Modelado dinámico de accesos a memoria para paralelización especulativa

Tesina de grado presentada
por

Esteban Campostrini

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de
Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura.
Universidad Nacional de Rosario.

Diciembre 2014

Director
Dr. Philippe Clauss
philippe.clauss@inria.fr

Co-director
Aravind Sukumaran Rajam
aravind.sukumaran-rajam@inria.fr

Co-director
Juan Manuel Martinez Caamaño
jmmartinez@dc.uba.ar

Equipo CAMUS
(Compilation pour les
Architectures MUlti-coeurS)
https://team.inria.fr/camus

Universidad de Strasbourg
Pôle API
300 Bd S. Brant - BP 10413, 67412 Illkirch cedex, Francia
+33 (0)3 68 85 45 54
http://icube-icps.unistra.fr

Resumen

La alta disponibilidad de los procesadores multinúcleos así como su amplio rango de aplicación —desde uso cotidiano en tabletas y celulares, hasta uso profesional con gran demanda de cálculos intensivos— hace que se lleven a cabo esfuerzos orientados a maximizar el aprovechamiento del hardware disponible. En este contexto, la paralelización de código surge naturalmente como uno de los principales recursos a explotar. Generalmente, los ciclos juegan un rol central en el tiempo de ejecución de un programa, encargados de de realizar computaciones repetitivas que los convierte en grandes candidatos a la hora de tener en cuenta la paralelización de un programa. Es por esto que muchas herramientas especialmente orientadas a la paralelización de ciclos han sido desarrolladas.

Un modelo mátematico, el *modelo del poliedro*, cubre la teoría detras del análisis y transformación de ciclos. Este modelo está originalmente limitado a ciclos que pueden ser precisamente analizados en tiempo de compilación. Es decir que, por ejemplo, no puede tratar con ciclos que contienen accesos a memoria de manera indirecta o a través de punteros.

APOLLO (*Automatic POLyhedral Loop Optimizer*) es un framework que permite realizar instrumentación en tiempo de ejecución y paralelización dinámica con el objetivo de adaptar el modelo del poliedro de manera especulativa. De esta forma, este framework permite el tratado de ciclos que, originalmente, no pueden ser paralelizados utilizando técnicas tradicionales.

APOLLO extiende el modelo del poliedro a códigos a los que anteriormente no podía ser aplicado. Sin embargo, entre el resto de las condiciones que sigue imponiendo el modelo, se encuentran el hecho de que los accesos a memoria tienen que ser lineales. Esta tesis continua el trabajo hecho en APOLLO de la siguiente forma:

- Extendiendo el modelado de operaciones de lectura y escritura a memoria realizado por APOLLO, con el objetivo de permitir la paralelización de loops en los que no todos los accesos a memoria son lineales pero que sin embargo no ponen en riesgo la validez de las transformaciones realizadas mediante el modelo.

- Incorporando un sistema de decisión que permite elegir en forma dinámica entre dos estrategias distintas para el manejo de accesos a memoria no lineales de acuerdo comportamiento no linear observado durante la instrumentación.

Estas extensiones permiten que el modelo del poliedro pueda aplicarse a loop nests que no cumplen estrictamente (tanto de manera estática como dinámica) las condiciones necesarias. Al mismo tiempo, amplían el espectro de códigos que APOLLO es capaz de paralelizar.

La efectividad de las extensiones fue evaluada empíricamente mediante benchmarks bien conocidos.

Acknowledgements

To Phillip and Cedric who gave me the chance of being part of their team during the last months, I learned a lot and met really nice people. To all the professors that are part of LCC, they do what they love and love what they do, and that's more than I can ask as a student. To the great people with whom I shared almost all of the courses, and most important, asados and midnight teas: Lucia, Fepi, Qk, Javier, Julián, Mariano, el Pajaro and Tincho. To Aravind and Juan, for their patience and their guidance. To Willy, for cheap tabaco and nice break time talks. And to the most important persons in this world, my parents, I wouldn't have been able of doing anything without them.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Contexto del trabajo

Hoy en día los procesadores multinúcleo están en todas partes. Su alta popularidad y disponibilidad así como el potencial que poseen, demandan nuevas estrategias para explotar de manera ventajosa el hardware subyacente y lograr una buena performance en el software. Dos aspectos importantes cuando de esto se trata, son paralelismo y la localidad de datos. A pesar de la existencia de muchos lenguajes orientados a la paralelización, como OpenMP, MPI, Cilk o CUDA, la programación paralela es en general difícil ya que el programador debe tener en cuenta detalles complejos, tales como seleccionar un algoritmo de paralelización adecuado, analizar las dependencias entre las distintas partes del código y usar un modelo de programación adecuado, entre otros. Como resultado de esto, el desarrollo de compiladores dedicados a paralelizar automáticamente es un área de interés en el mundo de la programación paralela. Varios compiladores enfocados a paralelización automática han sido desarrollados, como por ejemplo SUIF [18] y PIPS [3].

Otro recurso para la paralelización de código que ha sido desarrollado en los últimos años es la *paralelización especulativa*. De manera general, esta técnica consiste en paralelizar código antes de tener la completa certeza de que el código puede ser paralelizado. En el contexto general de la paralelización especulativa, una dirección que ha sido ampliamente estudiada es la especulación a nivel de hilos, o *thread-level speculation*(TLS) [2] [11] [15] [16] [17]. Típicamente un framework TLS realiza una ejecución optimista en paralelo de regiones de código en las que no *todas* las dependencias son conocidas, llevando registro de los accesos a memoria para determinar si ocurre alguna violación de dependencia. Generalmente estos frameworks tienen un sistema de verificación centralizado o requieren de una gran cantidad de comunicación *inter-thread* lo cual va en contra de la programación paralela de alta performance. Más aun, generalmente aplican solo una estrategia de paralelización que consiste en dividir el nido de ciclos externo, o *outermost loop* en varios threads especulativos paralelos.

Debido a que, generalmente, durante la ejecución de un programa la mayor parte del tiempo transcurre dentro de ciclos que normalmente ejecutan código -epetitivo", estos han sido blanco de estudio, dando lugar a técnicas de análisis especializadas. El análisis preciso de datos hace que sea posible sacar ventaja de transformaciones para paralelización avanzadas como *tiling*, *fusion* o *loop splitting*. La teoría que trata el análisis de ciclos y sus transformaciones esta unificada en un marco teórico matemático llamado *modelo del poliedro* [6]. El modelo del poliedro permite capturar el comportamiento de un programa por medio de funciones afines que luego pueden ser empleadas para realizar análisis complejos y optimizar el código mediante transformaciones. A pesar de ser un recurso poderoso, solo es aplicable a nidos de ciclos cuyo comportamiento lineal puede ser determinado en tiempo de compilación. Ciclos que incluyen accesos de memoria de manera indirecta o a través de punteros, exhiben su *linealidad* en tiempo de ejecución. Es por esto que no pueden ser tratados mediante esta técnica en tiempo de compilación.

Para superar esta limitación, una técnica de paralelización especulativa fue desarrollada [10]. Esta técnica aprovecha la información disponible en tiempo de ejecución para paralelizar de manera *on-the-fly* ciertas partes del código. La información necesaria para completar el modelo del poliedro es obtenida mediante la instrumentación de código. Debido a que parte del modelo está basada en un *sampleo* de la ejecución del programa, existe la posibilidad de que el comportamiento cambie durante la misma. Esto puede poner en riesgo la validez del modelo, haciendo que las transformaciones efectuadas dejen de ser válidas. Por esta razón un sistema de *verificación on-line* tiene que ser implementado capaz de llevar a cabo acciones de recuperación que devuelvan el programa a un estado válido. Este tipo de acciones de recuperación es llamado *rollback*.

En este contexto, APOLLO, *Automatic POLyhedral Loop Optimizer* [9], provee un framework orientado a *optimizar y paralelizar* código en tiempo de ejecución el cual permite explotar el paralelismo en códigos en los cuales a priori no es posible. APOLLO aplica de manera dinámica patrones de código llamados *skeletons*, los cuales permiten usar diferentes técnicas de paralelización tales como *loop tiling*, *skewing*, etc. Posee un sistema de verificación de-centralizado que representa una alternativa más eficiente que la que es generalmente usada por sistemas TLS a la hora de verificar código paralelizado especulativamente. En caso de una predicción errónea, se inicia un rollback y la memoria que había sido modificada es restaurada a un estado correcto gracias a un sistema de back-up on-line. Este sistema de back-up realiza una copia de las regiones de memoria que van a ser modificadas antes de iniciar la ejecución paralela. A modo de capturar la nueva fase del programa una re-instrumentación es iniciada, y nuevas optimizaciones y transformaciones son aplicadas. La Figura 1.1 representa globalmente el flujo de ejecución de APOLLO.

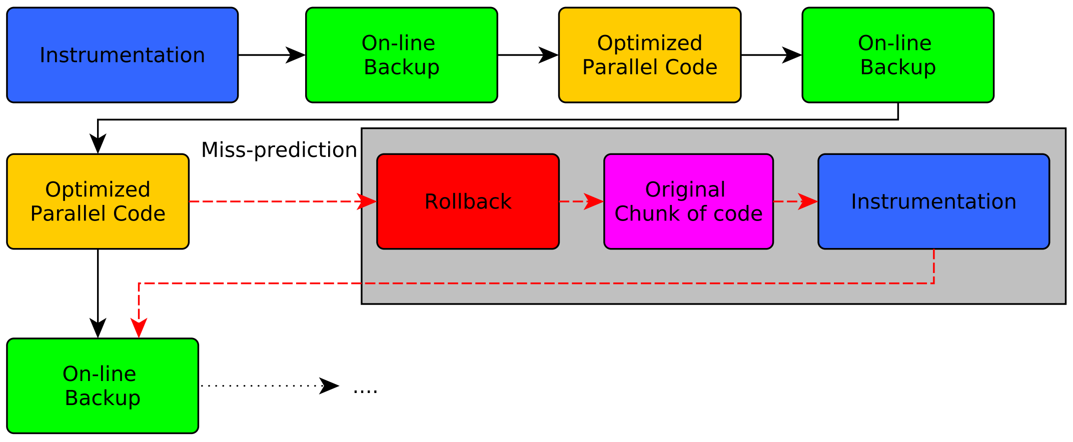


Figura 1.1: Vista general del flujo de ejecución de APOLLO

En definitiva, APOLLO permite llevar a cabo paralelización especulativa mediante la aplicación de transformaciones definidas por el modelo del poliedro.

Esta tesis continua el trabajo hecho en APOLLO extendiendo su alcance a ciclos que incluyan accesos de lectura y escritura de memoria no lineales, mediante:

- Extendiendo el modelado de operaciones de lectura y escritura a memoria realizado por APOLLO, con el objetivo de permitir la paralelización de loops en los que no todos los accesos a memoria son lineales pero que sin embargo no ponen en riesgo la validez de las transformaciones realizadas mediante el modelo.

- Incorporando un sistema de decisión que permite elegir en forma dinámica entre dos estrategias distintas para el manejo de accesos a memoria no lineales de acuerdo comportamiento no linear observado durante la instrumentación.

Las extensiones propuestas tienen impacto mayormente en el sistema de verificación de APOLLO.

Parte I

Preliminares

Capítulo 2

Breve introducción al modelo del poliedro

El modelo del poliedro es un framework matetmático geométrico utilizado para optimización de programas y de ciclos. Provee una forma homogénea y compacta de representar la información necesaria para realizar análisis preciso de dependencias. Geométricamente, un poliedro representa un objeto con caras planas que puede existir en un número arbitrario de dimensiones. Un poliedro acotado es un politopo. En el caso de APOLLO, solo los politopos convexos son tenidos en cuenta.

Abstraer un ciclo de manera poliedral es equivalente a asociar a cada statement del ciclo un poliedro en el cual cada punto entero representa una instancia del statement durante cada iteración del ciclo. Una exhaustiva presentación de aplicaciones del modelo del poliedro para optimización de programas es dada por Feautrier en varios trabajos [4, 5, 6, 7].

El modelo del poliedro puede ser aplicado solamente a ciclos de loops en los cuales:

- Las cotas de los ciclos son funciones afines de las cotas de los ciclos envolventes.
- Cada predicado es una función afín de los ciclos envolventes y de parámetros globales.
- Las estructuras de datos presentes en el nido de ciclos son arreglos multidimensionales y escalares de cualquier tipo.
- Los índices de los arreglos son funciones afines de los ciclos envolventes y de los parametros.

```
for(i = 1; i ≤ N; ++i) {
    for(j = 1; j ≤ N; ++j) {
        S: A[i][j] = A[i-1][j] + A[i][j-1]
    }
}
```

Figura 2.1: Ejemplo de ciclos anidados

2.1. Representando loops en el modelo del poliedro

Para un análisis preciso de cada statement dentro del modelo, tres cosas son necesarias: su dominio de iteración, el patrón de acceso a memoria de la misma y el orden de ejecución con respecto a los otros statements.

2.1.1. El dominio de iteración

En la Figura 2.1 se ve que el statement etiquetado como S puede tener diferentes instancias a lo largo de la vida del ciclo, a saber, $S_{(1,1)}$, $S_{(1,2)}$, $\ldots$, $S_{(N,N)}$. Cada instancia del statement tiene un vector de iteración (i, j) asociado. El conjunto de todos los vectores de iteración de un statemet es llamado *dominio de iteración*. El dominio de iteración de S es:

$$D_S = \{(i, j) \in Z^2 | 1 \leq i \leq N, 1 \leq j \leq N\}$$

2.1.2. Funciones de acceso de datos

En el modelo del poliedro, los accesos a memoria deben estar expresados en términos de funciones afines de los indicies de los ciclos envolventes. El statement S tiene una operación de escritura correspondiente a $A[i][j] = \ldots$ y dos operaciones de lectura que corresponden a $A[i-1][j]$ y $A[i][j-1]$. Las funciones de acceso correspondientes a estos accesos de memoria son:

$$f_{W^A}(i,j) = \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$f_{R^A}(i,j) = \begin{pmatrix} i-1 \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

$$f_{R^A}(i,j) = \begin{pmatrix} i \\ j-1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

2.1.3. Statement scheduling

Ni el dominio, ni las funciones de acceso de datos proveen un orden de ejecución para un statement. La *función de scheduling* se encarga de mapear puntos del dominio de iteración D_S a *time stamps* multidimensionales, es decir, asigna una fecha lógica a cada instancia dinámica de cada statement. En general para un statement S, su time-stamp θ_S está definido por:

$$\forall \vec{x} \in D_S \; \theta_S(\vec{x}) = T\vec{x} + \vec{t}.$$

Donde T is una matriz de $d \times d$, $\vec{t}$ es un vector de tamaño d y d es la profundidad dentro del nido de ciclos de S

2.2. Análisis de dependencias

Para que la transformación del ciclo sea válida, la semántica del código original debe ser preservada.

Definition 9 (*Dependencia de instancias de statements*) Dos statements S y R son dependientes si existen dos instancias $S(\vec{x_S})$ y $R(\vec{x_R})$, donde $\vec{x_S}$ y $\vec{x_R}$ pertenecen al dominio de iteración de S y R, de manera tal que $S(\vec{x_S})$ y $R(\vec{x_R})$ acceden a la misma posición de memoria y al menos uno de ellos es una operación de escritura.

Para preservar la semántica, el orden de ejecución de dos statements dependientes debe ser el mismo en la versión original secuencial y en la versión paralela. Por otro lado, dos statements independientes pueden ser ejecutados en un orden arbitrario, pudiendo ser paralelizados. Vale la pena notar que cuando dos statements son operaciones de lectura, no hay dependencia ya que la memoria no es modificada. Sin embargo tales accesos son tenidos en cuenta a la hora de mejorar la localidad de datos. Las dependencias son clasificadas en tres categorías, según el orden de las operaciones de lectura y de escritura:

- RAW: read-after-write, or *flow dependence.*
- WAR: write-after-read, or *anti dependence.*
- WAW: write-after-write, or *output dependence.*

Una vez que las dependencias entre iteraciones fueron computadas, es posible aplicar transformaciones poliedrales (schedules) de manera que no ocurran violaciones de dependencias.

Capítulo 3

APOLLO

3.1. Limitaciones del modelo del poliedro

La aplicación original del modelo del poliedro está orientada a optimización y paralelización de ciclos en tiempo de compilación. Es decir, que trabaja con códigos estáticos que están en concordancia con las exigencias del mismo. Sin embargo, la aplicación estática del modelo del poliedro está limitada por factores como: cotas que no pueden ser estáticamente predichas, flujos de control complejos, presencia de punteros no resueltos, etc.

Si bien no son analizables de manera estática, estos códigos pueden exhibir aun un comportamiento lineal (durante su ejecución) que cumpla las condiciones establecidas por el modelo del poliedro. O sea que, contando con suficiente información sobre los patrones de acceso a memoria así como de las cotas de los ciclos, el análisis poliedral puede ser llevado a cabo junto con sus consiguientes transformaciones y optimizaciones. Para lograr modelar el comportamiento del programa en tiempo de ejecución, un pequeño conjunto de iteraciones son instrumentadas (instrumentación mediante sampleo). Dado que la construcción del modelo está basada en la observación de una pequeña parte de la ejecución, cualquier cambio en esta puede afectar la validez de las transformaciones. Esto hace que sea necesaria alguna forma de verificación que asegure la continua correctitud del modelo, más precisamente:

- Verificación en tiempo de ejecución es necesaria para validar las transformaciones especulativas de código.
- Un sistema de recuperación on-line, que sea ejecutado en caso de haber una mala predicción;
- El sistema debe ser lo suficientemente *liviano* para opacar el costo que conllevan las acciones anteriores.

A continuación, un framework que aplica el modelo del poliedro de manera especulativa, teniendo en cuenta los requerimientos anteriormente mencionados, es presentado.

3.2. Apollo

APOLLO (Automatic POLyhedral Loop Optimizer) es un framework que permite realizar instrumentación y paralelización especulativa de ciclos. Como optimizador poliedral, se encarga de aplicar optimizaciones poliedrales antes de iniciar la paralelización y como framework de paralelización especulativa se encarga de manejar nidos de ciclos que no son analizables de manera estática. Esto último lo logra mediante la generación de diferentes versiones de una misma porción de código, siendo cada una de ellas adecuada para diferentes tipos de ejecuciones (por ejemplo: una versión puede estar orientada a paralelización mientras que otra a instrumentación). En las secciones siguientes se verá en más detalle las diferentes partes de APOLLO así como también la forma en la que adapta el modelo del poliedro para realizar paralelización especulativa.

3.2.1. Arquitectura

La implementación consta de dos grandes partes: una parte estática, implementada mediante pasadas del compilador **LLVM** [12], y una parte dinámica escrita enteramente en C++.

Para utilizar APOLLO el programador tiene que enmarcar el código usando un pragma especifico como se muestra en la Figura 3.1. Este código será compilado usando la suite de compilación de APOLLO que corre sobre el compilador *clang-llvm*. El resultado de la compilación es un ejecutable. Gracias al diseño desacoplado de LLVM, el *front-end* puede ser reemplazado por uno de otro lenguaje siempre que produzca código intermedio de LLVM (*llvm-ir*), haciendo APOLLO aplicable a un gran conjunto de lenguajes de programación.

```
#pragma apollo dcop
{
     loop 1
          loop 2
               ...
}
```

Figura 3.1: Ciclo de nidos anotados para paralelización especulativa con APOLLO

La componente estática está mediante pasadas de compilación de LLVM. La principal tarea de estas componentes es generar diferentes versiones de cada ciclo que es anotado.

- Una versión está dedicada a realizar el *profiling* (ver Sección 3.2.3). Esta versión contiene código para capturar datos relevantes para la construcción del modelo, que están disponible a través de las iteraciones del ciclo.
- Una versión original del ciclo.
- Distintos *code-skeletons* que implementan distintas transformaciones paralelas.

La componente dinámica o *runtime*, está encargado de recolectar la información que está disponible solamente en tiempo de ejecución. Usando estos datos, el runtime construye funciones lineales de interpolación que describen el comportamiento del nido de ciclos actual. Luego de esto, un análisis de dependencias es llevado a cabo que luego será entregado a PLUTO [1] que es el encargado de realizar las transformaciones poliedrales. Una vez obtenidas las transformaciones, el code-skeleton adecuado es seleccionado para ser instanciado y posteriormente ejecutado.
Este procedimiento está descompuesto en diferentes pasos, cada uno de ellos aislados en diferentes módulos los cuales tienen una tarea especifica a cumplir, por ejemplo: realizar el muestreo de datos a partir de los cuales las funciones afines de interpolación son construidas, realizar el análisis de dependencias, llevar a cabo el back-up de las posiciones de memoria que van a ser modificadas durante la ejecución especulativa, etc.

3.2.2. Multi-versionado

Como fue mencionado anteriormente, APOLLO genera diferentes versiones del código a paralelizar. La idea detrás de este *multi-versionado* consiste en compilar y embeber en el binario final diferentes versiones del código que permiten, posteriormente, seleccionar de manera dinámica la versión que mejor se ajusta al comportamiento actual del nido de ciclos. El mecanismo de switching de APOLLO permite pasar de una versión a otra sin interrumpir la ejecución del programa.

APOLLO genera tres tipos de versiones a partir de cada nido de ciclos anotado: una versión original, una versión de instrumentación y diferentes code-skeletons que implementan diferentes clases de transformaciones y optimizaciones. Al momento de ejecución de los ciclos, el runtime se encarga de alternar entre las diferentes versiones según sea necesario.

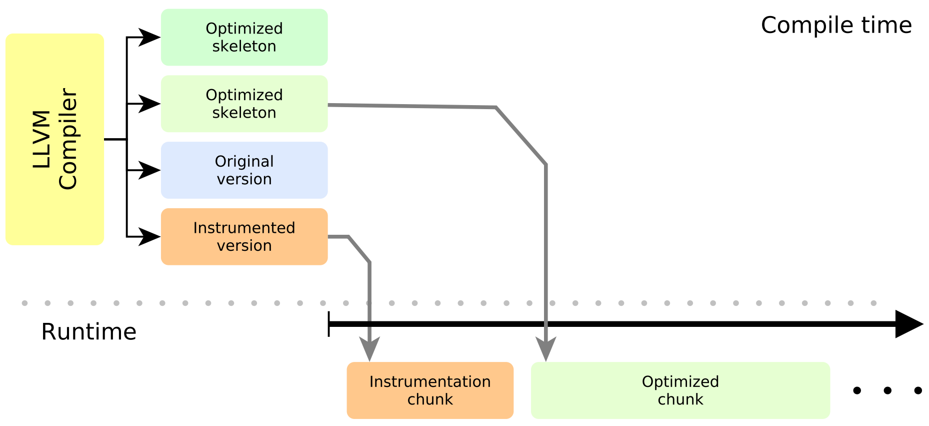


Figura 3.2: Alternando entre diferentes versiones durante la ejecución de un ciclo paralelizado especulativamente

3.2.3. Instrumentación

La estrategia usada para aplicar el modelo del poliedro en ciclos que no pueden ser completamente analizados de manera estática consiste en ejecutar especulativamente. La especulación se basa en valores observados durante la ejecución de una pequeña parte de los ciclos. Si durante esta pequeña ejecución se observa un comportamiento lineal, se realiza la suposición de que este comportamiento se mantendrá durante toda la ejecución de los ciclos. Naturalmente, es necesario un sistema de verificación en tiempo de ejecución para controlar la validez de la especulación durante la ejecución especulativa, ya que sino se corre el riesgo de que no se preserven las dependencias del código.

La información no-estática es recogida a través de una fase de *profiling on-line* que monitorea e instrumenta los accesos a memoria así como las cotas de los ciclos. Para minimizar el costo causado por la instrumentación, solo un pequeño número de iteraciones es observado del ciclo externo (outermost loop).

Si el número de iteraciones llevadas a cabo por el nido de ciclos es alta (lo cual es esperable en código que va a ser paralelizado), el costo de la instrumentación se hace despreciable y altamente compensado por la performance ganada.

Durante la fase de instrumentación, APOLLO recoge la siguiente información:

- Valores de escalares [1] de interés.
- Direcciones de memoria accedidas.

[1]Los escalares son variables unidimensionales cuyos valores se actualizan típicamente entre dos iteraciones del ciclo del que forman parte

- El número de iteraciones realizadas por los ciclos internos.

Los valores recogidos son utilizados para calcular las funciones de interpolación.

3.2.4. Iteradores Virtuales

Para poder manejar de manera uniforme cualquier tipo de ciclos (for-loops, do-while loops) contadores adicionales llamados *iteradores virtuales* son insertados en los skeletons. Hay un iterador virtual por cada ciclo en el nido y todos son incrementados de a una unidad, tomando el cero como valor inicial. Los iteradores virtuales tienen especial importancia ya que permiten manejar ciclos que originalmente no controlan sus iteraciones mediante un contador, sino que mediante alguna expresión booleana.

3.2.5. Transformaciones de paralelización de código

Una vez que el modelo de predicción fue armado, el runtime realiza un análisis de dependencias para determinar si el nido de ciclos anotados puede ser paralelizado y que transformaciones pueden ser aplicadas a el. Una transformación poliedral consiste en cambiar el orden en el cual las iteraciones son ejecutadas (el orden en el que el dominio de iteración es recorrido). La transformación es representada mediante una matriz y es generada por PLUTO (a partir de las dependencias las funciones de interpolación). Solo el kernel del *scheduler* de PLUTO es utilizado y fue ligeramente adaptado para que tomara como entrada el análisis de dependencias generado por APOLLO. Dado un nido de ciclos de profundidad dos con iteradores $\binom{i}{j}$ y matriz de transformación T, transformaciones poliedrales como loop skewing o loop interchange son obtenidas de la siguiente forma:

$$T \times \binom{i}{j} = \binom{x}{y}$$

Las cotas de los ciclos en el espacio transformado son obtenidas mediante el algoritmo de eliminación Fourier-Motzkin, a través de la biblioteca FMlib [14].

3.2.6. Verificación de la especulación poliedral paralela

Para asegurarse que el modelo de predicción usado durante la ejecución especulativa es válido, ciertas partes del código deben ser verificadas durante la ejecución del *skeleton paralelo* que aplica las optimizaciones. Para esto, se inserta código dedicado en tiempo de compilación, que luego es instanciado en tiempo de ejecución. Este código genera un rollback en caso de que el valor esperado no sea el mismo que el valor actual. Los tres aspectos ejecución que son verificados son: accesos a memoria, los escalares y las cotas de los ciclos.

Verificación de accesos a memoria

El modelo de predicción representa la secuencia de direcciones accedidas por una determinada instrucción del ciclo como una función afín de los iteradores virtuales. Por lo tanto, le verificación de los accesos a memoria se realiza comparando, para cada instrucción de lectura o escritura, la posición realmente accedida contra el valor predicho por la función de interpolación.

Inicialización y verificación de escalares

Variables escalares como contadores y acumuladores tienen que ser tenidos en cuenta por el sistema de verificación ya que también acarrean dependencias a través de las iteraciones. Los escalares típicamente toman un valor inicial que es actualizado a través de las iteraciones del ciclo del cual forman parte. Generalmente la actualización se realiza al comienzo de cada iteración utilizando el último valor que el escalar contenía al terminar la iteración anterior. Esto mismo es realizado por APOLLO para su verificación: al final de cada iteración, la predicción para el siguiente valor del escalar (obtenida mediante la correspondiente función de interpolación linear) es comparada contra el valor actual (o sea el último valor que toma en esa iteración). Nuevamente, si la verificación falla, un rollback es generado.

Verificación de la cota de los ciclos

Otra de las condiciones del modelo del poliedro es que las cotas de los ciclos deben ser funciones afines de los *enclosing loops* o ciclos *padres*. En este caso, es posible que pueda modelarse el patrón del *outermost loop* pero si su valor final no puede obtenerse en tiempo de compilación entonces solo puede conocerse al alcanzar el final de la ejecución del mismo. Es por esto que no puede usarse para generar transformaciones, pero como se verá mas adelante, el sistema de chunking de APOLLO permite que se ejecute en paralelo, permitiendo aplicar transformaciones a los ciclos hijo. De todas maneras, es posible obtener una función de interpolación lineal que es usada para controlar el progreso de los iteradores.

3.2.7. Code Skeletons

Las transformaciones obtenidas son aplicadas a través de los *code skeletons*. Estos son nidos de ciclos parametrizados, derivados del código original. Pueden ser vistos como *templates* que envuelven el código original aplicando transformaciones poliedrales así como transformaciones al cuerpo del ciclo (loop interchange, skewing, etc). Aportan una gran ventaja a la hora de generación de código optimizado mediante transformaciones ya que representan un compromiso entre generación estática y dinámica de código, dándoles flexibilidad en cuanto a los diferentes tipos de código que pueden generar (al ser instanciado con las transformaciones poliedrales) y bajo costo (al ser generados en la etapa compilación).

La lógica de los skeletons se divide en tres partes: la primera parte escanea el espacio de iteración transformado y recupera los valores de los iteradores virtuales en el espacio original; la segunda lleva a cabo la computación original; y la tercera verifica que las especulaciones hechas siguen siendo correctas en la iteración actual.

3.2.8. El sistema de chunking

Como fue anteriormente mencionado, la estrategia para aplica el modelo del poliedro a código que presenta accesos a memoria dinámicos, consiste en encontrar el modelo afín subyacente mediante la instrumentación de un número determinado de iteraciones del *outermost loop*. Suponiendo que el nido de ciclos responde al modelo poliedral, diferentes versiones transformadas y paralelas son generadas mediante los *code skeletons*. Con el fin de proveer una ejecución flexible, capaz de adaptarse a los cambios de fase de la ejecución del ciclo anotado, la ejecución es realiza de a tramos. Estas partes en las que se divide la ejecución son llamadas *chunks*. Un chunk representa un conjunto de iteraciones del *outermost loop*. Con la ejecución dividida en varias partes, es necesario algún mecanismo que se encargue de orquestar la ejecución de los distintos *chunks* de manera correcta. Para lograr esto APOLLO utiliza un mecanismo de *chunking* que está ilustrado en la Figura 3.3.

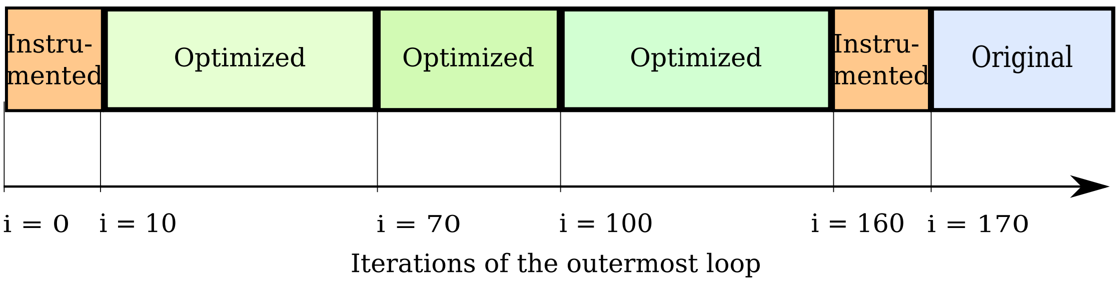


Figura 3.3: Ejecución del código dividida en subpartes llamadas *chunks*

Los *chunks* son orquestados por APOLLO de la siguiente manera:

- La primer secuencia de iteraciones se ejecuta de manera controlada. Es aquí donde se lleva a cabo la instrumentación.
- Se seleccionan el code skeleton paralelo y se lo instancia con la correspondiente transformación obtenida por PLUTO.
- Se lleva a cabo el back-up, guardando las posiciones de memoria que, según el modelo, serán modificadas. A continuación se ejecuta el *chunk paralelo*,

que tiene un tamaño considerablemente mayor al utilizado para la instrumentación.

- Si la verificación falla, la ejecución del chunk es cancelada, la memoria se restituye y un chunk que contiene la versión secuencial original es ejecutada. Luego de este, se vuelve a ejecutar un chunk de instrumentación a modo de capturar la nueva fase del programa. Si, en cambio, la verificación es exitosa, un nuevo chunk paralelo con la misma transformación es lanzado.

3.2.9. Visión general

Para concluir la introducción a APOLLO, una visión general del sistema es mostrada en la Figura 3.4

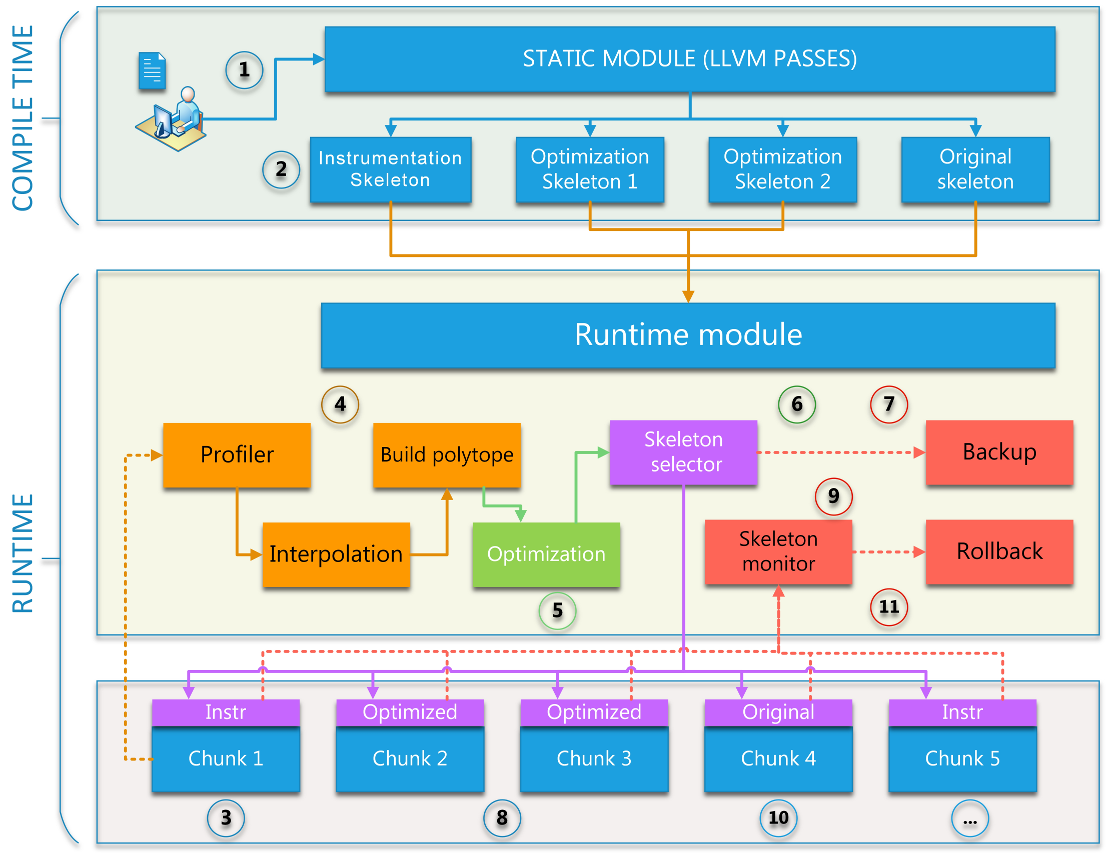


Figura 3.4: Visión general del framework APOLLO

Parte II

Contribuciones

Capítulo 4

Modelando comportamiento no linear

Como fue explicado en capítulos anteriores, a pesar de su gran poder, el modelo del poliedro esta limitado a una clase particular de códigos de computación intensiva cuyo comportamiento puede ser caracterizado por un conjunto de funciones afines. Sin embargo, códigos que no encajan perfectamente en el modelo, aun pueden beneficiarse de este. Mediante on-line profiling, APOLLO saltea las limitaciones del análisis estático, permitiendo aplicar el modelo del poliedro a códigos en los que originalmente era imposible debido a la falta de información. No obstante, entre el resto de las condiciones que siguen siendo impuestas por el modelo, se encuentra el hecho de que el patrón de accesos a memoria tienen que ser lineales. El trabajo de esta tesis consiste en extender la aplicación del modelo del poliedro que hace APOLLO, a códigos que presentan accesos a memoria no lineales.

4.1. Chunks y dispatcher-chunks

Las nociones de chunk y dispatcher-chunk serán usadas en las secciones siguientes así en esta sección se procede a una explicación de las mismas. La Figura 4.1 representa el espacio de iteraciones dividido en pedazos consecutivos.

Estos pedazos representan la división del espacio de iteraciones que hace APOLLO. Cada uno de ellos es denominado *chunk*. Los chunks son ejecutados en orden secuencial. Dentro de cada chunk, la ejecución es en paralelo. Cada uno de los threads que se ejecuta dentro de cada chunk, es denominado *dispatcher-chunk*. La Figura 4.2 toma como base la Figura 4.1 e ilustra los dispatcher-chunks dentro de cada chunk.

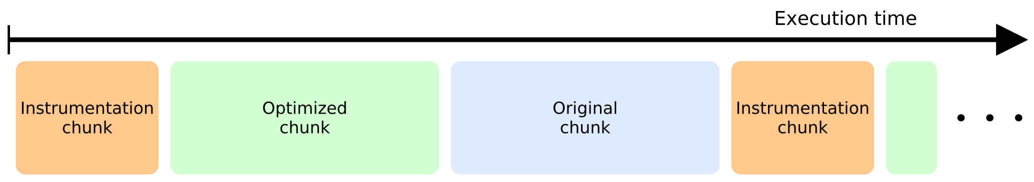


Figura 4.1: Particionado en chunks del espacio de iteraciones

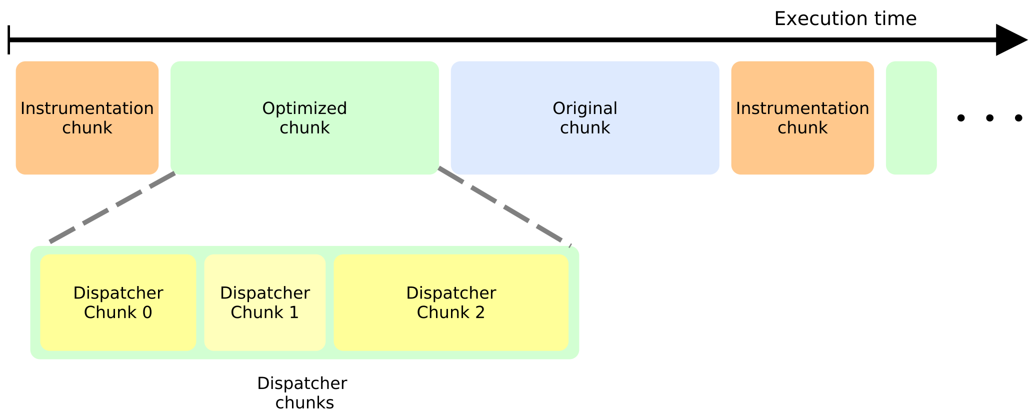


Figura 4.2: Dispatcher chunks

A partir de aquí los términos dispatcher-chunk y thread serán usados de manera intercambiada y mientras que para las porciones más grandes que se ejecutan secuencialmente, el término chunk será usado.

4.2. Accesos de lectura no lineales

Consideremos el nido de loops de la Figura 4.3

Este código pertenece al kernel de un filtro de imagen. La matriz $filter$ representa la mascara que es aplicada sobre la imagen, la cual es también representada por una matriz rectangular de enteros. Las lineas 7 y 9 inicializan en cada iteración la posición de la imagen que será usada en la computación del nuevo valor del píxel indicado por x e y.

La operación de modulo al final de cada expresión en las lineas 7 y 9 hace

```
 1 for(int x = 0; x < imageWidth; x++) {
 2   for(int y = 0; y < imageHeight; y++) {
 3     double red = 0.0  , green = 0.0, blue = 0.0;
 4
 5     for(int filterX = 0; filterX < filterWidth; filterX++)
 6       for(int filterY = 0; filterY < filterHeight; filterY++) {
 7         int imageX = (x - filterWidth / 2 + filterX + imageWidth)
 8     %imageWidth;
 9         int imageY = (y - filterHeight / 2 + filterY + imageHeight)
10     %imageHeight;
11         red += image[imageX][imageY].r * filter[filterX][filterY];
12         green += image[imageX][imageY].g * filter[filterX][filterY];
13         blue += image[imageX][imageY].b * filter[filterX][filterY];
14       }
15
16     result[x][y].r = min(max(factor * red + bias, 0), 255);
17     result[x][y].g = min(max(factor * green + bias, 0), 255);
18     result[x][y].b = min(max(factor * blue + bias, 0), 255);
19   }
20 }
```

Figura 4.3: Kernel de un filtro de imagen con lecturas de memoria no lineales

que el flujo de valores calculados por esas expresiones sea no lineal. El sistema de verificación actual va a iniciar un rollback ya que en algún momento los accesos a memoria que son indexados por las variables *imageY* y *imageX* no van concordar con sus funciones de interpolación. Sin embargo, viendo las lineas 11 a 13 puede observarse que el acceso a memoria mal predicho corresponde a una operación de escritura. Recordando la noción de dependencia explicada en la sección 2.2, si no hay una operación de escritura modificando la misma posición en algún punto de la ejecución de los ciclos, no se genera otra dependencia. En otras palabras, un acceso de lectura no lineal no va a afectar la correctitud del modelo, siempre que no haya un acceso de escritura en la misma posición en algún momento dado de la ejecución del chunk.

Con esto en mente, una primera implementación de la verificación para casos de lectura de memoria no lineales fue llevada a cabo, de la siguiente manera:

- El skeleton al encontrar una predicción errónea realiza una llamada al runtime indicando el *id* de la instrucción fallida así como la dirección de memoria en la que se encontró la no linealidad.

- Si la instrucción es de lectura, el runtime verifica que la dirección no pertenezca a los rangos de direcciones que serán modificados por las instrucciones de escritura durante la ejecución del chunk actual (estos rangos pueden ser obtenidos mediante las funciones de interpolación)

- Si la verificación es exitosa el runtime comunica al skeleton que es seguro continuar. Si no, un rollback es iniciado.

Vale aclarar que si dos accesos de lectura no lineales acceden a la misma posición de memoria, no la modifican lo cual no implica un riesgo para el model. Por lo tanto en este caso no hay necesidad de almacenar ni comparar contra las direcciones de los accesos de lecturas previos.

4.3. Accesos de escritura no lineales

La Figura 4.4 muestra un kernel de suma de matrices.

```
void kernel (int **C, int **A, int **B){
  for (i = 0; i < M; i++)
    for(j = 0; j < N; j++)
  C[i][j] = A[i][j] + B[i][j];
}
```

Figura 4.4: Kernel de suma de matrices

Si la memoria alocada para el array C no es contigua, los accesos a C serán no lineales en algún punto de la ejecución, por lo que eventualmente se producirá una mala predicción durante un acceso de escritura. Al mismo tiempo, se puede observar que esto no debería representar un riesgo para el modelo [1] ya que cada dirección que es accedida a través de alguna operación no lineal será accedida en no más que una ocasión. Es decir, no se generarán nuevas dependencias. Se implementó una estrategia con este tipo de casos en mente.

Hay algunos detalles a tener en cuenta:

1. **Tanto las lecturas como las escrituras a memoria no lineales tienen que ser registrada**, de manera opuesta a como hay que hacer cuando solo se tienen en cuenta accesos de lectura no lineales. El hecho de que la memoria es modificada introduce nuevas restricciones a tener en cuenta para asegurar

[1] Suponiendo que C no es alias de A ni de B

que el modelo especulativo sigue siendo válido: Si una lectura no lineal ocurre en la posición de memoria *pos* la cual pertenece a alguno de los rangos que serán escritos y/o leídos (según el modelo lineal) o si esa posición ya fue accedida mediante una instrucción de lectura o escritura no lineal, luego una nueva dependencia aparece, poniendo en riesgo la validez del modelo.

2. **Se debe hacer un back-up de la memoria que es modificada mediante accesos no lineales**. Si dos accesos de lectura no lineales modifican la misma dirección, se debe iniciar un rollback. Es por esto que la memoria debe ser restaurada al estado que tenía antes de comenzar la ejecución del chunk.

Siguiendo estos dos puntos, se implementó un sistema de verificación que registra los accesos no lineales y restaura la memoria en caso de que sea necesario hacer un rollback.

De la misma manera que con los accesos de lectura no lineales, la verificación se realiza mediante una llamada al runtime pasándole la información necesaria. El sistema de verificación procesará de manera distinta los accesos no lineales según sean de lectura o escritura.

Si el causante es una operación de lectura luego el análisis descrito en la sección anterior es llevado a cabo con el adicional de que la posición de memoria es guardada y que además se chequea con los accesos de escritura no lineales que, como se explica a continuación, también son registrados.

Si el acceso no lineal es causado por una operación de escritura, las siguientes acciones son llevadas a cabo.

- Se verifica que la posición de memoria no pertenezca a ningún rango de lectura ni de escritura lineal así como también se compara con las direcciones de los accesos de lectura no lineales que han ocurrido hasta el momento. Si esta verificación es exitosa, entonces la dirección es guardada por el thread que ejecutó la instrucción, se hace un back-up de la correspondiente región de memoria y se continúa con la ejecución. Cada thread cuenta con su propia tabla en la cual mantiene registro de todos los accesos de escritura no lineales.

- Los valores en estas tablas son comparados al final de la ejecución de cada chunk para ver si dos threads modificaron de manera no lineal la misma dirección de memoria. Si es así, la memoria es restaurada (con el back-up de la primera modificación hecha) y un rollback es iniciado. Luego se ejecuta el chunk original.

4.4. Chequeo rápido de rangos

En las estrategias explicadas en las secciones anteriores, se realiza un chequeo de las direcciones de los accesos no lineales contra todos los rangos de memoria que serán modificados por accesos de lectura y de escritura lineales. Teniendo la información precisa de los rangos que serán accedidos y modificados durante la ejecución del chunk actual resulta trivial calcular la posición más baja y más alta de memoria que será modificada en cada tipo de acceso (lectura y escritura). Si la posición de memoria que intento ser leída o escrita de manera no lineal no se encuentra dentro de estos dos rangos *globales*, entonces el chequeo puntual contra los rangos de escritura y lectura de cada instrucción no es necesario.

Por ejemplo, en el caso en el que solo se manejan los accesos de *lectura* a memoria no lineales, la condición mostrada en la Figura 4.5 es testeada.

```
if( addr_stmt_i <= write_UB && write_LB <= addr_stmt_i ){
  /*Chequeo mas refinado*/
}else{
  /*es seguro continuar*/
}
```

Figura 4.5: Chequeo global de rango a nivel de chunk

$write_{LB}$ y $write_{UB}$ corresponden a la cota superior y la cota inferior de *todas* las posiciones de memoria que serán modificadas, según el modelo lineal, durante la ejecución de chunk *actual*.

Para ver por que es seguro continuar con la ejecución si la dirección correspondiente al acceso no lineal no se encuentra dentro del rango global veamos los posibles resultados al chequear la condición:

1. La dirección mal predicha se encuentra dentro del rango indicado por las cotas globales.

2. La dirección mal predicha se encuentra fuera del rango global.

En el primer caso, se procede a realizar un cheque más refinado. Se compara contra todos los rangos de escritura como fue anteriormente explicado. Si la posición de memoria no pertenece a ninguno de estos rangos, es seguro continuar con la ejecución.

En el segundo caso, estamos seguros que la posición no lineal no va a interferir con las direcciones modificadas linealmente por los accesos de escritura del chunk actual, pero todavía está la posibilidad de que interfiera con las posiciones modificadas por los accesos de escritura de otros chunks. Sin embargo, como fue explicado en la sección 4.1, los chunks son ejecutados de manera secuencial con respecto a ellos, por lo que su orden relativo de ejecución siempre va a ser correcto. O sea que no hay riesgo de que una nueva dependencia aparezca entre dos chunks distintos poniendo en peligro la validez del modelo.

El chequeo de rangos rápido también es hecho en el caso de accesos de escritura no lineales.

4.5. Optimizando el registro de accesos no lineales

Proporcionar soporte para accesos de lectura y de escritura no lineales al mismo tiempo introduce la necesidad de registrar los accesos a memoria. En el caso de accesos de escritura no lineales, todas las direcciones accedidas junto con el valor original de esa posición de memoria tienen que ser guardados, para poder hacer rollback y restaurar la memoria en caso de que una nueva dependencia aparezca en el chunk actual. En el caso de lectura no lineales¡, solamente se guardan las direcciones accedidas para comparar con los accesos lineales y no lineales de escritura. El caso general es ilustrado en la Figura 4.6

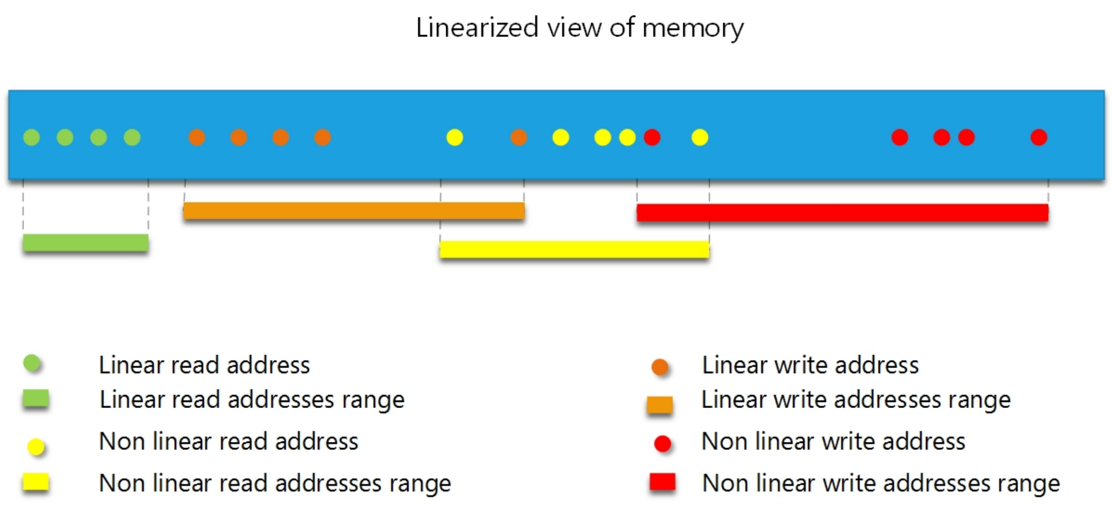
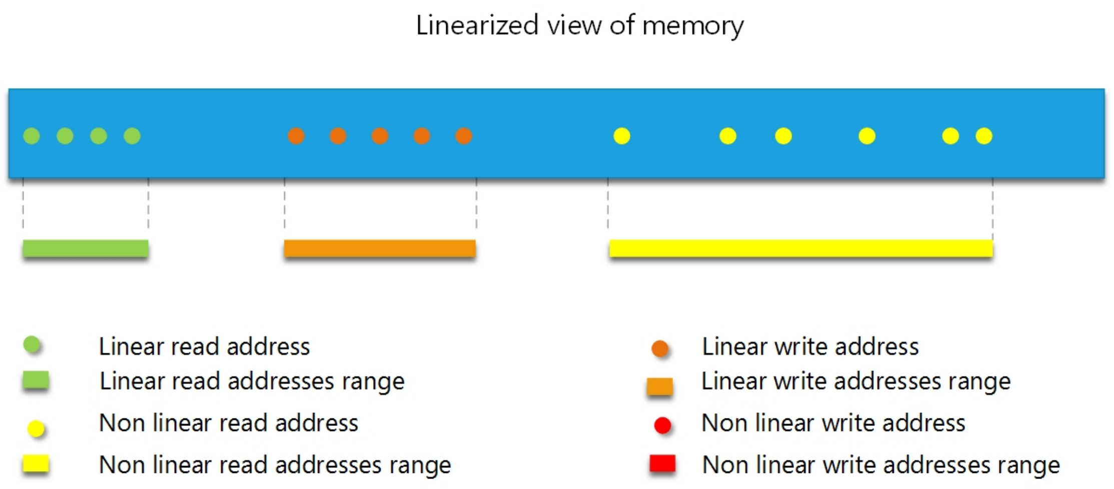

Linearized view of memory
Linear read address
Linear read addresses range
Non linear read address
Non linear read addresses range
Linear write address
Linear write addresses range
Non linear write address
Non linear write addresses range

Figura 4.6: Accesos de lectura y escritura no lineales solapados

Este registro no solo demanda más memoria sino que también más recursos de CPU para realizar las comparaciones. Sin embargo en muchos casos este registro es evitable.

4.5.1. Sólo hay accesos de lectura no lineales (junto con accesos de lectura y escritura lineales)

Como fue anteriormente explicado, para que ocurra una dependencia, debe haber al menos dos accesos a la misma posición de memoria en de los cuales al menos uno tiene que ser un acceso de escritura. Si los accesos no lineares corresponden solamente a accesos de lectura, basta con verificar estos no pertenecen a ningún rango de escritura lineal. Este chequeo es posible ya que gracias a las funciones de interpolación permiten calcular estos rangos. Teniendo esto en cuenta, se adaptó el sistema de verificación para que si durante la instrumentación no se encuentran accesos de lectura no lineales, se haga solamente un chequeo de rangos cada vez que un acceso de lectura no lineal ocurre, en vez de mantener registro de todas las direcciones accedidas lo cual requiere más cálculos y memoria. Obviamente se corre el riesgo de que un acceso de escritura no lineal ocurra. En este caso se inicia un rollback. La Figura 4.7 muestra patrón de memoria que ilustra este ejemplo.



Linearized view of memory
Linear read address
Linear read addresses range
Non linear read address
Non linear read addresses range
Linear write address
Linear write addresses range
Non linear write address
Non linear write addresses range

Figura 4.7: Solo accesos de lectura no lineales (junto con accesos de lectura y escritura lineales)

4.5.2. Ninguno de los accesos de lectura no lineales se solapa con los accesos lineales o no lineales de escritura

Para el manejo de accesos de escritura no lineales los detalles mencionados en la sección 4.3 tienen que ser tenidos en cuenta. Para reducir la cantidad de accesos de lectura que son registrados, se implementó la siguiente heurística. Si durante la instrumentación, como se ejemplifica en la Figura 4.8, se observan accesos de lectura no lineales que no se solapan con los accesos tanto lineales como no lineales de escritura (también observados durante instrumentación), entonces se toma el valor más bajo y el valor más alto observado. De esta manera solo se lleva registro de los accesos de lectura no lineales que no pertenecen a este rango. En caso de un acceso de escritura no lineal, el chequeo con los rangos de lectura no lineales se reduce drásticamente. Si bien este método carece de precisión, ya que el manejo del rango da lugar a falsos positivos, reduce el uso de CPU y accesos a memoria. En los ejemplos testeados mostró aumentos en la performance de hasta un 100 % con respecto a cuando se lleva exacto registro de todos los accesos.

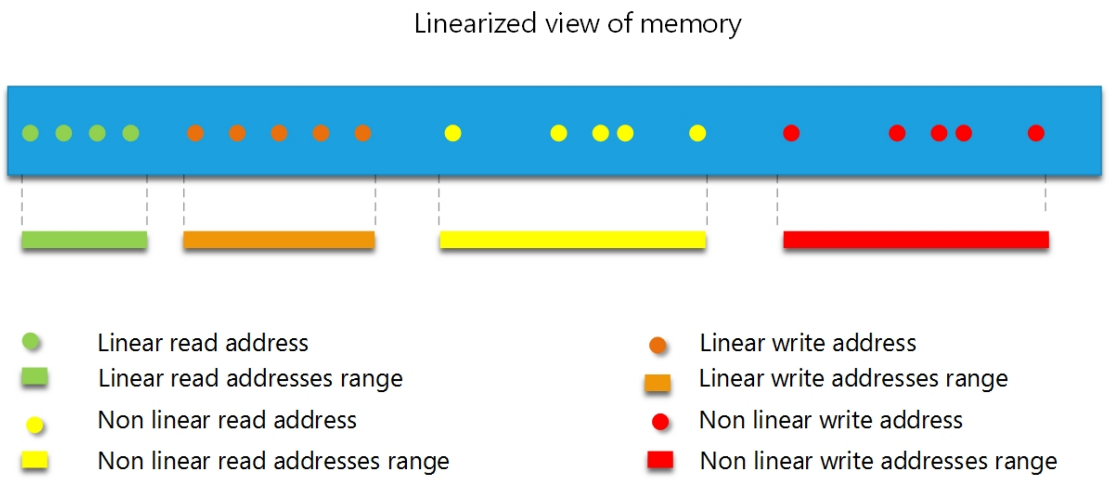

Linearized view of memory
Linear read address
Linear read addresses range
Non linear read address
Non linear read addresses range
Linear write address
Linear write addresses range
Non linear write address
Non linear write addresses range

Figura 4.8: Accesos de lectura lineales no se solapan con accesos de escritura tanto lineales como no lineales durante la instrumentación

4.5.3. Selección automática de estrategia

Basado en los 3 escenarios descriptos, se incorporó un sistema de selección automática de estrategia a APOLLO para decidir, con respecto a los accesos de lectura no lineales, entre mantener las direcciones exactas, usar la heurística del rango descrita en 4.5.2 o no salvar ninguna dirección en absoluto 4.5.1. Durante la instrumentación se observa la presencia o no de accesos de escritura no lineales. Si este tipo de accesos no ocurre, luego la estrategia que no mantiene registro de las direcciones de escritura no lineales es elegida. Si, por el contrario, se observan accesos de escritura lineales, se chequea si estos caen dentro del rango comprendido entre la dirección más baja y la dirección más alta de que fue accedida mediante lectura no lineal. Si hay solapado, entonces se llevará registro exacto de las direcciones de lectura no lineales. Si no hay solapado entonces la heurística de 4.5.2 es usada.

Vale la pena notar, que esta selección automática de estrategia extiende la flexibilidad del sistema de chunking explicado en 3.2.8 ya que, por ejemplo, si un rollback es causado por un acceso de escritura no lineal mientras la estrategia que consiste en no llevar registro de las direcciones de lectura no lineales estaba siendo utilizada, es probable que este comportamiento se mantenga también durante la instrumentación que es relanzada luego de que el rollback fue ejecutado. Esto quiere decir que en la nueva fase de instrumentación, la estrategia de verificación de accesos de lectura no lineales probablemente va a cambiar adaptándose al nuevo comportamiento de manera dinámica.

Parte III

Resultados y conclusiones

Capítulo 5

Resultados

En esta sección presentamos los resultados obtenidos al aplicar el framework de paralelización especulativa brindado por APOLLO a códigos que exhiben comportamiento no lineal (lectura, escritura o ambos) durante los accesos a memoria. La mayoría de los kernels utilizados fueron sacados de la suite de benchmarks Polybench [13]. A estos códigos se les reemplazo las variables automáticas por punteros para poder manejarlos mediante análisis dinámico. El ejemplo del filtro fue sacado de [8].

Los distintos speed-ups obtenidos al ejecutar los códigos en APOLLO están mostrados en la Figura 5.1. Están expresados en términos de cuantas veces más rápido fue el código paralelo en comparación con el código original. La comparación fue hecha contra el mismo código pero compilado con GCC con las optimizaciones de nivel 3, *-O3*. La versión de GCC usada es *4.8.2*. Los tests fueron ejecutados en una máquina con la siguiente configuración: dos procesadores AMD Opteron 6172, de 12 núcleos cada uno, con una frecuencia de 2.1 Ghz y 32 Gb de RAM.. El sistema operativo es Linux 3.2.0-36-generic x86 64.

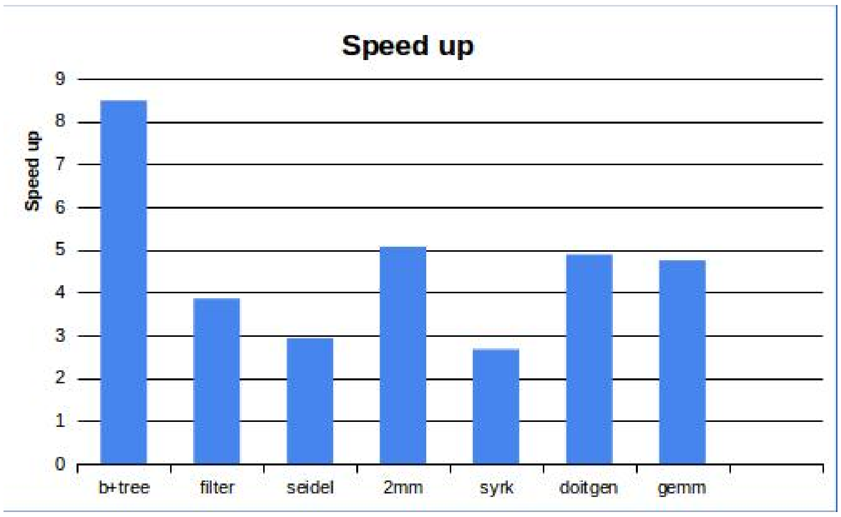


Figura 5.1: Speed-ups obtenidos

La tabla 5.1 muestra información aproximada sobre la cantidad de accesos de lectura y escritura no lineales.

Cuadro 5.1: Número aproximado de accesos de lectura y escritura no lineales

Program	Original (in sec)	Apollo (in sec)	# NL-R accesses (approximated)	# NL-W accesses (approximated)	speed up
b+tree	8.4	0.99	8128000000	0	8.48
filter	21.2	5.5	6000000000	0	3.85
seidel	12.01	4.1	35892081	3988009	2.92
2mm	36.04	7.11	65536	16512	5.06
syrk	20.75	7.76	3221225472	1073741824	2.67
doitgen	10.27	2.1	16777216	4294967296	4.89
gemm	18.06	3.8	1073741824	0	4.75

Capítulo 6

Conclusiones

El trabajo aquí presentado contribuye al tópico de paralelización especulativa extendiendo la aplicación del modelo del poliedro. Más precisamente, permiten determinar si una operación no lineal de lectura o escritura invalida las transformaciones poliedrales en las cuales la paralelización está basada. También se introdujo un sistema de selección automática que permite adaptar la granularidad del sistema de verificación de a cuerdo al comportamiento dinámico del código. Esto permite evitar hacer comparaciones y accesos a memoria lo cual representa una importante ganancia en la performance del código paralelizado por el framework. La efectividad de estas estrategias fue testeada en códigos pertenecientes a un conjunto de benchmarks bien conocidos para los cuales la performance de ejecución aumento obteniendo speed-ups de hasta 8x.

En conclusión, las extensiones realizadas en esta tesis permiten que el modelo del poliedro pueda aplicarse a ciclos que no cumplen estrictamente (tanto de manera estática como dinámica) las condiciones necesarias. Al mismo tiempo, amplían el espectro de códigos que APOLLO es capaz de paralelizar obteniendo empíricamente resultados alentadores.

Bibliografía

[1] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, 2008.

[2] Derek Bruening, Srikrishna Devabhaktuni, and Saman Amarasinghe. Softspec: Software-based speculative parallelism. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, Monterey, California, Dec 2000.

[3] Laurent Daverio, Corinne Ancourt, Fabien Coelho, Stéphanie Even, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Frédérique Silber-Chaussumier. PIPS – An Interprocedural, Extensible, Source-to-Source Compiler Infrastructure for Code Transformations and Instrumentations. Tutorial at PPoPP, Bengalore, India, January 2010; Tutorial at CGO, Chamonix, France, April 2011. `http://pips4u.org/doc/tutorial/tutorial-no-animations.pdf` presented by François Irigoin, Serge Guelton, Ronan Keryell and Frédérique Silber-Chaussumier.

[4] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[5] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, 1991.

[6] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 1 : one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[7] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 2 : multidimensional time. *International Journal of Parallel Programming*, 21(6), 1992.

[8] Image filtering. http://lodev.org/cgtutor/filtering.html., 2010.

[9] Alexandra Jimborean, Philippe Clauss, Juan Manuel Martinez, Aravind Sukumaran-Rajam, and Wolff Willy. Speculative program parallelization with scalable and decentralized runtime verification. *5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014*, 8734:pages 124–139, 2014.

[10] Alexandra Jimborean, Matthieu Herrmann, Vincent Loechner, and Philippe Clauss. Vmad: A virtual machine for advanced dynamic analysis of programs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 125–126, Washington, DC, USA, 2011. IEEE Computer Society.

[11] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[12] The LLVM compiler infrastructure. `http://llvm.org`.

[13] Polybenchs 1.0. http://www-rocq.inria.fr/pouchet/software/polybenchs.

[14] Louis-Noël Pouchet. FM: the Fourier-Motzkin library. `http://www.cse.ohio-state.edu/ pouchet/software/fm`.

[15] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[16] Easwaran Raman, Neil Va hharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 175–184, New York, NY, USA, 2008. ACM.

[17] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[18] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The suif compiler system: a parallelizing and

optimizing research compiler. Technical report, Stanford University, Stanford, CA, USA, 1994.

[redacted]

DYNAMIC MODELING OF MEMORY ACCESSES FOR EFFICIENT SPECULATIVE PROGRAM PARALLELIZATION

Tesina de grado presentada
por

Esteban Campostrini

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de
Licenciado en Ciencias de la Computación

A thesis submitted for the degree of Doctor
of Philosophy

Abstract

The high availability of multi-core processors as well as their wide range of application — from everyday gadgets as tablets and cell phones, to professional and scientific compute-intensive applications — has led to an effort towards making the most of the available hardware. In this context, code parallelization is naturally on of the main resources to exploit. Typically, loops play a central role on the execution time of a program, doing *repetitive* computations which turn them into targets when it comes to parallelizing a program. This is why many tools specially oriented to loop parallelization have been developed.

A mathematical framework, the *polytope model*, covers the theory behind loop analysis and transformation. This model is originally limited to loops that can be precisely analyzed during compile-time. This means that it cannot handle loops that, for example, contain indirect memory accesses as or pointer.

APOLLO, *Automatic POLyhedral Loop Optimizer*, is a framework which allows to perform instrumentation during run-time in order to apply the polytope model dynamically and perform a speculative parallelization. In this way, this framework allows to parallelize loops that, originally, were not able to be parallelized using traditional techniques.

APOLLO extends the polytope model to codes on which originally was not possible to be applied. Nevertheless, among the rest of the conditions the model imposes, the memory accesses have to be linear. This thesis continues the work done in APOLLO in the following way:

- Extending the modeling of read and write memory accesses done by APOLLO with the purpose of allowing the parallelization of loops in which although the not all the memory accesses are linear, the validity of the model is not put at risk.

- Incorporating a decision system which allows to choose dynamically between two strategies to handle the non-linear memory accesses, according to the non-linear behavior observed during the instrumentation.

These extensions allow to apply the polytope model on loop nests that do not strictly gather the necessary conditions (statically as well dynamically.) At the same time they widen the range of codes APOLLO is able to parallelize.

The effectiveness of the extension was evaluated empirically on codes taken from a well known benchmark suite.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context of the work

Nowadays multi-core processors are ubiquitous. Their high popularity and availability as well as the potential they have, claim for new strategies for reaching good software performance and exploit advantageously the underlying hardware. Two important aspects when it comes to this are parallelism and data locality. Even if many parallel programming languages are available, such as OpenMP, MPI, Cilk or CUDA, parallel programming is in general difficult since the programmer must handle complex issues, such as selecting a convenient algorithm for parallelization, analyzing the dependences between parts of the code, use a suitable programming model, among many others. As a result of this, the development of compilers dedicated to automatic parallelization is an area of interest in the world of parallel computing. Many compilers that put the focus into automatic parallelization have been developed, examples of such are SUIF [24] and PIPS [7].

Since generally a program spends a great deal of its execution time in loops, they have been widely studied giving place to dedicated analysis techniques. Precise data analysis makes possible for loops to take advantage of advanced parallelizing transformations as tiling, fusion or loop splitting. The theory which deals with loop analysis and transformations is unified in a well-known mathematical framework called *the polytope model* [10]. The polytope allows to capture the behavior of a program by means of affine functions and which can then be used to perform complex analysis and look for optimizing transformations. Although it is a powerful resource, it is only applicable to loops whose linear behavior can be determined at compile-time. Loops that include memory accesses through pointers or indirect array references may exhibit their linearity only at execution-time. Thus they cannot be handled by this technique during the compilation phase.

In order to overcome this limitation, a *speculative* parallelizing technique has been developed [14]. This technique takes advantage of the information available at runtime to automatically parallelize on-the-fly some code parts. The information needed to complete the polytope model is obtained by means of code instrumentation. Since a part of the model is built based on the behavior observed on a sample of the whole execution, there is no guarantee that it will hold during the whole execution since the behavior may change. For this reason an *on-line verification* system has to be implemented in order to launch recovery actions in case previously speculated information is invalidated. This recovery actions generally consist of restarting the computations from the last correct state and constitute what is commonly called a *rollback*.

In the general context of speculative parallelization, a well-researched direction is thread-level speculation (TLS) [4] [15] [19] [20] [22]. Typically a TLS framework allows optimistic execution of parallel code regions before *all* code dependences are known, keeping track of memory accesses to determine if any dependence violation occurs. Generally, these frameworks have a centralized verification system that requires great amount of inter-thread communication which goes against the basic principles of high performance parallel computing. Moreover, they generally apply a unique parallelizing strategy which consists of dividing a loop nest in several speculative parallel threads.

In this context, APOLLO, *Automatic POLyhedral Loop Optimizer* [13] provides a framework oriented to *Optimize and parallelize* code at runtime and which allows to *exploit* parallelism in codes which could not be parallelized in the original form. APOLLO allows to dynamically apply parallel code patterns called *skeletons* which allow to use different parallelizing techniques such as *loop tiling*, *skewing*, etc. It has a de-centralized verification system which provides a more efficient alternative when it comes to verifying the code parallelized speculatively. In case of a mis-prediction, a rollback is initiated and the modified memory is restored thanks to an on-line backup system. This system saves the memory regions that are going to be modified prior to the launch of the parallel code. In case of a mis-prediction a re-instrumentation is launched in order to capture the *new phase* of the code and new optimizations and transformations are applied. Figure 1.1 shows a global view of the execution flow of APOLLO.

All in all, APOLLO provides speculative parallelization by applying transformations defined in the polytope model.

The contributions of this work focus on extending APOLLO's scope to a wider class of loops which may include non-linear memory reads and writes, as well as non-linear scalars assignments. The proposed extensions of APOLLO mostly impact the verification system.

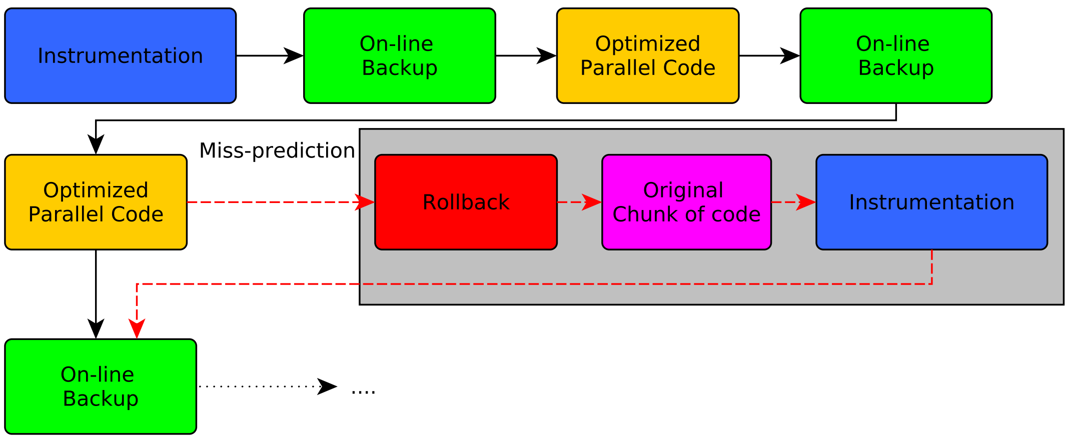


Figure 1.1: General view of the execution flow of APOLLO

1.2 Related works

As it was mentioned, a well known technique in the field of speculative parallelization is thread-level speculation (TLS). They apply different verification systems at runtime but, in general, they treat the problem of a mis-prediction on a different way from the one presented here.

In this section, we mention approaches taken by some TLS frameworks when a mis-prediction occurs.

ParExC [23] targets automatic speculative parallelization of code that has been optimized at compile time. It speculates on a failure free execution and and aborts as soon a mis-speculation is encountered, relying on a software transactional memory (STM) based solution. The areas that have to be checked for conflicts are indicated by means of explicitly STM primitives. In case of a conflict a dedicated STM system must may initiate a rollback or delay the conflicting transaction. This involves the active participation of the programmer as well as a checking for every access, things that avoided in the solution presented in the current work.

MiniTLS [25] treats mis-predictions on a different way. The verification system focus on a fast recuperation whenever a mis-peculation occurs by implementing a parallel rollback system. The extension presented here allows to avoid the rollback in many cases but it demands certain verifications that are not needed after a rollback. So, if, say, the amount of memory that has to be restored is *considerable* then continuing with the execution and doing a checking at the end, will be likely more beneficial than executing a rollback. On the other hand, if the amount of checking that has to be done at the end of the execution is extremely big and *if* the mis-peculation appeared at an early stage of the execution, the rollback may be a more beneficial alternative.

Softspec [4] Proposes speculative parallelization based on the construction of a model, as well as APOLLO, but the decision taken upon a mis-peculation is to rollback to the last correct state of the execution. Thus, with the extension presented in the current work, APOLLO outnumbers the types of codes that is able to handle without having to do a rollback.

The rest of the document is organized as follows: Section 2 gives an introduction to the polytope model, Section 3 explains the APOLLO framework in detail and Section 4 presents the contributions made in this thesis.

Part I

Preliminars

Chapter 2

The polytope model

The polytope model is a geometrical and arithmetical framework which allows to capture the execution of a program in a compact way. It is widely used in loop nest optimization since, among other things, it provides an homogeneous representation of loop information necessary for performing precise dependence analysis. In geometry, a polyhedron represents a geometric object with flat sides, which can exists in any number of dimensions. A bounded polyhedron is a polytope. In our case, we are only addressing convex polytopes.

Abstracting a loop in the polyhedral representation is equivalent to associate to each statement of the loop a polyhedron in which each integer point represents an instance of the statement during each loop iteration. An exhaustive presentation of applications of the polytope model in program optimizations are given by Feautrier in multiple works [8, 9, 10, 11]. For a complete presentation of the mathematical apparatus building the underlying background of the polytope model, the reader is referred to the monograph of Schrijver [21]. In what follows we present only an overview.

The polytope model can only be applied for loop nests where:

- The loop bounds are affine functions of the enclosing loop indices and parameters.
- Each test predicate is an affine function of the enclosing loop indices and parameters.
- The data structures in the nest are multi-dimensional arrays and scalars of any type.
- The array subscripts are affine functions of the surrounding loop indices and parameters.

2.1 Definitions

We denote by $\vec{v}$ a vector, by $|\vec{v}|$ its dimension, and by $\vec{v}$[i] the i^{th} element of $\vec{v}$.

Definition 1 (*Affine function*) A function $f : K^n \rightarrow K^m$ is said to be affine iff $\exists$ a matrix $A \in K^{n\times m}$ and a vector $\vec{b} \in K^n$ such that:

$$\forall \vec{x} \in K^n,\ f(\vec{x}) = A\vec{x} + \vec{b}.$$

Definition 2 (*Affine hyperplane*) An affine hyperplane of an n-dimensional affine space V is a subspace of dimension $n-1$, defined by a linear equation in $\vec{x} \in K^n$ of the form:

$$\vec{a} \cdot \vec{x} = b,$$

where $\vec{a} \in K^n$ (at least one element $\vec{a}$[i] $\neq 0$) and b is a scalar from K.

Definition 3 (*Affine half-space*) An affine hyperplane divides the space into two *half-spaces*, defined by the inequalities:

$$\vec{a} \cdot \vec{x} \geq \mathrm{b} \quad \text{and} \quad \vec{a} \cdot \vec{x} \leq \mathrm{b}$$

where $\vec{a} \in K^n$ (at least one element $\vec{a}$[i] $\neq 0$) and b $\in K$.

Definition 4 (*Convex polyhedron*) The intersection of a finite number of *affine* half-spaces defines a *convex polyhedron*, each half-space providing a face of the polyhedron. Formally, the polyhedron $P \subset K^n$ can be expressed as a set of m affine constraints in $A \in K^{m\times n}$ and $\vec{b} \in K^m$:

$$P = \{\vec{x} \in K^n | A\vec{x} + \vec{b} \geq 0\}$$

Definition 5 (*Parametric polyhedron*) A polyhedron P may be parametrized by a vector of parameters $\vec{p}$ and is denoted by $P(\vec{p})$. It can be defined by a matrix $A \in K^{m\times n}$, a matrix of symbolic coefficients $B \in K^{m\times p}$, where p is the dimension of the vector of parameters $|\vec{p}| = p$ and a vector of constants $\vec{b} \in K^m$ as:

$$P(\vec{p}) = \{\vec{x} \in K^n | A\vec{x} + B\vec{p} + \vec{b} \geq 0\}$$

Definition 6 (*Polytope*) A bounded polyhedron is called a *polytope.*

Definition 7 (*Iteration vector*) The *iteration vector* of a statement S, denoted by $\vec{x_S}$, is the n-dimensional vector of values of the iterators of the n loops enclosing S.

```
for(i = 1; i ≤ N; ++i) {
    for(j = 1; j ≤ N; ++j) {
        S: A[i][j] = A[i-1][j] + A[i][j-1]
    }
}
```

Figure 2.1: Loop nest example for the iteration domain.

2.2 Representing loops in the polytope model

For a precise analysis of each statement within the model, three pieces of information are needed: its iteration domain, the memory access it performs and the execution order in relation with the other statements. These three parts are explained in the following subsections.

2.2.1 The iteration domain

Let us consider the loop nest in figure 2.1:

Notice the statement labeled as S. Even if S is a single statement, it has several statement instances along the life time of the loop, namely $S_{(1,1)}$, $S_{(1,2)}$, ..., $S_{(N,N)}$. Each statement instance has an iteration vector (i, j) associated to it. The set of all iteration vectors for a statement is called *the iteration domain*. The iteration domain of S is:

$$D_S = \{(i,j) \in Z^2 | 1 \leq i \leq N, 1 \leq j \leq N\}$$

It is defined by a set of constraints. If these constraints are affine functions of the enclosing loop indices and parameters, they define a *polytope*. The matrix representation of D_S, taking $N = 10$, is the following and its geometrical representation is shown in Figure 2.2:

$$\left(\begin{array}{cc|c} 1 & 0 & -1 \\ -1 & 0 & 10 \\ 0 & 1 & -1 \\ 0 & -1 & 10 \end{array}\right) * \left(\begin{array}{c} i \\ j \\ \hline 1 \end{array}\right) \geq 0$$

2.2.2 Data access functions

In the polytope model, the memory access of a statement must be expressed in terms of an affine function of the enclosing loop indices.

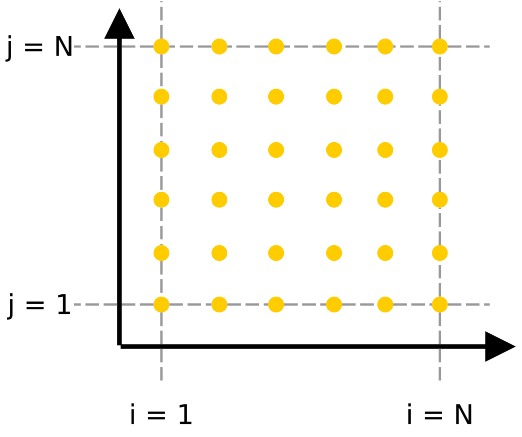


Figure 2.2: Representation of the iteration domain D_{S_1}.

Statement S has one write operation corresponding to $A[i][j] = \ldots$ and two read operations corresponding to $A[i-1][j]$ and $A[i][j-1]$. The access functions corresponding to these memory accesses are:

$$f_{W_A}(i,j) = \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$f_{R_{A_1}}(i,j) = \begin{pmatrix} i-1 \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

$$f_{R_{A_2}}(i,j) = \begin{pmatrix} i \\ j-1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

2.2.3 Statement scheduling

Neither the domain, nor the data access functions provide an execution order for a statement. The *scheduling function* maps the points in the iteration space D_S to multidimensional *time stamps*, i.e., it assigns a logical date to each dynamic instance of every statement. In general, for a statement S, its time stamp θ_S is defined by:

$$\forall \vec{x} \in D_S \ \theta_S(\vec{x}) = T\vec{x} + \vec{t}.$$

Where T is a $d \times d$ matrix, $\vec{t}$ is a d vector, and d is the depth of statement S.

The associated timestamps allow to order the instructions according to the lexicographic order, denoted as $\ll$, as component-wise comparison of vectors:

$$(a_1, \dots, a_n) \ll (b_1, \dots, b_n) \Leftrightarrow \exists i : 1 \leq i \leq n, \forall m : 1 \leq m < i, \, a_m = b_m \wedge a_i < b_i$$

The scheduling function for statement S in the example is:

$$\theta_{S_1}(i, j) = (0, i, 0, j, 0)$$

The constants between the iterators allow to express the order between statements at the same loop level. If there was another statement executed after S at the same loop level its scheduling function would be $(0, i, 0, j, 1)$

Applying the scheduling function $\theta(\vec{x}) = T\vec{x} + \vec{t}$ to the integer points of an iteration domain $\mathcal{D} = \{\vec{x} | A\vec{x} + \vec{b}\}$, is expressed as the polyhedron:

$$\left(\begin{array}{c|c} I & -T \\ \hline 0 & A \end{array}\right) \left(\begin{array}{c} \theta(\vec{x}) \\ \hline \vec{x} \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \left(\begin{array}{c} \vec{t} \\ \hline \vec{b} \end{array}\right)$$

Scheduling matrices

The scheduling function θ^S of a statement S is defined by a scheduling matrix Θ^S of S, such that:

$$\theta^S(\vec{x}) = \Theta^S \vec{x} = \vec{ts}$$

where $\Theta^S \in Z^{d^t \times (d+p+1)}$, with $d^t = |\vec{t}|$, $d = |\vec{x}|$, $p =$ dimension of the global variables vector, and $\vec{ts}$ is the resulting time stamp vector.

Canonical representation of the scheduling matrices

To make the scheduling matrix more meaningful, Cohen *et al.* [2, 6] propose a normalized representation. The encoding purposed is suitable for expressing compositions of transformations, as it is decomposable in three sub-matrices:

1. *The iteration ordering matrix* $A^S \in \mathcal{M}_{d^S, d^S}(Z)$ representing the iteration vectors;

2. *The matrix of parameters* $\Gamma^S \in \mathcal{M}_{d^S, d_g p+1}(Z)$, where $d_g p$ denotes the number of global parameters;

3. *The statement ordering vector* $\beta \in N^{d^S+1}$, which specifies the order of S among the other statements executed at the same iteration.

The structure of the canonical schedule matrix is:

$$\Theta^S = \left[\begin{array}{ccc|ccc|c} 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_0^S \\ A_{1,1}^S & \cdots & A_{1,d}^S & \Gamma_{1,1}^S & \cdots & \Gamma_{1,p}^S & \Gamma_{1,p+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_1^S \\ A_{2,1}^S & \cdots & A_{2,d}^S & \Gamma_{2,1}^S & \cdots & \Gamma_{2,p}^S & \Gamma_{2,p+1}^S \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{d,1}^S & \cdots & A_{d,d}^S & \Gamma_{d,1}^S & \cdots & \Gamma_{d,p}^S & \Gamma_{d,p+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_d^S \end{array}\right]$$

In this form, various transformations can easily be expressed: affecting A^S it is possible to define a loop interchange, skewing or loop reversal; altering Γ^S can define shifting transformations; and modifying β^S redefines the execution order of the instructions, equivalent to performing loop fission or loop fusion.

The schedule for the statement S it given by the following matrices:

$$A^{S_1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \beta^{S_1} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \Gamma^{S_1} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

The time stamp for S_1 is given by:

$$\left(\begin{array}{cc|c|c} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right) * \left(\begin{array}{c} i \\ j \\ \hline N \\ 1 \end{array}\right) = \begin{pmatrix} 0 \\ i \\ 0 \\ j \\ 0 \end{pmatrix}$$

2.3 Dependence Analysis

To be valid, the loop transformation must preserve the semantics of the original code.

Definition 9 (*Dependence of Statement Instances*) Two statements S and R are said to be dependent, if there exists two instances $S(\vec{x_S})$ and $R(\vec{x_R})$, where $\vec{x_S}$ and $\vec{x_R}$ belong to the iteration domains of S and R, such that $S(\vec{x_S})$ and $R(\vec{x_R})$ access the same memory location and at least one is a write. To preserve the semantics, the execution order of two dependent statements must be the same in the original sequential and in the transformed parallel order. On the other hand, two independent statements can be executed in arbitrary order, and this in parallel.

Note that when the two statements are read operations, there is no dependence since the memory is not modified. However such accesses to common memory locations can still be considered to improve data locality.

Dependences are classified in three categories, depending on the order of the read and write operations:

- RAW: read-after-write, or *flow dependence.*
- WAR: write-after-read, or *anti dependence.*
- WAW: write-after-write, or *output dependence.*

2.3.1 Dependence vectors

$R(\vec{j})$ dependes on $S(\vec{i})$ means that $S(i)$ is executed before $R(j)$. In *loop dependence analysis* [1], this is equivalent to say that iteration j of loop L depends on iteration i, where L contains statements S and R. The dependence between R and S is characterized by:

- the *distance vector* $\vec{d} = \vec{j} - \vec{i}$;
- the *direction vector* $\sigma = \mathbf{sig}(\vec{d})$;

  where
  $$sig(i) = \begin{cases} 1, & if\ i\ >\ 0, \\ -1, & if\ i\ <\ 0, \\ 0, & if\ i\ =\ 0. \end{cases}$$
  The sign of a vector $\vec{d} = (d_1, d_2, ..., d_m)$ is $\text{sig}(\vec{d}) = (\text{sig}(d_1),\ \text{sig}(d_2),\ ...,\ \text{sig}(d_m))$;
- the level $\mathbf{l} = \mathbf{lev}(\vec{d})$.

  Given that m is the depth of the loop L, for a distance vector $\vec{d} = (d_1, d_2, ..., d_m)$, the *leading element* is the first non-zero element. If this is d_l, then l represents the *level* of $\vec{d}$. The vector $\vec{d}$ is said to be *lexicographically positive* or *negative* if its leading element is positive or negative, respectively.

A distance vector or a direction vector of a dependence must always be lexicographically non-negative for a schedule to be semantically valid.

Using the distance vectors, one can compute the *dependence matrix* of the loop L, whose rows are the distance vectors of all the dependences in L.

Schedule validation

Once the dependences between iterations are computed, one can apply polyhedral transformations (schedules) such that no dependence violations occur.

To validate a schedule θ one computes the scalar product between each of the transformation matrices Θ_S and the dependence matrix. If in the resulting matrices, the first non-null component of each row is positive, the schedule is valid. This first strictly positive component defines the depth of the loop which carries the dependence. The outermost parallel loop level is given by the first column in the resulting matrix for which no loop is carrying a dependence at this depth.

2.3.2 Example of a transformation

Recall the example of figure 2.1. For a given iteration (i, j), to compute $A[i][j]$, it must read the values produced by iteration $(i - 1, j)$ and $(i, j - 1)$. This is represented by the dependency vectors $\vec{d_1}$ and $\vec{d_2}$ and the dependence matrix d_M.

$$\vec{d_1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \vec{d_2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$d_M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Notice that with these dependences there is no parallel loop since each loop carries a dependence. If we apply the transformation defined by the transformation matrix $T = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, we obtain:

$$d'_M = T * d_M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$d'_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} d'_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

The first non-null element of each new dependence vector is positive, so the transformation is valid. Also the outermost loop carries both dependences, thus, the innermost loop can be executed in parallel.

By applying this transformation we obtain a new iteration space with two new iterators, x and y such that $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} i + j \\ i \end{pmatrix}$. This geometrical transformation is

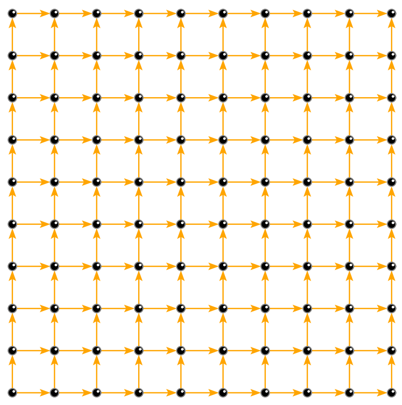


Figure 2.3: Representation of the iteration domain D_{S_1} with its dependences.

represented in figures 2.3 and 2.4, while the final resulting code is shown in figure 2.5.

The bounds of the loops are calculated using the Fourier-Motzkin algorithm. For the sake of brevity, since explaining the algorithm would incur in a long section which has no much relation with the work here presented, we will give an intuitive explanation of the reason of the bounds. For the outermost loop lets consider the new iterator, x, in terms of i and j:

$$x = i + j \text{ and } 1 <= i, j <= N \Rightarrow 2 <= i + j <= 2 * N \Rightarrow 2 <= x <= 2 * N$$

In this way we have the new bounds for the outermost loop. As for the innermost one, lets try to see the rational behind the bounds given by Fourier-Motzkin. According to the skewed iteration space given by the transformation and depicted in Figure 2.3, the parallel execution can be achieved starting from the bottom values and going up in columns. For example, for the first iteration of the outermost loop the points of the iteration space that are going to be visited are $(2, 1)$. For

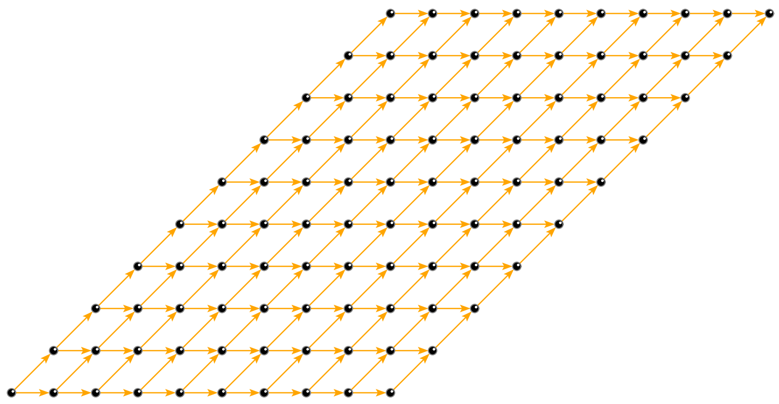


Figure 2.4: Representation of the new iteration domain for S_1 with its dependences.

```
for(x = 2; x ≤ 2N; ++x) {
    forall(y = max(1, x−N); y ≤ min(N,x−1); ++y) {
        i = y
        j = x − y
        S: A[i][j] = A[i−1][j] + A[i][j−1]
    }
}
```

Figure 2.5: Resulting code after applying the code transformation.

the second iteration of the outermost loop, the points are going to be: $(3, 1)$ and $(3, 2)$; for the third one $(4, 1)$, $(4, 2)$ and $(4, 3)$ and so on. Because of the skewing, they y iterator is going to assume 1 as its initial value just until the value of x is greater than N. Than can also be seen in Figure 2.3. From that on, the initial value of y always has to be incremented by one. This explains the intuition behind choosing **max(1, x - N)** as the lower bound. Regarding the upper bound a sim-

ilar reasoning can be made. One more time, taking Figure 2.3 as a reference and recalling that the parallel traversing order of the iteration space is by columns, we can see that y goes, during each iteration, one step further, i.e. its last value is incremented by one for each subsequent column. The figure also shows two more things: first, the las value it assumes it is always one less than the value of x, and second, when it reaches the value of N, it remains there for the rest of the iterations. This intuitively suggest the following expression as upper bound:
min(N, x - 1).

Chapter 3

The APOLLO framework

3.1 Limitations of the polytope model

Originally, the polytope model was applied for compile-time loop optimization and parallelization. It worked with static codes which are compliant with the model. However, the static application of the polytope model is limited by factors such as: bounds that cannot be statically predicted, complex flow controls, presence of unresolved pointers, etc.

Although not statically analyzable, these codes may still exhibit a linear behavior at runtime which complies with the polytope model. Thus, if enough information regarding the memory access patterns and loop bounds can be gathered, then polyhedral dependence analysis and parallelizing code transformations can be performed on these kind of codes. In order to extract the program behavior at runtime, a small set of loop iterations is instrumented (Instrumentation by sampling). Since the construction of the model is based on a small part of the execution, any change on the code behavior could affect the validity of the transformation. This calls for some form of verification that ensures the continuous correctness of the model, more precisely:

- Runtime verification is required to validate the speculative code transformations;
- An on-line recovery system, is triggered upon a mis speculation;
- The system must be lightweight enough to shadow the runtime overhead.

In what follows, the APOLLO framework which applies the polytope model in a speculative way, with all the requirements mentioned above, is presented.

3.2 APOLLO

APOLLO (Automatic POLyhedral Loop Optimizer) is a framework for speculative loop parallelization and instrumentation. As a polyhedral optimizer it applies polyhedral optimizations prior to parallelization and as a speculative parallelization framework it handles non-statically analyzable loop nests and memory accesses. It achieves the latter by means of generation of different versions of a same piece of code making it suitable to different kinds of executions and controls, e.g., instrumentation or parallelization. In the following sections, a detailed overview of the the APOLLO framework as well as how the polytope model is adapted for speculative parallelization are presented.

3.2.1 APOLLO Architecture

APOLLO is implemented as a static-dynamic framework and consists of two main parts: a static part implemented as passes of the **LLVM** compiler [16], and a dynamic part implemented as a system written in C++.
The procedure for using APOLLO is the following: the programmer annotates the source code using a specific *pragma* as shown in figure 3.1. This source code is compiled using the APOLLO compilation suite which runs on top of the *clang-llvm* C compiler. The compilation produces a binary ready to be executed. The front-end can be easily interchanged with another language front-end which outputs *llvm-ir*, making APOLLO available to a wide set of programming languages.

```
#pragma apollo dcop
{
    loop 1
        loop 2
            ...
}
```

Figure 3.1: Loop nest tagged for speculative parallelization.

The *static component* of the framework is implemented as a series of compiler extensions. The main purpose of these components is to generate several code versions from each target loop nest:

- One version is dedicated to profiling (See section 3.2.3). This version includes code to capture dynamic data of the execution of the loop nest through instrumentation.
- Another version, corresponding to the original version of the loop nest.

- Several code-skeletons matching different kinds of code transformations.

The *runtime component* is in charge of collecting dynamic information during the actual loop execution. Using this information, the runtime system builds interpolating linear functions to describe the behavior of that loop. Afterwards, dependence analysis is performed and an adequate code-skeleton is selected to be interpolated and to be executed. For achieving this, several steps are performed by the runtime component. Each of these are isolated in modules that have a very specific purpose, for example: doing the backup of the memory that is going to be modified during the speculative execution, calculating the linear functions corresponding to the different memory accesses, performing the dependence analysis, and so on.

3.2.2 Multi-versioning

As it was mentioned before, APOLLO produces different versions of the targeted region of code. The idea behind multi-versioning is to compile and embed in the binary different versions of a region of code. Having a set of versions of a loop nest allows one to select dynamically the version that suits best the current execution behavior. APOLLO presents a mechanism for switching from one version of code to another one without interrupting the execution of the program.

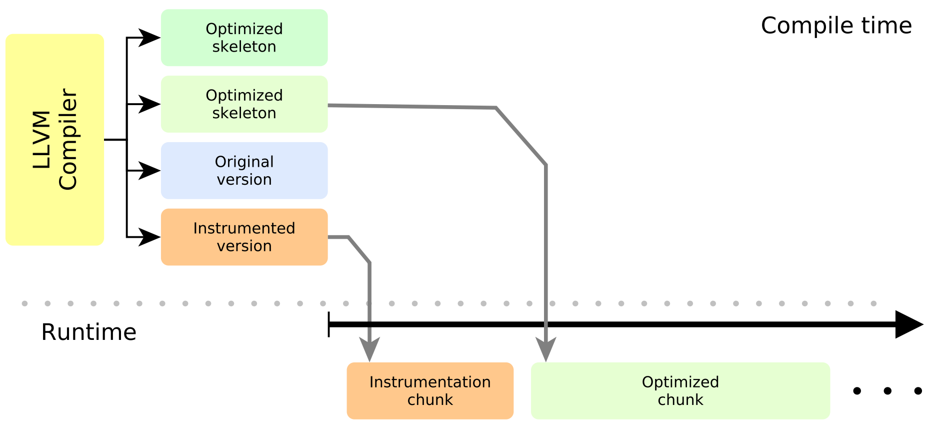


Figure 3.2: Alternating the execution of different versions, during one run of the loop nest.

APOLLO uses multi-versioning for generating four types of versions of each targeted loop nest: an original version, an instrumented version, and several code skeletons supporting different classes of optimizing transformations. At the moment of the execution of the loop nest, the runtime system will switch between the different code versions to achieve different behaviors.

3.2.3 Instrumentation of loop nests

The strategy followed to apply the polyhedral model to loop nests that cannot be statically analyzed relies on speculation. This speculation consists of observing initially the original code during a short sample of the whole run. If this sample exhibits a polyhedral behavior then the assumption that the behavior will remain the same during the whole execution is taken. Of course a runtime verification is required to check that this speculation still holds, otherwise there is the risk of not preserving the semantics of the code.

Non-statically available information is collected through an *online profiling phase* which is intended to monitor and instrument the memory accesses as well as the loop bounds. In order to avoid the overhead caused by instrumentation, only a sample of a small number of iterations is instrumented.

Provided that the number of iterations executed by the loop is high, the instrumentation cost becomes negligible in practice, and widely compensated by the optimization.

During the instrumentation phase, APOLLO collects the following data:

- Values taken by some scalars.
- Addresses of the accessed memory locations.
- The number of iterations executed by the inner-loops.

The values collected are used for building interpolating linear functions.

3.2.4 Virtual Iterators

In order to handle uniformly any kind of loops, e.g. for loops or do-while loops, additional counters called *virtual iterators* are inserted. There is one virtual iterator for each loop in the nest and they are expressed in a canonical form. The initial value they take is 0 and they are incremented with a step of 1. Although they have a special importance when it comes to handling loops which do not have iterators (as *do-while* or *while* loops), the virtual iterators form the mathematical referential domain of the dynamic polyhedral model interpolated in APOLLO. Thus, they are the variables of the interpolating functions and are used to apply new structures to the original loop nest.

3.2.5 Parallelizing code transformation

Once the prediction model is built, APOLLO's runtime performs a dependence analysis in order to determine if the target loop nest can be parallelized and what transformation can be applied for this purpose. A polyhedral transformation consists of changing the order in which the iterations are executed (i.e. the order in which the iteration domain is traversed.) The transformation is represented as a matrix and is obtained by passing the collected information (affine functions and dependencies) to Pluto [3] which is the polyhedral parallelizer that takes care of finding the transformation. Only the scheduler kernel of Pluto is used and it has been slightly customized to consume the dependence analysis of APOLLO.
Given a loop nest of depth two with iterators $\binom{i}{j}$ and a transformation matrix T, polyhedral transformations such as loop skewing or loop interchange are obtained as:

$$T \times \binom{i}{j} = \binom{x}{y}$$

For obtaining the loop bounds in the transformed space an implementation of the Fourier-Motzkin elimination algorithm is used through the FMlib library [18].

3.2.6 Verification of the speculative polyhedral parallelization

In order to ensure that the prediction model used during the speculative execution still holds, some characteristics must be verified during the optimized chunk execution. To achieve this, dedicated code is inserted in the parallel code skeletons at compile-time, and instantiated at runtime. This code will trigger a rollback as soon as the verification fails. The three aspects of the execution that are verified are: the memory accesses, the scalars, and the loop bounds.

Memory access verification

The prediction model represents the sequence of the addresses accessed by an instruction as an affine function of the virtual iterators. This linearity of the memory access allows to perform a dependence analysis which in time allows to apply optimizing transformations to the target code, that will be semantically correct as long as the predicted dependencies hold. Hence, verifying that the predicted dependencies are correct translates to verifying that all memory accesses follow their associated affine function. This is done by comparing, for each memory instruction, the actual accessed address, against the value predicted by the interpolating affine function. If this comparison fails, a more specific checking is performed by the runtime. The developing and implementation of this checking is the main subject of this thesis and it will be explained in detail in the following chapter. It is worth to mention, that the original implementation was initiating a rollback.

Scalars initialization and verification

Scalar variables such as counters and accumulators have to be taken into account as well since they also carry dependences across iterations. Scalars typically take an initial value which is updated among the different loop iterations. In order to remove their associated dependences by predicting their values, they must exhibit an affine behavior as well, so a linear interpolating function can be constructed. APOLLO first tries to do this at compile time by using the *scalar evolution pass*[1]. In case this pass fails to statically produce the corresponding affine expression, instrumentation by sampling is performed by the runtime in order to collect the successive values the scalar assumes during the different iterations and to construct a speculative affine function.
As with the memory accesses, a verification, has to be performed on all scalars to be sure they are being correctly modeled.
Since the code transformations may not follow the original iteration order, scalar variables must be initialized at their correct starting values in the header of each iteration. These scalar initializations are speculative so they must be verified. The scheme used by APOLLO for doing this is: at the very end of each iteration, the prediction for the next iteration initial value is compared to the actual value of the scalar. If the verification fails, a rollback is triggered and the original code is executed.

Loop bounds verification

The polyhedral model imposes loop bounds to be affine functions of the enclosing loop iterators. These bounds can either be extracted at compile time by the scalar evolution pass or must be built at runtime through interpolation and handled speculatively. When unknown at compile time, the bounds cannot be used by APOLLO to analyze and transform speculatively the code. But as it was previously mentioned, the target loop nest is split into chunks consisting of slices of the outermost loop, and the bounds of this chunks are used instead. When the original loop exit condition is met during the run of a chunk and before its completion, a rollback is initiated and the last chunk is run again in the serial original order. In this way, loops such as whiles with unknown exit conditions can be handled. When outermost loop bound can be discovered statically the last chunk is adjusted to the precise size in order to avoid the final rollback.

[1]`http://llvm.org/devmtg/2009-10/ScalarEvolutionAndLoopOptimization.pdf`

3.2.7 Code Skeletons

The interpolating functions and distance vectors obtained with the information collected during the instrumentation phase are, in time, employed for dependence analysis so as to select a proper parallelizing transformation. In order to generate parallel code at almost no cost, a technique based on code skeletons is used. A code-skeleton is a parametrized loop-nest derived from the original user code. One code skeleton can instantiate different transformations —interchange, skewing, etc...— just by changing its parameters. Thus the cost of code-generation is exactly the cost of setting these parameters.

Skeletons consist of three parts: the first part scans the transformed iteration space and recovers the values of the virtual iterators in the original space; the second part performs the original computation; and the third one verifies that the speculations are still correct. Since they play an essential role in the way program transformations are carried out by APOLLO, a more detailed explanation is given.

Figure 3.3 shows a typical matrix multiplication kernel

```
for (i = 0; i < M; i++)
  for(j = 0; j < N; j++)
    for(k = 0; k < P; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Figure 3.3: A matrix multiplication kernel

As it was mentioned, a skeleton is a code pattern, or template, that takes parameters, providing a compromise between flexibility and fast code generation. More specifically, a skeleton will receive the transformation matrix T, its inverse T^{-1}, and the loop bounds in the transformed space which are calculated by means of the Fourier-Motzkin algorithm.

Going back to the multiplication kernel, the loop bounds in the transformed space, from outermost to innermost loops are: $FM_{lb_x}()$ and $FM_{ub_x}()$, $FM_{lb_y}(vi_x)$ and $FM_{ub_y}(vi_x)$, $FM_{lb_z}(vi_x, vi_y)$ and $FM_{ub_z}(vi_x, vi_y)$ [2]. Thus, the first part of the skeleton, scans the transformed iteration space, and recovers the values of the virtual iterators in the original space, using the inverse of the transformation matrix T. It is represented in Figure 3.4 in source code for pedagogical reasons, although code skeletons are actually manipulated in LLVM IR form by APOLLO.

[2] FM represents a call to the implementation of the Fourier-Motzkin algorithm mentioned in section 3.2.5

```
 1
 2  for ($vi_x = FM_{lb_x}()$; $vi_x <= FM_{ub_x}()$; $++vi_x$){
 3      for ($vi_y = FM_{lb_y}(vi_x)$; $vi_y <= FM_{ub_y}(vi_x)$; $++vi_y$) {
 4          for ($vi_z = FM_{lb_z}(vi_x, vi_y)$; $vi_z <= FM_{ub_z}(vi_x, vi_y)$; $++vi_z$) {
 5              $\begin{pmatrix} vi_i \\ vi_j \\ vi_k \end{pmatrix} = T^{-1} * \begin{pmatrix} vi_x \\ vi_y \\ vi_z \end{pmatrix}$
 6
 7              //initialization code
 8              i = $a * vi_i + b_i$
 9              j = $a * vi_i + b * vi_j + b_j$
10              k = $a * vi_i + b * vi_j + c * vi_k + b_k$
11              ...
```

Figure 3.4: Obtaining the values of the iterators in the original iteration space

Note that since virtual iterators where used, lines 8, 9 and 10 obtain the values of the original iterators through the interpolating functions.

The next two steps consist of verifying that the speculations made by the model are correct, and to perform the computation in the original space. Figure 3.5 continues with the skeleton body and illustrates this part. The skeleton also performs other verifications in order to ensure the legality of the ongoing speculative execution. In the case of the matrix multiplication example, the loop bounds and the scalars, which in this case are the original loop iterators, have to be verified. Figure 3.6 illustrates these verifications.

3.2.8 The chunking system

As mentioned earlier, the strategy followed to apply the polyhedral model to code with dynamic memory behavior consists of finding the underlying affine model of these dynamic portions of code by means of instrumenting a determined number of iterations. Provided that parallelizing optimizations can be applied to the loop nest, several parallel transformed versions are generated from the code skeletons. It is worth to mention that during its execution, a code may display several phases, characterized by different memory behaviors. APOLLO handles this by performing a re-instrumentation so as to generate new transformations and parallel code which adapts to the new current behavior. All in all, the code ends up split in several subparts called *chunks* which are a set of consecutive iterations of the outermost

```
...
//calculation of the predictions
$pred_C = \vec{v_{pred_C}} * \begin{pmatrix} vi_i \\ vi_j \\ vi_k \end{pmatrix} + b_{pred_C}$
$pred_A = \vec{v_{pred_A}} * \begin{pmatrix} vi_i \\ vi_j \\ vi_k \end{pmatrix} + b_{pred_A}$
$pred_B = \vec{v_{pred_B}} * \begin{pmatrix} vi_i \\ vi_j \\ vi_k \end{pmatrix} + b_{pred_B}$

//memory access verification
if(&C[i][j] != $pred_C$) rollback()
if(&A[i][k] != $pred_A$) rollback()
if(&B[k][j] != $pred_B$) rollback()

//original computation
C[i][j] = A[i][k] * B[k][j]
...
```

Figure 3.5: Verifying the speculated memory accesses and performing the original computation

loop, many of which may represent different versions of the same part of the code. Figure 3.7 illustrates this. With the code divided in many subparts, there has to be some mechanism that takes care of coordinating the execution of these chunks by linking the different subparts in an appropriate way. To achieve this, APOLLO utilizes a *chunking mechanism.*

The chunking is orchestrated by APOLLO in the following way:

- At startup, a small chunk running the instrumented version is launched in order to build the prediction model and perform dependence analysis.
- The transformation suggested by Pluto from the dependence information is then used to instantiate the code skeletons which are in charge of the corresponding class of transformations.
- The memory locations predicted to be updated by the model are backed-up and the parallel code is launched inside a chunk (which is significantly larger than the instrumentation chunk.)
- If the verification of the speculation detects an unpredicted behavior, the execution of the current chunk is canceled, memory is restored and a chunk

```

//loop bounds verification
if((j <= P) != ($vi_j$ < $ub_{vi_j}$ )) rollback()
if((k <= N) != ($vi_k$ < $ub_{vi_i}$ )) rollback()

//compute the next iteration
$\begin{pmatrix} vi_{i_{next}} \\ vi_{j_{next}} \\ vi_{k_{next}} \end{pmatrix} = \begin{pmatrix} vi_i + (vi_j == ub_{vi_j} \,\&\&\, vi_k == ub_{vi_k}) \\ vi_j + 1 * (vi_j != ub_{vi_j}) \\ vi_k + 1 * (vi_k != ub_{vi_k}) \end{pmatrix}$

//scalar verification
if ( i != $\vec{v_i} * \begin{pmatrix} vi_{i_{next}} \\ vi_{j_{next}} \\ vi_{k_{next}} \end{pmatrix} + b_i$) rollback()
if ( j != $\vec{v_j} * \begin{pmatrix} vi_{i_{next}} \\ vi_{j_{next}} \\ vi_{k_{next}} \end{pmatrix} + b_j$) rollback()
if ( k != $\vec{v_k} * \begin{pmatrix} vi_{i_{next}} \\ vi_{j_{next}} \\ vi_{k_{next}} \end{pmatrix} + b_k$) rollback()
```

Figure 3.6: Verifying iterators and loop bounds

which contains the original serial version is launched in order to overcome the faulty execution point.

- Finally, a new instrumented chunk is launched again to capture the changing behavior and build a new prediction model.

If no mis-prediction is detected during the running of a parallel chunk, a next chunk using the same parallel code and running the next slice of the loop nest is launched.

3.2.9 Some notes

In the case of the presence of global variables or function calls that may cause side effects that go beyond the scope of the current loop nest, then APOLLO skips the corresponding loop nest. Those cases can be detected before doing any poliedral analysis of the code. For example, callings to IO functions are easily identifiable and global variables are going to be treated only if they are modified just by the current module, which means that presence of variables marked as *volatile* causes the framework to skip the current loop nest.

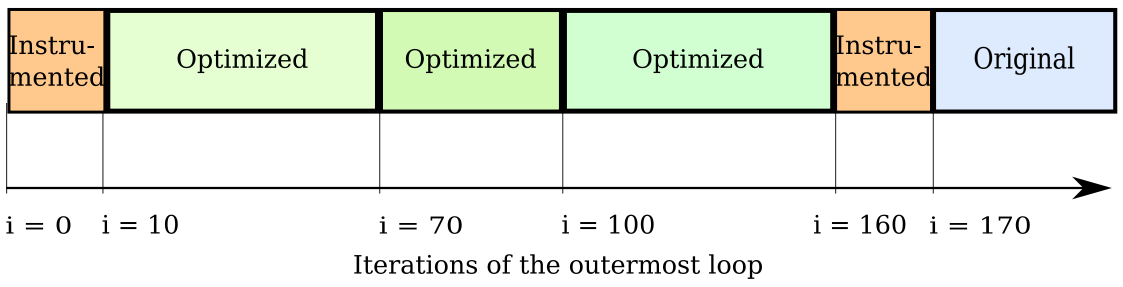


Figure 3.7: The execution of the code split in subparts called *chunks*

3.2.10 General overview

To conclude the introduction to APOLLO, a general overview of the system is depicted in Figure 3.8.

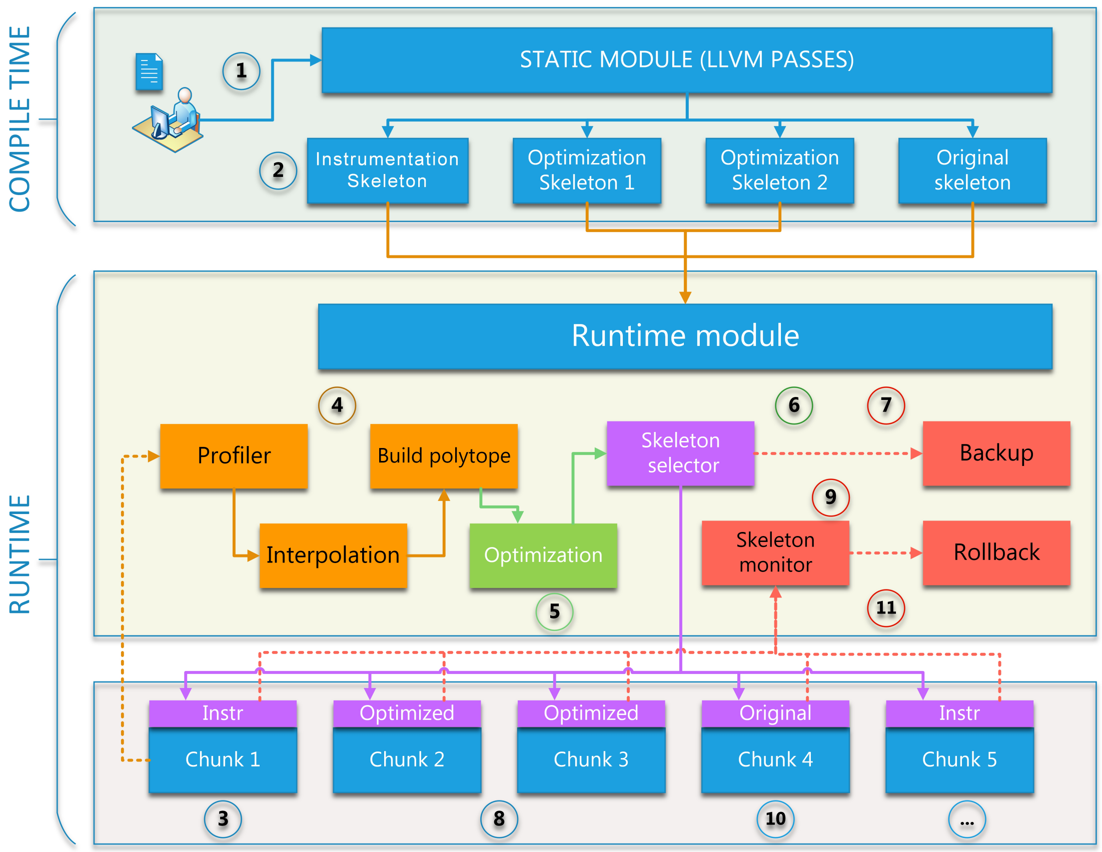


Figure 3.8: General overview of APOLLO

Part II

Contributions

Chapter 4

Modeling non-linear behavior

As explained in previous chapters, the polytope model, although very powerful, is limited to a particular class of compute-intensive codes, whose behavior can be precisely characterized by a set of linear functions. Nevertheless, codes that do not perfectly fit into the polytope model may still benefit from it. By means of on-line profiling, APOLLO overcomes the limitations of static analysis, allowing to apply the polytope model in cases where it was originally impossible due to missing information. Even though this extends the applications of the polytope model at runtime, it still carries others limitations, in particular, the code being modeled has to exhibit exclusively an *affine behavior*. In what follows, two strategies aimed at overcoming this limitation in APOLLO are introduced.

4.1 Motivation

APOLLO relies on instrumentation by sampling for building the linear prediction model for the loop bounds and the memory accesses. This makes the model speculative, since it is based on a small observation of the execution of the program. Consequently, a verification system [13] is implemented to check that the assumptions made during the polyhedral transformations hold and the semantics of the target code is not put at risk due to a unpredicted behavior. This verification is part of the code skeletons, as was explained in section 3.2.7.

As shown in Figure 3.5, if the value predicted by the corresponding interpolating function does not match the observed code behavior, a rollback is triggered. The recovery process explained at the end of section 3.2.8 is launched. A general overview of the process is depicted in Figure 4.1

The main reason for launching the recovery process is that when a mis-prediction

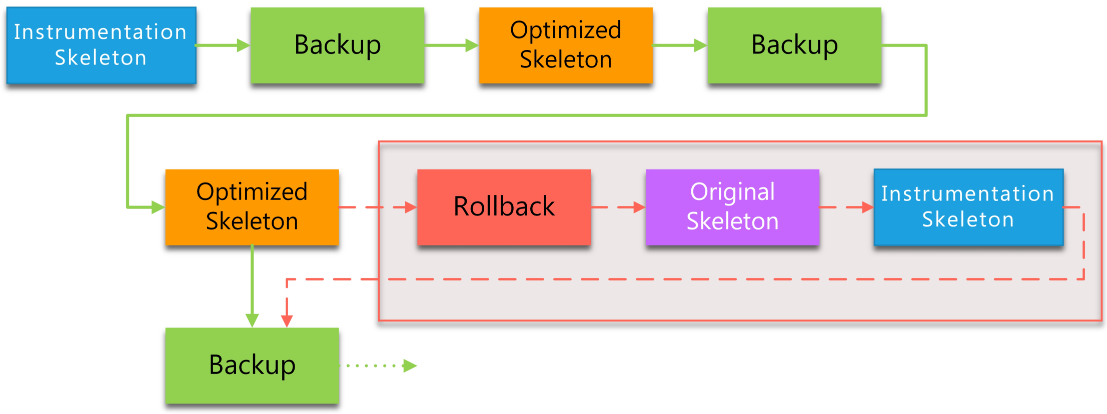


Figure 4.1: General overview of the execution flow in terms of *chunks*

occurs, memory can store incorrect values and new dependencies that were not taken into account during the generation of the transformations may arise rendering the model unsafe. Although this approach ensures the correctness of the executed code, it is rather conservative and it will generate a rollback even in situations in which no new dependencies were introduced by the mis-predicted instruction or when the new dependences do not invalidate the current parallel schedule. The contribution of the current work was to extend APOLLO in order to handle a class of non-linear memory accesses. In what follows, the approaches developed for non-linear *writes* and *reads* are explained.

4.2 Chunks and dispatcher-chunks

The notion of *chunk* and *dispatcher-chunk* are going to be used in the following Sections so they are explained here.

Let us consider the chunked representation of the iteration space presented in Figure 4.2.

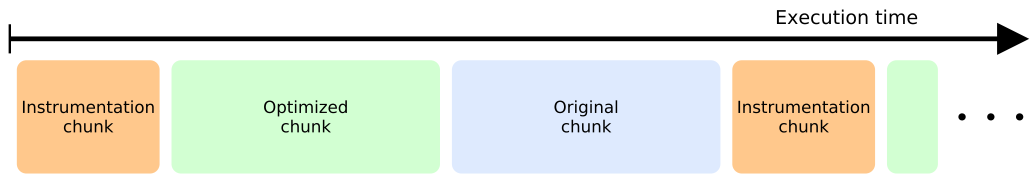


Figure 4.2: Partitioning in chunks of the iteration space

According to the chunking system of APOLLO, the chunks in which the space is divided are going to be run in sequential order regarding each other (i.e. the instrumentation chunk is going to be run first, followed by the optimized chunk and so on.) But inside each optimized chunk, the computations are going to be run in parallel. Each of the threads that is going to be run in parallel within each of the chunks is called *dispatcher chunk*. Figure 4.3 takes as a reference Figure 4.2 and adds the dispatcher chunks to it.

From now on, in order to refer to the small parallel chunks, the term *dispatcher chunk* is going to be used. Otherwise, the term *chunk* is going to be employed.

4.3 Non-linear Reads

Let's consider the following loop nest shown in Figure 4.4:

This code corresponds to the kernel of an image filter. The matrix $filter$, represents the mask that is applied over the image, which is also represented by a rectangular matrix of integer values. Lines 7 and 8 initialize at each iteration the position of the image that is going to be used in the computation of the new value of the pixels indicated by x and y.

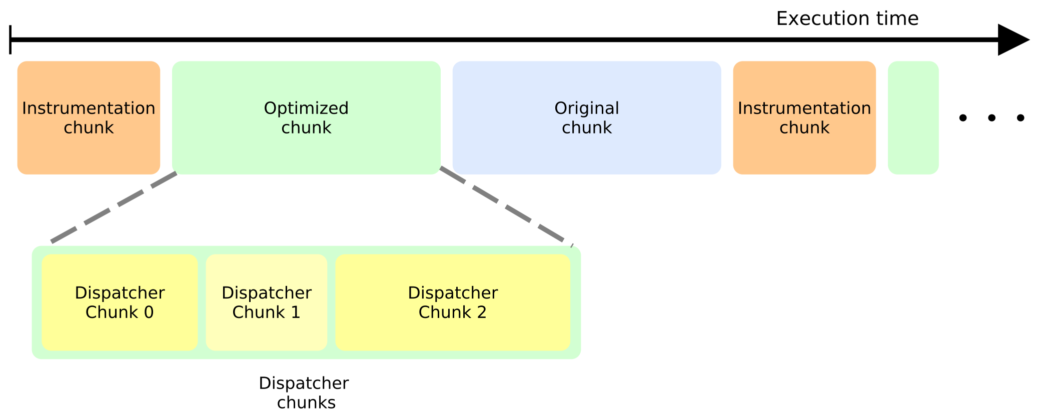


Figure 4.3: Dispatcher chunks

The modulo at the end of each expression makes the stream of values, non-linear. The current verification system will eventually launch a rollback since the memory accesses that are indexed by *imageY* and *imageX* will not match, at some point, with the predicted values obtained by the interpolating functions. However, by looking at lines 9 to 10, it can be observed that the mis-predicted memory accesses occur during a *read* operation. Recalling the notion of dependency explained in section 2.3, if there is no write statement modifying the same memory location at any point of the loop nest execution, then there is no new dependency. In other words, a read mis-prediction will not affect the correctness of the model, as long as there is no other statement writing in the same memory location.

The cost of checking this absence of interaction is lower than the cost of stopping the execution of all the parallel threads, load the backed-up values and launching the serial original version, leading to an opportunity for improving the performance achieved by the framework. This led to a first implementation of a runtime verification system which in case of a detected mis-prediction of a memory position that is being read performs the following:

- A call to the runtime is made, indicating the id of the faulty instruction and the memory position *addr* that was mis-predicted.
- Provided the instruction is a read, the runtime checks that *addr* is not part of the memory modified by the *write* instructions.
- If the verification is successful then the runtime communicates to the main

```
for(int x = 0; x < imageWidth; x++) {
  for(int y = 0; y < imageHeight; y++) {
    double red = 0.0 , green = 0.0, blue = 0.0;

    for(int filterX = 0; filterX < filterWidth; filterX++)
      for(int filterY = 0; filterY < filterHeight; filterY++) {
        int imageX = (x - filterWidth / 2 + filterX + imageWidth)
          % imageWidth;
        int imageY = (y - filterHeight / 2 + filterY + imageHeight)
          % imageHeight;
        red += image[imageX][imageY].r * filter[filterX][filterY];
        green += image[imageX][imageY].g * filter[filterX][filterY];
        blue += image[imageX][imageY].b * filter[filterX][filterY];
      }

    result[x][y].r = min(max(factor * red + bias, 0), 255);
    result[x][y].g = min(max(factor * green + bias, 0), 255);
    result[x][y].b = min(max(factor * blue + bias, 0), 255);
  }
}
```

Figure 4.4: Kernel of an image filter

program that it is safe to continue with the current parallel schedule.

Notice that even if two mis-predicted *reads* access the same memory location, they do not modify it so this does not implies a risk for the model. Therefore, there is no need to compare against the addresses of the previous mis-predicted reads. Even when the runtime verification is called from several threads at the same time, information is not modified, eliminating any chance of having inconsistent data. A general view is shown on Figure 4.5 and a more detailed view of the new verification part of the skeleton, based on the *C-like* skeleton presented on section 3.2.7 , is shown in Figure 4.6.

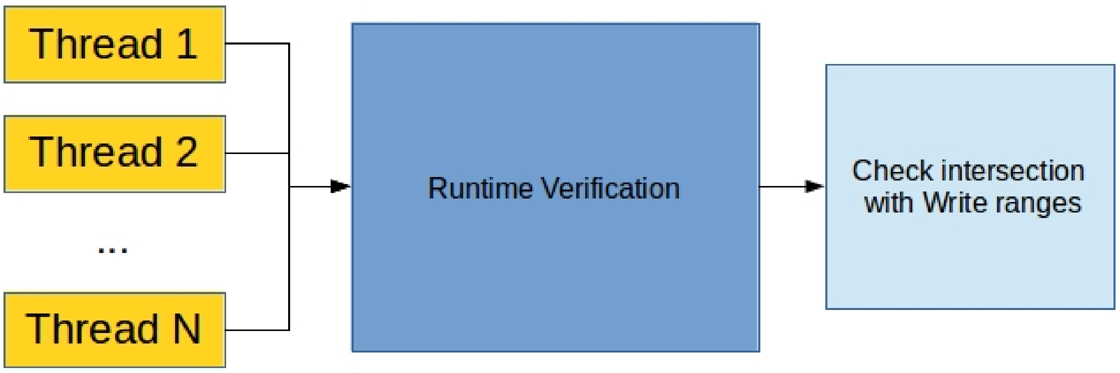


Figure 4.5: De-centralized read verification scheme

```
...
$pred_C = \vec{v_{pred_C}} * \begin{pmatrix} vi_i \\ vi_j \\ vi_k \end{pmatrix} + b_{pred_C}$
$pred_A = \vec{v_{pred_A}} * \begin{pmatrix} vi_i \\ vi_j \\ vi_k \end{pmatrix} + b_{pred_A}$
$pred_B = \vec{v_{pred_B}} * \begin{pmatrix} vi_i \\ vi_j \\ vi_k \end{pmatrix} + b_{pred_B}$

//memory access verification
if(&C[i][j] != $pred_C$)
        if(!apollo_run_verif_stmt($C_{id}$, &C[i][j])
                rollback()
if(&A[i][k] != $pred_A$)
        if(!apollo_run_verif_stmt($A_{id}$, &A[i][k])
                rollback()
if(&B[k][j] != $pred_B$)
        if(!apollo_run_verif_stmt($B_{id}$, &B[k][j])
                rollback()

C[i][j] += A[i][k] * C[k][j]
...
```

Figure 4.6: New verification applied inside the code skeleton

4.4 Non-linear Writes

Let us consider the matrix addition code kernel showed in Section 4.7

```
void kernel (int **C, int **A, int **B){
  for (i = 0; i < M; i++)
    for(j = 0; j < N; j++)
        C[i][j] = A[i][j] + B[i][j];
}
```

Figure 4.7: A matrix addition code kernel

If the memory allocated for array C is not contiguous, then accesses to C will not be linear at some point of the execution. Thus, a mis-prediction will appear during a write operation. By looking further at the code, it can be observed that this should not represent a risk to the model [1] since each address that is accessed through some non-linear operation will be accessed at most once. In the previous version of APOLLO this scenario would have triggered a rollback. In what follows, an extension to handle mis-predicted *writes* is presented.

In order to handle non-linear writes, more details than with non-linear reads have to be taken into account:

1. **All non-linear reads as well as non-linear writes have to be tracked** as opposed to what is done when handling just non-linear reads. The fact that memory is being modified introduces new constraints in order to ensure the validity of the speculative model: if a mis-predicted write occurs at a memory position *pos* which belongs to the ranges that are going to be read or written (according to the model) or which was already accessed by a non-linear read or non-linear write, then a dependence that was not taken into account by the model appears (*flow dependency* or *output dependency* or *anti-dependence*), putting at risk the legality of the model.

2. **Modified memory must be backed-up** . If an *output dependency* appears because of two mis-predicted writes to the same memory address, the model is no longer safe as it was explained in the previous point, and a rollback should be performed. The memory positions modified by mis-predicted write operations were possibly not taken into account by the back-up that was done prior to the execution of the current chunk by APOLLO, thus those changes must also be reverted.

[1]Assuming C does not aliases with A nor with B

According to these two points, a verification system was implemented which tracks the non-predicted operations and restores the memory when a rollback is required.

Every time a mis-predicted operation occurs a call to the runtime system is performed as depicted on Figure 4.6. The runtime system processes the mis-prediction differently depending on whether the prediction failure was caused by a read instruction or by a write instruction.

If the culprit was a read instruction, then the analysis described in the previous section is performed with an additional step that stores the memory address in a table.

If the responsible of the failure is a write instruction that tried to write at the memory location *loc*, then the following steps are performed:

- A fast range check is carried out. Here a comparison between the lower address and upper address that will be read and written during the chunk execution is performed. If *loc* does not fall within those ranges, then it is safe to continue.

- If the previous check fails, then a more refined check against the ranges of *each* read and write instruction of the current chunk is performed. Provided this check is successful (i.e. *loc* does not falls into any of those ranges) then the write operation is registered by the thread that executed it and the execution continues. Each thread handles one table on which it keeps its non-linear writes along with a global time-stamp [2]. The values in these tables are compared at the end of the execution of the current chunk to see whether two threads modified the same memory location. If this is the case the memory is restored with the value that contains the lower time-stamp and a rollback is executed.

Inside each chunk, several threads are executed. Thus, provided that non-linear writes are allowed, it is possible that two different threads will cause a mis-prediction by trying to write at the same memory address. This is why each thread keeps a time-stamp of each non-linear write.

4.4.1 Some notes

It is worth to note that, for example in the example presented in this section, the memory position being accessed is what is being taking into account for building the affine interpolating function. So, lets consider a version of the code from above which seems to overcome a priori this problem, presented in Figure 4.8:

[2]The time-stamp is taken from a global integer counter

```
for(i = 0; i < M; i++)
  int *CC=C[i], *AA=A[i], *BB=B[i];
  for(j=0; j < N; j++)
    CC[j] = AA[j] + BB[j];
```

Figure 4.8: Another version of the above code

At first sight it looks like the memory accesses performed by *CC* are linear but when we decompose the accesses with pointer arithmetic we have that:

```
CC == C[i] == *(BaseAddressOfC + i)
```

and

```
CC[j] == *(BaseAddressOfCC + j) == *(*(BaseAddressOfC + i) + j)
```

On the other hand, taking the memory access from 4.7, we have:

```
C[i][j] == *(*(BaseAddressOfC + i) + j)
```

since what we are accessing is a dynamically defined array. Thus, both memory accesses are performed in the same way and the *indirection* remains in both cases.

4.5 Fast range check

The verification strategy explained in the previous section consists of checking the mis-predicted address against the ranges that every write instruction of the current chunk is going to modify. Thanks to the interpolating functions, it is possible to speculatively predict the memory range that each write operation is going to modify during the execution of the current chunk. Having this information, it is trivial to obtain the minimum global address and the maximum global address that are going to be modified by linear writes and reads within the execution of the current chunk. If the memory address that caused the mis-prediction does not fall in this *global* range, then it is safe to continue. If this verification is successful, then the checking against *all* the ranges that will be modified is avoided, thus leading to a faster verification. Essentially, before checking against all the write ranges, the condition shown in Figure 4.9 is tested.

```
if( addr_stmt_i <= write_UB && write_LB <= addr_stmt_i ){
  /*more refined check*/
}else{
  /*it is safe to continue*/
}
```

Figure 4.9: Condition for a more general check

$write_{LB}$ and $write_{UB}$ correspond to the lower bound and upper bound of *all* the memory positions that are going to be modified during the execution of the *current* chunk.

To see why it is safe to continue with the execution if the mis-predicted address falls outside the global range, let us consider the the possibles result for the condition showed in Figure 4.9.

1. The mis-predicted address lies within the range indicated by the write bounds.

2. The mis-predicted address lies outside of the range indicated by the bounds.

In the first case, a function call to the runtime is made as it was explained in the previous section. The verification performed is the same, the mis-predicted address is checked against all the write ranges. If no intersection is found, then it is safe to continue.

In the second case, we are sure that the mis-predicted value is not going to interfere with the addresses modified by the *writes* of the current chunk, but still there is the possibility that it may interfere with the addresses accessed by write instructions that reside in another chunk. Here the risk consists of having a *new* **read-after-write** or a **write-after-read** dependency. Nevertheless, as it was explained in Section 4.2, the chunks are executed sequentially with respect to each other. So, for both operations [3] that are executed in different chunks, their relative execution order is always going to be correct. Thus, whenever a new dependence arises between them we are sure it is going to be respected.

This fast range checking is also performed in case of a non-linear write. In this case both, linear *write* and linear *read* ranges have to be taken into account.

4.6 Optimizing the non-linear book-keeping

Handling non linear accesses (supporting both non-linear writes and non-linear reads at the same time) introduces the need for additional book keeping of memory

[3]It can be the same operation but instantiated differently

accesses. For non linear write accesses, all the addresses accessed along with their original memory values have to be saved, so that if the prediction fails, a rollback can be performed. For all non linear read accesses, only the accessed address have to be saved; in-order to check the prediction validity against linear and non linear writes. The general case is shown in Figure 4.10

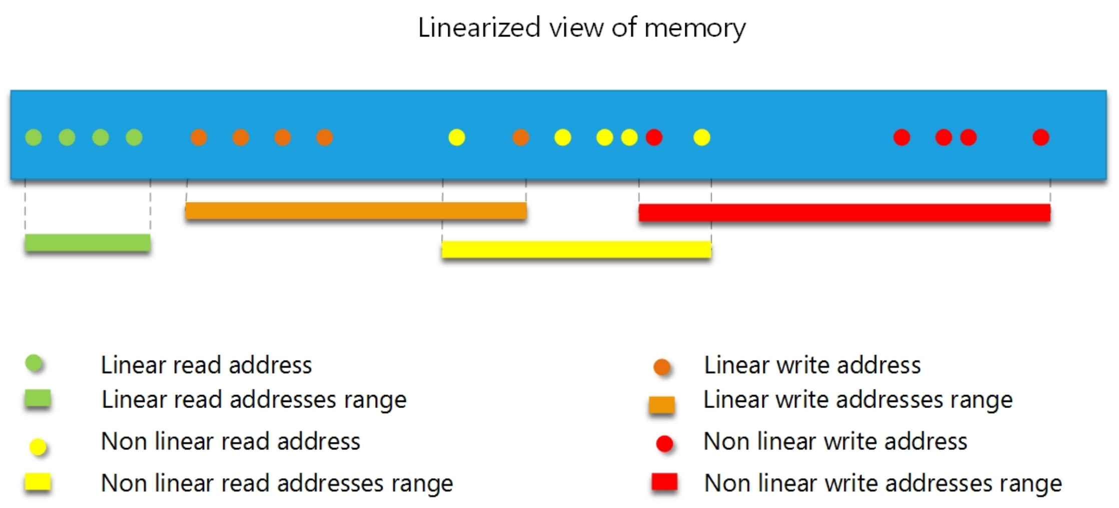

Linearized view of memory
Linear read address
Linear read addresses range
Non linear read address
Non linear read addresses range
Linear write address
Linear write addresses range
Non linear write address
Non linear write addresses range

Figure 4.10: Overlap between non-linear read access with linear and non-linear writes

This book keeping not only demands more memory but also more CPU resources for comparisons. However, in many cases the entire book keeping is not required.

4.6.1 Only non linear read accesses are present (along with linear read and write accesses)

As explained in the previous sections, for a dependence to occur, there should be at least one read access and at least one write access to the same memory location. If only non linear read accesses are present, the system has to verify that, none of the non linear read accesses overlap with any of the linear write accesses. Thanks to the linear predicting functions, APOLLO has the exact information regarding the linear write regions accessed by a chunk. If during instrumentation there are not non-linear writes, then the verification systems does only a range checking every time a non-linear read occurs, instead of keeping track of the address which requires more memory and computations. If the test turns out to be true, a rollback is triggered. If there is a non-linear write then a rollback it is also triggered. The expected memory behavior is depicted in Figure 4.11

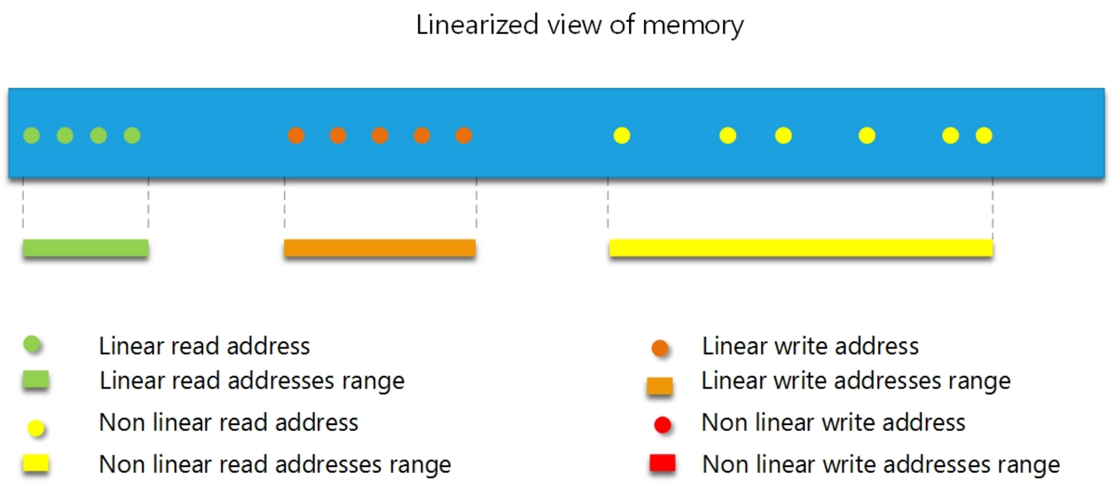

Linearized view of memory
Linear read address
Linear read addresses range
Non linear read address
Non linear read addresses range
Linear write address
Linear write addresses range
Non linear write address
Non linear write addresses range

Figure 4.11: Only non-linear reads are presented(along with linear reads and writes)

4.6.2 None of the non linear read access ranges overlap with any of the linear or non linear write ranges

If non-linear writes appear during instrumentation then the details mentioned in Section 4.4 have to be taken into account. Nonetheless, a faster checking can be achieved by replacing the exact book-keeping of non-linear read addresses with just a single range per thread (which consists of the lower bound and the upper bound of non-linear read accesses.) The verification against the linear read ranges is still necessary but for non-linear reads the computations are drastically reduced. Although this method lacks precision, as only a range is saved giving space to false rollbacks, it saves CPU time and reduces the memory accesses as well as the memory usage. The expected memory behavior is depicted in figure 4.12

Just keeping the range, typically improves the performance by at-least 100% in most cases, when compared against keeping the exact accesses.

4.6.3 Automatic strategy selection

An automatic mechanism was incorporated into APOLLO to decide between saving the exact addresses or just the range, for the non-linear read case. During the

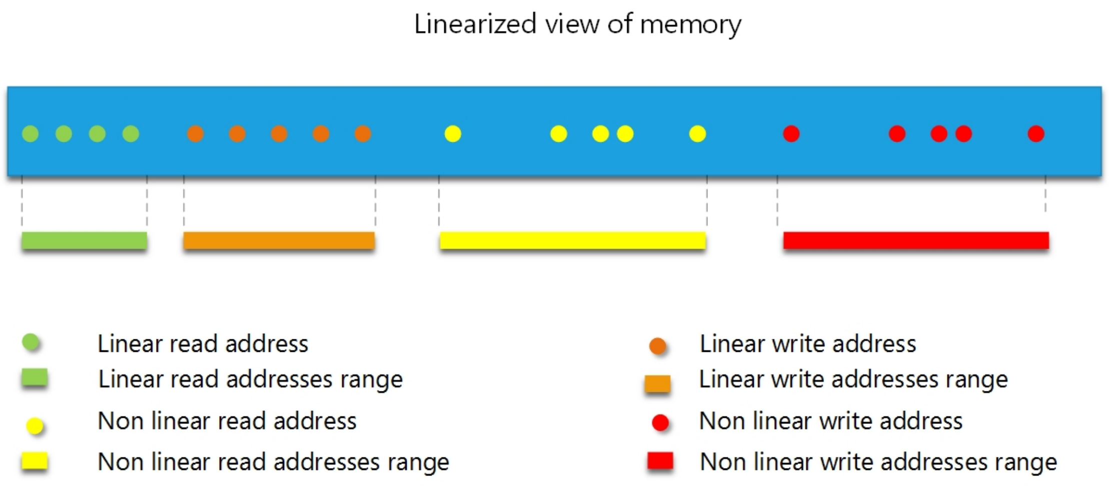

Linearized view of memory
Linear read address
Linear read addresses range
Non linear read address
Non linear read addresses range
Linear write address
Linear write addresses range
Non linear write address
Non linear write addresses range

Figure 4.12: No overlap between non-linear access with linear and non-linear writes

instrumentation phase, at first, the presence of any non linear writes is tested. If there are no non linear write accesses, then the strategy which requires no book keeping information is selected.

If non linear writes are present, then exact data for non linear reads are checked against the write ranges (linear and non linear). If this succeeds, then the same check is repeated with range information of all non linear read accesses. If the exact checking succeeds and so does the range checking, then APOLLO just keeps the range of all non linear reads; otherwise, APOLLO keeps the exact addresses accessed.

Notice that these two strategies also extend the flexibility of the chunking system explained in Section 3.2.8, since for example, if a rollback is caused by a mis-predicted write while the strategy explained in 4.6.1 was being used, then the behavior will probably be the same during the new instrumentation chunk launched after the rollback was performed. This means that during the new instrumentation phase the strategy explained in Section 4.6.2 will be selected, adapting the verification system to the new behavior.

Chapter 5

Ongoing and future work

Following the lines of the extensions presented in the previous chapter, one interesting further development is to enable parallel execution when there are non-linear scalars in the body of the loop. There is already work in progress in this direction, and the main ideas behind it are going to be presented in this chapter.

5.1 Extending the scalars verification

The strategies presented so far focus on managing dynamically different cases where the memory accesses do not strictly follow the linear prediction model inferred by the framework. Memory accesses are one kind of instructions that are considered by the prediction model. Another kind of access corresponds to what are called *scalars*[1]. Scalars are variables that are typically updated between 2 iterations of the loop they are part of. For APOLLO, the scalars of particular interest are those required to maintain the control flow or to compute memory addresses. These scalars, as well as the memory accesses, are analyzed statically by *scalar evolution* or instrumented if the analysis fails. They are detected at compile-time since they are represented by phi-nodes in the SSA form of the LLVM Intermediate Representation.

```

for (int i = 0; i < no_of_nodes; i++) {
  ptr = ptr->next;
  foo = ... ptr->bar ...
}
```

Figure 5.1: Scalar used to access a memory location

[1]In the context of APOLLO they are called *basic scalars*

In figure 5.1 the l-value of line 3, *ptr*, represents a basic scalar. Its value is updated at each iteration and it is used to perform a memory access. It also carries a read-after-write dependence since the value of *ptr* assigned at the previous iteration is used. In order to parallelize the loop this RAW dependence has to be removed by predicting the values of *ptr*. For this purpose APOLLO interpolates, if possible, the values of *ptr* using a linear function[2].

If, for example, the pattern of addresses on which the different nodes in the *linked list* reside do not follow an exactly linear behavior, like the one showed in the Figure 5.2, then the framework will not be able to handle the loop nest.

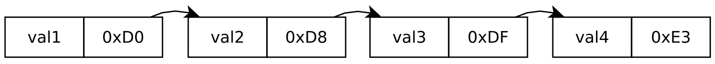


Figure 5.2: Quasi-linear behavior

The technique proposed here consists of modeling the behavior of a quasi-linear scalar by means of a regression line, allowing to parallelize the loop into parallel slices of the original loop.

5.1.1 Modeling quasi linear scalars

Let us suppose that there is a scalar whose values evolve following an almost linear pattern similar to what is shown in the Figure 5.3.

Patterns of values that are too distant are not taken into account since they would produce a very unbalanced regression line. For this purpose, a bound for the maximum possible distance between the points and the regression line is set, giving the *tube-like* form shown in Figure 5.4.

The steps performed to recognize and to process a *quasi-linear* scalar are described in the following.

[2]A consequence of a scalar responding to a linear behavior is that any linear transformation from it will be linear thus allowing to avoid the verification of memory accesses that depend on it

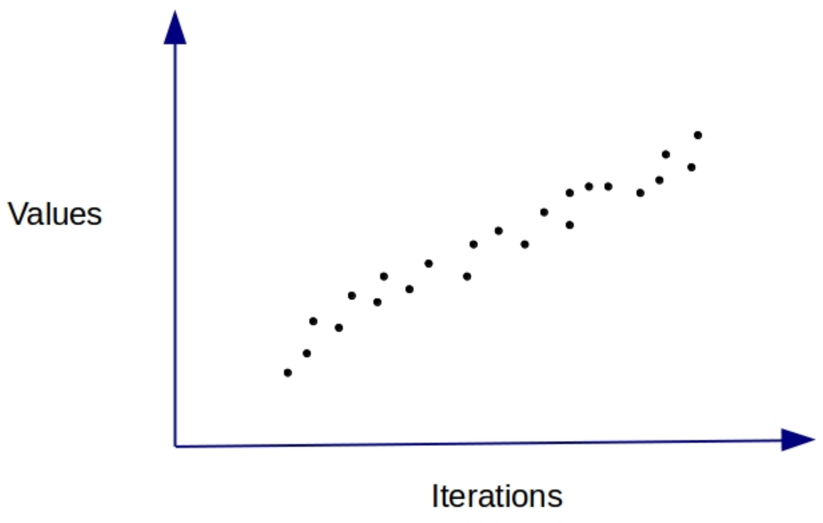


Figure 5.3: Quasi-linear behavior

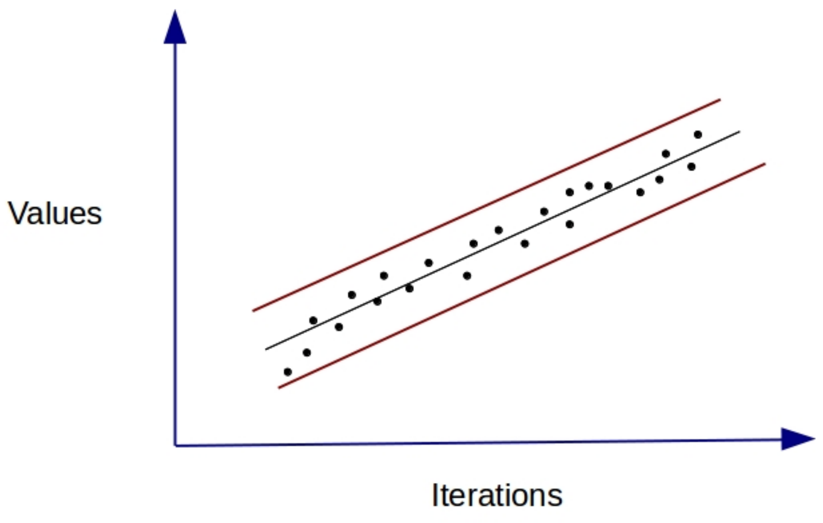


Figure 5.4: Regression line and bounds providing a *tube modeling*

5.1.2 Creating the regression line

Since a quasi-linear scalar can never be analyzed statically, it is only at runtime that the regression function can be built. Throughout the instrumentation chunk, the code is executed in the original sequential way. During this phase, a scalar which was marked as not-resolved by the static part is tested against the bounds

of the tube. If at least one point lies outside of it, then it is not handled. If all the points lie within the *tube* then the regression line is computed and the scalar is marked as quasi-linear.
The regression line is obtained using the least square method.

5.1.3 Parallel chunks with quasi-linear scalars

This section focuses on the way a quasi-linear scalar is handled during the execution of the chunks.
Once the regression line has been obtained, it is used to allow the parallel execution of the chunks as it happens with the affine predicting functions but in a slightly different way. Let us first consider the chunk that is executed immediately after the instrumentation phase. Since this chunk follows the original sequential execution, then it is possible to keep track correctly of the values that the scalar takes. In particular it is possible to know the value reached at the end of this chunk. As it was explained in the introduction of Chapter 5.1, scalars are updated at each iteration of the loop they are part of. So, for *thread 0* from the first parallel chunk that is executed after the instrumentation, the initial value of the scalar is already know. However, for the rest of the threads, the initialization of the scalar is done by instantiating the regression line with the proper values of the virtual iterators (i.e. the values the virtual iterators take at the beginning of the thread execution). Figure 5.5 illustrates the different initializations the scalar has in the different parallel chunks.

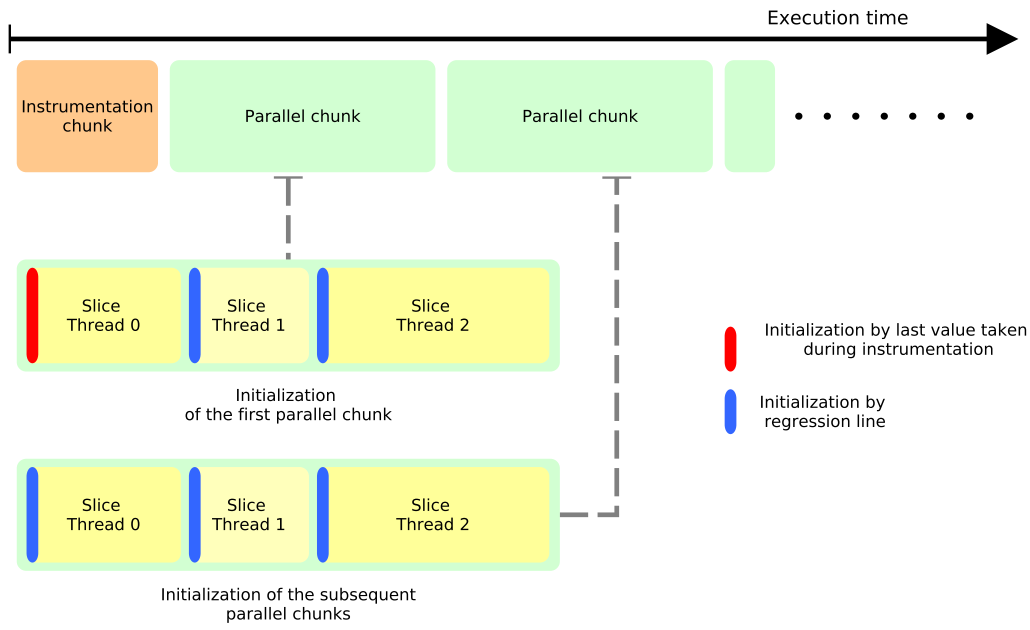

Execution time
Instrumentation chunk
Parallel chunk
Parallel chunk
Slice Thread 0
Slice Thread 1
Slice Thread 2
Initialization of the first parallel chunk
Slice Thread 0
Slice Thread 1
Slice Thread 2
Initialization of the subsequent parallel chunks
Initialization by last value taken during instrumentation
Initialization by regression line

Figure 5.5: Initialization of a quasi-linear scalar on different parallel chunks

Once the initialization is done, a slice of the original sequential code is executed inside each thread. For the first thread of the first parallel chunk the initial value is correct since it was taken from the last value reached in the instrumentation chunk. For the rest of the threads, the initialization value is *speculative* and thus can be erroneous. The quasi-linear scalar is verified as follows:

1. Throughout the execution of the first thread, the scalar does not need to be verified since the initialization was based on the value reached at the end of the instrumentation chunk which is obviously correct.

2. For the second thread, the value assumed by the scalar is *speculative* and needs verification. As it was explained in the previous point, the scalar had a necessarily correct initialization for the first thread. This makes it possible to check the initialization of the scalar in the second thread with the last value reached in the first one. If this verification is correct, then the thread being verified can continue its execution (in case it has not finished yet.) If the verification fails, a rollback is triggered for the thread and a new thread with the correct initialization is launched.

3. Once the first thread has been used to verify the second one, the second

one can be used to verify the third one in the same way the second one was verified and so on. Although thread number n has to be checked with thread number $n-1$, this does not means that the threads of a particular chunk are going to be executed in parallel, since a thread can start its execution taking a value for the non-linear scalar from the regression line. It is just the verification that has necesarily to be done in a sequential way.

Part III

Results and conclusions

Chapter 6

Evaluation of the strategies introduced

In this section we present the results we obtained while applying the speculative parallelization framework provided by APOLLO to codes which exhibits non linear behavior (non-linear write, non-linear read or both) during their memory accesses. The kernels in which the strategies were tested come mostly from the polybench benchmarks suite [17]. These are versions that replace automatic variables with pointers in order to be able to handle them through the dynamic analysis. The *filter* example was taken from [12]. *b+tree* was taken from the Rodinia Benchmark Suite [5].

The different speed-ups obtained by executing the codes in APOLLO are shown in Figure 6.1. They are expressed in terms of how many times faster the code run in comparison with the original. The comparison was made against the same code compiled with GCC using the flag -*O3*. The version of GCC used is *4.8.2*. The configuration on which the tests were run embeds two AMD Opteron Processors 6172, of 12 cores each, at 2.1 Ghz, running Linux 3.2.0-36-generic x86 64, and 32 Gb of RAM.

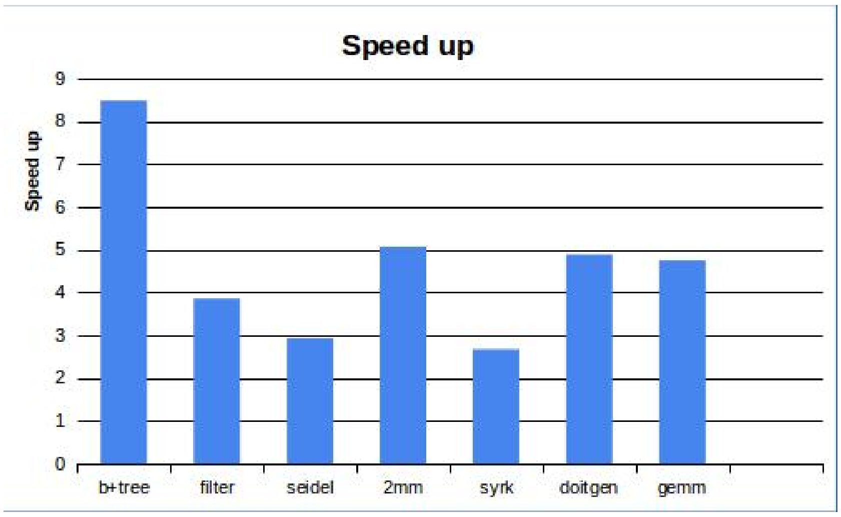


Figure 6.1: obtained speed-ups

Table 6.1 shows more detailed information which includes an approximation of number of non-linear memory reads and writes.

Table 6.1: approximated number of non-linear reads and writes

Program	Original (in sec)	APOLLO (in sec)	# NL-R accesses (approximated)	# NL-W accesses (approximated)	speed up
b+tree	8.4	0.99	8128000000	0	8.48
filter	21.2	5.5	6000000000	0	3.85
seidel	12.01	4.1	35892081	3988009	2.92
2mm	36.04	7.11	65536	16512	5.06
syrk	20.75	7.76	3221225472	1073741824	2.67
doitgen	10.27	2.1	16777216	4294967296	4.89
gemm	18.06	3.8	1073741824	0	4.75

The different benchmarks tried corresponds to the following categories:

- **Linear algebra kernel:** which usually involves operation among matrices (sums, multiplications, etc). The benchmarks used that fall in this category are: *2mm, syrk, doitgen, gemm.*
- **Stencils:** it comprehends a class of iterative kernels that update array elements according to some fixed pattern. *seidel* corresponds to this category. It implements an iterative method to solve a linear system of equations.

- **Image processing kernel:** they generally consist of changing an image by applying some transformation, represented as a matrix. pixel by pixel. The kernel from *filter* represents a classic convolution filter.
- **Searching algorithms:** *b-tree* is the benchmark that corresponds this category. It implements a search method to a generalization of binary trees.

For *seidel* the following transformation was applied:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix}$$

As a side note, without the extensions presented in this work, APOLLO would not even try to execute a parallel version of the code, it would resort directly to the original sequential version.

Chapter 7

Conclusion

The work presented in the current thesis contributes to the topic of speculative parallelization by extending the scope of the application of the polytope model. More precisely, it allows to determine if a non-linear memory access (a reading access or a writing access) invalidates the polyhedral transformations applied to the code.

A method for dynamically adapt the granularity of the verification system according to the ongoing behavior of the code was also introduced, which allows to choose between two different ways of verification. The two of them have a different impact on the performance on the system and are oriented to reduce the amount of comparisons and memory accesses performed during verification, which represents a considerable improvement on the performance of the code parallelized by the framework.

The effectiveness of these strategies was tested on codes that belong to well-know benchmark suites, obtaining speedups that range from 2.6x to 8x.

All in all, the extensions presented in this thesis allow to apply the polytope model on loop nests that do not gather all the necessary conditions (statically and dynamically.) At the same time, they widen the range of codes that APOLLO is able to parallelize, obtaining encouraging results.

Bibliography

[1] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[2] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral transformations to work. In *LCPC'16 Intl. Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, October 2003.

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, 2008.

[4] Derek Bruening, Srikrishna Devabhaktuni, and Saman Amarasinghe. Softspec: Software-based speculative parallelism. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, Monterey, California, Dec 2000.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Albert Cohen, Sylvain Girbal, and Olivier Temam. A polyhedral approach to ease the composition of program transformations. In *in: Euro-Par'04, no. 3149 in LNCS*, pages 292–303. Springer-Verlag, 2004.

[7] Laurent Daverio, Corinne Ancourt, Fabien Coelho, Stéphanie Even, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Frédérique Silber-Chaussumier. PIPS – An Interprocedural, Extensible, Source-to-Source Compiler Infrastructure for Code Transformations and Instrumentations. Tutorial at PPoPP, Bengalore, India, January 2010; Tutorial at CGO, Chamonix, France, April 2011. `http://pips4u.org/doc/tutorial/`

`tutorial-no-animations.pdf` presented by François Irigoin, Serge Guelton, Ronan Keryell and Frédérique Silber-Chaussumier.

[8] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[9] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, 1991.

[10] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 1 : one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[11] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 2 : multidimensional time. *International Journal of Parallel Programming*, 21(6), 1992.

[12] Image filtering. http://lodev.org/cgtutor/filtering.html., 2010.

[13] Alexandra Jimborean, Philippe Clauss, Juan Manuel Martinez, Aravind Sukumaran-Rajam, and Wolff Willy. Speculative program parallelization with scalable and decentralized runtime verification. *5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014*, 8734:pages 124–139, 2014.

[14] Alexandra Jimborean, Matthieu Herrmann, Vincent Loechner, and Philippe Clauss. Vmad: A virtual machine for advanced dynamic analysis of programs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 125–126, Washington, DC, USA, 2011. IEEE Computer Society.

[15] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[16] The LLVM compiler infrastructure. `http://llvm.org`.

[17] Polybenchs 1.0. http://www-rocq.inria.fr/pouchet/software/polybenchs.

[18] Louis-Noël Pouchet. FM: the Fourier-Motzkin library. `http://www.cse.ohio-state.edu/ pouchet/software/fm`.

[19] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[20] Easwaran Raman, Neil Va hharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 175–184, New York, NY, USA, 2008. ACM.

[21] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[22] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[23] Martin Süsskraut, Stefan Weigert, Ute Schiffel, Thomas Knauth, Martin Nowack, Diogo Becker Brum, and Christof Fetzer. Speculation for parallelizing runtime checks. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '09, pages 698–710, Berlin, Heidelberg, 2009. Springer-Verlag.

[24] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The suif compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford University, Stanford, CA, USA, 1994.

[25] Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown, and Mikel Luján. Optimizing software runtime systems for speculative parallelization. *ACM TACO*, 9(4):39:1–39:27, January 2013.