El optimizador de bucles del compilador Open64/ORC (parte 2)

Eduard Santamaria Marta Jiménez Agustín Fernández
Josep Ma. LLabería

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

5 de septiembre de 2005

Índice general

Capítulo 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.

Este documento continúa el trabajo iniciado en los technical reports "Introducción al compilador Open64/ORC" [10] y "El optimizador de bucles del compilador Open64/ORC (parte 1)" [11]. El primero describe los componentes del compilador y la representación intermedia que se utiliza como interficie común entre ellos. El segundo documento se centra específicamente en uno de los componentes del compilador: el optimizador de bucles.

1.1. Etapas del optimizador de bucles

Las principales tareas realizadas por el optimizador de bucles comprenden (ver figura 1.1):

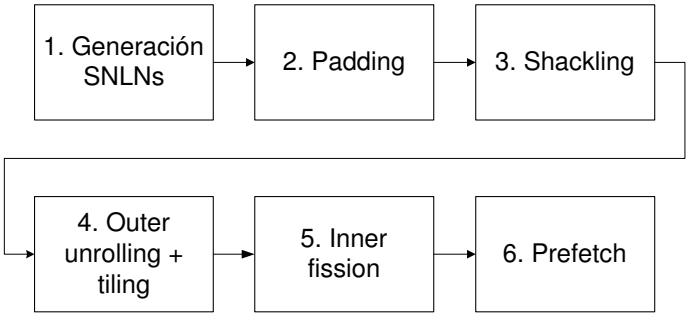
1. Construcción de anidaciones de bucles casi perfectas, denominadas SNLNs, utilizando las transformaciones fusión y distribución (Etapa 1 - Generación SNLNs).

2. Aumento del 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 para mejorar los accesos a la cache de segundo nivel (Etapa 3 - Shackling).

4. Aplicación de intercambio, outer loop unrolling y tiling en los SNLNs internos para mejorar la localidad de los accesos a registros, y a los distintos

Figura 1.1: Etapas del optimizador de bucles



niveles de cache (Etapa 4 - Outer unrolling + tiling).

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

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

Los *Singly Nested Loop Nests* (SNLNs[1]) son anidaciones de bucles casi perfectas. Esto es, anidaciones donde sólo vamos a encontrar un bucle en cada nivel de profundidad, pero con la posibilidad de que aparezcan sentencias intercaladas entre los bucles. Además deben cumplir condiciones adicionales como no contener gotos, returns o llamadas a función.

La primera tarea llevada a cabo por el optimizador de bucles consiste en la generación de SNLs de profundidad máxima. Estos SNLs son tratados durante la etapa de shackling para intentar mejorar la localidad de los accesos a nivel de L2, pero el principal responsable de mejorar la localidad a todos los niveles de la jerarquía de memoria es la etapa 4 mediante las transformaciones de intercambio, outer loop unrolling y tiling. Para obtener la mejor combinación de estas transformaciones se utiliza un algoritmo de búsqueda que enumera las distintas posibilidades. Un modelo del procesador junto con un modelo de la cache realizarán una estimación del coste de ejecución del código transformado que servirá para intentar seleccionar la mejor opción.

Las etapas 1 (generación de SNLNs) y 3 (shackling), junto con la construcción de los grafos de dependencia de datos, son tratados en [11]. Este documento continúa la descripción del optimizador de bucles mostrando los análisis y transformaciones que se realizan durante la etapa 4 (outer unrolling + tiling).

En [13] se presentan los principales elementos que intervienen en esta etapa: modelo del procesador, modelo de la cache y algoritmo. Este documento profundiza en los detalles de implementación y añade información acerca de comprobaciones preliminares. Todo ello se organiza siguiendo el orden de ejecución, de modo que se pueda establecer una relación más directa entre los contenidos de este documento y su implementación en el compilador.

[1] También denominados SNLs para mayor brevedad.

1.2. Estructura del documento

Antes de decidir qué combinación de intercambio, outer unroll y tiling se aplicará a un SNL se analiza la anidación para determinar qué bucles se pueden transformar. Estas comprobaciones, que se describen en el capítulo 2, servirán para obtener los siguientes parámetros:

- el subconjunto de bucles del SNL totalmente permutables,
- el número de bucles donde intentaremos intercambio y
- el número de bucles a los que intentaremos aplicar unroll.

El primero de estos puntos indica a qué bucles podremos aplicar tiling de forma legal. El criterio de legalidad no es el único que se tiene en cuenta para decidir qué transformaciones se intentará aplicar. Otras características del código pueden llevarnos a limitar el número de bucles que se podrán intercambiar o el número de bucles a los que se aplicará unroll.

Como se ha comentado, el compilador basa su decisión sobre la combinación de transformaciones que se aplicará en un algoritmo de búsqueda que enumera y evalúa las distintas posibilidades. En el capítulo 3 se presentan el algoritmo de búsqueda y el modelo del procesador. Este modelo realiza una estimación del coste de ejecución del código transformado sin tener en cuenta los fallos en cache y se utilizará para determinar los factores de outer unroll para cada posible bucle interno.

El capítulo 4 describe el modelo de la cache. Dados un bucle interno y los factores de outer unrolling, este modelo será el encargado de decidir como se hará el tiling para los distintos niveles de cache. Los resultados de este modelo se sumarán a los del modelo del procesador y servirán para comparar entre las distintas posibilidades.

En el capítulo 5 veremos con algunos ejemplos como se modifica el código una vez decidido qué transformaciones aplicar.

Por último el capítulo 6 presenta algunas conclusiones y direcciones de trabajo futuro.

Capítulo 2

Selección de los bucles

La optimización de bucles se describe en [13] como un procedimiento que consta de tres pasos:

1. Generación de SNLNs (etapa 1 del optimizador de bucles).
2. Selección de la combinación de intercambio, outer unrolling y tiling, y transformación del código intermedio (etapa 4).
3. Distribución de los bucles internos para reducir la presión sobre los registros (etapa 5).

La función Phase_123(), definida en lnopt_main.cxx, controla la ejecución de estas etapas. Si bien durante la ejecución de Phase_123() también se llevarán a cabo el padding y el shackling, estas dos transformaciones no se tienen en cuenta en esta numeración.

La etapa 4, que es la que nos ocupa, se inicia con una llamada a SNL_Phase(). SNL_Phase() implementa un recorrido de los SNLs generados en la etapa 1 y, para cada uno de los SNLs internos, llama a SNL_Transform(), que será la encargada de optimizar la anidación.

Dado un SNL interno, el primer paso consistirá en determinar el subconjunto de bucles del SNL que vamos a intentar transformar. Existen distintos motivos para descartar bucles o limitar el tipo de transformación, entre ellos la presencia de directivas de compilación, el análisis de legalidad y la tipología de los bucles. Con este propósito se realizan las comprobaciones que se describen en este capítulo.

La sección 2.1 describe el tratamiento de la directiva de compilación $BLOCKABLE, que permite al usuario corregir situaciones donde el compilador asume dependecia de forma conservadora.

La clase SNL_NEST_INFO (be/lno/snl_nest.h) encapsula distintos análisis necesarios para determinar qué bucles podremos transformar y qué tipo de transformación se aplicará. Estos análisis, que se presentan en la sección 2.2, comprenden el estudio de las dependencias entre escalares, la clasificación de los bucles en invariantes y generales, el análisis del código imperfecto y la recopilación de información acerca de los límites de los bucles.

La sección 2.3 describe cómo se analizan las dependencias entre accesos a arrays. Como en la sección 2.2, también existe una clase que encapsula estos análisis, que en este caso se denomina SNL_ANAL_INFO (be/lno/snl_deps.h).

La información recogida en los análisis anteriores se utiliza, tal como se explica en la sección 2.5, para inicializar los siguientes parámetros:

- el subconjunto de bucles del SNL totalmente permutables,
- el número de bucles donde intentaremos intercambio y
- el número de bucles a los que intentaremos aplicar unroll.

La sección 2.5 incluye también la descripción del tratamiento de expresiones invariantes. La presencia de expresiones invariantes se debe tener en cuenta en la estimación del coste de ejecución. Por este motivo se construye una lista de expresiones invariantes que constituye un parámetro adicional que se pasa al modelo que va a decidir qué transformaciones aplicar.

2.1. Tratamiento de la directiva $BLOCKABLE

Para que la aplicación del tiling sea legal, es necesario que todos los bucles implicados sean totalmente permutables. Esto significa que cualquier permutación posible de dichos bucles debe respetar el sentido de las dependencias. Para efectuar esta comprobación el compilador utiliza la información del grafo de dependencias entre arrays. En [11] se detalla como se calculan las dependencias. En ocasiones puede ocurrir que, de forma conservadora, el compilador asuma dependencias que impidan la aplicación del tiling. El usuario puede enmendar esta situación haciendo uso de la directiva $BLOCKABLE.

La función Fix_Blockable_Dependences() (be/lno/snl_test.cxx) es la encargada de ver qué bucles están afectados por dicha directiva y de modificar el grafo de dependencias en consecuencia. El algoritmo 1 muestra como se lleva a cabo esta tarea.

Algoritmo 1: Corrección de las dependencias
Entrada: El conjunto L de bucles afectados por la directiva.
El grafo de dependencias entre arrays *depg*.
Salida: El grafo de dependencias *depg* corregido.

```
(1)  foreach nodo whirl nw contenido en L
(2)      if nw tiene asociado un vértice v en depg
(3)          foreach arista de salida a de v
(4)              dva = lista de vectores de dependencia de a
(5)              if dva es bloqueable
(6)                  Ir a siguiente iteración
(7)              else
(8)                  Corregir dva
```

Para que una lista de vectores de dependencia sea bloqueable es necesario que cada uno de sus vectores de dependencia lo sea. Sean J el bucle más externo afectado por la directiva y K el bucle más interno afectado por la directiva, diremos que un vector de dependencias $(d_1, \ldots, d_j, \ldots, d_k, \ldots, d_n)$ es bloqueable si $\exists p \,|\, p < j$ y $signo(d_p) > 0$, o bien, $\forall p \,|\, j \leq p \leq k$ se cumple $signo(d_p) \geq 0$.

Corregir la lista de vectores de dependencia consiste en tratar cada uno de sus vectores de dependencia por separado y modificar cada dimensión afectada según la siguiente tabla:

original	corregido
DIR_POSNEG (<>)	DIR_POS (<)
DIR_NEGEQ (>=)	DIR_EQ (=)
DIR_STAR (*)	un vector con DIR_POS y otro con DIR_EQ
otros	no se modifican

Al final de este proceso se habrán eliminado aquellas dependencias de signo negativo que aparecen por las limitaciones en el cálculo de dependencias. Es decir, si tenemos un vector $(<, *)$ donde, de forma conservadora, se ha asumido dependencia en la segunda dimensión, sustituiremos este vector por los vectores $(<, <)$ y $(<, =)$ ya que el usuario informa que los bucles sí son bloqueables.

2.2. Información acerca de los bucles

Esta sección presenta un conjunto de comprobaciones que, junto con el análisis de dependencias entre arrays descrito en la sección 2.3, permite al compilador decidir qué bucles se pueden transformar. Estas comprobaciones, que son llevadas a cabo por una instancia de la clase SNL_NEST_INFO, consisten en:

- Estudiar las dependencias entre escalares. Se genera una lista indicando qué escalares se pueden privatizar.
- Clasificar los bucles en invariantes y generales. Esta clasificación se tiene en cuenta para decidir qué bucles se podrán intercambiar y desenrollar, y también afecta al proceso de transformación de la representación intermedia.
- Analizar el código imperfecto para determinar si podrá ser trasladado fuera de la anidación mediante distribución. El resultado de este análisis puede modificar la clasificación anterior.
- Recopilación de información acerca de los límites de los bucles. Esta información será utilizada más adelante al transformar la representación intermedia. También es necesaria porque hay casos en que se debe garantizar la ejecución de al menos una iteración de un bucle.

2.2.1. Expansión de escalares

Las dependencias entre escalares debidas al reuso de valores se deben preservar, pero cuando estas dependencias aparecen debido a la reutilización de posiciones de memoria, podemos eliminarlas creando variables adicionales. De este modo, cada iteración del bucle que acarrea la dependencia accederá a su propia variable escalar. En el contexto de la paralelización de bucles esta transformación se denomina privatización. Éste no es el objetivo de los análisis que tratamos en este capítulo. En nuestro caso, para romper la dependencia y proveer a cada iteración con su propia posición de memoria se aplicará expansión de escalares, transformación que consiste en sustituir la variable escalar por un vector.

Dados dos bucles L1 y L2 pertenecientes al SNL, el compilador construye una lista de escalares con información indicando si se pueden privatizar. Se

Figura 2.1: Casos en que no es necesaria la expansión de escalares

```
do I
  do J
    t = t + A(I,J)
  enddo
enddo
```

(A)

```
do I
  do J
    t =
      = t
  enddo
enddo
```

(B)

```
do I
  t =
    = t
  do J
    (no hay defs.
     ni usos de t)
  enddo
enddo
```

(C)

asume que las transformaciones que se vayan a realizar sólo afectarán a bucles pertenecientes al subconjunto delimitado por L1 y L2.

Para cada escalar se determinarán, entre otros atributos:

- Los bucles que podremos transformar sin necesidad de aplicar expansión de escalares (Outer_Se_Not_Reqd).
- Los bucles que podremos transformar si aplicamos expansión de escalares (Outer_Se_Reqd).

Si una variable no aparece en la lista, puede ser ignorada a efectos de la legalidad de la transformación.

El compilador considera que no es necesaria la expansión de escalares en los siguientes casos:

- Cuando no hay definiciones, o bien, todas las definiciones forman parte de un mismo tipo de reducción (figura 2.1 (A)).
- Todas las definiciones de un escalar privatizable están contenidas en el bucle más interno (figura 2.1 (B)).
- Dado un escalar privatizable, todas sus definiciones y usos se encuentran por encima o todas por debajo del siguiente bucle de la anidación (figura 2.1 (C)).

Será necesario aplicar expansión de escalares cuando exista una definición privatizable en los bucles externos. Una definición es privatizable cuando es la primera referencia al escalar y cumple, además, que no hay más definiciones del escalar en los bucles internos, o bien, todos los usos y definiciones forman parte de un mismo tipo de reducción. Por ejemplo:

```
do I
  t =
  do J
    t = t + ...
  enddo
enddo
```

En el siguiente ejemplo, aunque podríamos aplicar la transformación que se muestra, el compilador considera que no es transformable (es un caso particular no tratado):

```
do I                    do I
  t = 1                   t[i] = 1
  do J                  enddo
    t = -t              do I
  enddo                   do J
enddo                       t[i] = -t[i]
                          enddo
                        enddo
```

2.2.2. Clasificación de los bucles en invariantes y generales

La información recogida sobre los escalares se utiliza para descartar los bucles externos que no se pueden transformar debido a dependencias. Seguidamente, se intenta clasificar los bucles restantes en:

Invariantes: Cuando los límites no dependen de bucles externos y, además, todo el código imperfecto se puede distribuir.

Generales: Cuando los límites dependen de algún bucle externo o existe código imperfecto que no se puede distribuir. En éste último caso es necesario garantizar que al menos se puede distribuir en uno de los dos lados y que, como mínimo, se va a ejecutar una iteración del bucle interno.

El número de bucles de cada tipo se guarda en los campos *_nloops_general* y *_nloops_invariant*. La clasificación de los bucles en generales e invariantes se realiza en tres pasos:

1. Se inicializan los valores de *_nloops_general* y *_nloops_invariant* teniendo en cuenta, solamente, los límites de los bucles.

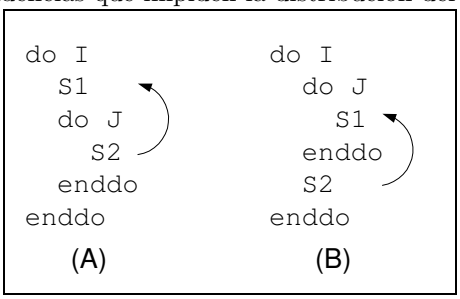
2. Las condiciones adicionales relativas al código imperfecto se analizan utilizando el algoritmo 2 (pág. 11), donde, para cada nivel de profundidad (empezando por los bucles más internos) inicializamos las variables *above_ok* y *_below_ok*, que indican si el código imperfecto se puede distribuir. Hecho esto se actualizan los valores de *_nloops_general* y *_nloops_invariant*. En cuanto una de las dos variables (*above_ok* o *_below_ok*) devuelva falso, sólo podremos encontrar bucles generales. Cuando ambas dan falso a la vez, se descartan el resto de bucles y dejamos de iterar. En la sección 2.2.3 se presenta el test que permite determinar cuando el código imperfecto es distribuible.

3. El último paso consiste en garantizar que en los bucles generales se ejecuta al menos una iteración de los bucles internos (ver sección 2.2.4).

Dado el siguiente código de la descomposición LU:

```
do i1 = 1, N
  do i2 = i1 + 1, N
    A(i2, i1) = A(i2, i1) / A(i1, i1)
    do i3 = (i1 + 1), N
      A(i2, i3) = A(i2, i3) - A(i2, i1) * A(i1, i3)
    enddo
  enddo
enddo
```

Figura 2.2: Dependencias que impiden la distribución del código imperfecto



el resultado de la clasificación es *_nloops_general* = 3 y *_nloops_invariant* = 2. Si no tenemos en cuenta el bucle $i1$ y nos limitamos a aplicar transformaciones a los bucles $i2$ e $i3$, estos bucles son invariantes. No dependen de ningún bucle externo que también se vaya a transformar y el código imperfecto se puede distribuir, ya que las dependencias acarreadas por $i1$ se ignoran. Si incluimos $i1$, entonces se trata de tres bucles generales, ya que los límites de $i2$ e $i3$ dependen de $i1$ y el código imperfecto no se puede distribuir.

2.2.3. Distribución del código imperfecto

Para determinar si el código que aparece intercalado entre bucles se puede distribuir se buscan dependencias "hacia atrás" entre accesos a array. En el caso de código imperfecto situado sobre el bucle interno, esto significa buscar dependencias con destino en el código imperfecto y origen dentro o debajo del bucle interno (figura 2.2 (A)). Si el código imperfecto está por debajo del bucle interno buscamos dependencias con origen en el código imperfecto y destino dentro o encima del bucle interno (figura 2.2 (B)). Las dependencias acarreadas por bucles externos que no se vayan a transformar se ignoran.

Previamente se ha producido una reordenación de las sentencias que garantiza que la presencia de una dependencia hacia atrás implica la existencia de un ciclo [11]. La existencia de alguna dependencia como las descritas indica que el código imperfecto no se podrá distribuir. Este análisis se lleva a cabo utilizando la información del grafo de dependencias entre arrays.

Como muestra el algoritmo 2, al mismo tiempo que se lleva a cabo la clasificación entre bucles invariantes y generales, se calculan los valores de los atributos _above_is_distributable y _below_is_distributable.

La figura 2.3 muestra el grafo de dependencias correspondiente al código de descomposición LU. Este caso correponde a la figura 2.2 (A) y podemos observar, como apuntábamos en el apartado anterior, que el código imperfecto sólo se puede distribuir cuando el bucle $i1$ no forma parte del conjunto de bucles que se desea transformar.

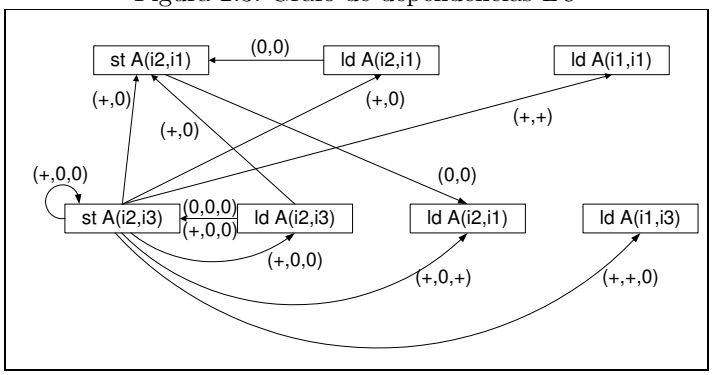
2.2.4. Límites de los bucles

Para terminar con la inicialización de SNL_NEST_INFO se recoge la información acerca de los límites de los bucles. Si no es necesario asegurar como mínimo una iteración, simplemente inicializamos el campo correspondiente creando un

Algoritmo 2: Clasificación de bucles en generales e invariantes
Entrada: El conjunto L de bucles de la anidación, *_nloops_general* y *_nloops_invariant*
Salida: *_above_is_distributable*, *_below_is_distributable*, *_nloops_general* y *_nloops_invariant*

```
(1)   _above_is_distributable = TRUE
(2)   _below_is_distributable = TRUE
(3)   nloops = Número de bucles en L
(4)   depth_inner = Profundidad del bucle más interno
(5)   for i = 2 to nloops
(6)       d = depth_inner + 1 - i
(7)       above_ok = _above_is_distributable
(8)       dd = d + 1
(9)       while above_ok and dd <= depth_inner
(10)          wn1 = Bucle a profundidad dd-1
(11)          wn = Bucle a profundidad dd
(12)          above_ok = Código entre wn1 y wn, sobre wn, es
              distribuible.
(13)          dd = dd + 1
(14)      below_ok = _below_is_distributable
(15)      dd = d + 1
(16)      while below_ok and dd <= depth_inner
(17)          wn1 = Bucle a profundidad dd-1
(18)          wn = Bucle a profundidad dd
(19)          below_ok = Código entre wn1 y wn, bajo wn, es
              distribuible.
(20)          dd = dd + 1
(21)      if not above_ok or not below_ok
(22)          if _nloops_invariant >= i
(23)              _nloops_invariant = i - 1
(24)          if i <= _nloops_general
(25)              if not above_ok and not below_ok
(26)                  _nloops_general = i - 1
(27)                  break
(28)              else if above_ok = FALSE
(29)                  _above_is_distributable = FALSE
(30)              else
(31)                  _below_is_distributable = FALSE
```

Figura 2.3: Grafo de dependencias LU



sistema de ecuaciones con todas las desigualdades derivadas de los límites de los bucles. Esta información será utilizada más adelante durante la transformación del código.

Cuando hay código imperfecto que no podemos distribuir, entonces es necesario asegurar que al menos se ejecutará una iteración de cada bucle. Creamos un sistema de ecuaciones, inicialmente vacío, y tratamos cada uno de los bucles de la anidación empezando por el más externo. A medida que se tratan los distintos bucles se van añadiendo al sistema las restricciones derivadas de sus límites. Para cada uno de los bucles:

1. Se generan dos expresiones (access vectors[11]) relacionando los límites del bucle:

   - $nox \equiv LB > UB$
   - $yesx \equiv LB \leq UB$

   donde LB y UB hacen referencia al límite inferior y al límite superior del bucle respectivamente.

2. Añadimos nox al sistema de ecuaciones y evaluamos su consistencia. La ecuación no va a permanecer en el sistema, una vez evaluada la consistencia se elimina.

   - Si el sistema es inconsistente, se garantiza que se ejecutará al menos una iteración. Añadimos las ecuaciones derivadas de los límites del bucle y pasamos a tratar el siguiente bucle de la anidación.
   - Si el sistema es consistente, cabe la posibilidad que no se ejecute ningua iteración del bucle. En ese caso hay dos posibilidades: la primera, que los límites del bucle que estamos tratando sean invariantes respecto a los bucles externos, sin tener en cuenta los bucles que no se desea transformar. Esta situación se intentará solventar durante la transformación del código generando dos copias de la anidación y protegiéndolas con condicionales de modo que en una de las copias se garantice al menos una iteración de cada bucle.
     La otra posibilidad es que los límites sí dependan de alguno de los

bucles externos. En este caso se analiza si haciendo peeling de la primera o última iteración del bucle externo podremos asegurar al menos una iteración. Esta transformación se aplicará sólo si el bucle del que extraemos la iteración es el más externo (del subconjunto que se intenta transformar) y el bucle en curso lo sigue immediatamente. Si no se puede hacer peeling o su aplicación no garantiza la ejecución de una o más iteraciones del bucle en curso, se decrementa el número de bucles generales y volvemos a iniciar la comprobación de todos los bucles.
Si los límites no dependen de los bucles externos o el peeling tiene éxito añadimos la ecuación $yesx$ y los límites del bucle al sistema y continuamos.

2.3. Análisis de dependencias

Después de las comprobaciones descritas en la sección 2.2, el compilador pasa el control a la función Do_Automatic_Transformation() (be/lno/snl_test.cxx), que dirige el resto de la etapa de tiling (etapa 4), empezando por el análisis de dependencias entre arrays y terminando con la transformación de la representación intermedia.

En el punto anterior se analizaban las dependencias entre escalares para ver qué bucles podíamos transformar y en qué casos se debía aplicar expansión de escalares. En este apartado veremos como se analizan las dependencias entre arrays para garantizar la legalidad de la transformación.

De modo similar a como se hace con las dependencias entre escalares, el compilador crea también una instancia de una clase, denominada SNL_ANAL_INFO, que recoge la información acerca de las dependecias entre arrays. Esta estructura de datos almacena de forma resumida las dependencias del grafo de dependencias entre arrays. Contiene los siguientes datos:

- Lista de dependencias lexicográficamente positivas entre sentencias del bucle interno, sin duplicados.
- Lista de dependencias lexicográficamente positivas del código imperfecto, también sin duplicados. Estas dependencias no se obtienen directamente del grafo, sino que son recalculadas suponiendo que el código imperfecto forma parte del bucle más interno.
- Tabla de hash que relaciona cada nodo whirl del SNL [11] (que tenga un vértice asociado en el grafo de dependencias) con una tupla <lex, depth>. El primer elemento de la tupla es un contador, y el segundo indica en qué nivel de profundidad de la anidación de bucles se encuentra el nodo whirl [10]. Los nodos whirl se añaden de forma ordenada, de modo que el contador lex, que se incrementa a cada inserción, nos va a permitir determinar la posición relativa de un acceso a memoria respecto al resto de las referencias y también respecto al bucle interno. Esta información se utiliza al recalcular las dependencias del código imperfecto.
- Valor de los contadores asociados al último nodo situado sobre el bucle interno y al primero situado por debajo ($\_lex\_last\_above\_innermost$ y $\_lex\_first\_below\_innermost$).

El siguiente paso consiste en trasladar esta información a formato matricial, más conveniente para realizar las comprobaciones de legalidad. La estructura de datos que se utiliza para este propósito es SNL_DEP_MATRIX (be/lno/snl_deps.h). Las dependencias entre sentencias del bucle interno y las dependencias del código imperfecto se guardan en matrices separadas, *smat* y *smati* respectivamente. Las filas representan vectores de dependencia y las columnas representan bucles. Cada elemento de la matriz contiene la distancia y el signo de la dependencia para el vector de dependencia y el bucle correspondientes.

Las matrices correspondientes al grafo de la figura 2.3 son:

$$\text{bucle interno: } \begin{bmatrix} 1+ & 1+ & 0 \\ 1+ & 0 & 1+ \end{bmatrix} \quad \text{código imperfecto: } \begin{bmatrix} 1+ & 0 & 1+ \\ 1+ & 1+ & 1+ \\ 0 & 0 & 1+ \end{bmatrix}$$

2.3.1. Legalidad de las transformaciones

Una transformación de una anidación de bucles es legal si respeta el sentido de las dependencias del código original. Puesto que en un programa legal todas las dependencias tienen sentido positivo[1], esto se traduce en que, antes de aplicar cualquier transformación, se deberá comprobar que tras su aplicación los vectores de dependencia continuarán siendo lexicográficamente positivos.

La condición que se debe cumplir para garantizar la legalidad del bloqueo y de cualquier intercambio arbitrario de bucles es que los bucles de la anidación sean totalmente permutables [12]. Una anidación es totalmente permutable si toda permutación posible de los bucles es una transformación legal.

Sean J y K el bucle más interno y más externo, respectivamente, del subconjunto de bucles de una anidación que se desea transformar. Y sea D el conjunto de vectores de dependencia que expresa las dependencias entre sentencias del subconjunto de bucles que queremos transformar. Diremos que los bucles delimitados por J y K son totalmente permutables si para todo $\vec{d} \in D$ se cumple:

$\exists p \,|\, p < j \,\text{y}\, signo(d_p) > 0$, o bien,

$\forall p \,|\, j \leq p \leq k$ se cumple $signo(d_p) \geq 0$.

2.3.2. Selección de los bucles

Dado el conjunto de bucles candidatos a ser transformados, analizaremos el contenido de *smat* y *smati* para determinar si son totalmente permutables. Empezando por el bucle más externo se irán descartando bucles hasta encontrar un subconjunto de bucles totalmente permutables.

Si el código imperfecto se puede distribuir, las dependencias de *smati* se ignoran. En caso contrario *smati* también debe cumplir las condiciones de legalidad.

[1] 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 tiene signo negativo en la primera dimensión corresponde a una antidependencia (o dependencia de flujo) con los signos del vector de dependencias invertidos.

2.4. Intercambio

Antes de pasar a describir los parámetros que recibirán los modelos encargados de decidir qué transformaciones aplicar, comentaremos algunos detalles acerca del intercambio. Si bien, una vez realizado el análisis de legalidad, cualquier permutación es válida, hay otros factores que se tienen en cuenta.

No se va a permitir intercambio si hay alguna directiva que lo prohiba explícitamente. Pero además, se analizarán los límites de los bucles procediendo del siguiente modo. Se permite intercambio cuando:

1. No se ha podido determinar el número de iteraciones de los dos bucles más internos pero se sabe que definen un espacio de iteraciones rectangular. Esta situación se puede producir debido a la aparición de parámetros no conocidos en tiempo de compilación.

2. O bien, los límites de todos los bucles de la anidación son invariantes (no dependen de variables de iteración de bucles externos).

Si no se garantiza ninguna de las condiciones anteriores estamos trantando una anidación general y, en el mejor de los casos, sólo se va a permitir intercambio de los dos bucles más internos. Dada una anidación general, considerando como último bucle el más interno, no se permite intercambio si se produce alguna de las siguientes situaciones:

1. El límite inferior o superior del penúltimo bucle contienen:

   a) las funciones MAX o MIN respectivamente,

   b) algún símbolo que no sea un índice de un bucle,

   c) o alguna expresión no lineal.

2. Se detecta la posibilidad que el bucle externo no ejecute ninguna iteración.

Para terminar se realizan las siguientes comprobaciones sobre el último bucle:

1. De igual modo que para el penúltimo bucle, sus límites no pueden contener símbolos que no sean índice de bucle ni expresiones no lineales.

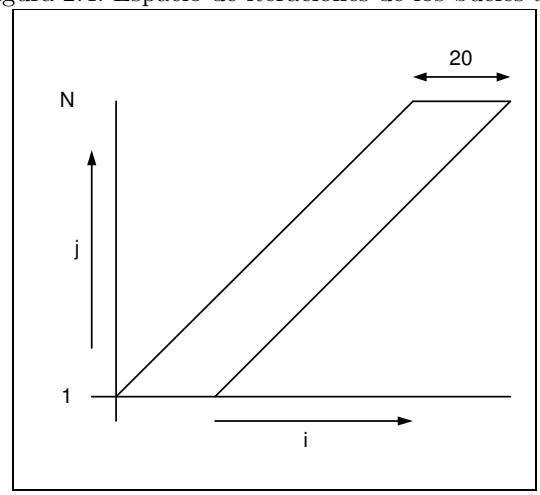
2. Sea i un bucle externo con muchas iteraciones y j un bucle interno con pocas iteraciones, el intercambio no garantiza que vayamos a tener pocas iteraciones en el bucle externo y muchas en el bucle interno. Si el compilador detecta que, después del intercambio, el número de iteraciones del bucle interno original se va a incrementar significativamente éste se desestima.

   El siguiente código presenta un ejemplo donde, para un valor elevado de N, no se va a permitir intercambiar. La figura 2.4 muestra el espacio de iteraciones:

```
do i = 1, N
  do j = i, i+20
    ...
```

Todas las comprobaciones que acabamos de detallar son realizadas por la función Est_Num_Iters_Suspect() (be/lno/snl_test.cxx).

Figura 2.4: Espacio de iteraciones de los bucles i, j



2.5. Parámetros del modelo

Para terminar, se inicializan los parámetros que se van a pasar al modelo que decidirá las transformaciones a aplicar.

Mediante los análisis de dependencias entre escalares y entre arrays descritos en los apartados anteriores se ha obtenido el subconjunto de bucles del SNL que pueden ser transformados. Este dato se pasa al modelo mediante la variable *outermost_can_be_tiled*, que indica la profundidad del bucle más externo que podemos bloquear.

El resto de parámetros que se van a pasar son:

wn: Apuntador al nodo whirl del bucle más interno del SNL.

can_be_inner[]: Vector que, para cada bucle transformable, indica si se puede considerar el bucle como candidato a bucle interno.

can_be_unrolled[]: Vector que, para cada bucle transformable, indica si se va a considerar su desenrosque.

array_graph: El grafo de dependencias entre arrays.

pi: Estructura de datos que indica qué escalares se pueden privatizar.

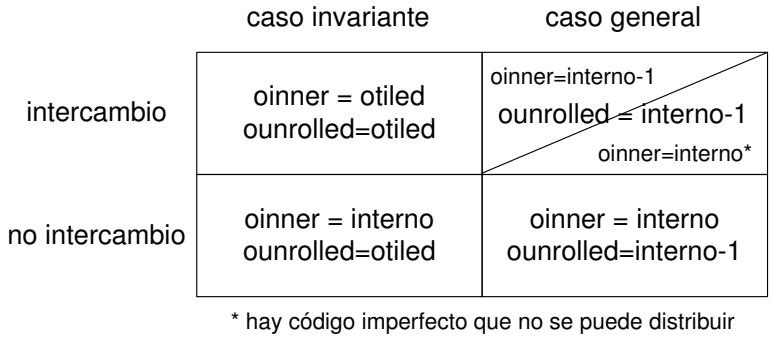
SNL_Depth: Número de bucles que podrán ser transformados.

invar_table: Tabla de hash que relaciona nodos whirl con un mapa de bits que indica respecto a qué bucles la expresión representada por el nodo whirl es invariante (ver sección 2.5.1).

La forma como se optimizan los bucles difiere en función del tipo de bucles que conforman la anidación. Tomando en consideración solamente aquellos bucles que se intentará transformar, se distingue entre los siguientes casos:

invariante: Todos los bucles de la anidación son invariantes.

Figura 2.5: Parámetros del modelo



general: La anidación contiene bucles generales.

Una vez conocidos los bucles totalmente permutables, el tipo de dichos bucles y si se va a permitir o no la aplicación de intercambio, se determina qué bucles serán candidatos a bucle interno y qué bucles se podrán desenrollar tal como se muestra en la figura 2.5, donde:

otiled: (*outermost_can_be_tiled*) indica a partir de qué bucle es posible aplicar tiling.

oinner: (*outermost_can_be_inner*) indica cual es el bucle más externo que podrá ser candidato a bucle interno.

ounrolled: (*outermost_can_be_unrolled*) indica cual será el bucle más externo que podremos desenrollar por aplicación de outer unrolling.

interno = profundidad del bucle más interno de la anidación.

Entre paréntesis aparece el nombre completo de la variable utilizada en el compilador. El caso menos restrictivo se produce cuando todos los bucles son invariantes y se permite intercambio. Entonces cualquiera de los bucles totalmente permutables podrá ser candidato a bucle interno y cualquiera (excepto el interno) podrá ser desenrollado. En el caso general, en cambio, como máximo se permitirá intercambiar los dos bucles más internos y sólo podremos aplicar outer unrolling al penúltimo bucle. Los vectores *can_be_inner*[] y *can_be_unrolled*[] se inicializan a partir del resultado de *outermost_can_be_inner* y *outermost_can_be_unrolled*.

2.5.1. Tratamiento de expresiones invariantes

Las expresiones invariantes no intervienen en la selección de los bucles a transformar, pero sí son uno de los parámetros que recibe el modelo. Al realizar las estimaciones del coste se debe tener en cuenta que estas expresiones se trasladarán fuera del bucle interno, cosa que incidirá en el número de operaciones, accesos a memoria y consumo de registros. El tratamiento de las expresiones invariantes se lleva a cabo en tres pasos:

1. Marcaje de las expresiones.
2. Reordenación de las expresiones modificando la representación WHIRL.
3. Traslado de las expresiones fuera de los bucles respecto a los cuales son invariantes (hoisting).

El hoisting de las expresiones invariantes se realiza después de haber aplicado el tiling. A continuación veremos como se detectan y reordenan las expresiones invariantes. Después esta información se pasa al modelo para que la tenga en cuenta en el cálculo de los costes de ejecución del código.

Marcaje de los invariantes

La información sobre los invariantes se mantiene en una tabla de hash donde, a cada nodo que representa una expresión o subexpresión, se le asocia un vector de bits. Cada bit representa uno de los bucles que contienen la expresión. Un valor igual a 1 indica que la expresión es invariante respecto a ese bucle.

La función Mark_Expression() (be/lno/oinvar.cxx) lleva a cabo el marcaje de las expresiones invariantes mediante un recorrido recursivo del árbol de la representación intermedia:

- El caso base del recorrido se alcanza cuando encontramos un load o un valor constante. En ese momento se crea e inicializa un mapa de bits. Si se trata de una constante, todas las entradas se pondrán a 1.
- Los vectores de bits de los nodos padre, se obtienen intersectando los mapas de bits de cada uno de los hijos. Si el resultado es un conjunto vacío, no añadimos la entrada a la tabla de hash.

El compilador intenta no añadir a la tabla de invariantes expresiones de direccionamiento que podrán ser simplificadas. Antes de realizar una inserción se llama a la función Contains_Work() (be/lno/oinvar.cxx), que devolverá cierto en los siguientos casos:

- El operador del nodo es distinto de load/paréntesi/parámetro y devuelve un elemento de tipo float.
- El operador es distinto de suma/resta/negación sobre un load indirecto que varía en cada iteración del bucle interno.
- Hay un load indirecto que varía en cada iteración del bucle interno en algún hijo y cualquier tipo de load en otro hijo.

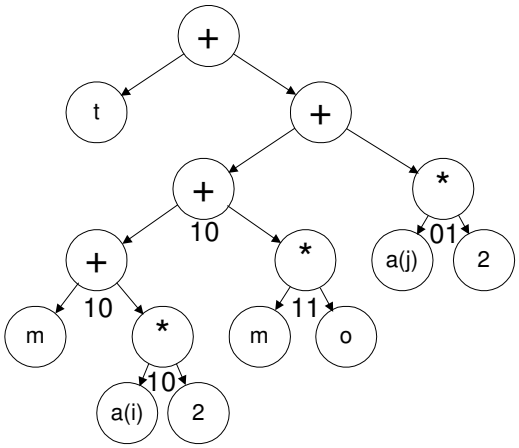
Si Contains_Work() devuelve falso, la entrada no se añade a la tabla, pero el valor del mapa de bits sí se propaga hacía arriba (en el recorrido recursivo).

La figura 2.6 muestra como se marcan los invariantes de un código de ejemplo. Los vectores de bits de las hojas, no se asocian a sus correspondientes nodos porque en todos los casos Contains_Work() devuelve falso, pero sí se tienen en cuenta para calcular los vectores de bits de los padres.

Observamos que la ordenación de los nodos puede afectar al resultado. Si, por ejemplo, intercambiáramos el término m con la t, la intersección con el nodo hermano daría nulo y este resultado se propagaría hacia arriba.

Figura 2.6: Marcaje de los invariantes

```
do I
  do J
    t = 2*a(i)+m+m*o+
        2*a(j)+t
  enddo
enddo
```



El mapa de bits asociado a los nodos indica respecto a qué bucles es invariante una subexpresión.

Ordenación de los invariantes

La ordenación de los invariantes intenta reorganizar las expresiones de modo que los términos con misma invariancia se ubiquen en posiciones adyacentes. Esta ordenación sólo es posible si se permite aplicar asociatividad.

Las funciones Sort_Invar_Expressions() y Sort_Invar_Expressions_Rec() recorren la representación intermedia en busca de expresiones. Para cada expresión se llama a Sort_Invar_Expression(), que recorre el cuerpo de la expresión y llama a Sort_Equivalence_Class() cuando detecta un operador +, *, MIN o MAX que coincide con el operador de uno de los hijos. Todas estas funciones se definen en el archivo be/lno/oinvar.cxx. La reordenación de los nodos sigue los siguientes pasos (ver figura 2.7):

1. Se modifica el árbol de la representación de modo que todos los nodos que no son loads aparezcan como hijo izquierdo.

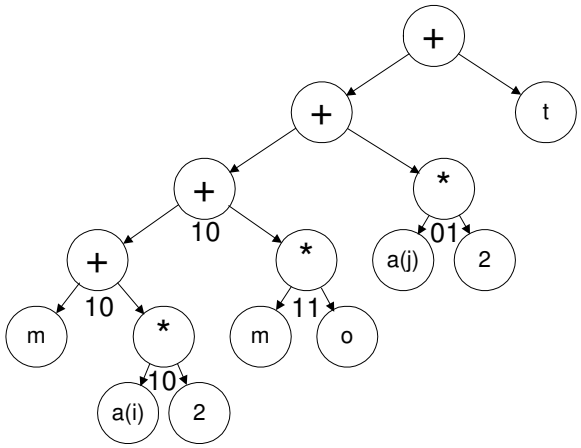
2. Se genera una lista con las subexpresiones en orden de aparición.

3. Se crea una nueva lista con las subexpresiones de la lista anterior clasificadas en grupos. Se añade una subexpresión a un grupo si su intersección con el mapa de bits correspondiente a ese grupo no es vacía. Si la subexpresión no se puede añadir a ningún grupo de los existentes se crea uno nuevo.

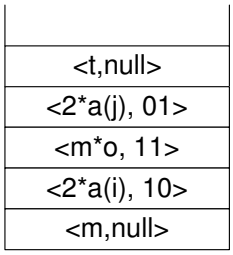
4. Se ordena la lista de grupos por número de elementos.

5. Se reorganiza el árbol de la representación de modo que las expresiones con igual invariancia sean adyacentes.

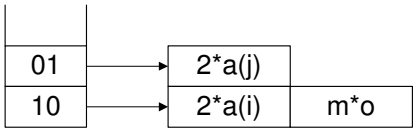
Figura 2.7: Pasos en la ordenación de los invariantes



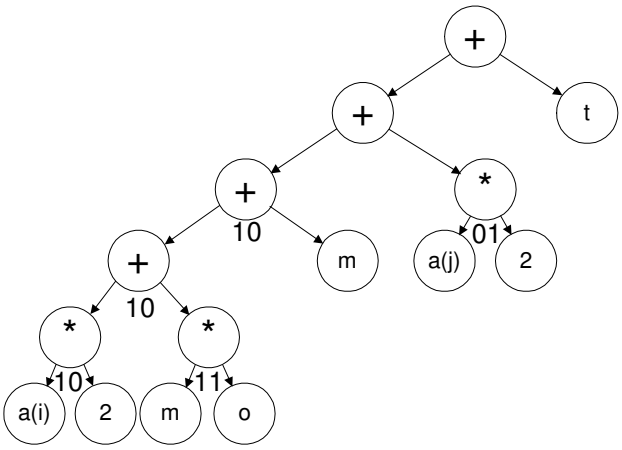
(1) Operador suma como hijo izquierdo.



(2) Lista de subexpresiones por orden de aparición.



(3,4)Clasificación en grupos y ordenación de los grupos.



(5)Modificación de la representación intermedia.

Hoisting de invariantes

En esta sección presentamos un ejemplo que muestra como se llevará a cabo el traslado de los invariantes fuera del bucle interno, pero debemos recordar que este proceso se realiza siempre después de aplicar tiling y outer unrolling.

El siguiente código contiene la expresión `A(J)*B(J)`, invariante respecto al bucle I, y la expresión `A(I)*o`, invariante respecto al bucle interno J:

```
do I = 1, N
  do J = 1, N
    t = t+A(J)*B(J)+A(I)*o
  enddo
enddo
```

Para apreciar más claramente el resultado del hoisting de invariantes hemos desactivado las optimizaciones de intercambio, outer unrolling y tiling. Tras la compilación, el resultado obtenido es el siguiente:

```
DO oinvar_tile_J__0 = 1, N, 1000
  se1__$stk__0 = OPR_ALLOCA(0)
  se1_F4 = OPR_ALLOCA(KZEXT(((MIN(N, (oinvar_tile_J__0 + 999))
                              -(oinvar_tile_J__0 +(-1))) * 4)))
  DO J = oinvar_tile_J__0, MIN(N, (oinvar_tile_J__0 + 999)), 1
    deref_se1_F__0((J - oinvar_tile_J__0) + 1) = (A(J) * B(J))
  END DO
  DO I__0 = 1, N, 1
    T___1 = (A(I__0) * 0)
    DO J__0 = oinvar_tile_J__0, MIN(N, (oinvar_tile_J__0 + 999)), 1
      T = (T___1 +(deref_se1_F__0((J__0 - oinvar_tile_J__0) + 1) + T))
    END DO
  END DO
  CALL OPR_DEALLOCA(se1__$stk__0, se1_F4)
END DO
```

Podemos observar que se ha creado un nuevo bucle J fuera de la anidación donde se inicializa el vector deref_se1_F__0 con los resultados de la expresión `A(J)*B(J)` para los distintos valores de J. Además se aplica tiling para limitar el número de elementos del nuevo vector. La figura 2.8 muestra, de forma esquemática, los pasos seguidos para transformar el código durante el hoisting de invariantes respecto a bucles externos.

Figura 2.8: Transformación de la representación intermedia durante el hoisting de invariantes respecto a los bucles externos.

```
do I
  do tile_J
    do J
      t = t+A(J)*B(J)+A(I)*o
    enddo
  enddo
enddo
```

(1) tiling

```
do tile_J
  do I
    do J
      t = t+A(J)*B(J)+A(I)*o
    enddo
  enddo
enddo
```

(2) intercambio

```
do tile_J
  do I
    do J
      temp = A(J)*B(J)
      t = t+temp+A(I)*o
    enddo
  enddo
enddo
```

(3) nueva sentencia

```
do tile_J
  do I
    do J
      se1(J-tile_J+1) = A(J)*B(J)
      t = t+se1(J-tile_J+1)+A(I)*o
    enddo
  enddo
enddo
```

(4) expansión

```
do tile_J
  do I
    do J
      se1(J-tile_J+1) = A(J)*B(J)
    enddo
  enddo
  do I
    do J
      t = t+se1(J-tile_J+1)+A(I)*o
    enddo
  enddo
enddo
```

(5) distribución

```
do tile_J
  do J
    se1(J-tile_J+1) = A(J)*B(J)
  enddo
  do I
    do J
      t = t+se1(J-tile_J+1)+A(I)*o
    enddo
  enddo
enddo
```

(6) finalización

Capítulo 3

Algoritmo y modelo del procesador

En este capítulo se presenta el algoritmo de búsqueda (sección 3.1) de la mejor combinación de transformaciones (intercambio, outer unroll y tiling). Para comparar entre las distintas posibilidades el compilador realiza una estimación del coste de ejecución del código transformado. Esta estimación se obtiene en base a dos modelos: el modelo del procesador y el modelo de la cache.

También en este capítulo describiremos el modelo del procesador (sección 3.2) que, teniendo en cuenta los recursos disponibles en la máquina y la longitud de las recurrencias, realiza una estimación del número de ciclos ignorando los fallos en cache.

El resto de secciones que conforman este capítulo (3.3 Inicializaciones, 3.4 Selección del bucle interno, 3.5 Selección de los factores de desenrosque y 3.6 Evaluación del coste) detallan distintos aspectos del modelo del procesador. En la sección 3.2 se indica el contenido de cada una de ellas.

El modelo de la cache se describe en el capítulo 4. El coste final se obtiene sumando el resultado de ambos modelos.

3.1. Algoritmo de búsqueda

El algoritmo que decide las optimizaciones a aplicar se basa en un recorrido del espacio de transformaciones. Los parámetros que definen el espacio de transformaciones son:

- El bucle interno.
- Los factores de desenrosque de los bucles externos.
- Los factores de bloqueo en la aplicación del tiling y la ordenación de los bucles externos.

Como muestra el algoritmo 3, para cada bucle interno, el compilador busca la mejor combinación de factores de desenrosque. Dado el bucle interno en curso y una vez determinados los factores de desenrosque, el siguiente paso consiste en encontrar un bloqueo que reduzca al máximo el coste de los accesos a la

jerarquía de memoria. El coste total se obtiene sumando al coste del código después del desenrosque el overhead introducido por los fallos en cache.

Algoritmo 3: Selección de la transformación
Entrada: Anidación de bucles, modelo del procesador y modelo de la cache.
Salida: Transformación a aplicar.

```
(1)   mejor_coste = infinito
(2)   mejor_trans = ninguna
(3)   foreach bucle interno posible
(4)     unroll_trans = calcular mejor outer unroll
(5)     unroll_cost = coste para este desenrosque
(6)     tile_trans = calcular mejor outer permutation y tiling dado unroll_trans
(7)     tile_cost = overhead por fallos de cache dados unroll_trans y tile_trans
(8)     if unroll_cost+tile_cost < mejor_coste
(9)       mejor_coste = unroll_cost+tile_cost
(10)      mejor_trans = unroll_trans y tile_trans
```

En la figura 3.1 podemos ver la secuencia de llamadas que se establece entre las funciones que implementan el algoritmo 3. El punto de entrada es el método LOOP_MODEL::Model() (be/lno/model.cxx), que recibe los parámetros listados en la sección 2.5. Esta función realiza una serie de inicializaciones y llama iterativamente a Try_Inner() para probar con cada uno de los bucles internos. Try_Unroll() es una función recursiva que enumera todos los posibles factores de desenrosque y evalúa cada combinación. Una vez establecido el mejor outer unroll para el bucle interno en curso, se llama a Cache_Model() (be/lno/cache_model.cxx) para determinar el tiling. Cache_Model() realiza una llamada a One_Cache_Model() para cada uno de los niveles de cache para los que se va a bloquear. Esta función enumera las distintas posibilidades de bloqueo de los bucles con más reuso y llama a Nest_Model() para que decida los factores de bloqueo y calcule el overhead de los accesos a memoria.

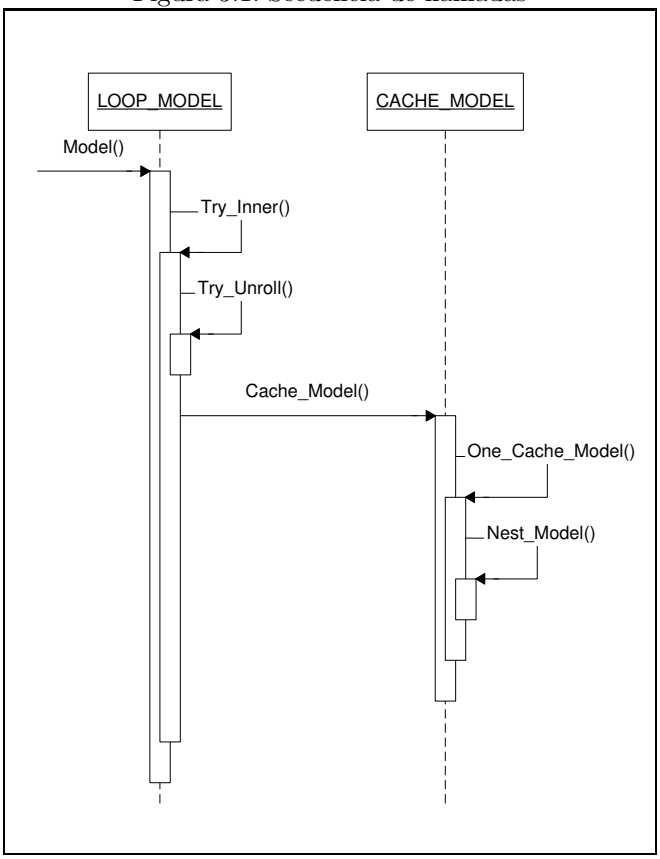
3.2. Modelo del procesador

El modelo del procesador permite estimar el coste de ejecución de una iteración del bucle interno sin tener en cuenta los fallos en los accesos a la cache. Dado un bucle interno, se utiliza este modelo para determinar los factores de desenrosque. El modelo del procesador contempla los recursos disponibles en la máquina (issue, unidades funcionales y registros) y la longitud de las recurrencias.

Al calcular los ciclos por limitaciones en el número de recursos se tienen en cuenta los siguientes parámetros:

OP_rcycles: Número de unidades funcionales y latencia de las operaciones. Se recorre el cuerpo del bucle y se obtiene el tiempo de ejecución teniendo en cuenta la ocupación de unidades funcionales de cada instrucción. No se tienen en cuenta las dependencias.

Figura 3.1: Secuencia de llamadas



$MEM\_rcycles$**:** Número de accesos a memoria. Se contabiliza un ciclo para cada acceso a memoria de coma flotante. El cálculo del número de accesos se realiza teniendo en cuenta la existencia de subexpresiones comunes y expresiones invariantes.

ALL_issue**:** Número de issues. El recorrido de las operaciones para contabilizar el número de ciclos por recursos se aprovecha para obtener el número de operaciones (OP_issue). Se añaden los accesos a memoria (MEM_issue) y el salto e incremento del bucle ($LOOP\_INIT\_issue$). $ALL\_issue = OP\_issue + MEM\_issue + LOOP\_INIT\_issue$.

Con todo, aplicamos la siguiente fórmula para obtener el número de ciclos por recursos:

$$\text{coste recursos} = MAX(OP\_rcycles, MEM\_rcycles, \frac{ALL\_issue}{ISSUE\_rate}) \quad (3.1)$$

Teniendo en cuenta que el generador de código intentará planificar las instrucciones de forma que se solape la ejecución de iteraciones consecutivas del bucle interno, el compilador calcula el número de ciclos que, como mínimo, deben transcurrir entre el inicio de una iteración y la siguiente para que las dependencias de datos sean respetadas. Este valor, que indica cada cuantos ciclos

se obtendrán resultados de una iteración del bucle interno, se toma como estimación del coste por recurrencias. Para calcular el coste por recurrencias el compilador construye un grafo de latencias que, posteriormente, se utiliza para obtener los componentes fuertemente conexos y aplicar un algoritmo de búsqueda del mínimo intervalo de iniciación.

Combinando el coste por recursos y el coste por recurrencias, el número de ciclos necesarios para ejecutar una iteración del bucle original después del desenrosque es:

$$ciclos = MAX(\text{coste recursos}, \frac{\text{coste recurrencias}}{\prod \text{factores de desenrosque}}) \qquad (3.2)$$

El modelo también realiza una estimación del consumo de registros para descartar las combinaciones de factores de desenrosque que produzcan una excesiva presión sobre el banco de registros.

En las siguientes secciones veremos como se obtiene el valor de cada una de las variables que intervienen en el cálculo del coste del modelo del procesador:

Inicializaciones (3.3): En primer lugar se calculan aquellos valores y variables auxiliares que no dependen de la elección del bucle interno:

1. OP_rcycles: Se precalcula este valor para todos los productos de factores de desenrosque.
2. _arl: Lista de referencias a memoria. Manipulando esta lista podremos simular el desenrosque sin tener que modificar el código original.
3. grafo de latencias: Será utilizado en el cálculo del coste por recurrencias.

Selección del bucle interno (3.4): Seguidamente se realiza el cálculo del coste por recurrencias, que sí depende de la elección del bucle interno.

Selección de los factores de desenrosque (3.5): Se exploran todas las combinaciones de desenrosque, evaluando el coste de cada una de ellas. Antes de realizar la evaluación (que se describe en el siguiente apartado) se desenrolla la lista _arl, se eliminan subexpresiones comunes y se marcan las expresiones invariantes.

Evaluación del coste (3.6): Por último, se calculan el resto de términos necesarios para determinar el coste según el modelo del procesador: MEM_rcycles, ALL_issue y presión sobre los registros.

Existe una relación uno a uno entre las secciones que acabamos de listar y los métodos de LOOP_MODEL que aparecen en la figura 3.1. Así pues la sección 3.3 corresponde a Model(), la sección 3.4 a Try_Inner() y la sección 3.5 a Try_Unroll(). La sección 3.6 corresponde a la función Evaluate() que no aparece en la figura 3.1 pero es llamada por Try_Unroll() para evaluar cada una de los distintas posibilidades de desenrosque.

3.3. Inicializaciones

La inicialización por defecto describe la situación en que no se aplica ninguna transformación, excepto cuando alguno de los parámetros del espacio de transformaciones ha sido especificado mediante flags o directivas de compilación.

Dentro de las incializaciones encontramos algunos datos que dependen de la arquitectura del procesador. En concreto:

_issue_rate: dos bundles con tres instrucciones cada uno (6 valor por defecto).

_num_mem_units: número de unidades de memoria (2 valor por defecto).

_base_int_regs: número de registros enteros reservados, en el caso de IA64, igual a 10 (r0, r1 (gp), r12 (sp), r13 (tp) más cinco de margen).

_base_fp_regs: número de registros de coma flotante necesarios para mantener el pipeline a máximo rendimiento (32).

_LOOP_INIT_issue: número de issues por la inicialización del bucle (2).

3.3.1. OP_rcycles

El cálculo del número de ciclos por iteración del bucle interno original, teniendo en cuenta solamente unidades funcionales y latencia de las operaciones, no depende de la elección del bucle interno. Para realizar este cálculo se traducen las instrucciones de alto nivel a instrucciones del procesador y se contabiliza, para cada tipo de recurso, el número de ciclos que estará ocupado por cada instrucción. Al mismo tiempo que realizamos esta operación, aprovechamos para contabilizar el número de issues (OP_issue).

Durante este recuento, las expresiones de indexación de arrays se ignoran. Tampoco contabilizamos consumo de recursos para las expresiones invariantes. Cuando el compilador encuentra alguna operación que no sabe como tratar (por ej. división de números complejos) devuelve error y se desestima la aplicación de outer unroll.

Para cada posible producto de factores de desenrosque se calcula el número de ciclos que se tardaría en ejecutar una iteración del bucle interno original. Por defecto, se establece que el producto de los factores de desenrosque (Max_Unroll_Prod) no va a ser superior a 16. Se inicializa cada entrada _loop_rcycles_unroll_by[i] como:

$$\frac{Min\_Cycles(\_Op\_res\_count \times (i+1) + \_Loop\_Inc\_Test\_Res)}{(i+1)},$$

donde $\_Op\_res\_count$ indica el número de ciclos consumidos para cada tipo de recurso, $\_Loop\_Inc\_Test\_Res$ representa el número de ciclos que se añaden por el incremento y el test sobre la variable de iteración, y la función Min_Cycles() toma el valor del número de ciclos consumidos en el recurso que actúa como cuello de botella. Cuando se dispone de más de una unidad de un determinado recurso, se realizan las divisiones pertinentes. Puesto que las entradas de _loop_rcycles_unroll_by se numeran desde 0 hasta $Max\_Unroll\_Prod - 1$ sumamos uno al índice i.

3.3.2. Lista de referencias a array

El compilador crea una lista ($\_arl$) donde se almacena, para cada array que aparece en el bucle más interno del SNL, la lista de referencias de ese array. Esta lista será utilizada posteriormente para probar las distintas posibilidades

de desenrosque. Manipulando copias de $\_arl$ simularemos el desenrosque sin necesidad de modificar el código original.

También se añaden a la lista los arrays que se crearán por aplicación de expansión de escalares:

- Una expresión invariante se trata como un array que varía en las dimensiones donde varía la expresión.
- Variables escalares a las que se debe aplicar expansión para que los bucles sean totalmente permutables.

Una vez construida la lista de referencias, se calculan los siguientes valores que se utilizarán en la evaluación del coste:

$\_scalar\_int\_regs$, $\_scalar\_fp\_regs$: Número de registros necesarios para almacenar escalares enteros y de coma flotante respectivamente. Se utilizarán en la estimación de la presión sobre los registros.

$\_num\_int\_array\_refs$, $\_num\_fp\_array\_refs$: Número de referencias a arrays de enteros y de coma flotante. Ambos intervienen en el cálculo del número de spills (sólo cuando no hay desenrosque). $\_num\_fp\_array\_refs$ también se utiliza para calcular el factor de corrección $minvar\_benefit$ que se aplica al coste.

3.3.3. Grafo de latencias

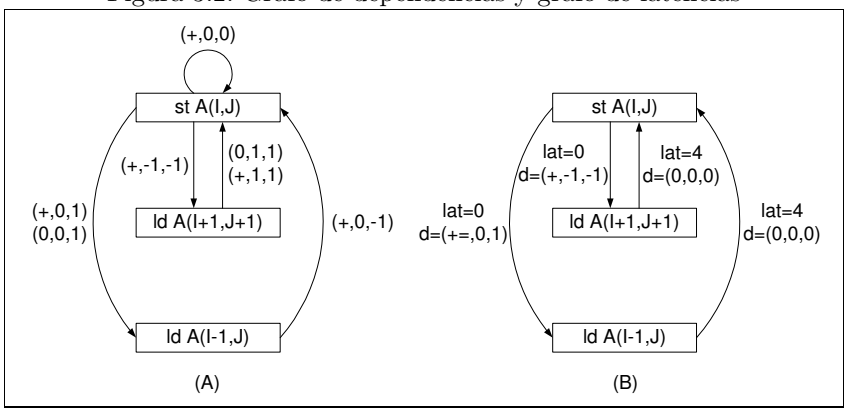
Otra estructura de datos que podemos generar sin necesidad de conocer cual será el bucle interno es el grafo de latencias, que contendrá:

- Un vértice para cada load/store de un array de valores en coma flotante.
- Para cada load, una arista desde el load hasta el store padre, etiquetada con distancia cero y con la latencia total de todas la operaciones que encontramos entre el load y el store.
- Aristas expresando las dependencias de flujo entre los stores y los loads. Estas aristas se etiquetan con un vector de dependencia que resume las dependencias expresadas en el grafo de dependencias. La latencia asignada a estas aristas es cero.

El grafo de latencias se genera a partir de un recorrido recursivo de la representación intermedia del código. Cuando se encuentra un store que tiene un vértice asociado en el grafo de dependencias entre arrays, se añade un nuevo nodo al grafo de latencias y se recorren los hijos del store hasta llegar a los loads. Durante el recorrido de los hijos se acumulan los costes de cada operación, que se utilizan para etiquetar la arista entre el load y el store. Los costes de las operaciones se obtienen traduciendo las instrucciones en WHIRL a instrucciones del lenguaje máquina.

Junto con el grafo de latencias se mantiene una tabla de hash donde cada entrada contiene una tupla que relaciona los vértices del grafo de dependencias entre arrays con su vértice respectivo en el grafo de latencias. Esta tabla de hash es utilizada para facilitar la localización de los vértices del grafo de latencias al

Figura 3.2: Grafo de dependencias y grafo de latencias



añadir las aristas que expresan las dependencias de flujo. Las dependencias de flujo entre escalares se ignoran.

La figura 3.2 muestra el grafo de dependencias (A) y el grafo de latencias (B) correspondientes al siguiente código:

```
do K = 1, N
  do J = 1, N
    do I = 1, N
      A(I,J) = A(I-1,J) + A(I+1,J+1)
    enddo
  enddo
enddo
```

3.4. Selección del bucle interno

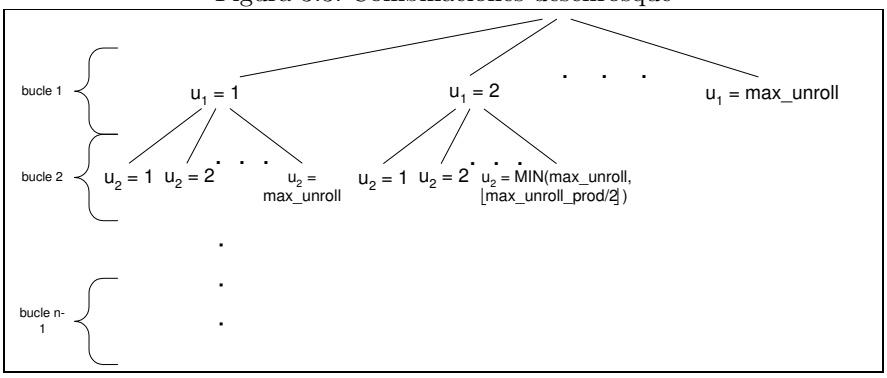
Tras las inicializaciones se procede a probar cada uno de los posibles bucles internos. Para cada uno de ellos se llama a la función Try_Inner() que, una vez seleccionados los factores de desenrosque, pasará el control al modelo de la cache para que decida el tiling (ver figura 3.1). Finalmente compara el coste con el mejor resultado obtenido hasta el momento y descarta las transformaciones que conllevan un mayor tiempo de ejecución.

3.4.1. Ciclos por recurrencias

Dado un bucle interno, el primer paso llevado a cabo en la función Try_Inner() es el cálculo del coste por recurrencias. Para ello, se genera una lista de todos los componentes fuertemente conexos (SCC) del grafo de latencias. Para cada uno de los SCCs, se aplica un algoritmo de búsqueda del mínimo intervalo de iniciación (ver [9], [4]). Este algoritmo recibe como parámetro de entrada una cota inferior cuyo valor inicial es $\_loop\_rcycles\_unroll\_by[Max\_Unroll\_Prod-1]$.

El resultado de este proceso es el valor máximo, entre todas las recurrencias, de la relación $\frac{\sum latencias}{\sum distancias}$.

Figura 3.3: Combinaciones desenrosque



Volviendo al ejemplo de la figura 3.2, el mínimo intervalo de iniciación son 4 ciclos para el bucle I. Si situamos J como bucle interno el mínimo intervalo de iniciación se reduce a 0 ciclos debido a que no hay dependencia entre iteraciones sucesivas de J. El bucle K no forma parte del conjunto de bucles totalmente permutables y, por tanto, no puede ser candidato a bucle interno.

3.5. Selección de los factores de desenrosque

La función Try_Unroll() es la encargada de explorar el espacio de posibles factores de desenrosque (ver figura 3.3). Esta exploración se lleva a cabo mediante un recorrido recursivo empezando por el bucle más externo. Sea l el bucle en curso, para cada posible factor de desenrosque de l se realiza una llamada recursiva pasando $l+1$. Al llegar al bucle más interno (que no se va a desenrollar) se llama a la función de evaluación del coste.

El factor de desenrosque que podemos aplicar a cada bucle depende de los flags y directivas de compilación. Por defecto se establece que el factor de desenrosque de un bucle puede ser como máximo de 10 iteraciones (max_unroll) y que el producto de los factores de desenrosque de todos los bucles no puede superar un máximo de 16 iteraciones (max_unroll_prod).

Las siguientes estrategias se utilizan para reducir la búsqueda:

- Sea l el bucle en curso, y $u_l = x$ su factor de desenrosque, si el retorno de la llamada recursiva para $l+1$ informa que no hay registros suficientes, no es necesario probar valores para $u_l > x$.
  Si, además, $x = 1$ se propaga este resultado informando a la llamada correspondiente a $l-1$.

- Si $u_l < max\_unroll$ y $u_l \times 2 > \_est\_num\_iterations[l]$, entonces $u_l = max\_unroll$.

- Finalmente, si al evaluar el coste (ver sección 3.6), se determina que se ha encontrado una combinación suficientemente buena, se abandona la búsqueda.

A medida que se recorre el espacio de factores de desenrosque se van generando copias "desenrolladas"de la lista de referencias a array ($\_arl$). De este modo, cada nodo del árbol de la figura 3.3 tiene asociada una lista de referencias a memoria que refleja los accesos del bucle interno tras aplicar la combinación de factores de desenrosque representada en el camino que lleva al nodo.

Antes de realizar la llamada recursiva, donde esta lista se pasa como parámetro, se eliminan las sub-expresiones comunes (CSEs) y se marcan las referencias invariantes.

Eliminación de CSEs: Dos referencias representan una subexpresión común si ambas acceden a la misma posición en un pequeño intervalo de iteraciones del bucle interno, por defecto igual a 6 (Max_Cse_Dist). Por ejemplo, las expresiones a[i] y a[i+1] pertenecen a la misma clase de equivalencia y contabilizarán como un sólo acceso a memoria, ya que el valor leído para a[i+1] será reutilizado para a[i] en la siguiente iteración. Si la clase de equivalencia contiene una lectura y una escritura y la primera lectura precede lexicográficamente la primera escritura se contarán dos accesos.

Marcaje invariantes: Se marcan las clases de equivalencia invariantes respecto al bucle interno. Estos accesos podrán sacarse fuera del bucle interno y no se contabilizan como acceso a memoria, aunque sí se deben tener en cuenta al calcular la ocupación de registros.

3.6. Evaluación del coste

En la sección 3.2 enumerábamos los elementos que intervienen en el modelo del procesador. Quedan por calcular el número de accesos a memoria y la ocupación de registros. Después se podrá comparar el resultado con el mejor obtenido hasta el momento. De todo ello, se encarga la función Evaluate().

Podemos distinguir tres partes en la función de evaluación:

1. En primer lugar se analiza la lista de referencias a memoria y se calcula el número de referencias y el número de registros necesarios para almacenar direcciones base y datos.

2. Calculamos el número de operaciones de memoria por iteración del bucle interno original y su coste en ciclos.

3. Tomando en consideración todos los elementos del modelo se calcula el coste, se le aplican distintos factores de corrección y se compara con el mejor resultado obtenido hasta el momento.

3.6.1. Número de registros y referencias a memoria

El número de registros y accesos a memoria se obtiene a partir de la lista de referencias a memoria. Esta lista contiene, para cada variable de tipo array referenciada en el cuerpo del bucle, una lista donde cada nodo representa una clase de equivalencia. Dos referencias pertenecen a la misma clase si acceden a la misma posición de memoria en un intervalo limitado de iteraciones del bucle interno.

El número total de registros y referencias a memoria se calcula como la suma de los registros y referencias de cada uno de los arrays de la lista. Para cada referencia (nodo que representa una clase de equivalencia) de cada array:

1. Si la referencia no es invariante, comprobamos si es necesario un nuevo registro base para el array. Si encontramos, entre las ya tratadas, una referencia que sólo difiera de la referencia en curso en la dimensión más interna y cuya diferencia sea constante y de valor absoluto inferior a 16, no será necesario un nuevo registro base.

2. Comprobamos el tipo de la referencia y el número de registros necesarios para almacenar un valor de dicho tipo (2 regs. en el caso de cuádruple precisión y complejos).

3. Actualizamos el número de registros (para datos) y accesos a memoria:

   - Si el nodo es invariante el acceso a memoria se realizará fuera del bucle interno y sólo incrementamos el número de registros.
   - Si el nodo es una subexpresión común (CSE) el número de registros necesarios se calcula como:

     $$MIN(Max\_Cse\_Dist, \_max\_inner\_offset - \_min\_inner\_offset) \times regs\_per\_ref$$

     donde:

     $\_max\_inner\_offset - \_min\_inner\_offset$ representa la distancia, en iteraciones del bucle interno que separa las dos referencias de la clase de equivalencia más alejadas entre sí.

     $regs\_per\_ref$ es el número de registros por referencia calculado en el punto 2.

     También se incrementan el número de accesos a memoria.
   - Si la clase de equivalencia contiene loads duplicados incrementamos el número de registros.
     Si contiene algún store incrementamos el número de accesos a memoria. Si contiene algún load y la primera ocurrencia no es un store se debe contabilizar la carga del dato incrementando el número de accesos.
   - Si no se trata de ninguno de los casos anteriores (invariante, CSE o duplicado) entonces incrementamos el número de accesos a memoria.

4. Para terminar, si un nodo contiene stores, comprobamos si se trata de una referencia invariante respecto al bucle interno.

Mediante el análisis que acabamos de describir se obtienen los siguientes datos:

num_fp_regs: número de registros necesarios para almacenar valores en coma flotante.

num_fp_refs: número de accesos a memoria de coma flotante.

num_fp_variant_stores: número de accesos a memoria no invariantes para stores de coma flotante.

num_fp_invariant_stores: número de registros para stores invariantes de coma flotante.

y también se obtienen los mismos datos para los enteros: **num_int_regs**, **num_int_refs**, **num_int_variant_stores** y **num_int_invariant_stores**.

3.6.2. Número de operaciones de memoria

Calculamos el número de operaciones de memoria por iteración del bucle original (MEM_issue):

$$MEM\_issue = \frac{num\_fp\_refs + num\_int\_refs}{\prod \text{factores de desenrosque}}$$

y el número de ciclos suponiendo un ciclo por operación:

$$MEM\_rcycles = \frac{MEM\_issue}{\text{número de unidades de mem}}$$

3.6.3. Estimación del número de ciclos

En este punto disponemos de todos los elementos que intervienen en la estimación del coste del código transformado.

$$ciclos = MAX(\text{coste recursos}, \frac{\text{coste recurrencias}}{\prod \text{factores de desenrosque}})$$

donde el coste por recurrencias es el valor calculado en la sección 3.4.1 y el coste por recursos se define como:

$$\text{coste recursos} = MAX(OP\_rcycles, MEM\_rcycles, \frac{ALL\_issue}{ISSUE\_rate})$$

Sea *unroll_prod* igual al producto de factores de desenrosque, los términos que intervienen en el cálculo de los ciclos por recursos toman los siguientes valores:

$OP\_rcycles$ = *_loop_rcycles_unroll_by*$[unroll\_prod - 1]$, calculado en la sección 3.3.1.

$MEM\_rcycles$ = valor que acabamos de calcular (sección 3.6.2).

$ALL\_issue = OP\_issue + MEM\_issue + \frac{LOOP\_INIT\_issue}{unroll\_prod}$, donde OP_issue proviene de la sección 3.3.1 y $LOOP\_INIT\_issue$ es un valor constante (ver sección 3.3).

Antes de incorporarlo a la función que devuelve el número de ciclos, el coste por recursos se ajusta restándole el término $minvar\_benefit$, que se obtiene como producto de los siguientes factores:

$$\begin{aligned} minvar\_benefit &= MINVAR\_MAGIC\_COEFF \\ &\times Inner\_Invariant\_Refs \\ &\times unroll\_product \\ &\times unequal\_unroll\_penalty \\ &\times Invariant\_Ref\_Coeff \end{aligned}$$

donde:

$MINVAR\_MAGIC\_COEFF = \_loop\_rcycles\_unroll\_by[Max\_Unroll\_Prod - 1] \times 0{,}20$. Limita el beneficio potencial a un veinte por ciento del número de ciclos ideal.

$Inner\_Invariant\_Refs$**:** Número de referencias a arrays invariantes respecto al bucle interno en curso. Se obtiene antes de empezar a probar factores de desenrosque y de clasificar los accesos en clases de equivalencia.

*unroll_product***:** Lo añadimos porque el valor anterior se obtiene sin tener en cuenta el desenrosque.

*unequal_unroll_penalty***:** Penalización por utilizar factores de desenrosque desiguales. Sean $un1$ y $un2$ los factores mayor y menor respectivamente, se obtienene como $(\frac{un2}{un1})^{0,3}$.

$Invariant\_Ref\_Coeff$**:** Coeficiente que aplicado a los factores anteriores nos servirá para estimar el beneficio que se puede obtener si se continúa aplicando desenrosque una vez obtenida una planificación ideal. Se define como $\frac{1}{\_num\_fp\_array\_refs \times Max\_Unroll\_Prod}$.

Por tanto, modificando la equación anterior:

$$ciclos = MAX(\text{coste recursos} - minvar\_benefit, \frac{\text{coste recurrencias}}{\prod \text{factores de desenrosque}})$$

Este valor no es definitivo ya que, antes de comprobar si esta transformación supera al mejor resultado hasta el momento, se le van a aplicar una serie de factores para penalizar situaciones no deseadas:

- Presión sobre el banco de registros: Si el número de registros de coma flotante necesarios iguala o supera el número de registros disponibles menos uno, incrementaremos el coste en un diez por ciento. Añadiremos otro diez por ciento si ocurre lo mismo respecto a los registros de enteros.

- Factores múltiplos del número de iteraciones: Al compilador le interesa que los factores de unroll sean múltiplos del número de iteraciones[1]del bucle, especialmente cuando el bucle que se va a desenrollar tiene pocas iteraciones. Cuando el bucle a desenrollar tenga menos de un cierto número de iteraciones (LNO_Outer_Unroll_Max, por defecto igual a 10) y el factor de desenrosque no sea un divisor se añadirá una penalización al número de ciclos.
  Para cada bucle se evalúa el peso del número de iteraciones sobrantes respecto al número de iteraciones del bucle. Entre todos los bucles, se toma aquel cuyo cociente es mayor.
  Si $\frac{\text{iteraciones sobrantes}}{\text{número de iteraciones}} > 0{,}1$ el número de ciclos se multiplica por 1,05.

- Número par de operaciones de coma flotante: En el caso de IA64, añadimos otra penalización multiplicando por 1,05 si el número de operaciones de coma flotante es impar. Esta penalización sólo se añade si el número de operaciones en el bucle interno supera un determinado valor, ya que si hay pocas operaciones se podrá continuar desenrollando durante la generación de código.

[1]Si el compilador no es capaz de calcular el número de iteraciones de un determinado bucle asume LNO_Num_Iters iteraciones (por defecto 100).

3.6.4. Estimación del consumo de registros

El número de registros de coma flotante necesario para la ejecución de una iteración del bucle interno en el código transformado se obtiene sumando $\_base\_fp\_regs + \_scalar\_fp\_regs + num\_fp\_regs$:

_base_fp_regs: Registros de coma flotante necesarios para mantener el pipeline a pleno rendimiento (sección 3.3). El número de registros para escrituras sobre referencias invariantes respecto al bucle interno no se debe contabilizar dos veces. Se aplica el siguiente criterio:
Si $num\_fp\_invariant\_stores > 4 \times num\_fp\_variant\_stores$ entonces:

$$new\_base\_regs = \frac{\_base\_fp\_regs}{3}$$

_scalar_fp_regs: Registros necesarios para almacenar escalares de coma flotante. Ver sección 3.3.2.

num_fp_regs: Registros necesarios para almacenar elementos de un array. Valor calculado en 3.6.1. Si hay productos de números complejos son necesarios más registros temporales. Añadimos a este término el número de multiplicaciones de complejos (obtenido en la sección 3.3.1) multiplicado por el producto de factores de desenrosque.

Si el bucle tiene pocas iteraciones (menos que LNO_Small_Trip_Count, por defecto igual a 5) o se le va a aplicar software pipelining sólo se tiene en cuenta _scalar_fp_regs.

En el caso de los enteros, los términos a sumar son:

_base_int_regs: Registros reservados. Ver sección 3.3.

_scalar_int_regs: Registros necesarios para almacenar escalares enteros. Ver sección 3.3.2.

num_int_regs: Valor calculado en 3.6.1.

Si el número de registros necesarios no supera el número de registros disponibles, la función de evaluación compara la estimación del coste con el mejor coste obtenido hasta el momento.

En el caso en que el número de registros disponibles no sea suficiente hay que distinguir entre dos situaciones:

1. Los factores de unroll son iguales a uno o describen un outer unroll predeterminado (mediante directivas o flags de compilación).
   Cuando esto ocurre el compilador calcula el número de spills que se van a tener que realizar y contabiliza, en paralelo, el número de ciclos necesarios teniendo y sin tener en cuenta los spills. Si al final el número de ciclos en uno y otro caso coincide, las operaciones de spill no están en el camino crítico y el número de registros necesarios será igual al número de registros disponibles. El número de registros necesarios se tiene en cuenta al decidir qué transformación es mejor.

2. Si los factores de unroll describen una situación donde efectivamente hay unroll (distintos de uno o de valores predeterminados) y el número de registros disponibles es insuficiente, directamente, descartamos esta transformación.

En ambos casos se pone a falso la variable *can_reg_allocate* para indicar que no es necesario probar factores de desenrosque superiores. Si para la combinación que describe la ausencia de desenrosque se detecta que no hay suficientes registros el valor de *can_reg_allocate* se irá propagando hacia las llamadas recursivas de Try_Unroll() anteriores provocando que termine la exploración de combinaciones de desenrosque (ver sección 3.5).

3.6.5. Mejor transformación

Los siguientes atributos de la clase LOOP_MODEL almacenan el mejor resultado obtenido hasta el momento por el modelo del procesador para un determinado bucle interno:

_inner_loop_inner: bucle interno.

_block_number_inner: factores de outer unroll.

_num_cycles_inner: número ciclos por iteración del bucle original.

_num_fp_regs_inner: registros de coma flotante utilizados en el código desenrollado.

_num_fp_refs_inner: accesos a memoria de coma flotante en el código desenrollado.

_num_int_regs_inner: registros de enteros.

_num_int_refs_inner: accesos a memoria de enteros.

_unroll_prod_inner: producto de factores de desenrosque.

_model_limit: Indica si se ha conseguido el número de ciclos ideal, si está limitado por latencias o si está limitado por recursos.

Sustituiremos la mejor combinación hasta el momento por la combinación en curso si ésta mejora la anterior en al menos un uno por ciento:

$$1{,}01 \times ciclos < \_num\_cycles\_inner$$

o , a igual número de ciclos, si hemos encontrado una combinación que consume menos registros o el producto de factores de desenrosque es menor.

Estos atributos se incializan con valores por defecto en cada llamada a Try_Inner() y sólo sirven para comparar entre los distintos factores de desenrosque dado un bucle interno. Para comparar entre distintos bucles internos también hay que tener en cuenta el resultado del modelo de la cache.

Capítulo 4

Modelo de la cache

En el capítulo 3, se ha visto como la elección de las transformaciones a aplicar sobre una anidación de bucles se lleva a cabo mediante un recorrido del espacio de transformaciones. Para cada una de las posibilidades, se estima el coste de ejecución utilizando dos modelos:

1. El modelo del procesador, que contempla issue, unidades funcionales, latencias y registros.

2. El modelo de la cache, que, basándose en una descripción de la jerarquía de memoria, evalúa el coste debido a fallos de cache/tlb e inicialización de los bucles internos.

La estimación del tiempo de ejecución se obtiene sumando el resultado de ambos modelos.

En este capítulo se describe el modelo de la cache. En la sección 4.1 se presenta el algoritmo utilizado para estudiar las distintas posibilidades de bloqueo. En la sección 4.2 se comenta brevemente como se aplica el algoritmo para bloquear para distintos niveles de cache.

El primer paso del algoritmo consiste en determinar en qué bucles se podrá sacar provecho de la aplicación de tiling. Para ello se debe analizar el reuso que presentan los distintos bucles de la anidación. Este análisis se describe en la sección 4.3.

En la sección 4.4 se describe el cálculo del footprint. Éste concepto, que informalmente podríamos definir como la cantidad de datos accedidos en todas las iteraciones de un determinado bucle, es la base de la fórmula paramétrica que nos permitirá obtener una estimación del coste en función de los factores de bloqueo.

La construcción de dicha fórmula, que contempla el coste por fallos en cache, el coste por fallos en el tlb y el overhead introducido al incrementar el número de bucles de la anidación, se describe en la sección 4.5.

En la sección 4.6 se enumeran y comentan brevemente los distintos parámetros devueltos por el modelo. Por último, la sección 4.7 presenta la descripción de la jerarquía de memoria utilizada por el modelo en la toma de decisiones.

4.1. Algoritmo

El modelo de la cache es el responsable de calcular un bloqueo que minimice el coste por fallos en cache y tlb. El punto de entrada es la función Cache_Model() (be/lno/cache_model.cxx), que recibe como parámetros, entre otros, el bucle interno y los factores de desenrosque de los bucles externos. Esta función se llamará para cada uno de los posibles bucles internos (ver figura 3.1). Para cada uno de ellos el modelo del procesador habrá precalculado el "mejor" outer unroll. El objetivo será calcular el mejor tiling dados el bucle interno y los factores de desenrosque. Se procede como indica el algoritmo 4.

Se empieza seleccionando los bucles de la anidación que acarrean algún tipo de reuso, que son los que nos van a permitir sacar provecho del tiling. A continuación, para cada subconjunto de los bucles con reuso, calculamos los factores de bloqueo que minimicen el coste por fallos en cache y comparamos con el mejor resultado obtenido hasta el momento.

Algoritmo 4: Selección del bloqueo para un nivel de cache
Entrada: Anidación de bucles, bucle interno y factores de outer unroll.
Salida: Mejor tiling y su coste.

```
(1)  mejor_coste_bloq = infinito
(2)  mejor_trans_bloq = ninguna
(3)  bucles_loc = subconjunto de bucles de la anidación que
     acarrean más reuso (hasta un máximo de cuatro)
(4)  foreach (subconjunto ⊆ bucles_loc) ∪ bucle interno
(5)      trans_bloq = cacular mejores tamaños de bloque para este tiling/unrolling
(6)      coste_bloq = calcular cache y loop overhead para este tiling/unrolling
(7)      if coste_bloq < mejor_coste_bloq
(8)          mejor_coste_bloq = coste_bloq
(9)          mejor_trans_bloq = trans_bloq
```

La ordenación (permutación) de los bucles después del tiling se obtiene al desplazar los bucles que forman parte del bloque hacia las posiciones más internas. En el siguiente ejemplo:

```
do i=1,L
  do j=1,M
    do k=1,N
      do l=1,P
```

sean i, j, l los bucles que participan en el bloqueo, la posición final de los bucles será:

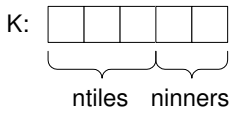
```
do k=1,N
  do j'=1,M,B1
    do l'=1,P,B2
      do i=1,L
        do j=j',MIN(j'+B1,M)
          do l=l',MIN(l'+B2,P)
```

Por tanto, se desplazan los bucles bloqueados hacia el interior de la anidación, pero no se modifica su posición relativa.

Los bucles j', l', que implementan el recorrido de los bloques, se colocan a continuación del último bucle que forma parte del bloque, en este caso el bucle i. Al bucle más externo del bloque, denominado middle loop, no se le aplica strip mining.

4.1.1. Recorrido de los subconjuntos de bucles con reuso

En el algoritmo 4 vemos que se calculará un bloqueo para cada subconjunto de los bucles que acarrean reuso.



Si representamos el conjunto de bucles que conforman la anidación mediante un mapa de bits, podemos realizar un recorrido por todos los subconjuntos posibles mediante un bucle cuya variable de iteración K toma valores entre cero y $2^{ninners+ntiles} - 1$. En esta expresión $ninners$ representa el número de bucles internos sobre los que podemos aplicar intercambio y $ntiles$ el número de bucles a los que hemos decidido no aplicar intercambio pero que podemos bloquear. Desde un punto de vista de legalidad de la transformación todos los bucles que participen en el bloqueo deben ser totalmente permutables, pero en la sección 2.5 hemos visto que podemos decidir limitar la aplicación de intercambio a los bucles más internos.

Al iterar para los distintos valores de K se descartarán aquellos subconjuntos que incluyan bucles sin reuso (excepto cuando se produzca una de las situaciones descritas en el apartado 4.1.2), o que no incluyan el bucle interno.

4.1.2. Inclusión de bucles sin reuso

Hay dos circunstancias que pueden provocar la inclusión de bucles que no acarrean reuso dentro del bloque:

1. Cuando el bucle que no acarrea reuso es el bucle interno.
2. Cuando la inclusión de un bucle que no acarrea reuso va a permitir poder continuar bloqueando.

El segundo caso se produce cuando se incluye en el bloque un bucle que no se va a intercambiar. Entonces será necesario que el bloque incluya también todos los bucles que se encuentran por debajo de éste.

4.2. Bloqueando para distintos niveles de cache

El algoritmo anterior lleva a cabo el bloqueo para un nivel de cache, pero es probable que el procesador disponga de múltiples niveles de cache y que se desee realizar el bloqueo para distintos niveles. En ese caso, el procedimiento consiste en aplicar el mismo algoritmo para cada uno de los distintos niveles empezando por el nivel más próximo al procesador.

Recuperando el ejemplo de la sección 4.1, una vez calculado el bloqueo para L1:

```
do k=1,N
  do j'=1,M,B1
    do l'=1,P,B2
      do i=1,L
        do j=j',MIN(j'+B1,M)
          do l=l',MIN(l'+B2,P)
```

la visión de la anidación que tendrá el compilador al bloquear para L2, será la de un conjunto de bucles cuyo bucle más interno es i (el middle loop del bloqueo anterior), en cada iteración del cual se accede a bloques de tamaño $B1 \times B2$. Por tanto, en el ejemplo, los bucles a considerar al bloquear para L2 son k, j', l' e i.

4.3. Análisis del reuso

Una referencia presenta self-reuse cuando accede al mismo dato, o a datos que comparten la misma línea de cache, en iteraciones distintas. Por ejemplo a(i), dentro de una anidación i, j, presenta self-reuse a lo largo del bucle j. También a lo largo del bucle i si hay más de un elemento por línea de cache.

Un conjunto de referencias presentan group-reuse si referencias distintas acceden al mismo dato o a la misma línea de cache. Por ejemplo a(i,j) y a(i,j+1) presentan group-reuse en j.

El compilador analiza la existencia de reuso en dos pasos: en primer lugar comprueba la presencia de self-reuse para cada una de las referencias y, a continuación, se toman todas las referencias por parejas y se procede a la detección de group-reuse.

Self-reuse: Una referencia presenta self-reuse a lo largo de todos aquellos bucles cuyas variables de iteración no aparecen indexando la referencia. Se considera que los bucles que aparecen en la *stride one dimension*[1]tienen reuso debido a la reutilización de la línea de cache si el coeficiente que los acompaña no supera un determinado valor (Happy_Coefficient).

Group-reuse: Una pareja de referencias presentan group-reuse a lo largo de un determinado bucle si el índice de dicho bucle indexa la *stride one dimension* de ambas referencias con un mismo coeficiente y, además, dicho coeficiente es divisor de $const\_offset_1 - const\_offset_2$. Esta última expresión, que debe ser distinta de cero, es la diferencia entre los offsets constantes que aparecen en la *stride one dimension* de cada una de las referencias.

Cuando el número de bucles con reuso es superior a cuatro, se toman los cuatro con más reuso. Se calcula el footprint de cada uno de los bucles con reuso suponiendo que dicho bucle es el más interno y se toman los cuatro bucles con un tamaño de footprint menor.

[1]En Fortran, donde las matrices se almacenan por columnas, la *stride one dimension* de una referencia es la dimensión situada más a la izquierda. Debido a que elementos consecutivos están en posiciones adyacentes, es a lo largo de esta dimensión donde podemos sacar partido del reuso de la línea de cache.

4.4. Footprints

Un concepto muy importante en el cálculo del tamaño de bloque y que también interviene en el análisis del reuso es el *footprint*. En [14] se define el footprint de una referencia como la porción de un array accedida por dicha referencia. En el código:

```
do I = 1,N
  A(I,1) = B(2,I) * C(I)
```

el footprint de la referencia A(I,1) es la primera columna de A, mientras que el footprint de B(2,I) es la segunda fila de B. Cuantos elementos de la fila o columna incluye el footprint depende del tamaño de N.

En [12] se define el footprint $Fp(u)$ de una variable u para el bucle p como la fracción de la cache utilizada por la variable u en una iteración del bucle p. A diferencia de la anterior, esta definición hace referencia a la ocupación de la cache. Este aspecto es relevante porque, para seleccionar unos factores de bloqueo adecuados, lo que nos interesa no es tanto cuantos elementos se van a acceder como qué cantidad de cache van a ocupar. En el compilador el footprint de un bucle se refiere a todas sus iteraciones.

4.4.1. Cálculo del footprint

El cálculo del footprint es la base de la fórmula del coste del modelo de la cache. A continuación pasamos a describir como se calcula el footprint y, posteriormente, veremos como se utiliza en el cálculo del coste. Estas descripciones pretenden dar una idea bastante aproximada del proceso que se lleva a cabo en el compilador, pero se quedan fuera muchos detalles y tratamientos de casos especiales que sí se tienen en cuenta en la implementación. Podemos encontrar más información en los comentarios del archivo cache_model.cxx.

Tomando una anidación de bucles con un acceso a un array que representamos de la siguiente forma:

```
for i⃗
  x [H  ×  i⃗  +  c⃗ ]
```

donde H es una transformación lineal y $\vec{c}$ un vector de constantes. Sea $Ker(H)$ el núcleo de H y $Ker^{\perp}(H)$ el subespacio ortogonal al núcleo de H, el punto de partida para el cálculo del footprint es la expresión:

$$Footprint = \prod_{\vec{v_i} \in Ker^{\perp}(H)} \left(\sum_{c_{ik} \in \vec{v_i} | c_{ik} \neq 0} \frac{P \times A_k}{|c_{ik}|} \right)$$

donde:

$P = \prod_{c_{ij} \neq 0} |c_{ij}|$,

y $A_k = N_k$, es el número de iteraciones del bucle k.

El footprint de un bucle se representa como una expresión paramétrica que depende de los factores de bloqueo. En el siguiente ejemplo:

```
do k=1, N
  do j=1, B1
    do i=1, B2
      a(i,j)
```

tenemos que:

$$H = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, ker(H) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ y } ker^{\perp}(H) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Sustituyendo en la fórmula por los valores correspondientes se obtiene que el valor del footprint de la referencia $a(i, j)$ es igual a $B2 \times B1$.

Esta expresión se debe modificar para tener en cuenta el tamaño y el reuso de la línea de cache. En primer lugar definimos H_s igual a la matriz H eliminando la fila que corresponde a la *stride one dimension* (ver sección 4.3). A continuación comprobamos la condición $dim(ker(H)) < dim(ker(H_s))$. Un resultado negativo indicaría que la eliminación de la fila correspondiente a la *stride one dimension* no comporta un aumento del reuso. En ese caso tomamos la fórmula anterior y multiplicamos por el número de elementos de la línea de cache.

En caso contrario definimos:

$$X = min(|k_j|, cls)$$

donde cls es el número de elementos de la línea de cache. Y aplicamos la siguiente modificación al cálculo de A_k:

$$A_k = \begin{array}{ll} N_k & \text{, para } k \text{ distinto del } \textit{stride one loop} \\ (N_j - 1) \times X + cls & \text{, cuando } j \text{ es el } \textit{stride one loop} \end{array}$$

donde el *stride one loop* se refiere al bucle que produce un desplazamiento menor de la dirección accedida por una referencia al incrementar su variable de iteración. En el ejemplo, el *stride one loop* de $a(i, j)$ es el bucle i, pero no siempre coincidirá con el bucle más interno.

Si repetimos el cálculo del footprint de $a(i, j)$ tras esta modificación el resultado es $((B2 - 1) \times 1 + cls) \times B1$. Teniendo en cuenta que la línea de cache va a ser reutilizada, podemos interpretar X como el número de elementos que debemos contabilizar por cada incremento del *stride one loop*.

Cuando el código contiene múltiples referencias, si éstas acceden a arrays distintos, podemos calcular el footprint total como la suma del footprint de cada array. Pero si hay referencias distintas que acceden a un mismo array se debe tener en cuenta la posible existencia de reuso entre ellas (group-reuse).

Supongamos que nos encontramos con las referencias $a(3 * i)$ y $a(3 * i + 1)$. No podemos calcular el footprint de cada una de ellas por separado, y después sumar, porque no se tendría en cuenta el reuso de la línea de cache. Modificamos la definición de X para contemplar este caso:

$$X = min(|k_j|, cls + spread)$$

donde *spread* es la diferencia entre el valor máximo y mínimo de los términos constantes. Tras esta modificación el footprint de ambas se calcula como si se tratara de una sola referencia.

Por último modificamos también la definición de A_k:

$$A_k = \begin{array}{ll} N_k + Max_k - Min_k & , k \text{ distinto del } \textit{stride one loop} \\ (N_j - 1) \times X + cls + Max_j - Min_j & , \text{cuando } j \text{ es el } \textit{stride one loop} \end{array}$$

Esta fórmula se aplicará para calcular el footprint de cada reference group. El footprint total se obtiene sumándolos todos ellos. En la siguiente sección se describe como se clasifican las referencias en reference groups y como se obtienen los valores Max y Min.

4.4.2. Clasificación de referencias en reference groups

Sea n la profundidad de una anidación, y d el número de dimensiones de un array A. Dos referencias $A[\vec{f}(\vec{i})]$ y $A[\vec{g}(\vec{i})]$, donde $\vec{f}$ y $\vec{g}$ son funciones de indexación $Z^n \rightarrow Z^d$, se denominan uniformemente generadas si

$$\vec{f}(\vec{i}) = H\vec{i} + \vec{c}_f \quad \text{y} \quad \vec{g}(\vec{i}) = H\vec{i} + \vec{c}_g$$

donde H es una transformación lineal y $\vec{c}_f$ y $\vec{c}_g$ son vectores constantes.

Como la posibilidad de que haya reuso explotable entre referencias que no son uniformemente generadas es muy pequeña, se agrupan las referencias en clases de equivalencia que acceden al mismo array y que comparten la matriz H. Estas clases de equivalencia se denominan *uniformly generated sets*.

Las referencias que no pertenezcan al mismo *uniformly generated set* se clasificarán en *reference groups* distintos. También se clasificarán en *reference groups* distintos si, aún cumpliendo la condición anterior, la ecuación $H_s \times \vec{i} = c2_s - c1_s$ no tiene solución (no hay reuso de la misma línea de cache). Si tomamos, por ejemplo, las referencias a(j, i, i+1) y a(j, i+1, i+1):

$$H = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}, c2 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, c1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$H_s = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}, c2_s = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, c1_s = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

éstas se clasificarán en *reference groups* distintos ya que no hay solución para:

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Cada *reference group* se representa en el compilador mediante un elemento de tipo RG que, a su vez, contiene una lista de elementos de tipo RG_NODE. Esta lista se utiliza para clasificar separadamente aquellas referencias que a priori pertenecen al mismo *reference group* pero cuyos términos constantes distan demasiado. El término *reference group* puede hacer referencia tanto a un RG como a un RG_NODE.

De entrada, dos referencias como $a(i)$ y $a(i + 100)$, se clasficarán en el mismo RG pero en RG_NODEs distintos. Si en el código aparecen otras referencias cuyos términos constantes son valores intermedios, a base de ir fusionando *reference groups*, $a(i)$ y $a(i + 100)$ pueden terminar compartiendo RG_NODE.

Cada *reference group* (RG_NODE) mantiene una lista con los valores Max y Min para cada uno de los bucles. Como no siempre habrá una correspondencia

directa entre el número de bucles y el número de dimensiones de las referencias, los Max y Min de cada bucle se obtienen resolviendo $H \times x = (c2 - c1)$, donde c1 corresponde al término constante de la referencia "representativa". Hay una referencia representativa por RG que se toma como punto de referencia para comparar el resto.

4.4.3. Cálculo del número de misses

Una vez calculados los footprints de los distintos bucles que intervienen en el bloqueo podemos empezar a construir la fórmula que nos permitirá estimar el coste por fallos en la cache. En concreto, en esta sección veremos como se obtiene la cantidad de fallos en número de bytes. En la sección 4.5.1 se explica como esta cantidad de fallos en número de bytes (en adelante *miss bytes*) se traduce a coste en número ciclos.

Supongamos que deseamos bloquear la siguiente anidación:

```
do i=1, N
  do j=1, B
    ...
```

Sean F_0 el footprint del bucle i y F_1 el footprint del bucle j podemos calcular ND, la cantidad de nuevos datos que se traen a la cache en cada iteración de i, como:

$$ND = \frac{F_0 - F_1}{v1 - 1} \tag{4.1}$$

donde $v1$ es el número de iteraciones de i ($v1 = N$). Podemos obtener esta fórmula aislando ND en la ecuación $F_0 = F_1 + ND \times (v1 - 1)$, que refleja que el footprint del bucle i es igual al footprint de j, que se carga en la primera iteración de i, más los nuevos datos que se llevan a la cache en el resto de iteraciones de i.

El siguiente paso consiste en el cálculo de la cantidad de cache necesaria para aprovechar al máximo el reuso. Si el bucle interno del código de ejemplo contuviera una referencia a[i] y una referencia a[i+5], para sacar partido del reuso entre iteraciones de i necesitaríamos que el valor accedido en a[i+5] se mantuviera en la cache, al menos, durante las cinco iteraciones posteriores. La siguiente fórmula, donde d representa esta distancia, calcula la cantidad de cache necesaria, a la que llamamos *requirements*:

$$requirements = F_1 + ND \times (d - 1) \tag{4.2}$$

donde d es el valor máximo de la diferencia $(Max_i - Min_i)$ entre todos los *reference groups*.

A continuación se calcula el número de misses. El coste de los fallos no es el mismo si el dato que vamos a reemplazar debe propagarse hacia niveles superiores de la jerarquía (dirty miss) que si éste puede ser sobrescrito (clean miss). Por este motivo el compilador intenta estimar qué cantidad habrá de cada tipo. Durante el cálculo de los footprints se mantiene por separado el footprint acumulado de los reference groups que contienen escrituras y de los que sólo contienen lecturas. La presencia de escrituras es lo que puede provocar que se produzcan dirty misses. La cantidad de dirty misses respecto al total se estima

aplicando la misma proporción del footprint con escrituras respecto al footprint total.

Para calcular el número de miss bytes por carga, la relación de lecturas y escrituras respecto al total se calcula de forma ponderada dando más peso a lo que ocurra en el bucle interno:

$$frac\_writes = \frac{WF_0 + var_0 \times WF_1 + var_0 \times var_1 \times WF_2 + ...}{F_0 + var_0 \times F_1 + var_0 \times var_1 \times F_2 + ...}$$

donde:

> WF_i indica el footprint acumulado, para el bucle i, de los reference groups que contienen escrituras.
>
> $var_0, var_1, etc.$ indican los factores de bloqueo de los bucles internos.

El número de dirty miss bytes debidos a carga por iteración del middle loop (bucle más externo del bloque) es:

$$reuse\_lost\_wbytes = \frac{F_0}{v1} \times frac\_writes \tag{4.3}$$

El número de clean miss bytes se calcula de forma análoga.

El siguiente paso consiste en calcular los fallos por conflicto y capacidad. Teniendo en cuenta el tamaño de la cache y su grado de asociatividad, el compilador calcula el tamaño de cache efectivo. Si la cantidad de cache que vamos a utilizar es inferior o igual al tamaño de cache efectivo podemos considerar que los fallos por conflicto serán mínimos. En la práctica esto se traduce en que el compilador utiliza dos factores distintos para calcular la cantidad de reuso perdido: un factor que se aplica cuando los requerimientos no superan el tamaño efectivo (*basic_middle_penalty*) y otro factor que se aplica al superarse este tamaño (*additional_middle_penalty*). Estos dos factores son constantes del programa cuyos valores son 0,20 y 1,00 respectivamente.

Para cada bucle de la anidación (exceptuando el más interno) calculamos la cantidad de reuso que se pierde debido a ese bucle. El resultado se expresa en bytes por iteración del middle loop, por tanto, para los bucles internos hay que multiplicar por el número de veces que se ejecutan.

Dado un bucle i, el reuso entre iteraciones del bucle se calcula como:

$$reuse_i = MAX(F_{i+1} - ND_i, cls) \tag{4.4}$$

A continuación obtenemos los factores que se van a aplicar para calcular la cantidad de reuso que se pierde:

$$frac\_reuse\_lost1_i = \frac{requirements_i \times basic\_middle\_penalty}{size}$$

$$frac\_reuse\_lost2_i = \begin{cases} \frac{(requirements_i - effective\_size) \times additional\_middle\_penalty}{size} \\ 0, \text{ si } requirements_i \leq effective\_size \end{cases}$$

donde $size$ es el tamaño de la cache. Finalmente aplicamos estos factores al reuso:

$$reuse\_lost_i = (frac\_reuse\_lost1_i + frac\_reuse\_lost2_i) \times reuse_i \tag{4.5}$$

Igual que antes, necesitamos dividir estos miss bytes entre dirty y clean misses. En el caso de los dirty misses:

$$reuse\_lost\_wbytes_i = reuse\_lost_i \times \frac{WF_i}{F_i} \tag{4.6}$$

De forma análoga se obtendría el número de clean miss bytes. El número de miss bytes total se obtiene sumando los miss bytes debidos a carga más los miss bytes que se pierden entre iteraciones de cada uno de los bucles. Los clean y dirty misses se acumulan de forma separada.

Antes de continuar, se realiza un cambio de unidades. Se pasa de número de miss bytes por iteración del middle loop a número de miss bytes por iteración del bucle interno del bloque. Para ello dividimos los clean y dirty miss bytes por el producto del número de iteraciones de los bucles internos.

4.5. Cálculo del tamaño de bloque

Partiendo de los footprints, el compilador construye una fórmula que devuelve el coste debido a la cache. Dicha fórmula incorpora el coste por fallos en cache, el coste por fallos en el TLB y el coste por el overhead introducido por el tiling. Al igual que el footprint, la fórmula del coste se expresa de forma paramétrica en función de los tamaños de bloque. El objetivo del compilador es encontrar unos tamaños de bloque que minimicen el valor de esta función.

4.5.1. Fórmula del coste

Podemos distinguir tres partes en la fórmula: cache, tlb y loop overhead. El coste final, que se expresa en ciclos por iteración del bucle interno original, se obtiene sumando la aportación de cada uno de estos elementos:

$$coste = Cache\_cpi + Tlb\_cpi + Loop\_Overhead \tag{4.7}$$

Acerca de las unidades

La fórmula del coste utilizada para comparar entre los distintos bloqueos se expresa en ciclos por iteración del bucle interno original. De momento disponemos de una estimación del número de miss bytes por iteración del bucle interno del bloque en curso. La fórmula del coste se construye tomando como referencia el bucle interno del bloque en curso. Sólo al final, antes de empezar a evaluar para distintos factores de bloqueo, se traducen las unidades a ciclos por iteración del bucle interno original. Por tanto, necesitamos conocer el número de iteraciones del bucle interno original que se ejecutarán por cada iteración del bucle interno del bloque en curso. Para calcular este valor, que se denomina *iters_inside*, se debe tener en cuenta los factores de desenrosque y de bloqueo que se aplicarán a cada uno de los bucles.

Veremos como se calcula *iters_inside* mediante un ejemplo. Supongamos que se va a bloquear la siguiente anidación para las caches de primero y segundo nivel (L1 y L2):

```
do i=1,L,2
  do j=1,M
    do k=1,N
```

Debido al desenrosque del bucle i, el número de iteraciones del bucle k antes de desenrollar por iteración de k tras la transformación es igual a 2. Si ahora aplicamos tiling a los bucles $\{j, k\}$ obtendremos algo como:

```
do i=1,L,2
  do k'=1,N,B
    do j=1,M
      do k=k',k'+B
```

Si deseamos continuar bloqueando para L2, los bucles a considerar serán i, k' y j. El bucle interno del bloque para L2 será j y el número de iteraciones del bucle original k por iteración del bucle j será $2 \times B$.

Cache cpi

Este término de la fórmula calcula el coste debido a los fallos en cache. De momento, supongamos que el cuerpo de los bucles sólo contiene loads y que el número de bytes por iteración del bucle interno que darán miss es *rcache_sreg*. En primer lugar pasamos de bytes a ciclos:

$$ccpi\_hidable\_sreg = rcache\_sreg \times cache\_clean\_cpb$$

donde *cache_clean_cpb* indica el coste de un acceso fallido en ciclos por byte. *cache_clean_cpb* se obtiene dividiendo el miss penalty por el tamaño de línea.

Si la cache soporta varias lecturas pendientes, sus tiempos de espera se van a solapar. En consecuencia, dividimos el valor anterior por el factor *Typical Outstanding Loads* que se obtiene de la descripción de la jerarquía de memoria:

$$ccpi\_hidable\_sreg = \frac{ccpi\_hidable\_sreg}{Typical\_Outstanding}$$

En la descripción de la jerarquía de memoria se definen los parámetros *Load_Op_Overlap_1* y *Load_Op_Overlap_2*. El primero indica qué fracción del primer ciclo de miss se podrá solapar con otras operaciones. El segundo indica qué fracción del último ciclo podremos solapar. En los ciclos intermedios esta fracción decrece de forma lineal. Puesto que deseamos calcular el número de ciclos que no se podrán solapar tomamos los complementarios:

$$counts0 = 1{,}0 - Load\_Op\_Overlap1$$
$$countsm = 1{,}0 - Load\_Op\_Overlap2$$

Aplicando los factores anteriores a cada uno de los ciclos indicados por *ccpi_hidable_sreg* se obtiene una sucesión aritmética, la suma de cuyos términos podemos simplificar y aproximar resultando en la siguiente expresión:

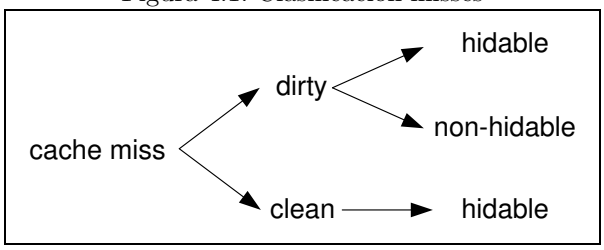
$$\text{ciclos no solapables} = \left(counts0 + \frac{countsm - counts0}{2} \right) \times ccpi\_hidable\_sreg$$

Finalmente, se añade un factor que relaciona *ccpi_hidable_sreg* con el número de ciclos necesarios para ejecutar una iteración del bucle interno, que se obtiene del modelo del procesador. Al final, el número de ciclos que añadiremos por fallos en cache es:

$$\text{ciclos} = \left(counts0 + \frac{ccpi\_hidable\_sreg}{mci} \times \frac{cdiff}{2} \right) \times ccpi\_hidable\_sreg \quad (4.8)$$

donde:

Figura 4.1: Clasificación misses



$cdiff = countsm - counts0$.

$mci = machine\_cycles \times iters\_inside$, es el número de ciclos del modelo del procesador (que se expresa en ciclos por iteración del bucle interno original) pasado a ciclos por iteración del bucle interno del bloque en curso.

La fórmula anterior sólo es válida cuando $mci > ccpi\_hidable\_sreg$. En caso contrario, aplicaremos la siguiente:

$$\text{ciclos} = \frac{counts0 + countsm}{2} \times mci + ccpi\_hidable\_sreg - mci \tag{4.9}$$

Hasta este momento hemos estado considerando el caso en que el bucle solamente contiene lecturas. Puesto que no hay escrituras que puedan modificar las líneas de cache, esto implica que todos los fallos serán clean miss. Además, se asume que el cien por cien de los ciclos de lectura son potencialmente solapables con otras operaciones (ver figura 4.1).

Cuando se calcula el footprint de un bucle, la aportación de los reference groups que contienen escrituras y las de los que sólo contienen lecturas se acumulan de forma separada. Durante el cálculo del coste debido a fallos en cache, la proporción del footprint con escrituras respecto al total se utiliza, aplicada al número de misses, para estimar la cantidad de dirty misses.

Los dirty misses (ver figura 4.1) provocarán accesos a la cache de coste superior, y se asume que un porcentaje fijo de los ciclos adicionales no serán solapables. Se define $cache\_dirty\_cpb\_nonhidable$ ($cdcn$ en la fórmula) como:

$$cdcn = (cache\_dirty\_cpb - cache\_clean\_cpb) \times \frac{Pct\_Excess\_Writes\_Nonhidable}{100{,}0}$$

donde:

$cache\_dirty\_cpb$ es el coste en ciclos por byte de un dirty miss.

$cache\_clean\_cpb$ es el coste en ciclos por byte de un clean miss.

$Pct\_Excess\_Writes\_Nonhidable$ es el porcentaje de la diferencia que se considera no solapable. Este dato se extrae de la descripción de la cache.

Sea $wcache\_sreg$, el número de bytes por iteración del bucle interno que provocarán un dirty miss, podemos calcular el siguiente término que sumaremos directamente al coste:

$$ccpi\_nonhidable\_sreg = \frac{wcache\_sreg \times cache\_dirty\_cpb\_nonhidable}{Typical\_Outstanding} \tag{4.10}$$

Para terminar tenemos que actualizar las equaciones 4.8 y 4.9 para contabilizar también los ciclos potencialmente solapables debidos a dirty misses. Tan solo hay que redefinir el cómputo de *ccpi_hidable_sreg* (*ccpi_hs* en la fórmula):

$$ccpi\_hs = \frac{rcache\_sreg \times cache\_clean\_cpb + wcache\_sreg \times cache\_dirty\_cpb\_hidable}{Typical\_Outstanding}$$

donde $cache\_dirty\_cpb\_hidable = cache\_dirty\_cpb - cache\_dirty\_cpb\_nonhidable$.

TLB cpi

El TLB es tratado como una cache completamente asociativa con un tamaño de línea muy grande. Las rutinas utilizadas para calcular el número de ciclos del coste debidos a fallos en el TLB son las mismas que se utilizan para la cache. Como en el caso de la cache, los pasos son:

1. Cálculo del footprint.
2. Estimación del número de miss bytes.
3. Cálculo del número de ciclos.

Una vez se dispone del número de miss bytes el cálculo del número de ciclos por fallos en el TLB se obtiene aplicando la siguiente fórmula:

$$tlb\_cpi = \begin{array}{c} rtlb\_sreg \times tlb\_clean\_cpb \times \frac{TLB\_Trustworthiness}{100} \\ + \\ wtlb\_sreg \times tlb\_dirty\_cpb \times \frac{TLB\_Trustworthiness}{100} \end{array}$$

donde:

rtlb_sreg, *wtlb_sreg* son el número de clean y dirty miss bytes respectivamente,

tlb_clean_cpb, *tlb_dirty_cpb* son la penalización por clean y dirty miss en número de ciclos por byte,

y $TLB\_Trustworthiness$ es un parámetro que se define en la descripción de la jerarquía de memoria que indica el grado de confianza en los resultados del TLB.

La falta de confianza en los resultados del TLB se debe a la limitada capacidad del modelo para determinar el reuso de la línea de cache para tamaños muy grandes de ésta. En una situación donde se defina un array almacenado por columnas como a(5, 5, 5, 100) el compilador no detectará reuso entre las referencias a(0,0,0,0) y a(0,0,0,1).

En el caso del TLB *basic_middle_penalty* y *additional_middle_penalty* se establecen en 0 y 0.20 respectivamente. Estas dos variables son los factores que se aplican para estimar la cantidad de reuso perdida entre iteraciones de un bucle (ver sección 4.4.3). En el caso del TLB sólo se penaliza cuando la cantidad de memoria requerida supera el tamaño del TLB.

Loop Overhead

Al bloquear una anidación de bucles la reducción en el número de iteraciones de los bucles internos provoca un incremento del coste asociado a su inicialización y finalización (preparación de las direcciones base, pipelining y copia de registros). El término *Loop_Overhead* refleja este incremento en la fórmula del coste.

El modelo de la cache incluye dos cálculos del coste asociado a la inicialización y finalización de los bucles. El primero de ellos es el que se añade como término a la fórmula junto con los ciclos por fallos en la cache y el tlb. Se expresa en función de los factores de bloqueo y, sumado al resto de términos, servirá para poder comparar y escoger estos factores.

El segundo cálculo del loop overhead lo realiza Compute_Do_Overhead(). Una vez determinado el bloqueo para un nivel de cache, el loop overhead calculado en la fórmula del coste se descarta y su valor se sustituye por el resultado de esta función. El coste total calculado por el modelo de la cache, se obtiene sumando el coste por fallos de cada uno de los niveles de cache más el loop overhead calculado por Compute_Do_Overhead(). Este coste no se calcula nivel a nivel sino que tiene en cuenta el conjunto de transformaciones para todos los niveles de la jerarquía de memoria (ver la sección 4.6).

En ambos casos, el punto de partida para el cálculo de este coste es la definición de un coste único para todos los bucles. Se define la variable *Loop_Overhead* como:

$$Loop\_Overhead = Loop\_Overhead\_Base + memrefs \times Loop\_Overhead\_Memref$$

donde:

Loop_Overhead_Base es un coste base pre-establecido.

$Loop\_Overhead\_Memref$ es el coste de cargar un dato en un registro.

$memrefs$ es el número de accesos a memoria del bucle interno tras la eliminación de subexpresiones comunes.

El término que incluimos en la fórmula del coste se calcula sumando el loop overhead de cada bucle expresado en ciclos por iteración del bucle interno:

$$\frac{\frac{\frac{\frac{Loop\_Overhead}{v1} + Loop\_Overhead}{var_0} + Loop\_Overhead}{var_1} + Loop\_Overhead}{\vdots}$$
$$var_n$$

donde:

$v1$ es el número de iteraciones del middle loop.

$var_0, var_1, ..., var_n$ son los factores de bloqueo de los bucles internos.

Como veremos en la sección 4.6, la función Compute_Do_Overhead() realiza el cálculo de forma similar.

4.5.2. Búsqueda de los tamaños de bloque

La búsqueda de los mejores factores de bloqueo se implementa mediante las funciones RSolve(), Rtry() y RSolve3() (be/lno/cache_model.h), que reciben un objeto de tipo FORMULA. La primera de ellas funciona del siguiente modo:

Sea b el factor de bloqueo cuyo valor queremos determinar, empezaremos por calcular el coste debido a la cache cuando $b = x$ y cuando $b = 2 \times x$.

1. Si $coste(x) \leq coste(2 \times x)$, entonces calculamos de forma iterativa el coste para $b = \frac{x}{2}, \frac{x}{4}, \ldots$ hasta que:
   a) encontramos un valor $\frac{x}{n}$ t.q. $coste(\frac{x}{n}) > coste(\frac{x}{n-1})$, cosa que indicaría que el valor óptimo de b se encuentra entre $\frac{x}{n}$ y $\frac{x}{n-2}$,
   b) o llegamos al valor mínimo permitido.
2. Si, por contra, $coste(x) > coste(2 \times x)$, entonces realizamos el cálculo para $b = 4 \times x, 8 \times x, \ldots$ hasta que:
   a) encontramos un valor $n \times x$ t.q. $coste(n \times x) > coste((n-1) \times x)$, cosa que indicaría que el valor óptimo de b se encuentra entre $n \times x$ y $(n-2) \times x$,
   b) o llegamos al valor máximo permitido.

Tanto si acabamos en la rama $1a$ como en la $2a$, llegamos a la situación que se plantea en la figura 4.2. A partir de ahí se trata de ir acotando la búsqueda hasta encontrar un mínimo local. De esta tarea se encarga el método RSolve3() que procede como sigue:

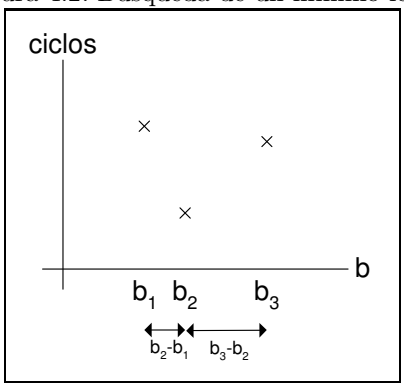
1. Compara los segmentos (b_1, b_2) y (b_2, b_3) y toma un punto x en la mitad del segmento más largo.
2. Calcula el coste para $b = x$.
3. Comprueba si el resultado es un nuevo mínimo de la función coste.
4. Realiza una llamada recursiva pasando el mejor tamaño de bloque encontrado hasta el momento y los puntos adyacentes.

Hasta este momento se ha planteado el problema para una sola incógnita b, pero qué pasa cuando hay que calcular más factores de bloqueo? La respuesta está en la función Rtry(), que es el método que evalúa la función de coste para un determinado valor de b. Al llamar a esta función se está eliminando una incógnita, si hay más valores por determinar simplemente se llama a RSolve() desde Rtry() para que repita el proceso que acabamos de describir pero para el siguiente factor de bloqueo.

Para reducir el tiempo de búsqueda se establece un número máximo de valores distintos para los factores de bloqueo (Max_Different_Blocksizes). De este modo, si bloqueamos n bucles y $n \geq Max\_Different\_Blocksizes$, los $n - Max\_Different\_Blocksizes + 1$ bucles externos compartirán el mismo factor de bloqueo.

Respecto a los valores que van a tomar los factores de bloqueo cabe comentar que éstos serán múltiple de los factores de desenrosque. De hecho el cálculo se realiza sin tener en cuenta los factores de desenrosque y al final se aplica el

Figura 4.2: Búsqueda de un mínimo local



producto de ambos. Esto significa que antes de empezar a calcular los factores de bloqueo tenemos que dividir el número de iteraciones estimadas de cada bucle por su factor de desenrosque.

4.6. Resultado del modelo de la cache

En esta sección describimos los parámetros de la función Cache_Model() que se utilizan para devolver el resultado del modelo de la cache. Se acompaña cada parámetro con los resultados obtenidos para el siguiente código:

```
DO K=1,2000
  DO I=1,2000
    DO J=1,2000
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

tras probar las distintas posibilidades de bucle interno y buscar los mejores factores de bloqueo para cada caso, los resultados obtenidos son:

new_order ({1,2,0}): Indica la permutación a aplicar a los bucles de la anidación. Se identifica cada bucle por su posición en la anidación inicial. En el ejemplo se obtiene la permutación {1,2,0}, que indica que el bucle k pasa a ser el bucle interno.

nstrips (4): Número de bucles de control (implementan el recorrido de los bloques). Se generan cuatro bucles de control.

stripdepth (0): En qué nivel de profundidad (después de reordenar) se colocarán los bucles de control. En el ejemplo se colocan como bucles más externos.

iloop ({2, 0, 1, 2}): Para cada bucle de control, indica a partir de qué bucle se obtiene. Como estos indices hacen referencia a la posición después de reordenar, los bucles de control se obtienen a partir de $\{k, i, j, k\}$.

stripsz ({84, 105, 12, 42}): Factores de bloqueo para cada uno de los bucles que aparecen en *iloop*.

striplevel ({2, 2, 1, 1}): A qué nivel de cache corresponde el bloqueo. Los bucles de control {2, 0} y {1, 2} de la lista *iloop* provienen de bloquear para L2 y L1 respectivamente.

cycles_per_iter: Ciclos por iteración del bucle interno original debidos a fallos en cache/tlb y loop overhead.

doverhead_best: Ciclos por iteración debidos a loop overhead.

El resultado de la aplicación de estas transformaciones al código inicial es el siguiente[2]:

```
DO tile2K__0 = 1, 2000, 84
  DO tile2I__0 = 1, 2000, 105
    DO tile1J__0 = 1, 2000, 12
      DO tile1K__0 = tile2K__0, MIN((tile2K__0 + 83), 2000), 42
        DO I = tile2I__0, MIN((tile2I__0 + 104), 2000), 1
          DO J = tile1J__0, MIN((tile1J__0 + 11), 2000), 1
            mi__0 = C(I, J)
            DO K = tile1K__0, MIN(MIN((tile2K__0 + 83),
                                      (tile1K__0 + 41)), 2000), 1
              mi__0 = (mi__0 +(A(I, K) * B(K, J)))
            END DO
            C(I, J) = mi__0
          END DO
        END DO
      END DO
    END DO
  END DO
END DO
```

El coste por fallos en cache/tlb es la suma del coste para cada uno de los niveles de cache. El coste de cada nivel se obtiene evaluando la fórmula para unos factores de bloqueo determinados. Sin embargo, una vez determinados los factores de bloqueo, el término correspondiente al loop overhead de la fórmula se descarta y, en su lugar, el coste devuelto es el calculado por la función Compute_Do_Overhead().

4.6.1. Cálculo del overhead asociado a los bucles

La función Compute_Do_Overhead() calcula el overhead debido a la inicialización de los bucles. Se asume un coste fijo (*Loop_Overhead*) para cada bucle que se reparte entre todas sus iteraciones. Este coste se expresa en ciclos por iteración del bucle interno original. Se obtiene el total sumando el coste asociado a cada uno de los bucles:

$$\sum_{\forall bucle} \frac{Loop\_Overhead}{\text{iteraciones de todos los bucles internos}}$$

[2]Este código se ha obtenido con la opción de outer unrolling deshabilitada.

donde (ver sección 4.5.1):

$$Loop\_Overhead = Loop\_Overhead\_Base + memrefs \times Loop\_Overhead\_Memref$$

Dado el siguiente ejemplo:

```
do k'' = 1, N3, B4
  do i'' = 1, N1, B3
    do j' = 1, N2, B2
      do k' = k'', k''+B4-1
        do i = i'', i''+B3-2, 2      (desenrosque 2)
          do j = j', j'+B2-2, 2      (desenrosque 2)
            do k = k', k'+B1-1
```

sea o el valor de *Loop_Overhead*, el resultado de Compute_Do_Overhead() se obtendrá sumando los siguientes términos:

$$\frac{o}{4\times B1} + \frac{o}{4\times B1\times(\frac{B2}{2})} + \frac{o}{4\times B1\times(\frac{B2}{2})\times(\frac{B3}{2})} + \frac{o}{4\times B4\times(\frac{B2}{2})\times(\frac{B3}{2})} + \\ + \frac{o}{4\times B4\times(\frac{N2}{2})\times(\frac{B3}{2})} + \frac{o}{4\times B4\times(\frac{N2}{2})\times(\frac{N1}{2})} + \frac{o}{4\times N3\times(\frac{N2}{2})\times(\frac{N1}{2})}$$

4.7. Descripción de la jerarquía de memoria

La descripción de la jerarquía de memoria se encuentra en el archivo config_cache_targ.cxx. En él se establece el valor de los parámetros que definen cada uno de los distintos niveles que se van a contemplar.

Para cada nivel se define:

Type: Indica si se está describiendo una cache o la memoria principal.

Size: Tamaño en bytes de la cache o memoria.

Line_Size: Número de bytes por línea.

Clean_Miss_Penalty: Número de ciclos necesarios para reemplazar una línea no modificada con datos del siguiente nivel de la jerarquia.

Dirty_Miss_Penalty: Número de ciclos necesarios para reemplazar una línea que ha sido modificada.

Associativity: Asociatividad de la cache.

TLB_Entries: Número de entradas del TLB. -1 si no hay TLB.

Page_Size: Cuantos bytes son mapeados por una entrada del TLB.

TLB_Clean_Miss_Penalty, TLB_Dirty_Miss_Penalty: Cuantos ciclos cuesta un fallo en el TLB.

Typical_Outstanding: Cuantos loads pendientes se soportan. Esta valor normalmente será inferior al máximo de la cache. 1 significa que no hay solapamiento.

Load_Op_Overlap_1, Load_Op_Overlap_2: Qué porcentaje de los ciclos de miss se puede solapar con operaciones del procesador. Load_Op_Overlap_1 indica qué cantidad del primer ciclo de miss se solapa, y Load_Op_Overlap_2 qué cantidad del último ciclo potencialmente solapable se solapa. Este porcentaje se reduce de forma lineal en los ciclos intermedios.

Pct_Excess_Writes_Nonhidable: Si restamos el coste de un clean miss del coste de un dirty miss, este valor indica qué porcentaje de la diferencia no será solapable. Estos ciclos no solapables se suman directamente a la fórmula del coste.

Adicionalmente, y no asociados a ningún nivel de la jerarquía en concreto, se definen las siguientes variables:

Non_Blocking_Loads: Cierto, si el procesador no se bloquea cuando se produce un miss.

Loop_Overhead_Base, Loop_Overhead_Memref: Teniendo en cuenta que el coste de inicialización de cada bucle se calcula como $Loop\_Overhead = Loop\_Overhead\_Base + memrefs \times Loop\_Overhead\_Memref$, el primero de estos parámetros es el coste base y el segundo el coste por referencia a memoria. $memrefs$ es el número de referencias después de eliminar subexpresiones comunes.

TLB_Trustworthiness: Toma valores entre 0 y 100 e indica el grado de confianza que se tiene en los resultados del TLB.

TLB_NoBlocking_Model: Cuando utilizamos el modelo para evaluar una situación donde no hay bloqueo, el modelo del TLB tiende a sobreestimar el número de misses. Si esta variable vale cierto, se favorece la opción de no bloquear ajustando a la baja el coste del TLB. Este criterio se aplica incluso cuando confiamos plenamente en los resultados del TLB (TLB_Trustworthiness = 100).

Capítulo 5

Transformación de la representación intermedia

El capítulo 2 presenta los análisis utilizados para seleccionar el subconjunto de bucles de un SNL que se intentará optimizar mediante intercambio, outer unrolling y tiling. Una vez seleccionados los bucles, el algoritmo descrito en el capítulo 3 decide qué transformaciones se van a aplicar.

En este capítulo vamos a ver como se lleva a cabo la transformación de la representación intermedia. Esta tarea se realiza en la parte final de la función Do_Automatic_Transformation() (be/lno/snl_test.cxx), que tras realizar el análisis de legalidad y llamar al modelo, dirige también el proceso de transformación.

Se distinguen claramente dos partes: la primera de ellas, que describimos en la sección 5.1 se encarga de la transformación de anidaciones invariantes, que son aquellas donde los límites de los bucles no dependenden de otros bucles de la anidación y que no contienen código imperfecto o, si lo contienen, éste se puede eliminar mediante distribución. El caso complementario corresponde a las anidaciones generales y su transformación, algo más compleja, se describe en la sección 5.2.

5.1. Transformación de bucles invariantes

La transformación de anidaciones de bucles invariantes se lleva a cabo en cuatro pasos:

1. Expansión de escalares y distribución del código imperfecto.
2. Permutación de los bucles.
3. Bloqueo a nivel de cache.
4. Bloqueo a nivel de registros.

Después del bloqueo a nivel de cache se lleva a cabo el hoisting de expresiones invariantes respecto a bucles externos. Este proceso se repite, después del bloqueo a nivel de registros, aplicado a las expresiones invariantes respecto al bucle interno.

A estos pasos podríamos añadir la aplicación de index set splitting en algunos casos y la eliminación de bucles con una o cero iteraciones. Estas dos transformaciones, que se aplican cuando ya se han realizado todas las transformaciones enumeradas anteriormente, las comentamos brevemente en la secciones 5.1.5 y 5.1.6 respectivamente.

La función SNL_INV_Transforms() (/be/lno/snl_inv.cxx) implementa estos cuatro pasos. Se distinguen tres casos:

1. La aplicación de tiling afecta a todos los bucles (del subconjunto de bucles totalmente permutables). En este caso, todos los pasos anteriores se realizan en una sola llamada a SNL_INV_Transforms().

2. El tiling no afecta a todos los bucles y no habrá bloqueo a nivel de registros. Cuando esto ocurra habrá una primera llamada a SNL_INV_Transforms() donde se aplicará la permutación de bucles y una segunda llamada donde se realizará el bloqueo a nivel de cache.

3. Por último, cuando el tiling no afecta a todos los bucles y sí hay bloqueo a nivel de registros, las transformaciones se llevan a cabo en dos llamadas. En la primera se reordenan los bucles y se aplica outer unroll a aquellos bucles que no participan en el tiling a nivel de cache (estos bucles serán externos respecto a los bucles de control introducidos por el tiling). En la segunda llamada se aplica tiling y outer unroll al resto de bucles.
   La aplicación de outer unroll en la primera llamada puede dar lugar a la aparición de nuevas anidaciones debido a iteraciones "sobrantes". A estas anidaciones también se les aplica tiling y outer unroll mediante llamadas adicionales a SNL_INV_Transforms().

A continuación veremos cada uno de los pasos de la transformación sobre el siguiente ejemplo de multiplicación de matrices, donde añadimos código imperfecto para que sea necesario aplicar distribución y expansión de escalares:

```
DO K=1,2555
  t = t + K
  DO I=1,2555
    DO J=1,2555
      C(I,J) = C(I,J) + A(I,K) * B(K,J) + t
    ENDDO
  ENDDO
ENDDO
```

Tras evaluar las distintas posibilidades que contempla el algoritmo de optimización, se resuelve que se aplicarán las siguientes transformaciones:

permutación:	tiling:
{2, 1, 0}	$\_iloop = \{2, 1, 2\}$
	$\_striplevel = \{2, 2, 1\}$
unroll:	$\_stripsz = \{56, 136, 28\}$
{4, 4, 1}	

- La permutación se especifica mediante una lista que indica qué posición ocupará cada uno de los bucles. Cada bucle se representa mediante un número que corresponde a su posición en el código original. En el ejemplo 2,1 y 0 hacen referencia a los bucles J,I y K respectivamente.

- El orden de la lista de factores de unroll corresponde al orden de los bucles después de aplicar la permutación. En este caso el factor de unroll es 4 para los bucles J e I, y 1 para el bucle K.

- En cuanto a la descripción del tiling tenemos, en primer lugar, una lista *_iloop* que indica a partir de qué bucles (después de reordenar) se generarán los bucles de control. *_stripsz* indica los factores de bloqueo y *_striplevel* indica para qué nivel de cache se realiza el bloqueo.

5.1.1. Expansión de escalares y distribución

El primer paso de la transformación del código consiste en la expansión de escalares. Para saber qué escalares es necesario expandir, el compilador utiliza la información recopilada en la sección 2.2.1.

En el ejemplo, se va a expandir la variable t. Se crea un nuevo array cuya primera posición se inicializa con el valor de t, se sustituye la definición y los usos de t por accesos a este array y, por último, se asigna a t su valor final después de la ejecución de los bucles.

```
deref_se1_F__0(1) = T
DO K = 1, 2555, 1
  deref_se1_F__0(K + 1) = (REAL(K) + deref_se1_F__0(K))
  DO I = 1, 2555, 1
    DO J = 1, 2555, 1
      C(I, J) = ((C(I, J) +(A(I, K) * B(K, J))) + deref_se1_F__0(K + 1))
    END DO
  END DO
END DO
IF(1 .LE. 2555) THEN
  T = deref_se1_F__0(2556)
ENDIF
```

A continuación se distribuye el código imperfecto. De este modo se obtiene una anidación perfectamente imbricada donde aplicaremos el resto de transformaciones.

```
deref_se1_F__0(1) = T
DO K = 1, 2555, 1
  deref_se1_F__0(K + 1) = (REAL(K) + deref_se1_F__0(K))
END DO
DO K = 1, 2555, 1
  DO I = 1, 2555, 1
    DO J = 1, 2555, 1
      C(I, J) = ((C(I, J) +(A(I, K) * B(K, J))) + deref_se1_F__0(K + 1))
    END DO
  END DO
END DO
IF(1 .LE. 2555) THEN
  T = deref_se1_F__0(2556)
ENDIF
```

5.1.2. Permutación

Para aplicar la permutación se genera una lista que aisla la representación WHIRL de cada uno de los bucles que vamos a reordenar. Se eliminan los bucles del código y se vuelven a colocar en el nuevo orden. Esta reordenación se debe reflejar también en la representación de las dependencias y algunos de los atributos asociados a los bucles, como por ejemplo su profundidad.

Después de realizar la permutación el código queda de la siguiente manera:

```
DO J = 1, 2555, 1
  DO I = 1, 2555, 1
    DO K = 1, 2555, 1
      C(I, J) = ((C(I, J) +(A(I, K) * B(K, J))) + deref_se1_F__0(K + 1))
    END DO
  END DO
END DO
```

5.1.3. Bloqueo a nivel de cache

Para cada uno de los bucles de control que se van a añadir, el compilador realiza los siguientes pasos:

1. Crea una nueva variable índice.

2. Genera la representación en WHIRL de los límites inferior y superior, copiándolos del bucle original, y del incremento del bucle utilizando el tamaño de bloque.

3. Construye un nuevo bucle con los elementos anteriores y lo añade al código original. Este bucle se coloca como bucle immediatamente superior al bucle más externo del subconjunto que estamos transformando. Los bucles de control se añaden empezando por el más externo.

4. Por último se modifica el bucle original para que su límite inferior sea el valor de la variable de iteración del bucle de control, y se genera la expresión del límite superior en función de la nueva variable y el tamaño de bloque. Si el tamaño de bloque no es divisor del número de iteraciones inicial, añadimos la función MIN.

A medida que se realizan estos cambios en la representación del programa, también se va actualizando la información sobre las cadenas de definición y uso, dependencias, etc.

Aplicando estas modificaciones al ejemplo obtenemos el siguiente resultado:

```
DO tile2K__0 = 1, 2555, 56
  DO tile2I__0 = 1, 2555, 136
    DO tile1K__0 = tile2K__0, MIN((tile2K__0 + 55), 2555), 28
      DO J = 1, 2555, 1
        DO I = tile2I__0, MIN((tile2I__0 + 135), 2555), 1
          DO K = tile1K__0, MIN(MIN((tile2K__0 + 55), (tile1K__0 + 27)), 2555), 1
            C(I, J) = ((C(I, J) +(A(I, K) * B(K, J))) + deref_se1_F__0(K + 1))
          END DO
        END DO
      END DO
```

```
    END DO
  END DO
END DO
```

5.1.4. Bloqueo a nivel de registros

Tras el bloqueo a nivel de cache se lleva a cabo el bloqueo a nivel de registros. Para cada uno de los bucles iniciales cuyo factor de bloqueo sea mayor que uno:

1. Se comprueba si el factor de bloqueo es divisor del número de iteraciones.
2. Si no se cumple la condición anterior se genera una copia del bucle. Se sustituye la variable de iteración en el segundo bucle por una nueva variable y se modifica su límite inferior para que tome el valor final de la variable de iteración del bucle original. Estas tareas se llevan a cabo en la función Wind_Down().
   Después de generar el bucle que ejecutará las iteraciones sobrantes, se modifica el límite superior del bucle original restando $u - 1$, donde u es el factor de bloqueo.
3. Se crean las copias adicionales del cuerpo del bucle original.
4. Se modifican las expresiones donde aparece el índice añadiendo el incremento correspondiente.
5. Por último se fusionan todas las copias.

El bloqueo a nivel de registros también se aplica dentro del bucle que ejecuta las iteraciones sobrantes.

El código resultante de la aplicación del bloqueo a nivel de registros en el ejemplo es el siguiente:

```
DO tile2K__0 = 1, 2555, 56
  DO tile2I__0 = 1, 2555, 136
    DO tile1K__0 = tile2K__0, MIN((tile2K__0 + 55), 2555), 28
      DO J = 1, 2552, 4
        DO I = tile2I__0, MIN((tile2I__0 + 132), 2552), 4
          DO K = tile1K__0, MIN(MIN((tile2K__0 + 55), (tile1K__0 + 27)), 2555), 1
            C(I,J) = C(I,J) + A(I,K) * B(K,J) + deref_se1_F__0(K+1)
            C(I,J+1) = deref_se1_F__0(K+1) + C(I,J+1) + A(I,K) * B(K,J+1)
            C(I,J+2) = deref_se1_F__0(K+1) + C(I,J+2) + A(I,K) * B(K,J+2)
            C(I,J+3) = deref_se1_F__0(K+1) + C(I,J+3) + A(I,K) * B(K,J+3)
                                ⋮
            C(I+3,J) = deref_se1_F__0(K+1) + C(I+3,J) + A(I+3,K) * B(K,J)
            C(I+3,J+1) = deref_se1_F__0(K+1) + C(I+3,J+1) + A(I+3,K) * B(K,J+1)
            C(I+3,J+2) = deref_se1_F__0(K+1) + C(I+3,J+2) + A(I+3,K) * B(K,J+2)
            C(I+3,J+3) = deref_se1_F__0(K+1) + C(I+3,J+3) + A(I+3,K) * B(K,J+3)
          END DO
        END DO
        DO wd_I__1 = I, MIN((tile2I__0 + 135), 2555), 1
          DO K = tile1K__0, MIN(MIN((tile2K__0 + 55), (tile1K__0 + 27)), 2555), 1
            C(wd_I__1,J) = C(wd_I__1,J) + A(wd_I__1,K) * B(K,J) + deref_se1_F__0(K+1)
            C(wd_I__1,J+1) = deref_se1_F__0(K+1) + C(wd_I__1,J+1) + A(wd_I__1,K) * B(K,J+1)
            C(wd_I__1,J+2) = deref_se1_F__0(K+1) + C(wd_I__1,J+2) + A(wd_I__1,K) * B(K,J+2)
```

```
            C(wd_I__1,J+3) = deref_se1_F__0(K+1) + C(wd_I__1,J+3) + A(wd_I__1,K) * B(K,J+3)
          END DO
        END DO
      END DO
      DO wd_J__0 = 2553, 2555, 1
        DO I = tile2I__0, MIN((tile2I__0 + 132), 2552), 4
          DO K = tile1K__0, MIN(MIN((tile2K__0 + 55), (tile1K__0 + 27)), 2555), 1
            C(I,wd_J__0) = C(I,wd_J__0) + A(I,K) * B(K,wd_J__0) + deref_se1_F__0(K+1)
            C(I+1,wd_J__0) = deref_se1_F__0(K+1) + C(I+1,wd_J__0) + A(I+1,K) * B(K,wd_J__0)
            C(I+2,wd_J__0) = deref_se1_F__0(K+1) + C(I+2,wd_J__0) + A(I+2,K) * B(K,wd_J__0)
            C(I+3,wd_J__0) = deref_se1_F__0(K+1) + C(I+3,wd_J__0) + A(I+3,K) * B(K,wd_J__0)
          END DO
        END DO
        DO wd_I__0 = I, MIN((tile2I__0 + 135), 2555), 1
          DO K = tile1K__0, MIN(MIN((tile2K__0 + 55), (tile1K__0 + 27)), 2555), 1
            C(wd_I__0,wd_J__0) = C(wd_I__0, wd_J__0) + A(wd_I__0,K) * B(K,wd_J__0) +
                                 deref_se1_F__0(K+1)
          END DO
        END DO
      END DO
    END DO
  END DO
END DO
```

5.1.5. Split inner tile loops

Esta transformación se aplica al bucle interno cuando la anidación tiene la siguiente forma:

```
DO tile_I = lb, ub, tile_size
  DO I = tile_I, MIN(tile_I + tile_size, ub), 1
    ...
  END DO
END DO
```

Consiste en eliminar la función MIN creando una réplica de la anidación y modificando los límites del bucle externo:

```
DO tile_I = lb, ub - (tile_size - 1), tile_size
  DO I = tile_I, tile_I + tile_size, 1
    ...
  END DO
END DO
DO tile_I_new = tile_I, ub, tile_size
  DO I = tile_I_new, ub, 1
    ...
  END DO
END DO
```

5.1.6. Remove useless loops

Si en tiempo de compilación se detecta que se ejecutará una única iteración de un bucle se elimina la estructura de control que lo conforma dejando sólo el código del cuerpo del bucle. Si se detecta que no se va a ejecutar ninguna iteración, el bucle se elimina completamente.

5.2. Transformación de bucles generales

Cuando la anidación que estamos tratando contiene bucles cuyos índices dependen de otros bucles de la anidación, o cuando ésta contiene código imperfecto que no ha sido posible distribuir, nos encontramos en el caso general. Como se ha visto en la sección 2.5 esto implica algunas limitaciones en las transformaciones que se van a llevar a cabo. En concreto, considerando como último bucle el más interno, sólo se aplicará outer unrolling al penúltimo bucle de la anidación y, en el mejor de los casos (cuando no hay código imperfecto), el intercambio se limitará a los dos bucles más internos.

Una vez decidido el conjunto de transformaciones a aplicar, los pasos seguidos en la transformación de una anidación general son:

1. Reordenación de los bucles de control para evitar intercambios implícitos.
2. Creación, si procede, de una copia de la anidación. Ambas anidaciones (original y copia) se protegeran con un condicional de modo que en una de las ramas se garantizará al menos una iteración del bucle más interno.
3. Expansión de escalares y distribución del código imperfecto.
4. Intercambio.
5. Bloqueo a nivel de cache.
6. Bloqueo a nivel de registros.

Al igual que en el caso invariante, tras estas transformaciones, se lleva a cabo la aplicación de index set splitting (en algunos casos) y la eliminación de bucles con una o cero iteraciones. Estas transformaciones se realizan de forma idéntica en ambos casos (ver 5.1.5 y 5.1.6).

5.2.1. Reordenación de los bucles de control

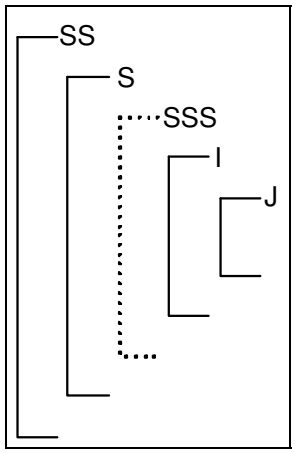
Cuando se aplica bloqueo para un solo nivel de cache los bucles de control siguen el mismo orden que los bucles originales. Este hecho, que no tiene importancia si estamos tratando una anidación invariante, en el caso general garantiza que no se vayan a producir intercambios implícitos en códigos no perfectamente anidados.

Si el bloqueo se va a realizar para más de un nivel, hay que revisar el orden de los bucles externos para evitar que se pueda producir un intercambio implícito. Por lo tanto, antes de proceder a la aplicación del tiling, y sólo en el caso general, se va a realizar una reordenación de los bucles externos. Sean I, J dos bucles internos tales que $I < J$ (J está a mayor profundidad que I), y sean S, SS dos bucles de control que recorren los bloques de I y J respectivamente, es necesario que se cumpla una de las siguientes condiciones (ver figura 5.1):

- $S < SS$
- o $SS < S$ y existe un bucle SSS que recorre bloques de J t.q. $S < SSS$.

Esta reordenación se lleva a cabo intentado realizar el menor número de cambios posible. Seria absurdo modificar la anidación:

Figura 5.1: Ordenación de los bucles de control



do j" do i' do j' do i do j

para obtener:

do i' do j" do j' do i do j

5.2.2. Condicionales

Si, ante la presencia de una anidación invariante (los límites de los bucles no dependen de otros bucles de la anidación) que contiene código imperfecto no distribuible, no ha sido posible garantizar como mínimo una iteración de cada uno de los bucles, el compilador crea una copia adicional de la anidación y protege ambas anidaciones con un condicional. Esta operación se realiza de modo que en una de las ramas siempre se garantiza al menos una iteración de todos los bucles. La anidación contenida en esta rama es la que se va a transformar. La copia contenida en la otra rama se mantiene sin modificaciones.

5.2.3. Expansión de escalares y distribución

De igual modo que en el caso invariante, en el caso general también se va a llevar a cabo la expansión de escalares y posterior distribución del código imperfecto (ver sección 5.1.1).

5.2.4. Intercambio

En el caso general se permite intercambiar, como máximo, los dos bucles más internos. Ante la presencia de código imperfecto siempre se va a respetar el orden original y, por tanto, nunca será necesario aplicar code sinking. Sin embargo, podemos encontrar una función inacabada (UT_Body_Imperfect() en snl_gen.cxx) que implementa esta transformación. Esta función nuna llega a ejecutarse.

Para ilustrar el funcionamiento de la aplicación de intercambio utilizaremos el siguiente código en Fortran:

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

sea $u = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ la matriz unimodular[12] que expresa el intercambio, los pasos que se van a seguir hasta obtener el código transformado son:

1. Cálculo de la matriz inversa de u, $uinv = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$.

2. Producto de los límites con la matriz inversa $uinv$.

$$\underset{Ale}{\begin{array}{c} K \quad I \quad J \\ \begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix} \end{array}} \leq \underset{Ble}{\begin{bmatrix} -1 \\ 2500 \\ 0 \\ 2500 \\ 0 \\ 2500 \end{bmatrix}} \Rightarrow \underset{Ale \times uinv}{\begin{array}{c} K \quad I \quad J \\ \begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 1 \\ 1 & -1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{array}} \leq \underset{Ble}{\begin{bmatrix} -1 \\ 2500 \\ 0 \\ 2500 \\ 0 \\ 2500 \end{bmatrix}}$$

3. Conversión de los límites a forma canónica:

$$\begin{array}{c} K \quad I \quad J \\ \begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \leq \begin{bmatrix} -1 \\ 2500 \\ 0 \\ 2500 \\ 0 \\ 2500 \end{bmatrix}$$

4. Reescritura de la representación intermedia tras la transformación:

```
DO K__0 = 1, 2500, 1
  DO fake_J__0 = K__0, 2500, 1
    DO fake_I__0 = K__0, 2500, 1
      C__0(fake_J__0, fake_I__0) = (C__0(fake_J__0, fake_I__0) +
        (A__0(fake_J__0, K__0) * B__0(K__0, fake_I__0)))
    END DO
  END DO
END DO
```

5.2.5. Bloqueo a nivel de cache

Según parece, el algoritmo de tiling implementado se basa en el descrito en [3]. Se contempla la posibilidad de aplicación de tiling no rectangular (bloques no

rectangulares), pero comentarios en el código nos informan que esa parte nunca ha llegado a funcionar. Así pues nos quedamos con una versión simplificada que, en todos los casos, generará bloques rectangulares.

La aplicación de tiling a una anidación general se lleva a cabo mediante los siguientes pasos:

1. Se monta el sistema de ecuaciones que describe las restricciones a las que van a estar sometidos todos los bucles del código transformado.

2. Se resuelve el sistema de ecuaciones eliminando los índices de los bucles internos.

3. Se generan los bucles de control en la representación del programa.

4. Se añaden otra vez las ecuaciones iniciales al sistema obtenido en el punto 2 y se calculan los límites para los bucles internos.

5. Se transforman los bucles originales (internos) y el cuerpo de la anidación.

Veremos el resultado de la aplicación de cada uno de estos pasos mediante el siguiente código de ejemplo:

```
DO K__0 = 1, 2499, 1
  A__0(K__0, K__0) = SQRT(A__0(K__0, K__0))
  DO I__1 = (K__0 + 1), 2500, 1
    A__0(I__1, K__0) = ((A__0(I__1, K__0) / A__0(K__0, K__0)))
    DO J__1 = (K__0 + 1), I__1, 1
      A__0(I__1, J__1) = ((A__0(I__1, J__1) -((A__0(I__1, K__0) * A__0(J__1, K__0)))))
    END DO
  END DO
END DO
A__0(2500, 2500) = SQRT(A__0(2500, 2500))
```

Tras la ejecución de los modelos de procesador y cache, el compilador decide aplicar tiling con los siguientes parámetros:

$\_iloop = \{2, 0, 1, 2\}$
$\_striplevel = \{2, 2, 1, 1\}$
$\_stripsz = \{104, 56, 40, 26\}$

Se van a generar cuatro bucles de control para realizar el recorrido de los bucles. Estos bucles provienen de cortar, respectivamente, los bucles 2, 0, 1 y 2 de la anidación original. Los niveles de memoria para los que se realiza el bloqueo son L2 para los dos bucles más externos y L1 para el resto. Por último los tamaños de bloque son 104, 56, 40 y 26.

1. Con la información anterior se procede, en el primer paso, a la generación del sistema de ecuaciones que refleja las restricciones sobre cada variable de iteración:

$$
\begin{array}{c}
\begin{array}{ccccccc} & & & & K & I & J \end{array} \\
\left[\begin{array}{ccccccc}
0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 \\
\hline
-104 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & -56 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & -40 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & -26 & 0 & 0 & 1 \\
104 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 56 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 40 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 26 & 0 & 0 & -1
\end{array}\right]
\leq
\left[\begin{array}{c}
-1 \\ 2499 \\ -1 \\ 2500 \\ -1 \\ 0 \\ \hline 103 \\ 55 \\ 39 \\ 25 \\ 0 \\ 0 \\ 0 \\ 0
\end{array}\right]
\begin{array}{l}
\\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\
\textit{bucles originales} \\ \\ \\ \\ \\ \\ \\ \\ \textit{bucles de control}
\end{array}
\\
\textit{sistema } s1
\end{array}
$$

El sistema contiene, en la parte superior, los límites de los bucles originales (que ya contemplan el intercambio si es que éste se ha producido) y, en la parte inferior, los límites de los bucles de control que se generan directamente a partir de los tamaños de bloque especificados en *_stripsz*.

2. El siguiente paso consiste en la eliminación, mediante proyección, de los índices de los bucles internos. Se aplica el algoritmo *row_echelon*[3] para poner a 0 la diagonal superior. De este modo se obtienen los límites de los bucles de control:

$$
\begin{array}{c}
\begin{array}{ccccccc} & & & & K & I & J \end{array} \\
\left[\begin{array}{ccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 \\
-13 & 7 & 0 & 0 & 0 & 0 & 0 \\
13 & 0 & -5 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 7 & -5 & 0 & 0 & 0 & 0 \\
-4 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & -20 & 13 & 0 & 0 & 0 \\
0 & 28 & 0 & -13 & 0 & 0 & 0 \\
-4 & 0 & 0 & -1 & 0 & 0 & 0
\end{array}\right]
\leq
\left[\begin{array}{c}
24 \\ 0 \\ 44 \\ 0 \\ 12 \\ 4 \\ 62 \\ 4 \\ 3 \\ 19 \\ 12 \\ 0
\end{array}\right]
\\
\textit{sistema } s1x
\end{array}
$$

3. Se generan los límites de los bucles externos en la representación intermedia:

```
DO tile__22 = 0, 24, 1
  DO tile__20 = 0, MIN(INTRN_I4DIVFLOOR(((tile__22 * 13) + 12), 7), 44), 1
    DO tile__11 = MAX(INTRN_I4DIVCEIL(((tile__22 * 13) + -4), 5),
                      INTRN_I4DIVCEIL(((tile__20 * 7) + -4), 5)), 62, 1
```

```
      DO tile__12 = MAX(INTRN_I4DIVCEIL(((tile__20 * 28) + -12), 13), (tile__22 * 4)),
                    MIN(((tile__22*4)+3), INTRN_I4DIVFLOOR(((tile__11*20)+19), 13)), 1
      END DO
    END DO
  END DO
END DO
```

4. Se añaden al sistema s1 las ecuaciones del sistema s1x y se vuelve a aplicar *row_echelon* para obtener los límites de los bucles internos:

$$
\begin{array}{c}
\begin{matrix} & & & & K & I & J \end{matrix} \\
\left[\begin{array}{rrrrrrr}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 \\
-13 & 7 & 0 & 0 & 0 & 0 & 0 \\
13 & 0 & -5 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 7 & -5 & 0 & 0 & 0 & 0 \\
-4 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & -20 & 13 & 0 & 0 & 0 \\
0 & 28 & 0 & -13 & 0 & 0 & 0 \\
4 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
0 & 0 & -40 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & -26 & 1 & 0 & 0 \\
0 & 56 & 0 & 0 & -1 & 0 & 0 \\
0 & -56 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 40 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 26 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 \\
0 & 0 & -40 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 26 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & -26 & 0 & 0 & 1
\end{array}\right]
\end{array}
\leq
\left[\begin{array}{r}
24 \\
0 \\
44 \\
0 \\
12 \\
4 \\
62 \\
4 \\
3 \\
19 \\
12 \\
0 \\
96 \\
\hline
38 \\
24 \\
0 \\
55 \\
2499 \\
-1 \\
0 \\
0 \\
2500 \\
-1 \\
39 \\
0 \\
0 \\
-1 \\
25
\end{array}\right]
$$

5. Se actualiza el código de los bucles internos en la representación intermedia:

```
DO tile__22 = 0, 24, 1
  DO tile__20 = 0, MIN(INTRN_I4DIVFLOOR(((tile__22 * 13) + 12), 7), 44), 1
    DO tile__11 = MAX(INTRN_I4DIVCEIL(((tile__22 * 13) + -4), 5),
                      INTRN_I4DIVCEIL(((tile__20 * 7) + -4), 5)), 62, 1
      DO tile__12 = MAX(INTRN_I4DIVCEIL(((tile__20 * 28) + -12), 13), (tile__22 * 4)),
                    MIN(((tile__22*4)+3), INTRN_I4DIVFLOOR(((tile__11*20)+19), 13)), 1
        DO K__0 = MAX((tile__20 * 56), 1), MIN(MIN(((tile__20 * 56) + 55),
                  MIN(((tile__11 * 40) + 38), ((tile__12 * 26) + 24))), 2499), 1
```

```
              imperf_L1___1 = (K__0 + 1)
              imperf_Lnew1___1 = MAX((K__0 + 1), MAX((tile__11 * 40), (tile__12 * 26)))
              imperf_Unew1___1 = MIN(((tile__11 * 40) + 39), 2500)
              IF(imperf_L1___1 .EQ. imperf_Lnew1___1 .AND.
                 imperf_Lnew1___1 .LE. imperf_Unew1___1) THEN
                imperf_L1___2 = (K__0 + 1)
                imperf_Lnew1___2 = MAX((K__0 + 1), (tile__12 * 26))
                imperf_Unew1___2 = MIN(imperf_L1___1, ((tile__12 * 26) + 25))
                IF(imperf_L1___2 .EQ. imperf_Lnew1___2 .AND.
                   imperf_Lnew1___2 .LE. imperf_Unew1___2) THEN
                  A__0(K__0, K__0) = SQRT(A__0(K__0, K__0))
                ENDIF
              ENDIF
              DO I__1 = MAX((K__0 + 1), MAX((tile__11 * 40), (tile__12 * 26))),
                        MIN(((tile__11 * 40) + 39), 2500), 1
                imperf_L2___2 = (K__0 + 1)
                imperf_Lnew2___2 = MAX((K__0 + 1), (tile__12 * 26))
                imperf_Unew2___2 = MIN(I__1, ((tile__12 * 26) + 25))
                IF(imperf_L2___2 .EQ. imperf_Lnew2___2 .AND.
                   imperf_Lnew2___2 .LE. imperf_Unew2___2) THEN
                  A__0(I__1, K__0) = ((A__0(I__1, K__0) / A__0(K__0, K__0)))
                ENDIF
                DO J__1 = MAX((K__0 + 1), (tile__12 * 26)),
                          MIN(I__1, ((tile__12 * 26) + 25)), 1
                  A__0(I__1,J__1) = ((A__0(I__1,J__1)-((A__0(I__1,K__0)*A__0(J__1,K__0)))))
                END DO
              END DO
            END DO
          END DO
        END DO
      END DO
    END DO
```

Antes de continuar, el compilador va actualizar toda la información asociada a la representación intermedia: access arrays, anotaciones, definiciones y usos, el grafo de dependencias, etc.

5.2.6. Bloqueo a nivel de registros

En el caso general la aplicación de outer unrolling se limita al penúltimo bucle de la andidación, considerando como último bucle el más interno. A continuación mostramos la aplicación de outer unrolling mediante el siguiente ejemplo:

```
DO K__0 = 1, 2499, 1
  A__0(K__0, K__0) = SQRT(A__0(K__0, K__0))
  DO I__1 = (K__0 + 1), 2500, 1
    A__0(I__1, K__0) = ((A__0(I__1, K__0) / A__0(K__0, K__0)))
    DO J__1 = (K__0 + 1), I__1, 1
      A__0(I__1, J__1) = ((A__0(I__1, J__1) -((A__0(I__1, K__0) * A__0(J__1, K__0)))))
    END DO
  END DO
END DO
A__0(2500, 2500) = SQRT(A__0(2500, 2500))
```

En primer lugar se realiza el desenrosque del bucle externo ignorando el código imperfecto:

```
DO K__0 = 1, 2499, 1
  A__0(K__0, K__0) = SQRT(A__0(K__0, K__0))
  DO I__1 = (K__0 + 1), 2492, 9
    DO J__1 = (K__0 + 1), I__1, 1
      A__0(I__1, J__1) = ((A__0(I__1, J__1) -((A__0(I__1, K__0) * A__0(J__1, K__0)))))
      A__0(I__1+1,J__1) = ((A__0(I__1+1,J__1)-((A__0(I__1+1,K__0) * A__0(J__1,K__0)))))
      A__0(I__1+2,J__1) = ((A__0(I__1+2,J__1)-((A__0(J__1,K__0) * A__0(I__1+2,K__0)))))
      A__0(I__1+3,J__1) = ((A__0(I__1+3,J__1)-((A__0(J__1,K__0) * A__0(I__1+3,K__0)))))
      A__0(I__1+4,J__1) = ((A__0(I__1+4,J__1)-((A__0(J__1,K__0) * A__0(I__1+4,K__0)))))
      A__0(I__1+5,J__1) = ((A__0(I__1+5,J__1)-((A__0(J__1,K__0) * A__0(I__1+5,K__0)))))
      A__0(I__1+6,J__1) = ((A__0(I__1+6,J__1)-((A__0(J__1,K__0) * A__0(I__1+6,K__0)))))
      A__0(I__1+7,J__1) = ((A__0(I__1+7,J__1)-((A__0(J__1,K__0) * A__0(I__1+7,K__0)))))
      A__0(I__1+8,J__1) = ((A__0(I__1+8,J__1)-((A__0(J__1,K__0) * A__0(I__1+8,K__0)))))
    END DO
  END DO
  DO wd_I__0 = I__1, 2500, 1
    A__0(wd_I__0, K__0) = ((A__0(wd_I__0, K__0) / A__0(K__0, K__0)))
    DO J__1 = (K__0 + 1), wd_I__0, 1
      A__0(wd_I__0, J__1) = ((A__0(wd_I__0, J__1) -((A__0(wd_I__0, K__0) *
                                                     A__0(J__1, K__0)))))
    END DO
  END DO
END DO
```

En el código resultante podemos observar como el número de sentencias en el cuerpo del bucle interno se incrementa por la aplicación del desenrosque, y también la aparición de una anidación adicional. Esta nueva anidación se introduce para ejecutar las iteraciones sobrantes debido a que el factor de bloqueo no es divisor del número de iteraciones.

A continuación el compilador actualiza el código para que se contemple la correcta ejecución del código imperfecto. El siguiente código refleja estos cambios:

```
DO K__0 = 1, 2499, 1
  A__0(K__0, K__0) = SQRT(A__0(K__0, K__0))
  DO I__1 = (K__0 + 1), 2492, 9
    lstar__0 = (K__0 + 2)
    ustar__0 = I__1
    IF(lstar__0 .LE. ustar__0) THEN
      DO r2d_I__0 = I__1, (I__1 + 8), 1
        A__0(r2d_I__0, K__0) = ((A__0(r2d_I__0, K__0) / A__0(K__0, K__0)))
        A__0(r2d_I__0, (K__0 + 1)) = ((A__0(r2d_I__0, (K__0 + 1)) -((A__0(r2d_I__0, K__0)
                                                    * A__0((K__0 + 1), K__0)))))
      END DO
      DO J__1 = lstar__0, ustar__0, 1
        A__0(I__1, J__1) = ((A__0(I__1, J__1) -((A__0(I__1, K__0) * A__0(J__1, K__0)))))
        A__0(I__1+1,J__1) = ((A__0(I__1+1,J__1)-((A__0(I__1+1,K__0) * A__0(J__1,K__0)))))
        A__0(I__1+2,J__1) = ((A__0(I__1+2,J__1)-((A__0(J__1,K__0) * A__0(I__1+2,K__0)))))
        A__0(I__1+3,J__1) = ((A__0(I__1+3,J__1)-((A__0(J__1,K__0) * A__0(I__1+3,K__0)))))
        A__0(I__1+4,J__1) = ((A__0(I__1+4,J__1)-((A__0(J__1,K__0) * A__0(I__1+4,K__0)))))
        A__0(I__1+5,J__1) = ((A__0(I__1+5,J__1)-((A__0(J__1,K__0) * A__0(I__1+5,K__0)))))
        A__0(I__1+6,J__1) = ((A__0(I__1+6,J__1)-((A__0(J__1,K__0) * A__0(I__1+6,K__0)))))
```

```
        A__0(I__1+7,J__1) = ((A__0(I__1+7,J__1)-((A__0(J__1,K__0) * A__0(I__1+7,K__0)))))
        A__0(I__1+8,J__1) = ((A__0(I__1+8,J__1)-((A__0(J__1,K__0) * A__0(I__1+8,K__0)))))
      END DO
      DO r2d_I__0 = I__1, (I__1 + 8), 1
        DO J__1 = (ustar__0 + 1), r2d_I__0, 1
          A__0(r2d_I__0, J__1) = ((A__0(r2d_I__0, J__1) -((A__0(r2d_I__0, K__0) *
                                                          A__0(J__1, K__0)))))
        END DO
      END DO
    ELSE
      DO r2d_I__0 = I__1, (I__1 + 8), 1
        A__0(r2d_I__0, K__0) = ((A__0(r2d_I__0, K__0) / A__0(K__0, K__0)))
        DO J__1 = (K__0 + 1), r2d_I__0, 1
          A__0(r2d_I__0, J__1) = ((A__0(r2d_I__0, J__1) -((A__0(r2d_I__0, K__0) *
                                                          A__0(J__1, K__0)))))
        END DO
      END DO
    ENDIF
  END DO
  DO wd_I__0 = I__1, 2500, 1
    A__0(wd_I__0, K__0) = ((A__0(wd_I__0, K__0) / A__0(K__0, K__0)))
    DO J__1 = (K__0 + 1), wd_I__0, 1
      A__0(wd_I__0, J__1) = ((A__0(wd_I__0, J__1) -((A__0(wd_I__0, K__0) *
                                                    A__0(J__1, K__0)))))
    END DO
  END DO
END DO
```

Podemos observar como antes de pasar a ejecutar el bucle $J\_\_1$, se añade un bucle ($r2d\_I\_\_0$) para la ejecución del código imperfecto, que debe ejecutarse para cada una de las iteraciones desenrolladas. Este bucle ejecuta también la primera iteración de la versión previa del bucle $J\_\_1$, cosa que produce que los límites de éste se vean modificados. Debido a que el límite superior de $J\_\_1$ depende del bucle $I\_\_1$, es necesario un bucle adicional, que sigue al bucle $J\_\_1$ y que va a realizar la ejecución de posibles restos. Tras la modificación de los límites de $J\_\_1$ ya no se garantiza que siempre se ejecute al menos una iteración de este bucle. Esta eventualidad se tiene en cuenta y por eso se añade un condicional que ejecutará una u otra versión del código dependiendo del valor de los límites de $J\_\_1$.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Al igual que su predecesor [11], el objetivo de este documento es dar a conocer con un elevado nivel de detalle el optimizador de bucles de los compiladores *Open64* y *ORC*.

En [11] se describe la detección y acondicionamiento de las anidaciones de bucles que posteriormente se intentará optimizar. Estas anidaciones reciben el nombre de Singly Nested Loop Nests (SNLs). En ese mismo documento encontramos también la descripción del Shackling, optimización que aplica bloqueo a los bucles de una anidación partiendo desde el punto de vista de los datos. Junto a estos contenidos, se describe también un elemento básico en el análisis de legalidad de distintas transformaciones como son los grafos de dependencias de datos.

En este documento retomamos la descripción del optimizador de bucles e intentamos describir con máximo detalle la selección y posterior aplicación de las transformaciones de intercambio, outer unroll y tiling.

En el capítulo 2 hemos visto como se analizan los SNLs para determinar cual va a ser el subconjunto de bucles de cada SNL susceptible de ser transformado.

Una vez decidido qué bucles se podrán transformar se pasa a la selección del conjunto de transformaciones que se aplicará. Esta selección se realiza mediante un algoritmo que enumera distintas posibilidades y que, basándose en las estimaciones del coste generadas por el modelo del procesador y el modelo de la cache, intenta escoger la mejor opción. El algoritmo y el modelo del procesador se describen en el capítulo 3 y el modelo de la cache en el capítulo 4.

Por último, el capítulo 5 muestra como la aplicación de estas transformaciones se refleja en el código del programa que se está compilando.

6.2. Trabajo futuro

Valorando conjuntamente los dos documentos realizados hasta el momento sobre el optimizador de bucles de los compiladores Open64 y ORC entendemos

que, si bien no se puede hablar de una descripción completa, sí se ha llegado a describir con un elevado nivel de detalle gran parte de esta etapa del compilador. Además esta descripción cubre la selección y aplicación de las que quizás podríamos considerar como las transformaciones más importantes, como son intercambio, outer unroll y tiling. Quedan en el tintero la aplicación de padding, inner fission y prefetch, siendo esta última transformación la que un mayor esfuerzo requeriría para llevar a cabo su documentación. Otra optimización de bucles que, por formar parte del generador de código, hasta el momento no hemos contemplado es la aplicación de software pipelining.

Si bien no descartamos que en algún momento se decida intentar documentar estas transformaciones, actualmente estamos centrando nuestros esfuerzos en el estudio de estrategias alternativas o complementarias a las que incorpora el compilador para implementar las transformaciones de intercambio, outer unroll y tiling. Concretamente, se está llevando a cabo la implementación de los algoritmos de tiling propuestos por Marta Jiménez en [6]. Creemos que la aplicación de estos algoritmos permitirá mejorar el rendimiento de, al menos, las anidaciones de bucles no rectangulares (que el compilador clasificaría como anidaciones generales).

Bibliografía

[1] http://ipf-orc.sourceforge.net. Web de ORC en SourceForge.

[2] http://open64.sourceforge.net. Web de Open64 en SourceForge.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26, pages 39–50, Williamsburg, VA, April 1991.

[4] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[5] 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.

[6] Marta Jiménez. *Multilevel Tiling For Non-Rectangular Iteration Spaces*. PhD thesis, Universitat Politècnica de Catalunya, 1999.

[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] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328. ACM Press, 1988.

[10] Eduard Santamaria, Marta Jiménez, Agustín Fernández, and Josep M. Llabería. Introducción al compilador Open64/ORC. Technical Report UPC-DAC-2003-20, Universitat Politècnica de Catalunya, 2003.

[11] Eduard Santamaria, Marta Jiménez, Agustín Fernández, and Josep M. Llabería. El optimizador de bucles del compilador Open64/ORC. Technical Report UPC-DAC-2004-44, Universitat Politècnica de Catalunya, 2004.

[12] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, 1992.

[13] 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.

[14] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM Press, 1989.