composición se refiere. Una vez encontrada una composición legal se retorna de la función sin agotar las iteraciones del bucle externo. Finalmente, hemos podido observar que si la variable de la referencia adicional no es la misma que la variable de la data-centric reference inicial, se produce un error debido a que el tamaño de bloque para dicha variable tendrá valor 0 (ver sección 4.4.4).

La función Augment_Simple_Basic_Shackle() implementa la composición de data shackles.

Algoritmo 13: Composición de data shackles
Entrada: Lista de sentencias. Lista de arrays $shackle\_info$.
Salida: Lista de sentencias con su lista de data-centric references actualizada.

$can\_be\_augmented = FALSE$
$step$ = primera sentencia
while ($step \neq NULL$) and ($can\_be\_augmented = FALSE$)
  if $step$ contiene referencia significativa adicional[2]
    $can\_be\_augmented = TRUE$
  else
    $step$ = siguiente sentencia
if $can\_be\_augmented = FALSE$
  Retornar
foreach referencia ref de $step$
  if ref es significativa dentro de $step$
    Añadir ref a las data-centric references de $step$
    foreach sentencia $step2$
      if $step \neq step2$
        $ref2$ = Buscar ref. significativa en $step2$
        if $ref2 = NULL$
          $ref2$ = data-centric ref. de $step2$
        Añadir $ref2$ a las data-centric references de $step2$
    if shackling legal
      Retornar
    else
      Eliminar última referencia de las data-centric references de cada sentencia

4.4.7. Reuso

Antes de aplicar la transformación al código del programa se realiza una última comprobación. Se trata de verificar que existe reuso de los datos que vamos a bloquear, ya que en caso contrario la aplicación del bloqueo no tiene sentido. La función que determina la existencia de reuso es Reuse_Exists_In_Loop(), mostrada en el algoritmo 14.

4.4.8. Aplicar transformación

Una vez determinados los parámetros de la transformación (data-centric references y tamaño de los bloques), verificada la legalidad del shackling y habiendo

[2] La sentencia contiene alguna referencia significativa respecto a la data-centric reference cuando alguno de sus access vectors no se puede obtener por combinación lineal de los access vectors de la data-centric reference.

Algoritmo 14: Existe reuso?
Entrada: Código al que vamos a aplicar la transformación.
Salida: Booleano indicando si hay reuso.
(1) $reuso = FALSE$
(2) **foreach** array que aparezca en una data-centric ref.
(3) **foreach** referencia ref del array
(4) $reuso\_en\_ref = TRUE$
(5) **foreach** dimensión de ref
(6) **if** coeficiente del bucle más externo $\neq 0$
(7) $reuso\_en\_ref = FALSE$
(8) **if** $reuso\_en\_ref = TRUE$
(9) $reuso = TRUE$

comprobado la existencia de reuso, se procede a la modificación del programa.

Aunque en este caso no se aplicaría shackling, puesto que tenemos una anidación de bucles perfecta, utilizaremos el siguiente ejemplo de multiplicación de matrices para ver como progresivamente se transforma el código.

```
DO I=1,512
  DO J=1,512
    DO K=1,512
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
 ENDDO
```

En un primer paso la función Create_Simple_Shackle_Loops() genera los bucles que van a realizar el recorrido de los bloques. El número de iteraciones se obtiene dividiendo el tamaño de cada una de las dimensiones de la estructura de datos que vamos a bloquear por el tamaño de bloque. Es posible que aparezcan más iteraciones de las que a priori se esperarían debido a que la aplicación del padding puede haber modificado el tamaño de las estructuras de datos. A falta de añadir los condicionales, el programa anterior quedará como sigue:

```
DO SC1 = 0, 13, 1
  IF(.TRUE.) THEN
    DO SC2 = 0, 14, 1
      IF(.TRUE.) THEN
        DO J0 = 1, 512, 1
          DO K = 1, 512, 1
            DO I0 = 1, 512, 1
              C(I0, J0) = (C(I0, J0) +(A(I0, K) * B(K, J0)))
            END DO
          END DO
        END DO
      ENDIF
    END DO
  ENDIF
END DO
```

Después de crear los bucles externos se añaden los condicionales que controlan la ejecución de las instancias de cada sentencia. Una sentencia sólo va a ejecutarse si la data-centric reference cae dentro del bloque en curso. La función que agrega los condicionales es Create_Shackle_If_All_Stmts(). Siguiendo con el ejemplo obtenemos:

```
DO SC1 = 0, 13, 1
  IF(.TRUE.) THEN
    DO SC2 = 0, 14, 1
      IF(.TRUE.) THEN
        DO J0 = 1, 512, 1
          DO K = 1, 512, 1
            DO I0 = 1, 512, 1
              IF(J0 .LE.((SC1 * 38) + 38)) THEN
                IF(J0 .GT.(SC1 * 38)) THEN
                  IF(I0 .LE.((SC2 * 38) + 38)) THEN
                    IF(I0 .GT.(SC2 * 38)) THEN
                      C(I0, J0) = (C(I0, J0) +(A(I0, K) * B(K, J0)))
                    ENDIF
                  ENDIF
                ENDIF
              ENDIF
            END DO
          END DO
        END DO
      ENDIF
    END DO
  ENDIF
END DO
```

Después de revisar todo el cuerpo de la función en curso y de haber aplicado shackling en aquellos puntos donde el compilador lo ha considerado necesario, se intenta mejorar la eficiencia del código eliminando los IFs añadidos por la transformación. En la sección 4.6 veremos como se lleva a cabo este proceso. Tras la eliminación de condicionales, la multiplicación de matrices del ejemplo tendrá el siguiente aspecto:

```
DO SC1 = 0, 12, 1
  DO SC2 = 0, 12, 1
    DO I0 = ((SC2 * 38) + 1), ((SC2 * 38) + 38), 1
      DO J0 = ((SC1 * 38) + 1), ((SC1 * 38) + 38), 1
        DO K = 1, 512, 1
          C(I0, J0) = (C(I0, J0) +(A(I0, K) * B(K, J0)))
        END DO
      END DO
    END DO
  END DO
  DO SC2 = 13, 13, 1
    DO I0 = ((SC2 * 38) + 1), 512, 1
      DO J0 = ((SC1 * 38) + 1), ((SC1 * 38) + 38), 1
        DO K = 1, 512, 1
```

```
          C(I0, J0) = (C(I0, J0) +(A(I0, K) * B(K, J0)))
        END DO
      END DO
    END DO
  END DO
END DO
DO SC1 = 13, 13, 1
  DO SC2 = 0, 12, 1
    DO I0 = ((SC2 * 38) + 1), ((SC2 * 38) + 38), 1
      DO J0 = ((SC1 * 38) + 1), 512, 1
        DO K = 1, 512, 1
          C(I0, J0) = (C(I0, J0) +(A(I0, K) * B(K, J0)))
        END DO
      END DO
    END DO
  END DO
  DO SC2 = 13, 13, 1
    DO I0 = ((SC2 * 38) + 1), 512, 1
      DO J0 = ((SC1 * 38) + 1), 512, 1
        DO K = 1, 512, 1
          C(I0, J0) = (C(I0, J0) +(A(I0, K) * B(K, J0)))
        END DO
      END DO
    END DO
  END DO
END DO
```

4.5. Caso no simple

En esta sección pasamos a describir la aplicación del shackling cuando en el código a optimizar se realizan escrituras sobre más de una variable. Las principales diferencias respecto al caso simple radican en la clasificación de sentencias y la selección de las data-centric references. Otra diferencia aparece cuando la transformación inicial no es legal. En el caso no simple se cambian las data-centric references y se realiza un segundo intento.

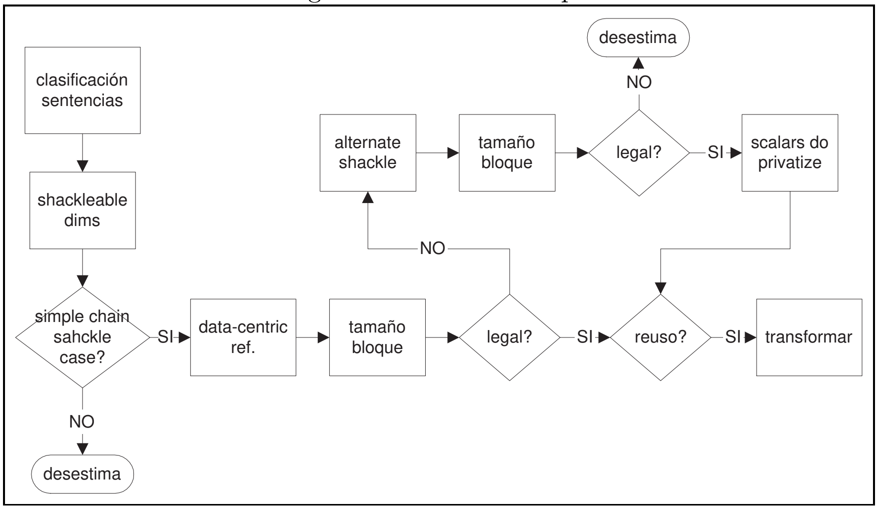
La figura 4.8 muestra un esquema de los pasos seguidos. En este caso, a diferencia de lo que podemos observar en la figura 4.4, el análisis de las dimensiones que se pueden bloquear (shackleable dims) se realiza justo a continuación de la clasificación de sentencias. Además aparece un test antes de proceder a la selección de las data-centric references. También se añade un camino alternativo para cuando la transformación inicial no es legal.

4.5.1. Clasificación de sentencias

El primer paso consiste en la clasificación de las sentencias en clases de equivalencia. Esta clasificación se realiza del siguiente modo:

1. Se busca una variable escalar, que no sea variable de iteración y que no esté indexando un acceso a array, dentro de la condición de un IF o como destino de una escritura.

Figura 4.8: Caso no simple



2. Se marcan las sentencias donde aparece el escalar como miembros de una misma clase.

3. De forma iterativa, se amplia la clase añadiendo las sentencias que contienen definiciones de escalares usados en ella. Añadimos también las sentencias que contienen usos de escalares definidos en sentencias que pertenecen a la clase de equivalencia.

Todas las sentencias de una misma clase de equivalencia, con la excepción del grupo 0, se bloquearán del mismo modo. Es decir, los condicionales que rodearán a cada una de las sentencias serán idénticos en todas ellas (ver sección 4.5.3). Esto implica que es posible que se aplique el shackling a una sentencia tomando como data-centric reference una referencia de otra sentencia.

El objetivo de esta actuación es respetar las dependencias entre escalares del programa original en el código optimizado. Dado el siguiente ejemplo:

```
    DO I
      DO J
S1      t = ...
        ...
Sn      ... = t
      ENDDO
    ENDDO
```

debemos asegurar que siempre que se ejecute una instancia de la sentencia S1 (donde se define el valor de t), se ejecutará también la correspondiente instancia de Sn (el uso de t). Esto se consigue clasificando ambas sentencias en la misma clase de equivalencia. Como veremos en el apartado 4.5.5 esta política no es suficiente.

El tratamiento de los escalares se describe en el apartado 3.4.2 de [9]. Respecto a los condicionales, cuando en el test interviene un escalar que no es una variable de iteración, se deben clasificar todas las sentencias del cuerpo del condicional en la misma clase de equivalencia. De este modo, se garantiza que cada vez que se cumple la condición se ejecutan todas las sentencias del cuerpo del condicional. Observamos que tal como se ha descrito el procedimiento de clasificación de sentencias esto no tiene porqué ocurrir. Por tanto, en lo que a los condicionales se refiere, el compilador no clasifica las sentencias tal y como se describe en [9].

Veamos como se clasificarían las sentencias del siguiente código:

```
    DO I=1,N
      DO J=1,N
        IF (MOD(I,2).EQ.0) THEN
S1        s=i*j
        ENDIF
S2      A(2*i,2*j)=s
S3      A(2*i,3*j+1) = B(i,j+1)
      ENDDO
    ENDDO
```

En S1 aparece una escritura sobre un escalar y se asigna a la clase de equivalencia 1. Se añade S2, que contiene un uso de la definición en S1. Éste es el resultado de la clasificación:

```
S = REAL((I0 * J0)):    1
A((I0 * 2), (J0 * 2)) = S:    1
A((I0 * 2), (J0 * 3) + 1) = B(I0, J0 + 1):    0
```

Para comprobar que no todas las sentencias de un condicional se van a clasificar dentro del mismo grupo, modificamos el ejemplo anterior añadiendo un condicional que depende de s.

```
    DO I=1,N
      DO J=1,N
        IF (MOD(I,2).EQ.0) THEN
S1        s=i*j
        ENDIF
        IF (s>100) THEN
S2        A(2*i,2*j)=s
S3        A(2*i,3*j+1) = B(i,j+1)
        ENDIF
      ENDDO
    ENDDO
```

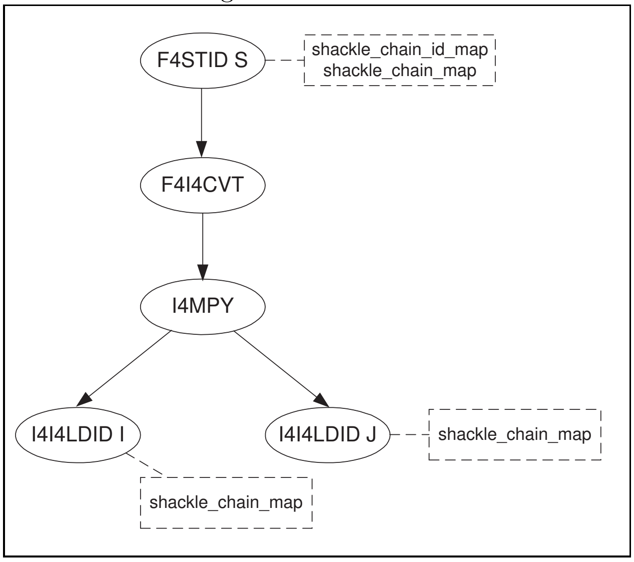
Tras la modificación del programa, la clasificación no varía:

```
S = REAL((I0 * J0)):    1
A((I0 * 2), (J0 * 2)) = S:    1
A((I0 * 2), (J0 * 3) + 1) = B(I0, J0 + 1):    0
```

El algoritmo de clasificación de las sentencias utiliza los siguientes mapeos:

shackle_chain_map: Marca cada nodo como visitado o no visitado.

Figura 4.9: S = I * J



shackle_chain_id_map: Asocia a cada sentencia el identificador de la clase de equivalencia a la que pertenece.

Con estos mapeos se añaden nuevos campos a todos los nodos, aunque sólo los vamos a utilizar para marcar los loads y stores. La figura 4.9 muestra la representación en WHIRL de la sentencia `S = REAL((I0 * J0))`. El nodo $F4STID\ S$ corresponde a la asignación sobre S. En este caso se utilizarán ambos mapeos. Los siguientes nodos representan la operación de conversión de entero a coma flotante y el producto. En los nodos correspondientes a los operandos sólo utilizaremos el mapeo *shackle_chain_map*.

De forma algo más detallada, los pasos seguidos por el algoritmo de clasificación son los siguientes:

1. Se busca un IF o un STORE que contenga una variable escalar que no sea de iteración y que no esté indexando un array. Inicializa *stmt* para que apunte a la sentencia encontrada.

2. Mientras $stmt \neq NULL$

   a) Incrementa el contador *shackle_chain_id_map_cnt*.

   b) Crea una lista con todas las ocurrencias en *stmt* de escalares que no sean variables de iteración. Marca estos escalares como visitados.

   c) Mientras la lista contenga elementos por tratar:

      Si es un load:
      - Obtiene la lista de definiciones.
      - Inicializa chain_id de la sentencia de asignación.
      - Añade store de la definición a la lista.

      Si es un store:

- Marca nodo como visitado.
- Obtiene la lista de usos.
- Para cada uso:
  Inicializa el chain_id de la sentencia que lo contiene.
  Marca todos los usos de escalares de la sentencia como visitados y los añade a la lista.

d) Busca una nueva sentencia con ocurrencias no visitadas de variables escalares no índice. Si no la encuentra $stmt = NULL$ y el algoritmo termina.

Aplicando este algoritmo al ejemplo

```
S1  x = y+z
S2  y = u+z
...
Sn  u = ...
```

podría ocurrir lo siguiente:

Primero generamos una lista con los escalares de S1 {*st_x, ld_y, ld_z*}. Cuando pasamos a tratar *ld_y* (suponemos que *st_x* ya se ha tratado) marcamos S2 como perteneciente a la misma clase que S1 y añadimos *st_y*. Tratamos *ld_y*, *ld_z* y cuando llegamos a *st_y* marcamos el nodo como visitado, buscamos los usos de *y*, los añadimos a la lista e inicializamos los chain_ids de las sentencias correspondientes. En este punto aparece un problema. Como los usos de *u* y *z* en S2 no se añaden a la lista, la sentencia Sn no va a clasificarse en el mismo grupo que S1 y S2. Sn será tratada al generar la siguiente clase de equivalencia.

4.5.2. Shackleable dims

Se genera la lista de arrays donde se indica qué dimensiones se pueden bloquear en cada uno de ellos tal como se explica en la sección 4.4.3.

En el ejemplo de descomposición LU que podemos ver en la figura 4.11 (A), el array A tiene una shackleable dimension, que corresponde a la segunda dimensión del array. La primera dimensión no puede bloquearse porque hay dos referencias de A donde la expresión que aparece en esta dimensión contiene una variable no índice. En ambos casos se trata de la variable M.

4.5.3. Data-centric reference

Antes de determinar cual será la data-centric reference de cada sentencia se debe inicializar la variable global $shackle\_chain\_info$. $shackle\_chain\_info$ es un vector que contiene una entrada para cada una de las clases de equivalencia (excepto la clase 0). En esta estructura de datos se guarda, para cada clase de equivalencia, la lista de las referencias a arrays contenidas en esa clase que son válidas para generar un data shackle.

Sea $A(expr_1, \ldots, expr_n)$ una referencia perteneciente a una clase de equivalencia distinta de cero, para que $A(expr_1, \ldots, expr_n)$ sea una referencia válida debe cumplir:

> Para todo j tal que la dimensión j del array A sea una shackleable dimension (ver sección 4.4.3), las variables que aparecen en $expr_j$ son todas

variables de iteración de bucles que contienen todas las referencias de la clase.

La inicialización de *shackle_chain_info* se realiza llamando a la función Shackleable_Refs_From_All_Chains(). Tras la inicialización se hacen las siguientes comprobaciones, que corresponden al test "simple chain shackle case?" de la figura 4.8:

1. Se obtienen las sentencias de la clase 0. Si el array que aparece a la izquierda de las asignaciones no es siempre el mismo abortamos el shackling.

2. Se comprueban el resto de clases. Todas ellas deben contener, como mínimo una ocurrencia válida del array destino de las escrituras de la clase 0. Esta información se obtiene consultando *shackle_chain_info*. Si alguna de las clases no cumple esta condición también abortamos el shackling.

Finalmente, mediante el mapeo shackle_shackle_map asociamos a cada sentencia su data-centric reference:

clase 0: la data-centric reference de cada sentencia es el destino de la asignación.

clases >0: la data-centric reference se obtiene de la lista de referencias de la clase almacenadas en *shackle_chain_info*. Es la primera ocurrencia en esta lista de la variable destino de la clase 0 y es la misma para todas las sentencias de la clase.

En la figura 4.11 (C) podemos ver a qué clase de equivalencia, o grupo, pertenecen las sentencias de la descomposición LU (A) y cual es la data-centric reference que se asigna a cada una de ellas. Todas las sentencias del grupo 1 tienen como data-centric reference la primera ocurrencia del vector A dentro de las referencias de este grupo. Aunque la primera dimensión de dicha referencia contiene una variable de iteración (I) de un bucle que no contiene a todas las sentencias del grupo 1, la referencia es válida porque la primera dimensión de A no se va a bloquear.

4.5.4. Tamaño de bloque

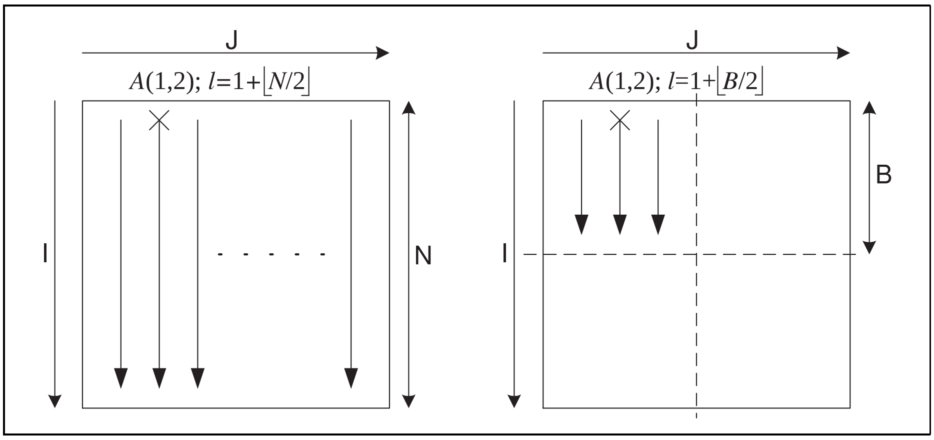
El cálculo del tamaño de bloque no varía respecto al caso simple. Consultar sección 4.4.4.

4.5.5. Legalidad de la transformación

Una vez seleccionadas las data-centric references y calculado el tamaño de bloque se procede a verificar la legalidad de la transformación.

A diferencia del caso anterior, en el caso no simple sí se deben tener en cuenta las dependencias entre escalares. En lo que a dependencias entre accesos a array se refiere, todo sigue como se ha descrito en la sección 4.4.5.

Figura 4.10: Bloqueo de la matriz A



Dependencias entre escalares

Cuando se valida la legalidad del shackling no se realiza ninguna comprobación especial en lo que a escalares se refiere. Se supone que la clasificación de las sentencias en clases de equivalencia, junto con la elección adecuada de la data-centric reference para cada clase (ver sección 4.5.3), darán lugar a un bloqueo donde las posibles dependencias entre escalares serán respetadas. Esto es cierto cuando nos encontramos ante variables privatizables, pero no en general. En el siguiente ejemplo mostramos como esta estrategia no es suficiente, ya que existe una dependencia de flujo entre iteraciones sucesivas que no va a ser respetada.

```
     l=1
     DO J=1,N
       DO I=1,N
S1       A(i,j) = l
         IF (MOD(I,2).EQ.0) THEN
S2         l = l+1
         ENDIF
S3       A(i,j)=2*A(i,j)
       ENDDO
     ENDDO
```

La implementación actual del shackling aplicaría bloqueo tal y como se muestra en la figura 4.10. Esta transformación es incorrecta ya que el valor de l que se recibe al calcular la posición $A(1,2)$ es distinto antes y después de aplicar bloqueo. Antes del bloqueo, cuando calculamos $A(1,2)$ se han ejecutado N iteraciones del bucle I. Como la sentencia condicional se ejecuta la mitad de las veces $l = 1 + \lfloor \frac{N}{2} \rfloor$. Tras aplicar la transformación, sea B el factor de bloqueo, al llegar al cálculo de $A(1,2)$, el valor de l será $l = 1 + \lfloor \frac{B}{2} \rfloor$.

Dependencias entre accesos a array

No hay cambios respecto al caso simple. Ver sección 4.4.5.

4.5.6. Shackle alternativo

En el caso no simple, si en el primer intento no se consigue un shackling legal, se buscan data-centric references alternativas y se procede a un segundo intento tal y como se describe en esta sección.

Obtención de data-centric references alternativas

Para cada sentencia que no pertenezca al grupo 0, se toma como data-centric reference alternativa la primera referencia encontrada (dentro de la sentencia) que acceda al mismo array que la data-centric reference original. Este proceso se lleva a cabo en la función Create_Alternate_Simple_Chain_Shackle().

Inicialmente, sólo eran válidas aquellas referencias indexadas por variables de iteración de bucles que contuvieran todas las sentencias de la clase de equivalencia. Al realizar la selección alternativa esta condición se ignora.

Es posible que la data-centric reference alternativa coincida con la data-centric reference original. También puede ocurrir que una sentencia no contenga ningún acceso al array de la data-centric reference original, en cuyo caso ésta no se modifica.

En la descomposición LU de la figura 4.11, la primera selección de data-centric references da lugar a una transformación no legal. En la última columna de la tabla que acompaña a la figura podemos ver cuales son las referencias alternativas que tomaremos para las sentencias del grupo 1.

Lista de escalares

Al mismo tiempo que se actualizan las data-centric references se crea una lista con las variables escalares no índice de bucle que aparecen dentro de las referencias alternativas que se van tomando. Todos los escalares de esta lista deben ser fácilmente privatizables, es decir, se pueden privatizar y además no es necesario generar el valor final de la variable a la salida del bucle. En caso contrario se aborta el shackling. Nos preguntamos porqué esta comprobación no se realiza también sobre los escalares que determinan las clases de equivalencia.

La lista de escalares del ejemplo de la figura 4.11 estaría formada por un único escalar que sería la variable M. El resto de variables que aparecen en las referencias del grupo 1 son todas variables de iteración.

Tamaño de bloque y legalidad

El siguiente paso consiste en recalcular el tamaño de bloque, cosa que en principio no parece necesaria, puesto que el tamaño de bloque no depende de cuales son las data-centric references escogidas.

A continuación se evalúa la legalidad de la transformación con las nuevas data-centric references.

Privatización de escalares

Con la elección inicial, todas las sentencias de un grupo distinto del cero comparten la misma data-centric reference. Sea L el bucle más interno que contiene todas las sentencias de la clase, si los escalares que aparecen son privatizables

Figura 4.11: Shackle alternativo

```
DO K = 1, 2000, 1
  TEMP = 0.0D00
  M = K
  DO I__0 = K, 2000, 1
    D = A(I__0, K)
    IF(TEMP .LT. ABS(D)) THEN
      TEMP = ABS(D)
      M = I__0
    ENDIF
  ENDDO
  IF(K .ne. M) THEN
    DO J__0 = K, 2000, 1
      TEMP__0 = A(K, J__0)
      A(K, J__0) = A(M, J__0)
      A(M, J__0) = TEMP__0
    ENDDO
  ENDIF
  DO I__1 = (K + 1), 2000, 1
    A(I__1, K) = (A(I__1, K) / A(K, K))
  ENDDO
  DO I__2 = (K + 1), 2000, 1
    DO J__1 = (K + 1), 2000, 1
      A(I__2, J__1) = (A(I__2, J__1) -
                (A(K, J__1) * A(I__2, K)))
    ENDDO
  ENDDO
ENDDO
```

(A) código que recibe la etapa de shackling

```
DO K = 1, 2000, 1
  TEMP = 0.0D00
  deref_se1_I__0(K) = K
  DO I__0 = K, 2000, 1
    D = A(I__0, K)
    IF(TEMP .LT. ABS(D)) THEN
      TEMP = ABS(D)
      deref_se1_I__0(K) = I__0
    ENDIF
  END DO
  IF(K .ne. deref_se1_I__0(K)) THEN
    DO J__0 = K, 2000, 1
      TEMP__0 = A(K, J__0)
      A(K, J__0) = A(deref_se1_I__0(K), J__0)
      A(deref_se1_I__0(K), J__0) = TEMP__0
    END DO
  ENDIF
  DO I__1 = (K + 1), 2000, 1
    A(I__1, K) = (A(I__1, K) / A(K, K))
  END DO
  DO I__2 = (K + 1), 2000, 1
    DO J__1 = (K + 1), 2000, 1
      A(I__2, J__1) = (A(I__2, J__1) -
                  (A(K, J__1) * A(I__2, K)))
    END DO
  END DO
END DO
```

(B) código después de aplicar expansión a M

	grupo	*data-centric ref.*	*ref. alternativa*
TEMP = 0.0D00	1	A(I__0, K)	A(I__0, K)
M = K	1	"	"
D = A(I__0, K)	1	"	"
TEMP = ABS(D)	1	"	"
M = I__0	1	"	"
TEMP__0 = A(K, J__0)	1	"	A(K, J__0)
A(K, J__0) = A(M, J__0)	1	"	"
A(M, J__0) = TEMP__0	1	"	A(M, J__0)
A(I__1, K) = (A(I__1, K) / A(K, K))	0	A(I__1, K)	A(I__1, K)
A(I__2, J__1) = (A(I__2, J__1) - ...	0	A(I_2,J_1)	A(I_2,J_1)

(C) clasificación en grupos y data-centric references respecto a L, esta elección de data-centric references garantiza que la transformación respetará las dependencias entre escalares. Dentro de una iteración de L el orden de ejecución de las sentencias de la clase no se ve alterado.

En el shackle alternativo se relajan las condiciones y se permite que las sentencias de un grupo distinto del cero puedan tener data-centric references distintas. Esto puede producir que, dentro de una iteración de L, algunas de las sentencias de la clase se ejecuten y que otras no lo hagan porque su data-centric reference no accede al bloque en curso. Parece que éste sería el motivo por el que se va a aplicar expansión de escalares. Los escalares que se van a expansionar son los que aparecen en alguna de las data-centric references alternativas.

No estamos seguros de que este mecanismo garantice la legalidad de la transformación respecto a los escalares. Creemos que podría darse una situación donde, por ejemplo, a causa del shackling una definición se retrasara respecto a alguno de sus usos dando lugar a código incorrecto.

La figura 4.11 (B) muestra el resultado de la expansión aplicada a la variable M, que aparece en dos de las data-centric references alternativas.

4.5.7. Reuso

Al igual que en el caso simple, antes de aplicar la transformación, se comprueba que exista reuso de los datos. Ver sección 4.4.7.

4.5.8. Aplicar transformación

La transformación del código para añadir el recorrido de los bloques y los condicionales que limitan los accesos al bloque en curso se lleva a cabo tal como se muestra en la sección 4.4.8.

4.6. Eliminación de condicionales

En la sección 4.4.8 hemos visto como se transforma el código al aplicar shackling. Las modificaciones consisten en la inclusión de bucles adicionales para el recorrido de los bloques y la protección de cada una de las sentencias mediante condicionales. En el programa resultante va a ser necesaria la evaluación de los condicionales en cada iteración de los bucles internos, cosa que va a afectar negativamente el rendimiento. En esta sección mostramos como se eliminan dichos condicionales. Para una descripción más formal y amplia consultar [9].

Cuando encontramos un condicional dentro de un bucle existen dos posibilidades (ver figura 4.12):

(a) El condicional es invariante respecto al bucle más interno.

(b) El condicional no es invariante respecto al bucle más interno.

En el primer caso, mover el IF fuera del bucle más interno no presenta ninguna dificultad (ver figura 4.13). Creamos dos copias del bucle, protegemos una de las copias con la condición que queremos sacar fuera y la otra con su complementario (ver figura 4.13 (b)). Hecho esto, podemos eliminar los condicionales del interior de los bucles, ya que su resultado siempre será el mismo (4.13 (c)). Aplicando este procedimiento al ejemplo de la figura 4.12 (a) obtendremos el resultado mostrado en la figura 4.14 (a).

Figura 4.12: Condicionales dentro de un bucle

```
do i=1,100
  S0(i)
  do j=1,200
    if (i<=50)
      S1(i,j)
    S2(i,j)
  S3(i)
```

(a)

```
do i=1,100
  S0(i)
  do j=1,200
    if (i<=j)
      S1(i,j)
    S2(i,j)
  S3(i)
```

(b)

Figura 4.13: Condicional invariante respecto al bucle interno

```
do i
  do j
    if (F(i)) S1
    S2
```

(a)

```
do i
  if (F(i))
    do j
      if (F(i)) S1
      S2
  if (!F(i))
    do j
      if (F(i)) S1
      S2
```

(b)

```
do i
  if (F(i))
    do j
      S1
      S2
  if (!F(i))
    do j
      S2
```

(c)

Aplicando este proceso repetidamente llegará un momento en que el condicional sí va a depender del bucle más interno que lo contenga (figura 4.14 (a)). Cuando esto ocurra podemos eliminar el condicional mediante la aplicación de *index set splitting*. Esta transformación consiste en la división del rango de valores del índice del bucle en dos intervalos: uno en que la condición siempre se cumple y otra en que ésta siempre dará falso. A partir de ahí se generan dos copias del bucle, una para cada uno de los intervalos.

Si tomamos la condición ($i \leq 50$) de la figura 4.14 (a), dividiremos las iteraciones iniciales [1..100] en los intervalos [1..50] y [51..100]. A continuación generaremos un bucle para cada uno de los intervalos (ver figura 4.14 (b)). Después de simplificar se obtiene el código de la figura 4.14 (c)).

Es importante observar que para una aplicación eficiente de *index set splitting* es necesario poder asegurar que el punto por donde se divide el rango de valores del índice se encuentra dentro de los límites del bucle.

A continuación pasamos a describir la implementación de la eliminación de condicionales. Todas las funciones referenciadas en las secciones 4.6.1 y 4.6.2 se

Figura 4.14: Condicional no invariante respecto al bucle interno

```
do i=1,100
  S0(i)
  if (i<=50)
    do j=1,200
      S1(i,j)
      S2(i,j)
  if (i>=51)
    do j=1,200
      S2(i,j)
  S3(i)
```

(a)

```
do i=1,50
  S0(i)
  if (i<=50)
    do j=1,200
      S1(i,j)
      S2(i,j)
  if (i>=51)
    do j=1,200
      S2(i,j)
  S3(i)
do i=51,100
  S0(i)
  if (i<=50)
    do j=1,200
      S1(i,j)
      S2(i,j)
  if (i>=51)
    do j=1,200
      S2(i,j)
  S3(i)
```

(b)

```
do i=1,50
  S0(i)
  do j=1,200
    S1(i,j)
    S2(i,j)
  S3(i)
do i=51,100
  S0(i)
  do j=1,200
    S2(i,j)
  S3(i)
```

(c)

definen en el archivo /be/lno/shackle_ifs.cxx.

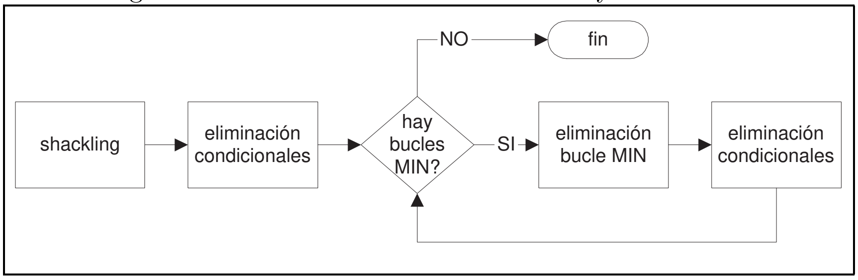
4.6.1. Algoritmo

El algoritmo que dirige la etapa de eliminación de condicionales consta de los siguientes pasos:

1. Construcción de la lista de sentencias.
2. Mientras queden condicionales por tratar:
   a) Seleccionar condicional.
   b) Tratar el condicional (existen dos posibilidades):
      - Aplicación de index set splitting.
      - Promoción de condicionales.
   c) Simplificar el código resultante (eliminación de casos triviales y código inútil).
   d) Construcción de la lista de sentencias.

Para garantizar la terminación del algoritmo se marcan los condicionales ya tratados, de modo que si no es posible eliminar un condicional éste no sea seleccionado una y otra vez.

Figura 4.15: Eliminación de condicionales y bucles MIN



Aquellos condicionales que se encuentren dentro de un bucle cuyo límite superior se define con la función MIN no pueden ser eliminados directamente. Primero debemos modificar el código para hacer desaparecer este tipo de bucles. Como veremos en la sección 4.7 al aplicar esta transformación también aparecen nuevos condicionales en el programa. Cada vez que se elimina un bucle que hace uso de la función MIN se repite el proceso de eliminación de condicionales. Ver figura 4.15.

La eliminación de condicionales no distingue entre los condicionales añadidos durante la etapa de shackling y aquellos condicionales que ya aparecían en el código. Además, la eliminación de condicionales se lleva a cabo sobre todo el cuerpo de la función en curso y no sólo en aquellas partes de la función donde se ha aplicado shackling. Si no se ha aplicado shackling a ningún punto de la función la eliminación de condicionales no se lleva a cabo.

El proceso de eliminación de condicionales se lleva a cabo en la función analyze_stmts_in_func_for_if(), definida en /be/lno/shackle_ifs.cxx. A continuación pasamos a detallar cada uno de los pasos del algoritmo.

Selección del condicional

Inicialmente, debido a que protegen sentencias, todos los condicionales añadidos durante el shackling están al nivel más interno posible. Sólo una vez iniciada la eliminación de condicionales podremos encontrar condicionales originalmente procedentes del shackling que contengan algún bucle.

El criterio seguido para la selección de condicionales es, en primer lugar, seleccionar un condicional no tratado que contenga algún bucle.

Si no se encuentra ninguno, tomaremos el primer condicional no tratado que encontremos.

De este modo, dejando a un lado posibles condicionales que ya estuvieran en el código (que también se van a tratar), el proceso consiste en seleccionar un condicional que inicialmente estará rodeando una sentencia e ir llevándolo hacia fuera tanto como sea posible.

Find_Unseen_If_Outside_Do() y Find_Stmt_With_Enclosing_If() son las funciones utilizadas en la selección del condicional. Estas funciones no devuelven directamente el condicional si no una sentencia. Posteriormente se toma el condicional más externo que no haya sido tratado y que contenga a dicha sentencia.

Tratamiento del condicional

Una vez determinado qué condicional se va a tratar, se comprueba que éste no contiene términos no lineales y que, por tanto, se trata de una expresión afín. Esta comprobación se hace sobre el access_array que expresa la condición.

A continuación se obtiene el bucle con el que se va a tener que combinar el condicional. Se selecciona el bucle más interno respecto al cual la condición es dependiente.

Si la condición es independiente de todos los bucles (todos los coeficientes de los índices igual a 0) hay dos posibilidades:

1. Se trata de una condición trivial tipo "IF (TRUE) THEN ..." o "IF (FALSE) THEN ...". Se elimina el condicional conservando o eliminando el cuerpo del IF respectivamente.

2. La condición no depende de ningún bucle. En este caso sólo vamos a mover la condición fuera de los bucles si no hay otras sentencias al mismo nivel que el condicional. Si hubiera otras sentencias tendríamos que replicar todo el código generando una versión para el caso en que se cumple el condicional y otra para cuando éste da falso. Cuando esto ocurra simplemente marcaremos el condicional como visitado sin modificar el código.

Cuando el condicional sí es dependiente de alguno de los bucles hay cuatro caminos posibles.

1. La condición siempre devolverá falso.

2. La condición siempre devolverá cierto.

3. Index-set splitting.

4. Promoción de condicionales.

Si bien en este punto (función Sink_If2do()) los dos primeros casos se contemplan, en la práctica nunca tomamos esos caminos porque estos casos ya son tratados durante la simplificación del código al final de cada iteración del algoritmo de eliminación de condicionales.

Cuando se aplica index set splitting se toma un bucle cuyo índice toma valores entre un límite inferior LB y un límite superior UB y se sustituye por dos bucles: el primero de ellos recorrerá las iteraciones que van desde LB hasta un determinado valor $UB(C)$, que depende de la condición C, el segundo recorrerá las iteraciones que van desde $LB(C)$ hasta UB, donde $LB(C) = UB(C)+1$. Para una correcta aplicación de index set splitting debemos asegurar que:

$$LB(C) \geq LB, \qquad \text{y} \qquad UB(C) \leq UB$$

El valor de $UB(C)$ y $LB(C)$ se obtiene aislando la variable de iteración del bucle dentro de la condición C (ver documento [9], página 85). La función encargada de comprobar que dichos valores caen dentro de los límites del bucle se llama Is_Provably_In_Bounds(). A continuación mostramos, mediante un ejemplo, el funcionamiento de esta función.

```
DO SA1=0,9
  DO SA2=0,9
    DO K=1,2000
      IF K<=SA1*218+218
      ...
```

Se calcula el valor de los límites inducidos por el condicional:

$$LB(C) = 218 \cdot SA1 + 219 \qquad ,\text{y} \qquad UB(C) = 218 \cdot SA1 + 218$$

Primero se intenta probar que $LB(C) \geq LB$. Para ello se genera un sistema de ecuaciones con las restricciones introducidas por los límites de todos los bucles y condicionales que contienen el IF que estamos tratando. A este sistema se le añade la ecuación $LB(C) < LB$. Si el sistema es inconsistente podemos asegurar que $LB(C) \geq LB$.

Siguiendo con el ejemplo, el sistema de ecuaciones generado es:

$SA1$	$SA2$	K	$\leq$	$ct.$			
		-1		-1	K	$\geq$	1
		1		2000	K	$\leq$	2000
	-1			0	$SA2$	$\geq$	0
	1			9	$SA2$	$\leq$	9
-1				0	$SA1$	$\geq$	0
1				9	$SA1$	$\leq$	9
218				-219	$LB(C)$	$<$	LB

Este sistema es inconsistente, por tanto, podemos asegurar que $218 \cdot SA1 + 219 \geq 1$.

Procedemos de forma análoga para probar que $UB(C) \leq UB$. El sistema está formado por las mismas condiciones que en el caso anterior, excepto la última, que en este caso va a expresar que $UB(C) > UB$.

$SA1$	$SA2$	K	$\leq$	$ct.$			
		-1		-1	K	$\geq$	1
		1		2000	K	$\leq$	2000
	-1			0	$SA2$	$\geq$	0
	1			9	$SA2$	$\leq$	9
-1				0	$SA1$	$\geq$	0
1				9	$SA1$	$\leq$	9
-218				-1783	$UB(C)$	$>$	UB

En este caso, la evaluación dará consistente y, en consecuencia, no podemos asegurar que $218 \cdot SA1 + 218 \leq 2000$. Por tanto no podemos aplicar index set splitting de forma directa. Esto nos lleva al último caso en el tratamiento del condicional, es decir, a la promoción.

La función Maybe_Handle_Sink_Promotion_Case() se encarga de acondicionar el código cuando no podemos aplicar index set splitting directamente. Esta función crea cuatro réplicas del bucle que, de momento, no podemos dividir y protege cada una de las copias del siguiente modo:

```
IF (UB(C)>=LB) THEN
  IF (UB(C)>UB) THEN
```

```
      ...
    IF (UB(C)<=UB) THEN
      // Aquí podremos aplicar index set splitting
      ...
  IF (UB(C)<LB)
    IF (UB(C)>UB)
      // Caso inconsistente
      ...
    IF (UB(C)<=UB)
      ...
```

Volviendo al ejemplo, este sería el resultado de su aplicación:

```
IF (SA1>=0) THEN          (SA1·218 + 218 ≥ 1)
  IF (SA1>8) THEN         (SA1·218 + 218 > 2000)
    ...
  IF (SA1<=8) THEN        (SA1·218 + 218 ≤ 2000)
    // ISS
    ...
IF (SA1<0) THEN           (SA1·218 + 218 < 1)
  IF (SA1>8) THEN         (SA1·218 + 218 > 2000)
    ...
  IF (SA1<=8) THEN        (SA1·218 + 218 ≤ 2000)
    ...
```

En iteraciones sucesivas se elimina el condicional original mediante la aplicación de index set splitting, o bien, simplificando el código en el resto de casos.

Simplificación del código

Finalmente, antes de pasar a la siguiente iteración, se eliminan todos aquellos condicionales que van a devolver siempre cierto o siempre falso, y cuyo resultado se puede determinar en tiempo de compilación.

La función Remove_Redundant_And_Inconsistent_If() realiza un recorrido en profundidad de la representación del código. Cada vez que encuentra un condicional se crea un sistema de ecuaciones que refleja todas las restricciones impuestas por los bucles y los condicionales que contienen el IF en curso.

Para saber si el condicional siempre va a devolver falso necesitamos demostrar que todas las restricciones $(r_1, ..., r_n)$ que llegan al IF más la condición C de éste no pueden darse simultáneamente. La función is_vector_inconsistent() genera el sistema de ecuaciones, al que se añade la condición del IF. Si el sistema es inconsistente, efectivamente, el condicional siempre va a devolver falso y el código incluido en él nunca va a ser ejecutado. Se elimina la condición y su cuerpo.

Para asegurar que el condicional siempre va a devolver cierto se debe demostrar que el cumplimiento de todas las restricciones que le afectan implican que éste también se cumple, es decir, que $r_1 \wedge \cdots \wedge r_n \rightarrow C$ o, lo que es lo mismo, que no es posible que $r_1 \wedge \cdots \wedge r_n \wedge \neg C$ se cumplan simultáneamente. La función is_vector_trivial() genera el sistema de ecuaciones, al que añade el complementario de la condición. Si el sistema es inconsistente, el IF siempre devolverá cierto y por tanto puede ser eliminado conservando su cuerpo.

4.6.2. Errores de implementación

En la siguiente sección comentamos dos errores de la eliminación de bucles que pueden dejar al compilador colgado en un bucle infinito.

Error al simplificar expresiones

La función Remove_Consts_From_Conditionals() realiza una simplificación de forma incorrecta. Veamos el siguiente ejemplo:

```
DO SA1=0,8
  DO SA2=0,SA1
    ...
    DO K=((SA1*218)+1), ((SA1*218)+218)
      IF ((K+2) .LT. ((SA2 *218) + 218)) THEN
        ...
```

En este caso no se puede aplicar index set splitting directamente, así que se van a generar cuatro copias del bucle protegidas por nuevos condicionales (ver sección 4.6.1). Al montar la condición $UB(C) \geq LB$ se genera $SA2 \cdot 218 + 215 \geq SA1 \cdot 218 + 1$, que al simplificar de forma errónea se convierte en $SA2 > SA1$. La condición $SA2 > SA1$ siempre va a devolver falso, mientras que la condición correcta $SA2 \geq SA1$ tiene solución cuando $SA2 = SA1$.

Debido a que los nuevos condicionales se generan de forma errónea, se produce una situación en que tras eliminar los condicionales redundantes e inconsistentes se vuelve al código original. De este modo se entra en un bucle infinito que repite el proceso una y otra vez.

Error al generar condicionales Too_Messy

Cuando no se puede aplicar index set splitting de forma directa, y el límite superior del bucle contiene la función MIN, vamos a encontrarnos con los siguientes problemas. El primero de ellos es que el compilador va a generar ecuaciones a partir de access vectors marcados como Too_Messy, cuyo valor es indeterminado. Podemos corregir esta situación descartando los IFs Too_Messy. El segundo problema es que, tanto si se generan ecuaciones erróneas como si no se generan ecuaciones para estos IFs, podemos entrar en un bucle infinito y que la compilación no acabe.

Como hemos visto en el apartado 4.6.1, cuando no podemos aplicar index set splitting se generan cuatro copias del bucle. Estas copias se protegen con condicionales y, puesto que nos referimos al caso en que el límite superior del bucle utiliza la función MIN, algunos de los IFs que se van a añadir van a contener también la función MIN.

Los access vectors de los condicionales que contengan la función MIN serán marcados como Too_Messy y su valor será indeterminado. Esto va a provocar que al realizar los análisis para comprobar si el condicional del interior del bucle puede ser eliminado, estos condicionales Too_Messy de valor indeterminado generen ecuaciones erróneas. Si el resultado del test es que el condicional no puede ser eliminado podemos entrar en un bucle infinito donde el condicional es seleccionado y tratado indefinidamente.

Ante un código como el siguiente:

```
DO I0 = J0, MIN((J0 + K), (K + 2000)), 1
  IF(I0 .LE.((SA2 * 218) + 218)) THEN
    IF(I0 .GT.(SA2 * 218)) THEN
      A(I0, J0) = (A(I0, J0) -(A(I0, K) * A(J0, K)))
    ENDIF
  ENDIF
END DO
```

El compilador no será capaz de determinar si el condicional interno puede ser eliminado, repitiendo el mismo proceso una y otra vez.

```
⋮
IF(((SA2 * 218) + 218) .GT. MIN((K + J0), (K + 2000))) THEN
  IF(((SA2 * 218) + 218) .GT. MIN((K + J0), (K + 2000))) THEN
    IF(((SA2 * 218) + 218) .GT. MIN((K + J0), (K + 2000))) THEN
      IF(((SA2 * 218) + 218) .GT. MIN((K + J0), (K + 2000))) THEN
        IF(((SA2 * 218) + 218) .GT. MIN((K + J0), (K + 2000))) THEN
          DO I0 = J0, MIN((J0 + K), (K + 2000)), 1
            IF(I0 .LE.((SA2 * 218) + 218)) THEN
              IF(I0 .GT.(SA2 * 218)) THEN
                A(I0, J0) = (A(I0, J0) -(A(I0, K) * A(J0, K)))
              ENDIF
            ENDIF
          END DO
```

Esta situación puede evitarse si no permitimos que se aplique promoción al condicional, y marcándolo como visitado, cuando detectamos que el bucle correspondiente contiene la función MIN.

4.7. Eliminación de bucles MIN

Por último, antes de abandonar la etapa de shackling, se procede a eliminar aquellos bucles cuyo límite superior se define con la función MIN (ver figura 4.15).

De esta tarea se encarga la función Convert_Do_Loops_Conditionals() que, al igual que analyze_stmts_in_func_for_if() (también referenciada en esta sección), se define en /be/lno/shackle_ifs.cxx. Los pasos seguidos son:

1. De forma recursiva, se recorre la representación del programa hasta encontrar un bucle cuyo límite superior se defina mediante MIN.

2. Dado el bucle *DO lím_inf,MIN(expr1,expr2)*, se protege el bucle con los siguientes condicionales:

   - *IF expr1≥lím_inf THEN ...*
   - *IF expr2≥lím_inf THEN ...*

3. Se crea una copia del bucle, condicionales incluidos. Se protege el bucle original con la condición *IF expr1 > expr2 THEN ...* y la copia con la condición *IF expr1 ≤ expr2 THEN ...*

4. Elimina la función MIN de los límites de los bucles.

5. Llama a analyze_stmts_in_func_for_if() para repetir el proceso de eliminación de condicionales.

6. Llama recursivamente a Convert_Do_Loops_Conditionals().

Este proceso termina cuando la función ya no contiene bucles cuyo límite superior se defina mediante la función MIN.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

El objetivo de este documento es dar a conocer con un elevado nivel de detalle algunas partes del optimizador de bucles de los compiladores *Open64* y *ORC*.

En la introducción enumerábamos las principales transformaciones realizadas durante la optimización de bucles.

En el capítulo 2 se han descrito los grafos de dependencias de datos utilizados durante la etapa de LNO. Éstos son el grafo de dependencias entre arrays y los grafos de niveles de dependencia. Hemos visto como se realiza el cálculo de dependencias para la construcción del grafo de dependencias entre arrays y como, a partir de éste, podemos obtener los grafos de niveles de dependencia, que incluyen también información de dependencias entre escalares.

A continuación, en el capítulo 3, hemos visto como el compilador identifica aquellas anidaciones de bucles que va a tratar y como aplica fusión y distribución para generar SNLNs de profundidad máxima.

Para terminar, en el capítulo 4, se ha visto como el compilador intenta bloquear los accesos a la cache de segundo nivel mediante la aplicación de shackling. Esta optimización sólo se aplica en algunas anidaciones imperfectas y hemos podido observar que presenta algunos problemas de implementación. Esos problemas se traducen en la posibilidad de que el compilador quede colgado ejecutando un bucle infinito o que, en algunos casos, se pueda llegar a generar código erróneo.

5.2. Trabajo futuro

Nuestro objetivo más inmediato es el estudio detallado y la documentación de las optimizaciones de tiling, intercambio y outer loop unrolling. Las decisiones sobre cómo aplicar estas transformaciones se basan en un modelo unificado que explora las distintas posibilidades e intenta encontrar la mejor combinación. Continuaremos con la aplicación de distribución en los bucles más internos, cuyo propósito es reducir la presión sobre los registros. Este material, junto con lo expuesto en este documento, completará los pasos descritos en [15].

Para una visión completa del optimizador de bucles tendremos que añadir, entre otras transformaciones, como se aplica el padding para reducir los conflictos en los accesos a la cache y como se añaden instrucciones de prefetch para esconder las latencias en los accesos a los datos.

Esperamos que toda esta documentación pueda servir de ayuda a aquellas personas interesadas en la utilización del compilador como herramienta de investigación.

Bibliografía

[1] http://ipf-orc.sourceforge.net. Web de ORC en SourceForge.

[2] http://open64.sourceforge.net. Web de Open64 en SourceForge.

[3] A. Bik and H. Wijshoff. Implementation of fourier-motzkin elimination. Technical Report TR94-42, Department of Computer Science, University of Leiden, 1994.

[4] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN symposium on Compiler contruction*, pages 162–175. ACM Press, 1986.

[5] Jim Dehnert. *Pro64: Performance Compilers For IA-64*, June 2000.

[6] Guang R. Gao, J.N. Amaral, J. Dehnert, and R. Towle. *Tutorial on the SGI Pro64 Compiler Infrastructure*. Disponible en http://www.cs.ualberta.ca/∼amaral/Pro64/index.html.

[7] Roy Ju, Sun Chan, Fred Chow, Xiaobing Feng, and William Chen. *Open Research Compiler (ORC): Beyond Version 1.0*, September 2002. Tutorial PACT-2002.

[8] Roy Ju, Sun Chan, Chengyong Wu, Ruiqi Lian, and Tony Tuo. *Open Research Compiler (ORC) for Itanium Processor Family*, December 2001. Tutorial MICRO34.

[9] Induprakas Kodukula. *Data-Centric Compilation*. PhD thesis, Cornell University, 1998.

[10] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–357, 1997.

[11] Induprakas Kodukula, Keshav Pingali, Robert Cox, and Dror E. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *International Conference on Supercomputing*, pages 482–491, 1999.

[12] Dror Eliezer Maydan. *Accurate Analysis of Array References*. PhD thesis, Stanford University, 1992.

[13] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[14] Eduard Santamaria, Marta Jiménez, Agustín Fernández, and Josep M. LLaberia. Introducción al compilador open64/orc. Technical Report UPC-DAC-2003-20, DAC-UPC, 2003.

[15] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *International Symposium on Microarchitecture*, pages 274–286, 1996.