Proyecto Fin de Carrera de Ingeniería en Informática

Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

DESARROLLO DE FASES DE COMPILACIÓN PARA DESCUBRIR EL TIEMPO DE EJECUCIÓN DE PEOR CASO

Marta López Ara

Director: Juan Segarra Flor

Área de Arquitectura y Tecnología de Computadores
Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Septiembre 2011

Curso 2010 / 2011

Agradecimientos

Una vez llegado al final de este proyecto, me gustaría expresar mi agradecimiento a todas aquellas personas que de una manera u otra han contribuido a su realización.

En primer lugar a mi director de proyecto, Juan Segarra Flor, por conseguir que llegara hasta aquí gracias a sus continuas explicaciones, su apoyo y su confianza depositada en mí.

A Raúl por su inestimable ayuda, paciencia, tranquilidad, por estar ahí cuando las fuerzas empiezan a flaquear y por infundir el color necesario para seguir trabajando cuando se ve todo de negro.

A mi padre y mi madre, por el apoyo, cariño y comprensión que me han dado durante todos estos años.

A mis abuelos, por su amor incondicional.

A todos mis amigos de la universidad, por todos estos años compartidos juntos que nunca olvidaré.

A mis amigos: compañeros de la escuela de idiomas, postgrado, monitores de tiempo libre y alguno más que me puedo dejar en el tintero, porque gracias a su amistad me han aportado algo más que buenos momentos.

¡¡Muchas gracias a todos!!

Desarrollo de fases de compilación para descubrir el tiempo de ejecución de peor caso

RESUMEN

Los dispositivos con requisitos de tiempo real son cada vez más utilizados, por ejemplo en automóviles (e.g. ABS), aeronáutica, electrodomésticos, etc. Para poder planificar los requisitos temporales de cualquier tarea, el primer paso es conocer (una cota superior de) su tiempo de ejecución en el peor caso (*worst case execution time* o WCET). Este cálculo depende de factores hardware y software, como por ejemplo de las memorias cache y del compilador utilizado, y debe conocerse previamente a su ejecución. Además, requiere información que maneja internamente el compilador pero no queda explícita en el ejecutable final, con lo que recuperarla es muy complejo.

Cuanto más ajustada sea la cota superior obtenida, mejor se aprovecharán los recursos del sistema, aumentando así la planificabilidad del mismo.

Por todo lo anterior, se ha realizado este proyecto de fin de carrera, cuyo objetivo principal ha sido la implementación de una serie de pasos (fases en la terminología usual de compiladores) que obtengan la información necesaria directamente en el proceso de compilación: reúsos de bloques de memoria y número máximo de iteraciones en bucles. Para ello se ha utilizado la infraestructura de compilación Low Level Virtual Machine (LLVM).

Se han creado dos bibliotecas para ayudar al cálculo de la cota superior de los procesos. Estas bibliotecas van a sacar a relucir los accesos a memoria que existen, pudiendo así saber el reúso de variables y constantes, tanto temporal como espacial, y el número de veces que se ejecuta cada bucle de instrucciones como máximo. En particular:

La biblioteca *libmarcarLoadsStores* localiza en el código intermedio de LLVM los accesos a memoria que existen (loads y stores) añadiéndoles los *metadatas* de depuración para su posterior reconocimiento con sus correspondientes instrucciones en el fichero que contiene el código ensamblador ARM.

La biblioteca *libbuclesReusos* analiza en profundidad el código LLVM Intermediate Repesentation (IR) en busca de iteraciones y subiteraciones, indicando en fichero ARM el máximo número de veces que se puede ejecutar un bloque básico de instrucciones, siempre que se sepa este dato en tiempo de compilación. Además, recoge los datos de los accesos a memoria, para poder identificar el reúso espacial y temporal, la variable o constante a la cual se refiere, y el desplazamiento que existe.

Índice general

Índice de figuras

Capítulo 1

Introducción

En este capítulo se comentarán los aspectos más generales del presente Proyecto Fin de Carrera, entre los que se encuentran: la motivación para la consecución del mismo, su contexto de realización, los objetivos definidos en la propuesta, las herramientas que se han utilizado durante su elaboración, las fases en las que se ha dividido el trabajo y, por último, una breve explicación de la estructura del documento.

1.1. Motivación

A la hora de elegir un proyecto, fueron varios los motivos que me hicieron decantarme por éste. Para empezar, el hecho de contribuir a la utilización de herramientas de software libre, en particular el proyecto Low Level Virtual Machine (LLVM). Otro aspecto interesante fue la posibilidad de ampliar mis conocimientos de ingeniería del software, así como de los lenguajes ensamblador y C++.

También consideré interesante para completar mi formación el hecho de incorporarme a un proyecto de investigación y tener que adaptarme a él y a las necesidades del mismo, puesto que en mi futura vida laboral es una situación más que probable. Y, por supuesto, el poner en práctica los conocimientos adquiridos durante la carrera, que considero debe ser el principal objetivo de cualquier PFC.

1.2. Contexto de realización

El presente proyecto de fin de carrera se enmarca en temas de investigación llevados a cabo dentro del grupo de Arquitectura de Computadores (gaZ) de la Universidad de Zaragoza. Parte de dicha investigación consiste en desarrollar metodologias para analizar el tiempo de ejecución en el peor caso (WCET) y proponer componentes hardware alternativos a los existentes cuyo análisis sea factible.

Para analizar el WCET en un sistema de tiempo real son necesarios parámetros como el máximo número de iteraciones de cada bucle. En caso de que el sistema

final disponga de cache de datos, también resulta imprescindible conocer el reúso de datos del programa a analizar.

En este PFC se obtienen dichos parámetros desde dentro del compilador, lo cual facilita el posterior análisis y proporciona mayor independencia respecto al repertorio de instrucciones final.

1.3. Objetivos

Los objetivos principales del proyecto son los siguientes:

1. Estudio del funcionamiento interno del compilador LLVM [1]
2. Localización de bucles y variables de iteración en código LLVM
3. *Backtracking* de variables de iteración para obtener el máximo número de iteraciones en bucles cuando sea posible
4. Identificación de reúso espacial y temporal básico en accesos a memoria en código LLVM
5. Generación de ensamblador ARM etiquetado con la información anterior (máximo número de iteraciones en bucles e información de reúso en accesos a memoria)

1.4. Herramientas utilizadas

Para realizar el desarrollo software habría bastado simplemente con un editor de texto y el entorno de desarrollo implementado por LLVM. Sin embargo, ha habido muchas otras herramientas que han sido muy útiles durante todo el desarrollo. Éstas han sido las herramientas utilizadas:

Gedit 2.30.4 [2]: Éste fue el editor elegido para codificar y manejar los diferentes tipos de archivos. Destaca por su simpleza y rapidez, y por sus múltiples funciones y plugins que ofrece a la hora de desarrollar. Y, sobre todo, por poder manejar a la vez múltiples tipos de archivos (C, C++, ensamblador ARM, LLVM IR) con facilidad.

GCC 4.5 [3]: Compilador de C/C++ del proyecto GNU/Linux. Usado para compilar las bibliotecas.

LLVM 2.8 [1]: Infraestructura de compilación, junto con todas sus herramientas llvm-gcc, llvm-dis, llc, opt. Estas herramientas permiten compilar programas, pasar de un código a otro y hacer pasadas de análisis y compilación sobre el código.

TEX Live 2009-11 [4]: Es una distribución de LaTeX para GNU/Linux, que permite la fácil descarga y actualización de paquetes y componentes. Fue necesaria para el desarrollo de la presente documentación.

Kile 2.1 beta 4 [5]: Es un editor de LaTeX para GNU/Linux, con licencia GPL, diseñado para trabajar con la distribución texlive. Conjuntamente con éste, fue utilizado para escribir el presente documento.

Gantt Project [6]: Herramienta de software libre para la gestión de proyectos, empleada en la generación de diagramas de Gantt.

1.5. Fases del trabajo

Una vez fijados los objetivos del proyecto, era el momento de comenzar el trabajo. Por supuesto, este debía tener un orden razonable, por lo que al principio el tiempo se empleó en adquirir los conocimientos que posteriormente se aplicarían durante el desarrollo del proyecto. Las principales etapas del trabajo realizado fueron las siguientes:

1. **Estudio de la infraestructura del compilador LLVM**: Antes de empezar, había que conocer exactamente el funcionamiento de las diferentes fases del compilador y sus herramientas.

2. **Estudio de la referencia del lenguaje intermedio LLVM**: Dado que el trabajo se iba a centrar en el análisis y la traducción de este lenguaje, era necesario aprender todo su repertorio de instrucciones, su estructura y lógica. En el Apéndice A se muestra una visión general del conjunto de instrucciones y de los tipos de datos de LLVM, centrándose en las instrucciones utilizadas en el proyecto.

3. **Estudio de parámetros de análisis y optimización**: Ya que la principal característica del compilador LLVM es el amplio repertorio de parámetros de análisis y optimización, y cómo interactúan entre sí, había que comprender ampliamente cada uno de éstos.

4. **Estudio de la API de LLVM y de cómo realizar una pasada de análisis sobre el código LLVM IR**: Para poder analizar las iteraciones y el reúso, había que estudiar en profundidad la API de LLVM, sus clases y métodos. Así como realizar una pasada de análisis usando las bibliotecas que había que crear para poder usar dicha API sobre las instrucciones LLVM IR.

5. **Estudio y comprensión del proceso de traducción**: esta fase se centró en el estudio del capítulo referente a la traducción del código de representación intermedia de LLVM a código ensamblador con el juego de instrucciones de ARM [7]. En este punto, era básico comprender exactamente cómo se traduce cada parte, para posteriormente poder realizar un diseño lo más adecuado posible a la hora de implementar las bibliotecas.

6. **Fase de implementación**: una vez estudiados todos los aspectos previos, era el momento de implementar la búsqueda y análisis de las iteraciones y de los accesos a memoria en LLVM IR y su traducción a lenguaje ensamblador ARM.

1.6. Estructura de la memoria

El presente documento se encuentra dividido en dos partes. La primera es la que engloba la memoria propiamente dicha, y la segunda parte está constituida por una serie de anexos.

La parte inicial está estructurada de la siguiente forma:

- **Capítulo 1. Introducción:** Es el presente capítulo. Como ya se ha visto, se detallan brevemente los objetivos del proyecto, las herramientas utilizadas y los contenidos de la memoria.
- **Capítulo 2. Planificación:** Este capítulo contiene la distribución de las diferentes tareas a lo largo de la duración del proyecto, y la organización de las mismas.
- **Capítulo 3. Conceptos:** A lo largo de este capítulo se va a explicar el comportamiento de la herramienta LLVM, y los resultados a los que se quería llegar.
- **Capítulo 4. Desarrollo:** En este capítulo se recogen los conocimientos adquiridos durante la realización del proyecto y que han sido utilizados para la realización del mismo.
- **Capítulo 5. Conclusiones:** Este último capítulo de la memoria es uno de los más importantes de la misma. En él se resume el trabajo realizado, se resaltan las dificultades encontradas y, por último, se presentan una serie de conclusiones.

La parte de los anexos tiene el siguiente contenido:

- **Apéndice A. LLVM Intermediate Representation:** En el primer anexo se habla del lenguaje LLVM IR. Su conjunto de instrucciones, sus tipos de datos y se explican en profundidad las instrucciones analizadas en el proyecto.
- **Apéndice B. Guía de comandos LLVM:** En este apéndice se detallan los diferentes programas de línea de comandos que provee el proyecto LLVM. Se detalla cómo compilar archivos fuente a diferentes lenguajes, cómo pasar de un lenguaje a otro y cómo realizar pasadas de compilación.
- **Apéndice C. Pruebas:** Aquí se presentan una serie de pruebas para la verificación del funcionamiento de la herramienta, analizando los resultados obtenidos.

- **Apéndice D. Manual de Uso:** En este último apéndice se explica cómo compilar y usar las bibliotecas realizadas, cómo pasarlas a los ficheros a analizar, y cómo borrar los ficheros resultantes de todas las compilaciones anteriores.

Capítulo 2

Planificación

A la hora de hablar del desarrollo del proyecto, podemos diferenciar entre el ciclo de vida seguido durante el desarrollo del mismo y la planificación del tiempo empleado.

2.1. Ciclo de vida

Se puede considerar que durante este proyecto se siguió un ciclo de vida incremental, puesto que los elementos se iban añadiendo secuencialmente, como se explica en el Capítulo 4. Por ejemplo, se empezó a trabajar con la cuenta de iteraciones de los bucles, y hasta que no se comprobó el correcto funcionamiento de la misma no se pasó a la siguiente fase. Además, para cada uno de los elementos traducidos se siguió un ciclo de vida en cascada, como se puede ver en la Figura 2.1. De esta forma se aseguraba que, si fallaba algo, era por el trabajo que se estaba realizando en ese momento y no debido a un fallo en la implementación de algún apartado anterior. Lógicamente, al estar todas las fases relacionadas entre sí, en las pruebas de cada elemento no se incluía solamente él, sino todos los anteriores, para comprobar que seguían funcionando adecuadamente.

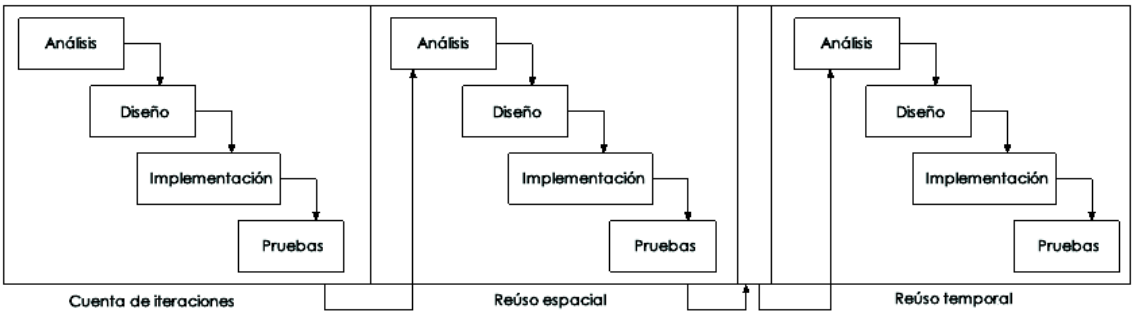

Figura 2.1: Ciclo de vida incremental. No se muestran todos los elementos del proyecto por claridad

2.2. Planificación del trabajo

Este proyecto comienza en septiembre de 2010, cuando tras hablar con el profesor Juan Segarra Flor me propone la realización del mismo. Se hizo un cálculo aproximado de la duración del mismo de unos seis meses, así que se puede considerar que se ha alargado más de lo deseado. Esto es debido a la dureza de la primera parte del proyecto, la relativa al estudio de toda la documentación, y de que se ha hecho a tiempo parcial. Una vez comenzada la implementación, el proyecto fue avanzando a más velocidad.

En la Figura 2.2 se puede ver el diagrama de Gantt final del proyecto. Como se observa, no es hasta principios de marzo de 2011 cuando se empiezan realmente a implementar las bibliotecas. Asimismo, conviene destacar que la tarea *Análisis y optimización* se extiende prácticamente a lo largo de todo el proyecto, puesto que después del análisis inicial, para cada nuevo elemento se volvían a estudiar las optimizaciones a la hora de implementarlo. Lo mismo ocurre con la tarea *Pruebas*, aunque ésta empieza algo más tarde y se extiende más allá de la fase de *Implementación*, para comprobar que el funcionamiento era correcto una vez finalizada esta tarea.

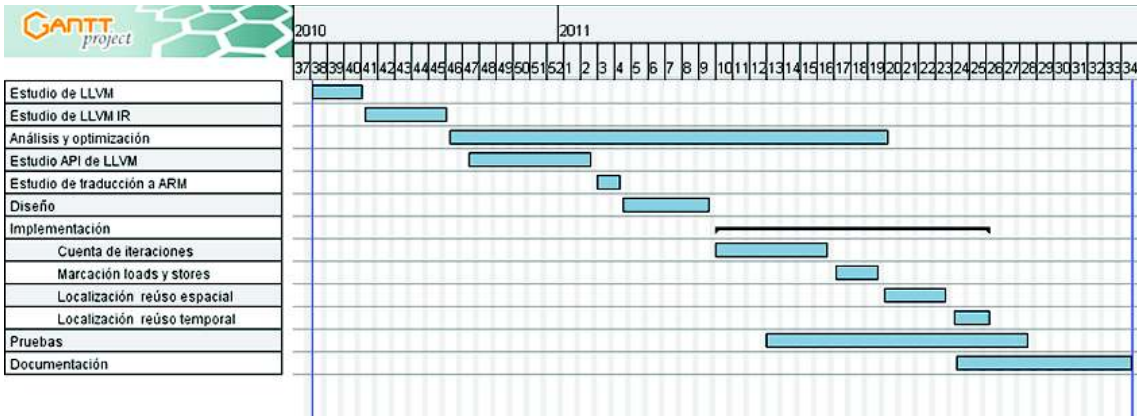

Figura 2.2: Diagrama de Gantt correspondiente a las fases de desarrollo

Capítulo 3

Conceptos

En este capítulo se recogen algunos de los conceptos más importantes en los que se basa este proyecto, para facilitar la comprensión de la herramienta LLVM y su funcionamiento, además de explicar con mayor profundidad lo que se ha conseguido.

3.1. Funcionamiento del compilador LLVM

El proyecto Low Level Virtual Machine (LLVM) es una colección de compiladores modulares y reutilizables. Primeramente el compilador transforma el código fuente de un programa (C, C++, Objetive C, etc.) en un lenguaje intermedio llamado LLVM Intermediate Representation (IR).

La representación del código LLVM está diseñada para ser utilizada de distintas formas:

- Como un lenguaje intermedio en memoria del compilador
- Como una representación bitcode en disco (ideal para una ejecución mediante máquina virtual Just-In-Time)
- Como un lenguaje legible por humanos

Esto permite transformación, visualización y análisis del código.

La representación del lenguaje es independiente de la máquina, de bajo nivel, estructurada y fuertemente tipada, lo que le hace ser muy potente. Este código intermedio puede generar códigos ejecutables para procesadores reales, como se puede ver en la Figura 3.1, entre los que está ARM, en el cual nos vamos a centrar.

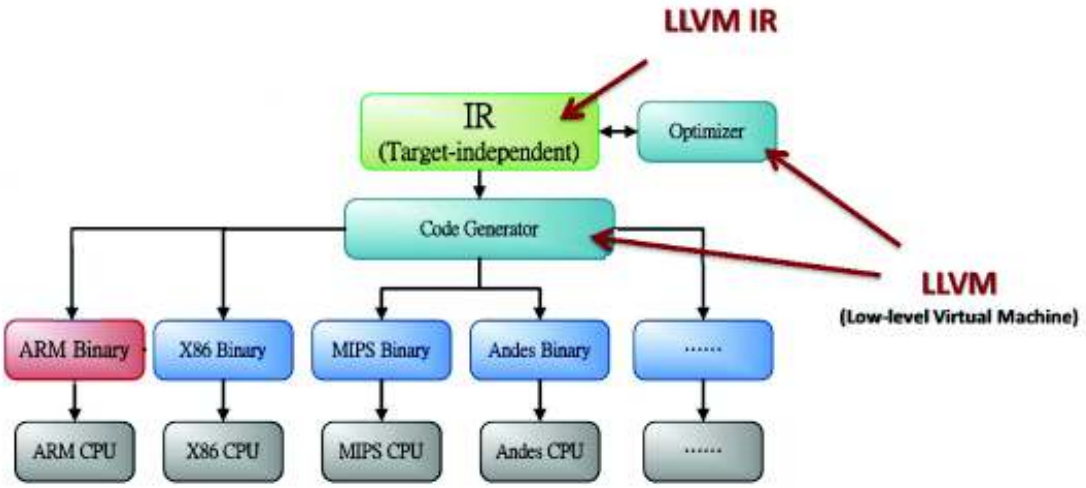

Figura 3.1: Arquitectura y módulos LLVM

En este proyecto nos vamos a basar principalmente en este lenguaje para analizar los diferentes tipos de instrucciones, para calcular el número de iteraciones en los bucles y los reúsos tanto temporales como espaciales. Una vez calculados y localizados se escribirán en el código ARM correspondiente.

Para entenderlo mejor, se va a mostrar el mismo código de un bucle con dos stores, tanto en lenguaje C como su transformación en lenguaje LLVM IR. En la Figura 3.2 vemos cómo sería el código en lenguaje C, en la Figura 3.3 vemos cómo sería el mismo código en lenguaje LLVM IR, y para terminar en la Figura 3.4 vemos su transformación en lenguaje ensamblador ARM.

```
for (varBucle=0;varBucle<Tam;varBucle++)
{
     Vector1[varBucle]=varBucle*5;
     Vector2[varBucle]=varBucle*5;
}
```

Figura 3.2: Código C de stores

En el código representado en la Figura 3.3, *bb* indica la etiqueta de un bloque básico que contiene todas las intrucciones siguientes del ejemplo. La instrucción *getelementptr* devuelve un puntero con la dirección de memoria. Las instrucciones *add* y *mul* son la suma y la multiplicación en código LLVM, mientras que *br* es el salto. La instrucción *store* es la instrucción *str* en código ensamblador ARM.

```
bb:                                               ; preds = %bb, %bb.nph9
  %varBucle.08 = phi i32 [ 0, %bb.nph9 ], [ %tmp1, %bb ]
  %scevgep12 = getelementptr [5000 x i32]* %Vector1, i32 0, i32 %varBucle.08
  %scevgep13 = getelementptr [5000 x i32]* %Vector2, i32 0, i32 %varBucle.08
  %tmp = mul i32 %varBucle.08, 5
  store i32 %tmp, i32* %scevgep12, align 4
  store i32 %tmp, i32* %scevgep13, align 4
  %tmp1 = add nsw i32 %varBucle.08, 1
  %exitcond11 = icmp eq i32 %tmp1, 5000
  br i1 %exitcond11, label %bb3, label %bb
```

Figura 3.3: Código LLVM de stores

```
.LBB0_1:                                @ %bb
                                        @ =>This Inner Loop Header: Depth=1
str r0, [r2], #4
str r0, [r1], #4
add r0, r0, #5
cmp r0, r3
bne .LBB0_1
@ BB#2:                                 @ %bb.bb3_crit_edge
add lr, sp, #1, 18 @ 16384
mov r4, #226, 30 @ 904
orr r4, r4, #1, 20 @ 4096
mov r5, sp
add r6, lr, #226, 28 @ 3616
ldr r7, .LCPI0_0
```

Figura 3.4: Código ARM de stores

3.2. Bucles y reúsos en código ARM

El código ARM es el lenguaje ensamblador propio de la arquitectura ARM. El código escrito en lenguaje ensamblador es complejo ya que es una reprensentación del lenguaje máquina, con instrucciones, registros y posiciones de memoria del procesador. Al tratarse de un nivel tan bajo y tratar con registros de la máquina es muy difícil poder ver con claridad las iteraciones de los bucles y reúsos mirando directamente el código ARM.

El siguiente sencillo ejemplo en lenguaje C, Figura 3.5, nos permitirá explicar a que nos referimos, mostrando bucles y reúsos espaciales. En el programa de ejemplo podemos ver que hay dos vectores y un bucle que se va incrementando de uno en uno. Dentro del bucle vemos que se produce una escritura en cada vector en la posición "i" que es la variable de iteración. También vemos que el bucle se repetirá cinco mil veces. Sabemos, por tanto, que se van a producir reúsos espaciales en los dos vectores ya que se accede a una posición del vector que se va incrementando de manera constante a lo largo del bucle.

```
#include <stdio.h>
#define Tam 5000

int main()
{
    int i=0;
    int A[Tam];
    int B[Tam];

    for (i=0;i<Tam;i++)
    {
         A[i]=i*5;
         B[i]=i*5;
    }
}
```

Figura 3.5: Código ejemplo en C

Sin embargo al compilarlo a código ARM, esa información es muy difícil de ver, como se puede observar en la Figura 3.6.

```
@ BB#0:                                 @ %bb.nph9
    stmdb   sp!, {r4, r5, r6, r7, r8, lr}
    sub    sp, sp, #113, 26    @ 7232
    sub    sp, sp, #2, 18    @ 32768
    add    lr, sp, #1, 18    @ 16384
    mov    r0, #0
    mov    r1, sp
    add    r2, lr, #226, 28    @ 3616
    mov    r3, #106, 30    @ 424
    orr    r3, r3, #6, 20    @ 24576
.LBB0_1:                                @ %bb
                                        @ =>This Inner Loop Header: Depth=1
    str    r0, [r2], #4
    str    r0, [r1], #4
    add    r0, r0, #5
    cmp    r0, r3
    bne    .LBB0_1
@ BB#2:                                 @ %bb.bb3_crit_edge
    add    lr, sp, #1, 18    @ 16384
    mov    r4, #226, 30    @ 904
    orr    r4, r4, #1, 20    @ 4096
    mov    r5, sp
    add    r6, lr, #226, 28    @ 3616
    ldr    r7, .LCPI0_0
```

Figura 3.6: Código ejemplo en ARM

Podemos ver que hay dos stores (instrucciones *str*) dentro de un bloque que se repite (*.LBB0_1*), con un salto (*bne*) con comparación (*cmp*). El bloque pertenece a un bucle, pero no podemos conocer fácilmente los reúsos de esos stores ni las repeticiones que hará dicho bloque. El objetivo del proyecto es precisamente mostrar claramente en el código ARM este tipo de información de forma clara y transparente. La siguiente figura muestra el resultado final en código ARM después de pasar las bibliotecas.

```
.LBB0_1:                                @ %bb
                                        @ Numero de vueltas=5000
                                        @ =>This Inner Loop Header: Depth=1
    str r0, [r2], #4  @ Store var "A". Reuso espacial. Var iteracion "i".
                                        Desplazamiento con "stride" 1
    str r0, [r1], #4  @ Store var "B". Reuso espacial. Var iteracion "i".
                                        Desplazamiento con "stride" 1
    add r0, r0, #5
    cmp r0, r3
    bne .LBB0_1
@ BB#2:                                 @ %bb.bb3_crit_edge
    add lr, sp, #1, 18    @ 16384
    mov r4, #226, 30    @ 904
    orr r4, r4, #1, 20    @ 4096
    mov r5, sp
    add r6, lr, #226, 28    @ 3616
    ldr r7, .LCPI0_0            @ Load Constante .LCPI0_0
```

Figura 3.7: Código ejemplo final ARM

Capítulo 4

Desarrollo

Este capítulo describe cómo se llevó a cabo el desarrollo del proyecto y las decisiones más importantes tomadas durante el mismo.

4.1. Análisis y diseño

Durante toda la fase de análisis y diseño, se trabajó estrechamente con el profesor Juan Segarra Flor para definir bien los siguientes puntos:

- **Alcance del Proyecto**: Se definió, describió y preparó el escenario de implementación.
- **Análisis del Proyecto**: Se especificaron las necesidades actuales del proyecto, y el encaminamiento para el futuro de la aplicación.
- **Diseño del Proyecto**: Se describió su solución y las actividades de implementación y testeo que se iban a hacer.

El objetivo de esta fase era poner en firme cuáles eran exactamente las especificaciones del proyecto, la entrega y la preparación de la implementación.

4.1.1. Alcance del proyecto

Se empezó a trabajar para determinar el alcance de la implementación del proyecto. En este paso, se determinó lo que cubriría el proyecto, y cómo se gestionaría el tiempo para ello.

El resultado final de esta fase, fue la generación de dos documentos:

- La propuesta del proyecto, que contiene el alcance descrito para el mismo.
- Una primera planificación de la distribución del proyecto.

4.1.2. Análisis del proyecto

En la fase del análisis se recogieron los requisitos necesarios. Para esto, se quedó varios días con el profesor Juan Segarra Flor, y así se delimitaron las necesidades halladas hasta el momento, las cuales cambiarían muy poco en todo el proceso.

Los requerimientos más importantes fueron:

- Localizar los bucles dentro de un código fuente.
- Calcular el número de iteraciones máximo, en tiempo de compilación, de cada bloque de instrucciones.
- Hallar el reúso espacial.
- Encontrar el reúso temporal.
- Mostrar todo lo anterior en el fichero ensamblador de ARM.

4.1.3. Diseño del proyecto

Posteriormente, una vez establecidos los requisitos y hecha la propuesta, se diseñó la biblioteca y sus dependencias.

Se llevó a cabo un mapeo de las tecnologías disponibles para su ejecución y un profundo estudio de la herramienta LLVM. Se siguieron haciendo reuniones para asegurar si el camino llevado hasta el momento era el correcto y para resolver dudas.

Se fijó que había que realizar una biblioteca que leyera el código intermedio LLVM IR y que usándola mediante una pasada de análisis sobre un programa pudiéramos leer los datos de sus instrucciones. Así se tendría acceso a la secuencia de instrucciones y podríamos analizarlas en profundidad.

Se estableció como había que realizar dicha biblioteca y las opciones del *makefile*. Se describieron los archivos *makefile*, tanto de la biblioteca como de los programas a analizar. Se establecieron también los comandos para poder realizar una pasada de análisis-optimización usando la biblioteca sobre el código LLVM IR. Se estudiaron todos los parámetros de optimización de LLVM y los diferentes niveles de optimización a la hora de compilar.

Se realizaron varios ejemplos básicos de bibliotecas que realizaban un recorrido básico sobre las instrucciones LLVM IR. Se comprobó así que teníamos acceso a dichas instrucciones y sus parámetros. Se fijaron también los diferentes tipos de programas-pruebas en C que habría que realizar para probar nuestra biblioteca: diferentes tipos de bucles (while, for, etc), acceso a vectores sobre la variable de iteración en los bucles y varios programas completos de prueba.

Es gracias a esta fase y a la de análisis, que la fase de implementación ha sido más corta de lo que se esperaba, teniendo una duración final esta última de dos meses a tiempo parcial, como se puede ver en la Sección 2.2.

4.2. Implementación

Fue requisito del proyecto que el lenguaje de desarrollo fuese C++, ya que es el lenguaje del código fuente de las bibliotecas de LLVM.

Para poder entender bien la implementación, se va a dividir esta sección en tres partes. En la primera, se va a explicar cómo se construyó la biblioteca para la cuenta de iteraciones de bloques. En la segunda, se mostrará cómo se creó la biblioteca *libMarcarLoadsStores*. Para finalizar, se detallará cómo se pasaron los reúsos temporales y espaciales marcados en el código LLVM IR al fichero ensamblador ARM, y varios detalles adicionales que se hicieron para complementar el trabajo realizado.

4.2.1. Cuenta de iteraciones

Después de estudiar el funcionamiento de LLVM y su API se decidió realizar primero el conteo de iteración de los bucles. Para ello, hubo que realizar una biblioteca, ya que se ésta se puede cargar en una pasada de análisis. Por consiguiente, se creó la biblioteca *libcuentaBucles*, que después se pasaría a llamar *libbuclesReusos*.

Para poder utilizarla, hubo que usar la herramienta "opt" que realiza pasadas de análisis y optimización sobre código LLVM IR, y a la que se le pueden añadir muchos parámetros según la necesidad que se tenga.

Para hallar el número de veces que se pasa por un bloque de instrucciones, fue de gran ayuda la clase "LoopInfo" de la API. Gracias a ella, se podía acceder a la información de cada bucle. Dentro de esa información, destacan los siguientes datos:

- Etiqueta del bloque de instrucciones
- Número de vueltas
- Variable de iteración

Además de cada una de las instrucciones incluidas en cada bloque.

De gran importancia, fue la optimización "indvars". Gracias a esto se pudieron convertir todos los bucles en bucles naturales con variable de iteración canónica, es decir, que todos empezaran en cero y fueran incrementándose de uno en uno, como se muestra en la siguiente figura.

```
for (i = 10; i < 5000; i+=2) => for (i = 0; i < 2495; i++)
for (i = 7; i*i < 1000; i++) => for (i = 0; i != 25; i++)
```

Figura 4.1: Optimización *indvars*

Con esto, ya se podía saber las iteraciones que se producirían en cada bucle y su etiqueta. El nombre de las etiquetas en código LLVM IR y en código ARM son idénticos, por lo cual, se podía escribir en el fichero ARM el número de vueltas calculado.

En la siguiente figura podemos ver un ejemplo de bucles anidados:

```
.LBB0_1:                                @ %bb2.preheader
                                        @ Numero de vueltas=5000
                                        @ =>This Loop Header: Depth=1
                                        @     Child Loop BB0_2 Depth 2
mov r7, #0
.LBB0_2:                                @ %bb1
                                        @ Numero de vueltas=3000
                                        @   Parent Loop BB0_1 Depth=1
                                        @ =>  This Inner Loop Header: Depth=2
mov r1, r7
mov r0, r5
add r7, r7, #1
bl printf
cmp r7, r6
bne .LBB0_2
```

Figura 4.2: Ejemplo de cuenta de iteraciones

4.2.2. Marcación de loads y stores

El siguiente paso fue localizar los reúsos, tanto temporales como espaciales, en el código LLVM IR.

Gracias a la biblioteca creada se podía acceder a las instrucciones y comprobar si éstas eran loads o stores. También, se podía saber cuál era la variable que se cargaba en cada momento, siempre hablando en tiempo de compilación, y si ésta se incrementaba con la variable de iteración. Pero como el objetivo del proyecto era localizarlos en el ARM aquí estuvo el primer gran problema.

El código LLVM IR usa nombres para variables mientras que el código ARM sólo usa los registros (r1, r2, r3, etc). Por este motivo, no se podía saber qué instrucción ARM correspondía con cada load o store localizado en el código LLVM IR.

Primero se pensó en añadir anotaciones o comentarios en las instrucciones del código LLVM para que al compilarlo a código ARM estuvieran allí en las

instrucciones correspondientes, pero los comentarios se eliminaban en el proceso de traducción.

Después de mucho investigar, se comprobó que la información del tipo "metadata" de LLVM (Figura 4.3) diseñada para DEBUG, y disponible desde la versión 2.7 de LLVM, se transformaba en comentarios en el código ARM (Figura 4.4) a la hora de compilar. Escribiendo un metadata en una instrucción LLVM se podía saber con qué instrucción o instrucciones se correspondía en el código ARM.

```
%tmp2 = load i32* %scevgep, align 4, !dbg !7
%tmp4 = load i32* %scevgep10, align 4, !dbg !8
```

Figura 4.3: Ejemplo de debug en loads y stores

```
declare i32 @printf(i8* nocapture, ...) nounwind
!1 = metadata !{i32 524329, metadata !"load", null, null}
!2 = metadata !{i32 524329, metadata !"store", null, null}
!3 = metadata !{i32 524299, null, i32 6, i32 0, metadata !1, i32 0}
!4 = metadata !{i32 524299, null, i32 6, i32 0, metadata !2, i32 0}
!5= metadata !{i32 0, i32 0, metadata !4, null}
!6= metadata !{i32 1, i32 0, metadata !4, null}
!7= metadata !{i32 0, i32 0, metadata !3, null}
!8= metadata !{i32 1, i32 0, metadata !3, null}
```

Figura 4.4: Ejemplo de metadatas

Así pues, había que escribir los metadatas en los loads y stores del código LLVM IR y después compilarlo para acceder a dicha información en el ARM. Como había que marcar los loads y stores, y después, compilar para obtener el ARM con la información necesaria para poder relacionar las instrucciones, se optó por crear dos bibliotecas en vez de una.

Por este motivo, se desarrolló una nueva biblioteca llamada *libmarcarLoadsStores* que se encargaría de escribir los metadatas correspondientes en los loads y stores del código LLVM. Así, cuando se compilara el código pasándole esta biblioteca, ya tendríamos el código LLVM con metadatas y faltaría pasarlo a código ARM. Es entonces, cuando se decidió que la biblioteca llamada *libcuentaBucles*, mencionada anteriormente, también fuera la encargada de realizar todo el proceso de comprobar los diferentes reúsos de los loads y stores (temporal y espacial), y pasó a llamarse *libbuclesReusos*.

4.2.3. Localización de reúsos espaciales

El reúso espacial que se ha calculado en este proyecto se centra en el acceso a diferentes posiciones de vectores, matrices o estructuras, en especial, cuando estas posiciones dependen directa o indirectamente de la variable de iteración de los bucles.

Ya que se había calculado el número de iteraciones en los bucles y se tenía la variable de iteración, se optó por continuar con la localización de reúsos espaciales.

Para continuar, lo primero que había que entender era lo que significaba cada parámetro de las instrucciones, en particular las de loads y stores. Estas instrucciones tienen dos parámetros. Uno es la variable o el valor de lectura, o de escritura, dependiendo de si es un load o un store. El otro parámetro es, en el caso de los reúsos espaciales, un puntero a una posición de un vector. Dicho puntero es el que nos interesaba analizar con profundidad.

Los punteros también tienen varios parámetros (ver Figura 4.5). El primero es el vector al que se quiere acceder, y los demás parámetros son variables, instrucciones o valores que nos indican los índices de desplazamiento sobre éste. Nos interesaba saber el nombre de la variable del vector y los parámetros de desplazamiento.

El principal objetivo aquí era saber cuántos parámetros tenía cada puntero de los loads y stores. Si tenía más de un parámetro, estábamos ante un vector multidimensional o una estructura.

```
%scevgep = getelementptr [100 x [100 x i32]]* @A, i32 0, i32 %i.119, i32 %k.016
%scevgep25 = getelementptr [100 x [100 x i32]]* @B, i32 0, i32 %k.016, i32 %j.117
```

Figura 4.5: Ejemplo de puntero con varios parámetros

Aunque no fue un requisito inicial, se prefirió dar una explicación más detallada de la instrucción a analizar. Por lo cual, además de avisar que había reúso espacial, a partir de ahora, también se indicaría si era de una o múltiples variables, y el nombre de las mismas o el valor, según el caso, como se muestra en la figura Figura 4.6.

```
ldr r6, [r4, -r12]    @ Load var "A". Reuso espacial. Multiples variables "i" "k"
ldr r7, [r2], #400    @ Load var "B". Reuso espacial. Multiples variables "k" "j"
```

Figura 4.6: Ejemplo de load con múltiples variables

En el caso de que fuera una sola variable la del desplazamiento sobre el puntero, se acordó que se mostraría el "stride", es decir, la separación entre una posición y la siguiente a la que se accediera, siempre expresado en elementos del tipo declarado en el vector.

```
str r0, [r2], #4     @ Store var "A". Reuso espacial. Var iteracion "i".
                                      Desplazamiento con "stride" 1
str r0, [r1], #4     @ Store var "B". Reuso espacial. Var iteracion "i".
                                      Desplazamiento con "stride" 1
```

Figura 4.7: Ejemplo de store con una variable

Así pues, hubo que comprobar las operaciones realizadas sobre esa variable dentro del bucle. Se vio que todas las variables de incremento constante tenían relación con el *phinode*[1] del bucle. Entonces, lo que se tenía que hacer era seguir todas las operaciones que se hacían sobre dicha variable hasta llegar a su phinode.

Debido a que se estaba usando la optimización "indvars", la variable directa de todos los phinodes se incrementaba de uno en uno, es decir, tenía "stride" uno. Por lo cual, teníamos una referencia clara.

Por esto se implementó una función en la biblioteca que recorriera todas las instrucciones realizadas sobre una variable, y sus resultados, hasta llegar al phinode. Aquí surgieron dos posibles resultados:

- Incremento constante
- Incremento variable

Dentro de los casos de incremento constante, cualquier variable que apuntara al phinode directamente tendría "stride" uno. Por el contrario, en muchas ocasiones, se realizan operaciones (desplazamiento de bits, multiplicación, etc) sobre el phinode, así que había que calcular el valor de dichas operaciones en base al valor uno del phinode, para poder saber su valor. Además, se indicó en el código ARM, que el incremento era debido a una variable calculada basada en la variable de iteración.

```
ldr r1, [r5], #20    @ Load var "A". Reuso espacial.
                                     Var calculada basada en var iteracion "i".
                                     Desplazamiento con "stride" 5
```

Figura 4.8: Ejemplo de incremento constante

Por otro lado, están los resultados en los que el incremento no es constante a lo largo del bucle. Esto sucede cuando la variable, o las instrucciones sobre ésta no apuntan al phinode o apuntan a un phinode que no es el principal del bucle. Aquí, hubo que indicar en el código ARM que el incremento no era constante.

```
ldr r1, [r7, r5, lsl #2]    @ Load var "B". Reuso espacial. Var iteracion "i".
                                            Desplazamiento con "stride"
                                            no constante
```

Figura 4.9: Ejemplo de incremento variable

[1] Ver Sección A.3 para más detalles

4.2.4. Localización de reúsos temporales

Para acabar el proyecto sólo faltaba localizar los reúsos temporales. Lo primero que hubo que hacer fue extraer los nombres de las variables de cada reúso temporal del código LLVM IR para su posterior indicación en el código ARM.

En este tipo de reúsos, se decidió mostrar el desplazamiento de bytes, aunque no fuera requisito incial. Este desplazamiento se extrajo directamente del código ARM.

```
strb r2, [r12], #1          @ Store var "outp.133.i". Desplazamiento 1 bytes
```

Figura 4.10: Ejemplo de reúso temporal

Posteriormente, se comprobó que en el código ARM existían más loads y stores que en el código LLVM IR. Esto es debido a que LLVM IR, al no trabajar con registros, no tiene ninguna limitación, mientras que ARM tiene un número de registros limitado, por lo cual necesita, en algunas ocasiones, reutilizarlos y crear más instrucciones. Igualmente pasa con las instrucciones de pila.

Por consiguiente, hubo que indicar en estos loads y stores el tipo de reúso y el desplazamiento. De igual modo, se indicaron los loads y stores en el caso de las constantes del código ARM.

```
str r2, [sp]                @ Store de pila. Desplazamiento 0
str r3, [sp, #8]            @ Store de pila. Desplazamiento 8 bytes
```

Figura 4.11: Ejemplo de store de pila

```
ldr r1, .LCPI0_0            @ Load Constante .LCPI0_0
```

Figura 4.12: Ejemplo de load de constante

4.3. Pruebas

Las pruebas del sistema son una parte muy importante del desarrollo software, ya que permiten comprobar el correcto funcionamiento del sistema. Estas pruebas han sido desarrolladas durante todo el proceso de desarrollo del proyecto y no solamente al final, para poder solucionar el mayor número de errores posibles en cada etapa, y no ir arrastrándolos a medida que se va avanzando.

Las diferentes pruebas realizadas han sido las siguientes:

Pruebas de unidad: En estas pruebas se han comprobado todas y cada una de las funcionalidades de las bibliotecas, verificando el comportamiento esperado de todos los métodos.

Pruebas de integración: Se ha comprobado exhaustivamente la combinación de las dos bibliotecas, constatando el correcto funcionamiento del proceso global.

En el Apéndice C se pueden ver algunos ejemplos de pruebas que se han realizado para la comprobación del desempeño de las bibliotecas.

Capítulo 5

Conclusiones

En este capítulo se resumirá el trabajo realizado en este proyecto. Además de la implementación, se señalarán también las dificultades más destacables, para finalizar con las conclusiones obtenidas.

5.1. Dificultades encontradas

La labor de cumplir los objetivos marcados al principio del proyecto y plasmados en la propuesta no ha sido fácil. Antes de empezar sabía que llegar a comprender todo el entramado de LLVM llevaría mucho trabajo, pero además la gran cantidad de documentación sobre muchos aspectos del mismo y la escasez de ejemplos hicieron que fuese más complicado de lo previsto.

El primer obstáculo destacable fue comprender toda la teoría relacionada con el compilador LLVM, tanto del *frontend* como del *backend*. El primer paso que se dio fue recordar los conocimientos ya adquiridos sobre lenguaje ensamblador y, además, adquirir algunos nuevos, propios del lenguaje de código intermedio de LLVM. Sin embargo, la parte más complicada de esto fue la comprensión de la traducción de código de alto nivel (por ejemplo, código en lenguaje C), al código intermedio de LLVM, ya que hubo que estudiar varios artículos.

Una vez comprendida esta traducción y la organización del código, era la hora de empezar a entender cómo se pasaba del código LLVM a código ensamblador ARM.

Otro aspecto importante fueron las optimizaciones precisas necesarias tanto a la hora de compilar los archivos fuente de las pruebas (-O0...-O3) como a la hora de hacer la pasada de optimización sin que afectara demasiado a las instrucciones tanto en LLVM IR como en ARM. Hubo que investigar mucho las diferentes opciones del optimizador de LLVM.

Por último, a la hora de implementar las bibliotecas, también aparecieron dificultades. La más destacable, por la cantidad de problemas que causó, fue el paso de los comentarios de las instrucciones del código LLVM IR al código ensamblador

de ARM. Al usar, este último, registros en vez de variables, no estaba claro qué instrucción correspondía con cada una del anterior código. Al final, se optó por utilizar los metadatas de depuración para poder relacionarlas.

5.2. Trabajo futuro

El trabajo futuro que se podría realizar está dividido en dos ramas:

- Analizar otros aspectos de un programa: Ya que tenemos acceso fácil a las instrucciones LLVM IR y su correspondencia en ensamblador, sería posible estudiar y anotar otro tipo de aspectos de los programas. Por ejemplo, el número de variables usadas, número de usos de esa variable, tamaño que ocupan en memoria dichas variables, diferentes saltos entre bloques. En definitiva cualquier dato del flujo del programa, de las variables y de las diferentes operaciones como multiplicaciones, sumas, llamadas a funciones, etc.
- Permitir trabajar con otras arquitecturas: Poder trabajar con otras arquitecturas y ensambladores diferentes de ARM como SPARC, MIPS, x86. Anotando en sus correspondientes archivos los diferentes datos calculados.

A parte de todo esto, se van a distribuir los códigos fuentes de las bibliotecas con licencia de software libre, para que se puedan utilizar en otras investigaciones.

5.3. Conclusiones

Durante la realización de este proyecto se han puesto en práctica parte de los conocimientos adquiridos durante la carrera, así como otros muchos que han sido adquiridos especialmente para él.

Considero especialmente interesante la práctica adquirida en el desarrollo en C++, ya que es el primer lenguaje orientado a objetos (OO) que conozco en profundidad, y uno de los más extendidos. Asimismo, el mayor conocimiento del compilador LLVM.

También he aprendido que una correcta planificación y documentación inicial es muy importante, especialmente cuando se habla de proyectos de cierta envergadura. Si no hubiera tanta documentación sobre LLVM me hubiera encontrado con muchos problemas que hubieran requerido emplear mucho tiempo en solucionarlos.

Al ser unas bibliotecas desarrolladas para ser integradas en un proyecto de investigación que llevan varias personas, ha tenido una gran importancia que las salidas fueran claras y concisas, según los requerimientos con el máximo detalle posible. Puede ser muy desesperante para alguien que continúa, o que integra, un proyecto encontrar cosas que no comprende y tener que estudiar casi todo el código para

cambiar los datos de salida a lo que necesita.

Por todo esto, creo que este proyecto ha contribuido a prepararme para un futuro entorno laboral, en el que espero poder poner en práctica algunos de los conocimientos aprendidos aquí.

5.4. Valoración personal

Una vez llegado al final de este proyecto, me gustaría hacer una valoración personal de lo que ha supuesto todo este tiempo. Respecto al trabajo realizado, me siento muy satisfecha por los resultados conseguidos. Los objetivos que se marcaron en un principio han sido superados, e incluso se ha realizado una pequeña ampliación de éstos llevando a realizar un trabajo muy completo.

El hecho de realizar un trabajo que puede ayudar a la investigación y que al final del mismo va a ser utilizado, es una experiencia muy enriquecedora. Añadir que me ha servido para ver muchas partes de la arquitectura de computadores que se dan sólo en teoría.

A nivel personal, la realización de este proyecto me ha enseñado muchísimo. He aprendido por primera vez a afrontar un proyecto de esta envergadura de principio a fin, pasando por todas las fases del mismo. He aprendido a aplicar y reforzar conocimientos de la carrera, así como muchísimos conceptos nuevos no enseñados en ella. Pero lo más importante que he aprendido es que gracias al trabajo diario y a la lucha continua se consigue llegar a la meta.

Además, he sabido aprovechar la experiencia y sabiduría de mi profesor que me ha enseñado y guiado en los momentos difíciles.