Introducción al compilador Open64/ORC

Eduard Santamaria Marta Jiménez Agustín Fernández
Josep Ma. LLabería

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

13 de mayo de 2003

Índice General

Capítulo 1

Introducción

1.1 Introducción

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

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

El conjunto de compiladores está disponible para correr sobre plataformas Linux IA-32 e IA-64. Para su utilización en una máquina Linux IA-32 es necesario el Native User Environment de HP.

Las prestaciones del compilador incluyen:

- Compilación y optimización para C, C++ y Fortran90/95
- Código generado compatible con otros compiladores que se ajusten a la ABI de Linux IA-64.
- Análisis y optimización inter-procedural (IPA)
- Acepta el mismo lenguaje C que el gcc de GNU.
- Acepta el mismo lenguaje C++ que el g++ de GNU.
- El lenguaje Fortran90/95 aceptado es ISO ANSI Fortran90 con extensiones de Fortran95.
- Los compiladores corren sobre Linux IA-32.
- Los compiladores corren sobre Linux IA-64.

Este documento pretende ser una herramienta que facilite la toma de contacto con estos compiladores. En él se describen los distintos componentes que los forman (cap. 2), la representación intermedia que se utiliza como interfaz entre dichos componentes (caps. 3 y 4), y la estructura de directorios que contiene los archivos fuente (cap. 5).

Capítulo 2

Componentes

En este capítulo veremos cual es el modelo de compilación del compilador, es decir, cuales son los pasos seguidos desde que se recibe un conjunto de archivos con código fuente hasta generar un programa o una librería ejecutable. Veremos también qué componentes actúan en las distintas etapas de la compilación y qué transformaciones se aplican a la representación del programa a medida que es tratada por cada uno de ellos.

2.1 Modelo lógico de compilación

Podemos considerar la compilación de un programa u objeto compartido dinámico (DSO o librería de carga dinámica) como un proceso compuesto por tres etapas (ver figura 2.1):

1. Para cada uno de los archivos fuente se realiza un análisis léxico, sintáctico y semántico, y se genera una representación intermedia del archivo de entrada. El componente responsable de llevar a cabo esta primera fase se denomina front-end.

2. Se procesa el archivo con la representación intermedia realizando las optimizaciones pertinentes, se traduce la representación intermedia a lenguaje ensamblador y se genera un archivo objeto relocalizable. El conjunto de componentes involucrados en esta fase recibe la denominación de back-end.

3. El enlazador (o linker) enlaza los distintos ficheros objeto para conseguir un programa ejecutable o un objeto compartido dinámico.

Existe un componente denominado driver que llama de forma ordenada a los componentes encargados de cada una de las fases.

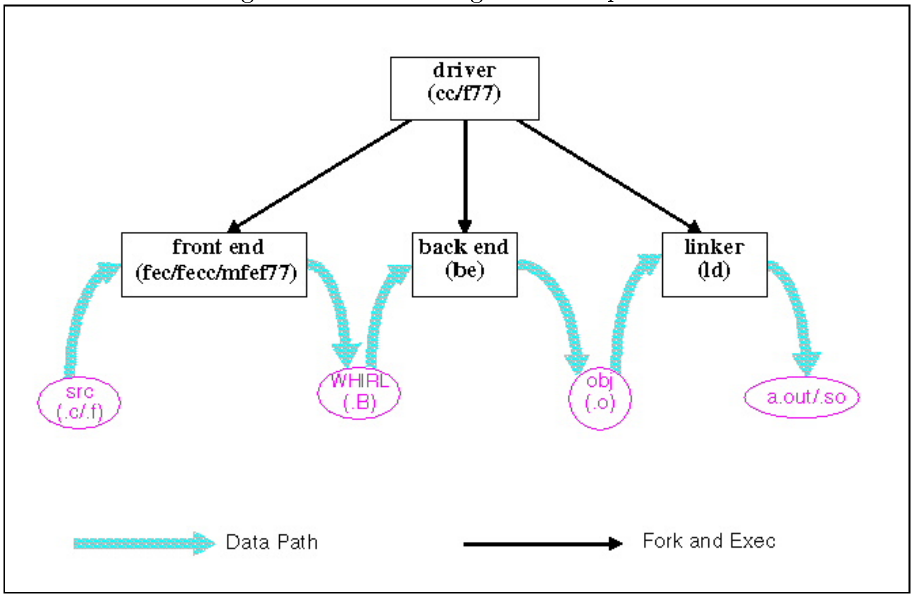
El compilador está formado por los siguientes componentes:

- Preprocesador de C y C++

- Front-ends para C, C++ y Fortran90/95

- Analizador/optimizador inter-procedural (IPA)

Figura 2.1: Modelo lógico de compilación



- Optimizador/paralelizador de bucles (LNO)
- Optimizador global
- Generador de código
- Ensamblador
- Linker

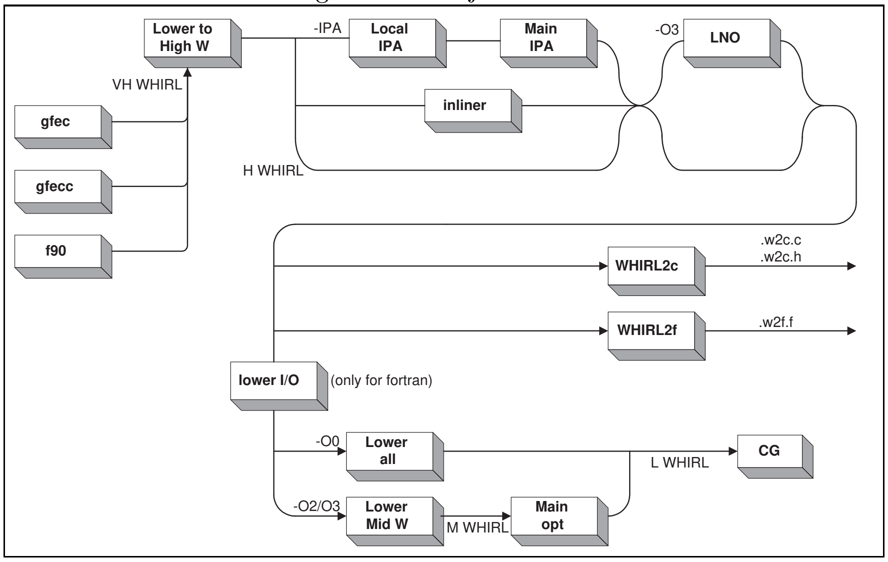
No todos los elementos aquí listados forman parte de la distribución del compilador. El preprocesador, el ensamblador y el linker se toman de las herramientas de compilación de GNU. Respecto al linker, se utiliza una versión modificada para soportar el optimizador inter-procedural (ver sección 2.5). Los componentes *optimizador inter-procedural*, *optimizador de bucles*, *optimizador global* y *generador de código* reciben conjuntamente la denominación de *back-end*.

2.2 Flujo de datos entre los componentes

La figura 2.2 nos muestra como se sucede la ejecución de los distintos componentes del compilador y el flujo que siguen los datos a medida que son procesados por cada uno de ellos.

En primer lugar, uno de los front-ends, el que corresponda según el lenguaje de programación, analiza el archivo de entrada y genera su representación intermedia. El formato utilizado para almacenar esta representación intermedia se denomina WHIRL (ver sección 2.3). Inicialmente partimos de una representación en WHIRL de muy alto nivel, próxima al lenguaje de programación, y

Figura 2.2: Flujo de datos



a medida que avanza la compilación se descendiende hasta llegar a WHIRL de muy bajo nivel, próximo a las instrucciones del procesador.

Después de la etapa de front-end encontramos un módulo que traduce la representación en VH WHIRL (Very High) a H WHIRL (High). Dependiendo de las opciones de compilación esta representación intermedia alimenta al componente encargado de las optimizaciones inter-procedurales (Local IPA y Main IPA), al stand-alone inliner (inlining a nivel de archivo) o bien pasa directamente al optimizador de bucles (LNO), cuya ejecución también depende de las opciones que se hayan especificado.

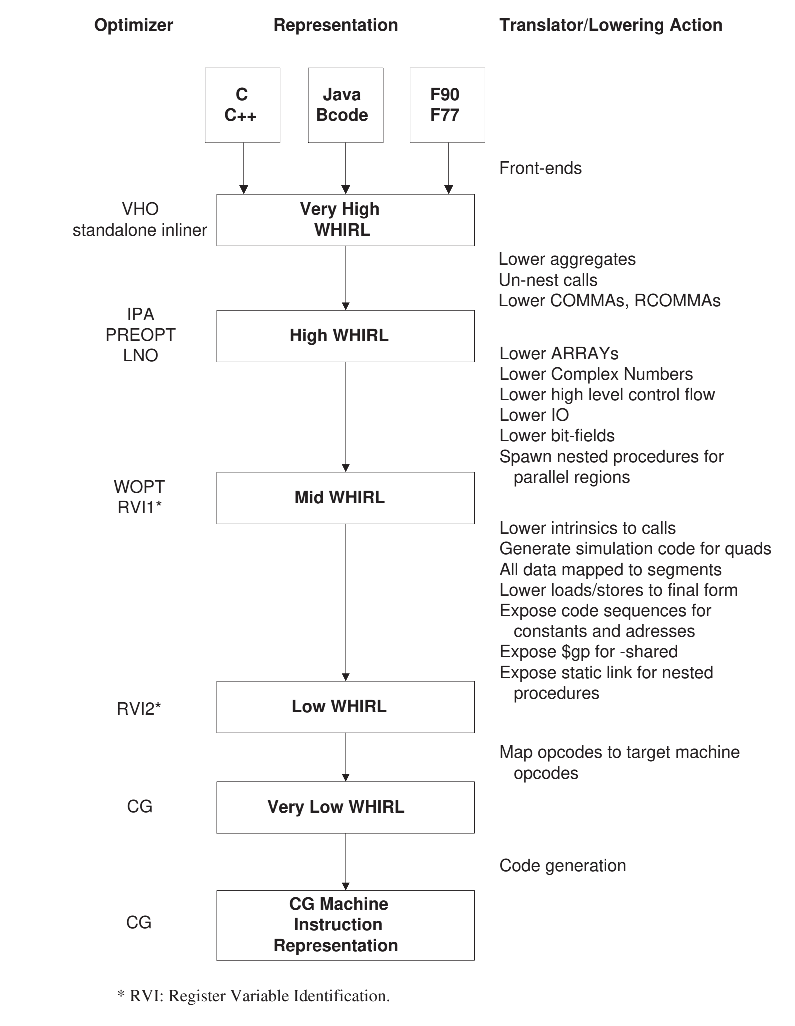
Superado este punto el compilador puede volver a generar un archivo en Fortran o C a partir de la representación intermedia (WHIRL2f/WHIRL2c), sin que esto impida continuar con el resto de la compilación. A continuación se traducen los operadores IO, que representan llamadas de E/S de Fortran, a construcciones de más bajo nivel[1]. Finalmente podemos pasar directamente a L WHIRL e ir a la generación de código (CG), o pasar primero por el optimizador global (Main opt) que actúa sobre M WHIRL (Medium).

2.3 Representación intermedia (WHIRL)

WHIRL es el lenguaje de representación intermedia utilizado por el compilador. Se trata de una representación en formato binario y estructura de árbol que ha sido diseñada para soportar las construcciones de distintos lenguajes (inicialmente C++, Java, Fortran77 y Fortran90). El hecho de utilizar una representación intermedia común permite, además, poder generar código para distintas plataformas.

[1]El principal motivo de la existencia de este operador específico para llamadas de E/S en Fortran es facilitar su posterior traducción a Fortran.

Figura 2.3: Descenso progresivo de la representación intermedia



Otra característica importante de WHIRL es que actúa cómo interfaz común entre las distintas etapas que conforman el back-end del compilador. Esto ofrece ventajas considerables, como por ejemplo la posibilidad de aplicar un mismo algoritmo en etapas diferentes sin necesidad de tener que reescribirlo para la representación vigente en ese momento.

Para una mejor adaptación a cada una de las fases del compilador existen distintos niveles de WHIRL. En total son cinco, y van desde WHIRL de muy alto nivel (Very High WHIRL) hasta WHIRL de muy bajo nivel (Very Low WHIRL), pasando por High , Mid y Low WHIRL. La figura 2.3 nos muestra a qué nivel actúan las distintas optimizaciones y las operaciones que se realizan para pasar de uno a otro.

Existen algunas herramientas para poder consultar la representación intermedia generada por el compilador. Para ello tendremos que indicar al compilador que no elimine los archivos que la contienen mediante la opción -keep. Los archivos WHIRL tienen extensión .B y podemos traducirlos a una representación en ASCII con el comando ir_b2a. Whirl2c y whirl2f traducen la representación intermedia a C y a Fortran respectivamente, aunque debemos tener en cuenta que esta conversión sólo es posible para representaciones en Very High WHIRL o High WHIRL, y que los archivos generados no siempre van a ser fácilmente compilables. Podemos ordenar al compilador que genere una representación en C o Fortran de forma automática mediante las opciones -clist o -flist.

A modo de resumen, estas serían las características más destacadas del lenguaje de representación intermedia:

- Interfaz común entre componentes
- Soporte para múltiples lenguajes y múltiples plataformas
- Una sola representación intermedia (IR) con cinco niveles
- Descenso progresivo de nivel a medida que avanza la compilación
- Las estrategias de optimización están ligadas a un determinado nivel de representación intermedia.

Para una especificación completa de WHIRL consultar [6]. Consultar [7] para información sobre la tabla de símbolos.

2.4 Front-ends

Existe un front-end para cada uno de los lenguajes soportados por el compilador. Su función consiste en realizar un análisis léxico, sintáctico y semántico de los archivos fuente, generando la representación intermedia en WHIRL como salida.

Los lenguajes soportados hasta el momento son C, C++ y Fortran90/95. Los front-ends para C y C++ están basados en los de los compiladores gcc y g++ de GNU respectivamente. El front-end para Fortran90/95 proviene del utilizado en los compiladores MIPSpro de SGI. Para agregar un nuevo lenguaje al compilador sería suficiente crear un nuevo front-end capaz de traducir código fuente escrito en ese lenguaje a WHIRL.

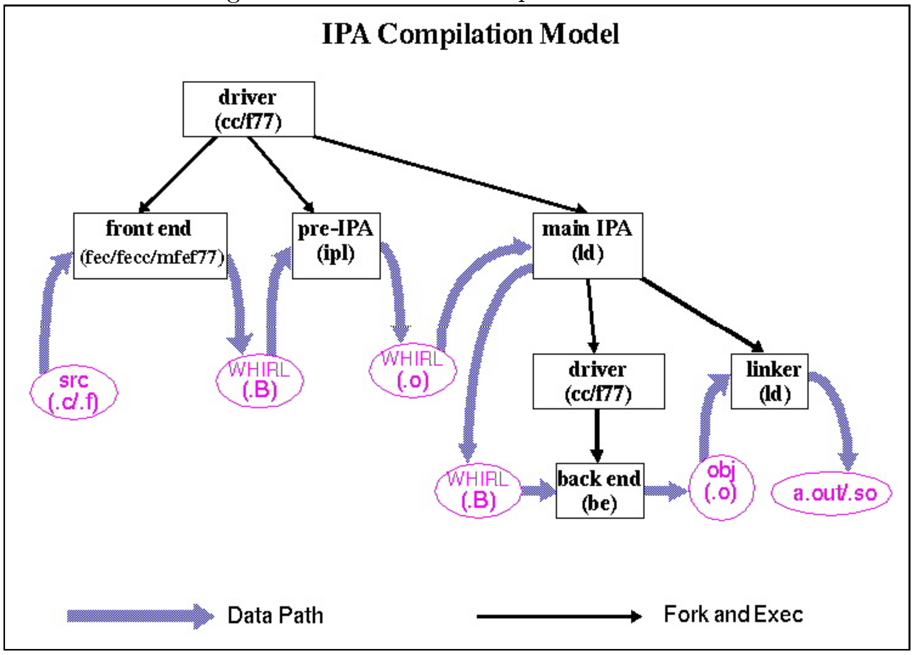
2.5 Analizador inter-procedural (IPA)

La mayoría de optimizaciones se realizan a nivel de rutina (función o procedimiento). De este modo se afrontan problemas más manejables y se favorece la compilación separada, puesto que se permite al compilador centrar su atención en el archivo fuente en curso.

Este enfoque tiene importantes inconvenientes. Frecuentemente, esta visión restringida forzará al compilador a trabajar en base al peor caso posible. El analizador interprocedural analiza más de una rutina (a poder ser, el programa completo) y transfiere información más allá de los límites de éstas. Los resultados obtenidos quedan disponibles para las siguientes fases de compilación.

La figura 2.4 muestra el modelo de compilación con IPA. Inicialmente, el front-end genera la representación intermedia de uno de los archivos fuente. A

Figura 2.4: Modelo de compilación con IPA



continuación, la etapa local (ipl) del analizador inter-procedural genera un archivo con extensión .o que no contiene código relocalizable, sinó una representación en WHIRL de sus procedimientos con información adicional.

Una vez hecho esto con cada uno de los archivos fuente, se llama al enlazador (main IPA/ld), que es el único componente que tiene una visión global de todos los objetos que participarán en la creación del programa o librería dinámica final, y éste invoca el IPA, que analiza y transforma todos los objetos WHIRL de entrada de forma conjunta, generando versiones modificadas de los mismos.

El siguiente paso consiste en llamar al compilador (driver) para cada uno de los archivos generados, produciendo los respectivos archivos objeto de código relocalizable. Finalmente, se llama al enlazador de nuevo para que enlace estos objetos y genere un ejecutable o librería dinámica como haría normalmente.

Como acabamos de ver, el IPA se ejecuta en dos etapas: una local, que viene a continuación del front-end y que recopila información local, y una fase principal, donde se lleva a cabo el análisis y la optimización.

El análisis y la optimización constituyen a su vez dos etapas dentro de la fase principal. Éstas son algunas de las tareas que se llevan a cabo:

- Análisis:

  PIC[2]symbol analysis Consiste en encontrar aquellos nombres que son declarados como externos y que, en consecuencia, se deben acceder mediante indirecciones. Intenta convertirlos en nombres internos permitiendo la realización de llamadas más simples y rápidas. De este modo también se reduce el número de entradas en la tabla global de offsets.

Constant global identification Identificación de constantes globales.

Scalar mod/ref Proporciona información sobre qué posiciones de memoria son modificadas o referenciadas por una llamada a función.

Array section Describe el conjunto de elementos de un array accedidos por el programa.

Code layout for locality Intenta ordenar los procedimientos de forma que se reduzcan los conflictos en la cache de instrucciones.

- Optimización

Inlining Llamadas a procedimiento se sustituyen por una copia debidamente modificada del procedimiento llamado, aunque este procedimiento se encuentre en otro archivo.

Intrinsic function library inlining Inlining de funciones intrínsecas.

Cloning El compilador crea copias especializadas de un procedimiento. Particionando los puntos desde los que se llama a la función, se crean nuevas oportunidades para la optimización.

Dead function, variable elimination Elimina funciones y variables que no son llamadas o utilizadas por el programa.

Constant propagation Los parámetros formales que siempre reciben un valor constante y aquellas variables que se inicializan con un valor que se mantiene constante pueden ser reemplazados por la constante.

Otras características destacables del IPA son que se trata de un proceso transparente al usuario, ya que éste sólo tiene que añadir la opción de compilación correspondiente, y que está integrado con el resto del compilador. La información obtenida durante esta fase se transmite al optimizador de bucles, al optimizador global y al generador de código. Además, el hecho de utilizar algún objeto .o o DSO normal no lo deshabilita.

2.6 Optimizador de bucles (LNO)

Los programas de cálculo numérico acostumbran a consumir buena parte de su tiempo en la ejecución de bucles. El componente LNO (Loop Nest Optimizer/Parallelizer) lleva a cabo optimizaciones de alto nivel que pueden mejorar considerablemente el rendimiento de un programa a través de un mejor aprovechamiento de la cache y el paralelismo a nivel de instrucción.

LNO se ejecuta por defecto al compilar con -O3. Puesto que forma parte del back-end y trabaja sobre la representación en WHIRL, es aplicable sea cual sea el lenguaje de programación. LNO analiza las dependencias a nivel de bucle, puede realizar transformaciones para mejorar el rendimiento en un procesador y paralelizar de forma automática o basándose en directivas OpenMP.

LNO realiza, entre otras, las siguientes optimizaciones:

Loop Fission Bucles de gran tamaño se dividen en bucles más pequeños y manejables.

[2]PIC = Position Independent Code.

Loop Fusion Se combinan bucles adyacentes en uno solo para mejorar el uso de la cache y facilitar la planificación de instrucciones.

Loop Unroll and Jam Se reduce el número de iteraciones replicando el cuerpo del bucle. De este modo podemos conseguir mejoras como un menor número de loads/stores o una planificación de instrucciones más eficiente.

Loop Interchange Intercambio de bucles anidados para conseguir una mejor localidad espacial.

Loop Peeling Consiste en sacar fuera la primera (o última) iteración del bucle. Puede usarse como paso previo para la aplicación de otras optimizaciones. Permite la ejecución de código invariante una sola vez.

Loop Tiling Incrementa la profundidad de una anidación de bucles. El espacio de iteraciones inicial se divide en subespacios más pequeños que intentan sacar el máximo partido de la localidad espacial y temporal.

Vector Data Prefetching Uso de instrucciones para traer datos de memoria a la cache de forma anticipada. La latencia de esta operación quedará escondida por la ejecución de otras instrucciones y los datos estarán disponibles en el momento en que se necesiten.

En lo que a paralelización se refiere, parece que se encuentra parcialmente implementada. Según hemos podido leer en la lista de distribución de soporte del compilador [4], el front-end de Fortran sí soporta OpenMP, pero no se puede decir lo mismo de los front-ends de C y C++ (recordamos que mientras el primero está basado en el de MIPSpro de SGI, los de C y C++ están basados en los de GNU).

2.7 Optimizador global (WOPT)

Los análisis y transformaciones aplicados por el optimizador global abarcan todo el cuerpo de un procedimiento, en contraposición a optimizaciones más locales efectuadas por el generador de código y a aquellas que van más allá de los límites de una función (optimizador inter-procedural).

Dependiendo de las opciones de compilación el optimizador global será llamado en distintas ocasiones. Las fases IPA y LNO se basan en el optimizador global para pre-acondicionar el código, y utilizan la información de use-def y alias generada por el optimizador global para realizar sus análisis y operaciones. Estas ejecuciones previas del optimizador global (PREOPT) no ejecutan el juego completo de optimizaciones. Antes de pasar a la generación de código se realiza la ejecución principal (MAINOPT).

El optimizador implementa las optimizaciones globales tradicionales mediante algoritmos basados en SSA (Static Single Assignment). En una representación de este tipo cada variable aparece como destino de una sola asignación. Esto simplifica algunos algoritmos permitiendo implementaciones más robustas. Cada optimización preserva la forma SSA y se puede volver a aplicar cuando sea necesario.

Algunas de las optimizaciones que se realizan durante esta fase son:

Dead Code Elimination Eliminación de variables e instrucciones cuyos valores no se van a utilizar.

Partial Redundancy Elimination Intenta minimizar la repetición de cálculos redundantes.

Strength Reduction Sustituye operaciones costosas por otras de menor coste.

Induction Variable Elimination Eliminación de variables de inducción.

Register Variable Identification (Register Promotion) Decidir qué variables se pueden pasar a registros.

2.8 Generador de código (CG)

El generador de código actúa sobre una representación intermedia propia (CGIR) próxima a instrucciones de lenguaje máquina. A continuación nombramos algunos de los análisis y transformaciones llevados a cabo durante la generación de código:

IF conversion Transforma dependencias de control en dependencias de datos. Elimina saltos condicionales.

Software pipelining Solapa la ejecución de distintas iteraciones de un bucle.

Predication Añade un operando boleano a algunas instrucciones. El valor del operando determina si la instrucción modifica el estado de la arquitectura.

Global instruction scheduling Planificación de instrucciones considerando más de un bloque básico.

Register allocation Assignación de variables a registros.

Local instruction scheduling Planificación de instrucciones a nivel de bloque básico.

La etapa de generación de código es la más afectada por las modificaciones de Intel y CAS y es, por tanto, donde pueden existir las mayores diferencias entre las distintas versiones del compilador.

Capítulo 3

WHIRL

En el capítulo de introducción hemos comentado ya algunos aspectos del lenguaje de representación intermedia del compilador. En este capítulo vamos a verlo con más detalle, pero para una descripción completa, el documento de referencia es [6].

El lenguaje de representación intermedia utilizado se denomina WHIRL. Se definen distintos niveles de WHIRL, desde Very High WHIRL hasta Very Low WHIRL. Cada optimización llevada a cabo actúa en un determinado nivel del lenguaje de representación intermedia. Los front-ends generan Very High WHIRL y a medida que avanza la compilación se produce un descenso progresivo del nivel de representación hasta llegar al generador de código. Éste traducirá el WHIRL de más bajo nivel a su propia representación interna. La figura 2.3 de la página 6 muestra el vínculo entre las distintas fases de optimización y el nivel de WHIRL en el que actúan, así como las operaciones que se deben llevar a cabo para pasar de un nivel a otro.

La siguiente tabla muestra las diferencias entre una representación intermedia de alto nivel y una de bajo nivel.

Caraterísticas	RI alto nivel	RI bajo nivel
tipo de construcciones	muchas	pocas
longitud de las secuencias de código	cortas	largas
estructura	jerárquica	plana

3.1 Estructura de un nodo

Las estructuras de datos que se utilizan para almacenar la representación intermedia de un programa consisten en un árbol de instrucciones para cada una de las unidades de programa (funciones) y unas tablas de símbolos (nos ocuparemos de ellas en el capítulo 4).

Cada uno de los nodos tendrá distintos campos, en función del tipo de operación que represente, pero todos ellos disponen de los siguientes campos comunes:

operator	operación que realiza la instrucción.
res	tipo del resultado de la operación.
desc	tipo de los operandos de la operación.
kid_count	número de hijos del nodo.
map_id	índice para dar soporte al mecanismo de mapeo.

Podemos clasificar los operadores en tres categorías:

1. operadores de control de flujo estructurado
2. sentencias
3. expresiones

Los operadores de control de flujo estructurado no pueden ser descendientes de sentencias, de modo similar éstas no pueden descender de expresiones. Además no se admite la anidación de sentencias. De todos modos, en VH y H WHIRL se permiten algunas excepciones a estas reglas.

El operador y los campos res y desc especifican de forma completa una operación (OPCODE).

El **mecanismo de mapeo** al que nos referimos en la descripción de map_id funciona de la siguiente manera: sea cual sea su implementación, podemos ver map_id como el índice de una fila dentro de una tabla donde cada columna representa un atributo del nodo. El compilador ofrece funciones para poder añadir columnas a esta tabla y de este modo ampliar la información sobre un nodo. Este mecanismo puede ser utilizado, por ejemplo, para asociar a determinados nodos un puntero hacia una estructura de datos con información adicional necesaria durante una determinada fase de la compilación. Superada esa fase, podemos liberar la columna correspondiente. Mediante este mecanismo se va adaptando la cantidad de información almacenada para cada nodo a las necesidades de cada etapa de la compilación.

Otros campos que vamos a encontrar frecuentemente son:

kids	vector de punteros a los hijos del nodo.
first	puntero al primer elemento de la lista de hijos.
last	puntero al último elemento de la lista de hijos.
next	siguiente sentencia.
previous	sentencia anterior.
offset	aplicable a los operadores de tipo load y store.

No todos los nodos contienen todos los campos que acabamos de mencionar, incluso algunos de ellos se excluyen mutuamente. Por ejemplo, el campo kids sólo es válido para aquellos operadores cuyo número de hijos es conocido en el momento de su creación. Si a priori no se conoce el número de hijos (el operador BLOCK es el único caso) deberemos utilizar los campos first y last. Podremos recorrer los hijos del operador BLOCK mediante los campos next y previous de cada uno de ellos. Los campos next y previous sólo están disponibles en los operadores de tipo sentencia (statement).

Podemos encontrar otros campos que no aparecen en la lista anterior, por ejemplo campos con la ubicación de una sentencia en el archivo fuente, o punteros a las tablas de símbolos.

3.2 Resumen de operadores

En esta sección veremos qué operadores podemos encontrar en la representación en WHIRL de un programa. Para ello los clasificaremos en distintos grupos. La siguiente tabla indica a qué catagoría pertenecen los elementos de cada grupo:

Grupo	*Categoría*
control de flujo estructurado agregados (where)	control de flujo estructurado
control de flujo no estructurado llamadas otras sentencias acceso a memoria (store)	sentencias
acceso a memoria (load) conversión de tipos operadores de expresión agregados (triplet, arrayexp, arrsection) constantes	expresiones

El archivo opcode_gen_core.cxx ofrece una clasificación exacta de cada operador según distintos criterios.

En algunos grupos aparece un apartado de operadores relacionados. Se trata de operadores que siempre aparecen acompañando a alguno de los operadores del grupo.

En las tablas que siguen a continuación la primera columna indica el nombre del operador, la segunda columna es una breve descripción y la última columna indica en qué nivel de WHIRL puede aparecer el operador. En algunos casos aparece una columna adicional al principio refinando la clasificación de los operadores.

3.2.1 Control de flujo estructurado

Construcciones de naturaleza jerárquica que controlan el flujo de ejecución del programa.

FUNC_ENTRY	Entrada a función. Raíz de todo árbol.	VH-VL
BLOCK	Lista de sentencias.	VH-VL
REGION	Sub-región anidada. Define una unidad de compilación (caja negra).	H-VL
DO_LOOP	Misma semántica que DO de Fortran.	VH-H
DO_WHILE	Bucle. La condición se evalúa al final.	VH-H
WHILE_DO	Bucle. La condición se evalúa al principio.	VH-H
IF	Sentencia condicional estructurada.	VH-H

Operadores relacionados:

ALTENTRY	Entrada alternativa a la función (Fortran).	VH-VL
IDNAME	Se utiliza para especificar parámetros formales en FUNC_ENTRY y ALTENTRY, y también para la variable de inducción en DO_LOOP.	VH-VL
RETURN	Retorno del procedimiento.	VH-VL
RETURN_VAL	Retorno devolviendo un valor.	VH-H
REGION_EXIT	Salida de una región (obligatorio). El resto de saltos deben producirse dentro de la región.	VH-VL
LOOP_INFO	Aporta información sobre un bucle.	H-VL

3.2.2 Control de flujo no estructurado

Estas son el resto de sentencias de control de flujo. Su naturaleza no es jerárquica.

	LABEL	Define una etiqueta.	VH-VL
salto incondicional	GOTO	Salto a la etiqueta dentro del procedimiento.	VH-VL
	GOTO_OUTER_BLOCK	Salto desde un procedimiento anidado a un procedimiento padre.	VH-VL
	AGOTO	Salto a una dirección calculada.	VH-VL
salto condicional	SWITCH	Sentencia switch.	VH
	CASEGOTO	Relaciona valor switch/destino salto.	VH
	COMPGOTO	Selecciona un salto a partir de un índice.	VH-M
	XGOTO	Sustituye a COMPGOTO en L y VL-WHIRL.	L-VL
	TRUEBR	Salto a la etiqueta si se cumple la condición.	VH-VL
	FALSEBR	Salto a la etiqueta si falla la condición.	VH-VL

3.2.3 Llamadas

Los operadores de este grupo se utilizan para especificar llamadas a función. En general, los lenguajes de programación permiten la anidación de llamadas. En WHIRL esto sólo se permite en su nivel más alto (VH), mediante la admisión de sentencias de tipo llamada como hijas de nodos de tipo expresión[1]. Durante la transformación a H-WHIRL estas anidaciones desaparecerán. Las llamadas a funciones se efectuarán de forma secuencial y sus valores de retorno se almacenarán en pseudo-registros (pregs).

directa	CALL	Llamada directa.	VH-VL
indirecta	ICALL	El hijo n-1 devuelve la dirección del procedimiento.	VH-VL
	VFCALL	Llamada a función virtual. Similar a ICALL.	VH-H
PIC	PICCALL	Llamada independiente de la posición. Proviene de una llamada CALL al compilar con la opción -shared.	L-VL
intrinsic	INTRINSIC_CALL	Llamada a procedimiento intrínseco. Dependiendo del procedimiento y opciones de compilación se generará una llamada o una secuencia de instrucciones.	VH-M
	IO	Procedimiento intrínseco de E/S en Fortran.	VH-H
ASM	ASM_STMT	Secuencia de instrucciones en ensamblador insertada en el código fuente.	VH-VL

Operadores relacionados:

[1]Se trata de una excepción, ya que en general no se permite la aparición de elementos de tipo sentencia como descendientes de una expresión.

PARM	Especifica un parámetro actual de una llamada.	VH-VL
COMMA	(ver operadores de expresión binarios)	VH
ASM_INPUT	Especifica un operando de entrada de un ASM_STMT.	VH-VL
INTRINSIC_OP	Aplica la operación intrínseca especificada por el campo intrinsic a los operandos especificados por los hijos.	VH-M
IO_ITEM	Representa un elemento especificado en una sentencia Fortran de E/S.	VH-H

3.2.4 Otras sentencias

Este apartado contiene aquellas sentencias que no son de control de flujo ni stores. Los stores se describen en el apartado 3.2.5.

EVAL	El hijo 0 es evaluado.	VH-VL
PRAGMA	Proporciona directivas de compilación.	VH-VL
XPRAGMA	Las directivas se especifican con respecto al subárbol determinado por el hijo 0.	VH-VL
PREFETCH	Prefetch de la dirección indicada por el hijo 0 más un offset.	H-VL
PREFETCHX	Proviene de un PREFETCH. La dirección se obtiene sumando sus dos hijos (que deben ser LDIDs).	M-VL
COMMENT	Comentario. No genera código ejecutable.	VH-VL
TRAP	El campo offset contiene un break code que determina como se va a generar el trap.	VH-VL
ASSERT	Si la condición especificada por el hijo 0 no devuelve cierto produce el mismo efecto que una instrucción de tipo TRAP.	VH-VL
AFFIRM	Afirma que la condición especificada por el hijo 0 siempre devuelve cierto, permitiendo así más optimizaciones.	VH-VL
FORWARD _BARRIER	Designa una barrera para el movimiento de instrucciones de acceso a memoria dentro del código en la dirección del control de flujo.	VH-VL
BACKWARD _BARRIER	Como FORWARD_BARRIER pero en dirección contraria.	VH-VL

3.2.5 Acceso a memoria

Una de las tareas que debe realizar el compilador es determinar la disposición de los datos en memoria. En WHIRL las variables y los datos estáticos se organizan en bloques de memoria. Para especificar un acceso directo a una variable, se utiliza la tripleta <st_idx (índice de la tabla de símbolos), offset, size>. Cada entrada de la tabla de símbolos contiene un campo que especifica el bloque donde se almacena la variable y otro campo que especifica el offset respecto al inicio del bloque. El offset total se calcula mediante la suma del offset contenido en el nodo y el del símbolo correspondiente en la tabla de símbolos. A medida que avanza la compilación los bloques de memoria se fusionan en bloques de mayor tamaño. Cuando esto ocurre sólo es necesario actualizar el offset de la tabla de símbolos.

A continuación listamos los distintos operadores disponibles para especificar accesos a memoria.

direct.	load	LDID	Load directo de una dirección.	VH-VL
	store	STID	Store directo del valor en hijo 0.	VH-VL
indirect.	load	ILOAD	Load de una dirección calculada (hijo 0+offset).	VH-VL
		ILOADX	Load de la dirección obtenida al sumar 2 pregs.	M-VL
		MLOAD	Load de múltiples bytes. Dirección: hijo 0+offset.	M-L
	store	ISTORE	Guarda hijo 0 en hijo 1+offset.	VH-VL
		ISTOREX	Store en la dirección obtenida al sumar 2 pregs.	M-VL
		MSTORE	Store de múltiples bytes.	M-L

Los campos de bits son campos de una estructura (tipo struct en C) donde se ha especificado la longitud en bits de cada uno de ellos. A partir de M-WHIRL (en circunstancias especiales, también en niveles superiores), utilizaremos las siguientes instrucciones para acceder a los campos de bits (bit-fields).

direct.	load	LDBITS	Load directo.	VH-VL
	store	STBITS	Store directo.	VH-VL
indirect.	load	ILDBITS	Load indirecto.	VH-VL
	store	ISTBITS	Store indirecto.	VH-VL

Otra tarea importante que se debe llevar a cabo durante el proceso de compilación es la asignación de registros. Los programas en WHIRL pueden utilizar un número ilimitado de pseudo-registros (pregs). Éstos se mapearán contra los registros físicos del procesador y algunos de ellos tendrán que ser volcados a memoria. Nos referiremos a los pseudo-registros mediante las instrucciones LDID y STID, pero no podremos utilizar LDA para obtener su dirección.

Para finalizar, nombraremos algunos operadores que, sin ser loads ni stores, también están estrechamente relacionados con el acceso a memoria.

LDA	Devuelve la dirección resultante de sumar el campo offset con la dirección del símbolo indicado por st_idx.	VH-VL
LDMA	Como LDA, pero no se puede tomar la dirección como constante porque esta puede cambiar (por cuestiones de enlace dinámico).	VH-VL
LDA_LABEL	Devuelve la dirección de la etiqueta, que se indica con número de etiqueta.	VH-VL
ILDA	Dirección en bytes obtenida al sumar el campo offset al hijo 0.	VH

ARRAY	Utiliza las reglas de direccionamiento de arrays para devolver una dirección.	VH-H
ALLOCA	Devuelve un apuntador a un bloque de espacio no inicializado en la pila local que se obtiene modificando el stack pointer.	VH-VL
DEALLOCA	Restaura el stack pointer con el valor contenido en su hijo 0. El valor de este hijo se debe haber obtenido mediante una llamada a ALLOCA pidiendo 0 bytes.	VH-VL

3.2.6 Constantes

Los operadores INTCONST y CONST nos permiten especificar valores constantes. El primero almacena un valor entero constante en el propio nodo WHIRL. El valor devuelto por CONST se obtiene de las tablas de símbolos. Se trata de operadores hoja, ya que nunca tendrán descendientes.

INTCONST	Devuelve un valor entero.	VH-VL
CONST	Devuelve un valor literal.	VH-VL

3.2.7 Conversión de tipos

WHIRL también aporta distintos operadores para especificar la conversión de tipos. Son los siguientes:

CVT	Convierte un valor de tipo desc a tipo res.	VH-VL
CVTL	El valor calculado por el hijo 0 es tratado como un valor con la longitud que indica el campo cvtl_bits.	VH-VL
TAS	Trata el valor calculado por el hijo 0 como si fuera de tipo res.	VH-VL

3.2.8 Operadores de expresión

Éste es el grupo más numeroso y contiene, entre otros, todos los operadores aritméticos y lógicos que nos van a permitir la construcción de expresiones en WHIRL. Los dividimos en tres subgrupos: operaciones unarias, operaciones binarias y operaciones ternarias.

Operaciones unarias

NEG	Negación aritmética.	VH-VL
ABS	Valor absoluto.	VH-VL
SQRT	Raíz cuadrada.	VH-VL
RECIP	Recíproco.	VH-VL
FIRSTPART	Parte real de un número complejo. Parte alta de un número en coma flotante SGI.	VH-M
SECONDPART	Parte imaginaria de un número complejo. Parte baja de un número en coma flotante SGI.	VH-M
PAREN	Paréntesis.	VH-VL
RND	Redondeo al entero más próximo.	VH-VL
TRUNC	Redondeo hacia 0.	VH-VL
CEIL	Redondeo hacia $+\infty$.	VH-VL
FLOOR	Redondeo hacia $-\infty$.	VH-VL
BNOT	Negación bit a bit.	VH-VL
LOWPART	Opera sobre un LDID de un preg que contiene el resultado de un XMPY o un DIVREM y devuelve la parte que representa la parte baja de la multiplicación o el cociente de la división respectivamente.	M-VL
HIGHPART	Parte que representa la parte alta de la multiplicación o el resto de la división.	M-VL
MINPART	Opera sobre un LDID de un preg que contiene el resultado de un MINMAX y devuelve la parte del mínimo.	M-VL
MAXPART	Devuelve la parte que representa el máximo.	M-VL
EXTRACT_BITS	Extracción de los bits indicados por bit_offset y bit_size.	VH-VL

Operaciones binarias

PAIR	Según el tipo res genera un número complejo o un número en coma flotante SGI.	VH-M
ADD	Suma.	VH-VL
SUB	Resta.	VH-VL
MPY	Multiplicación.	VH-VL
HIGHMPY	Devuelve la parte alta de la multiplicación.	VH-VL
XMPY	Multiplicación. Su resultado sólo puede ser accedido mediante LOWPART Y HIGHPART.	M-VL
DIV	División.	VH-VL
MOD	Módulo.	VH-VL
REM	Resto de una división.	VH-VL
DIVREM	Devuelve un resultado compuesto que incluye el cociente y el resto.	M-VL
MAX	Máximo.	VH-VL
MIN	Mínimo	VH-VL
MINMAX	Devuelve un resultado compuesto que incluye el mínimo y el máximo.	M-VL
EQ	Igual.	VH-VL
NE	Distinto.	VH-VL
GE	Mayor o igual.	VH-VL
GT	Mayor estricto.	VH-VL
LE	Menor o igual.	VH-VL
LT	Menor estricto.	VH-VL
BAND	AND bit a bit.	VH-VL
BIOR	OR bit a bit.	VH-VL
BNOR	NOR bit a bit.	VH-VL
BXOR	XOR bit a bit.	VH-VL
LAND	AND lógico.	VH-VL
LIOR	OR lógico.	VH-VL
CAND	Sólo evalúa el hijo 1 si el hijo 0 vale 1.	VH-H
CIOR	Sólo evalúa el hijo 1 si el hijo 0 vale 0.	VH-H
SHL	Desplazamiento hacia la izquierda.	VH-VL
ASHR	Desplazamiento aritmético hacia la derecha.	VH-VL
LSHR	Desplazamiento lógico hacia la derecha.	VH-VL
COMPOSE_BITS	Genera un nuevo entero por composición de los dos operandos.	VH-VL
RROTATE	Rotación a la derecha.	VH
COMMA	Ejecuta el bloque de instrucciones contenido en el hijo 0 y devuelve el valor de la expresión del hijo 1. Mediante este operador es posible la anidación de llamadas.	VH
RCOMMA	Evalua el hijo 0 (cuyo valor se devolverá) y después ejecuta el bloque de instrucciones del hijo 1.	VH

Operaciones ternarias

SELECT	Devuelve el resultado del hijo 1 o el hijo 2 en función del valor booleano que retorna el hijo 0.	H-VL
CSELECT	Como SELECT pero sólo se evalúa el hijo que indica la condición.	VH
MADD	Multiplicación y suma (Hijo 1 x Hijo 2) + Hijo 0.	VL
MSUB	Multiplicación y resta (Hijo 1 x Hijo 2) - Hijo 0.	VL
NMADD	Devuelve -(Hijo 1 x Hijo 2) + Hijo 0.	VL
NMSUB	Devuelve -(Hijo 1 x Hijo 2) - Hijo 0.	VL

3.2.9 Especificación de agregados

Las siguientes operaciones permiten expresar en WHIRL de alto nivel algunas construcciones de Fortran 90 relacionadas con el manejo de arrays.

expr.	TRIPLET	Array uni-dimensional de enteros en una progresión lineal. Hijo 0: valor base. Hijo 1: stride. Hijo 2: núm. de valores.	VH
	ARRAYEXP	Indica que el hijo 0 es una expresión que devuelve un array con un número de dimensiones igual a kid_count-1. Los hijos 1 hasta kid_count-1 indican el núm. de elementos en cada dimensión.	VH
	ARRSECTION	Corresponde a ARRAY, pero genera un agregado de direcciones.	VH
ctrl. flow	WHERE	Implementa el masked array assignment de Fortran 90.	VH

3.3 Ejemplo

A continuación mostramos el código en WHIRL generado para un programa de ejemplo. Para pasar la representación intermedia a formato texto, podemos utilizar el comando ir_b2a.

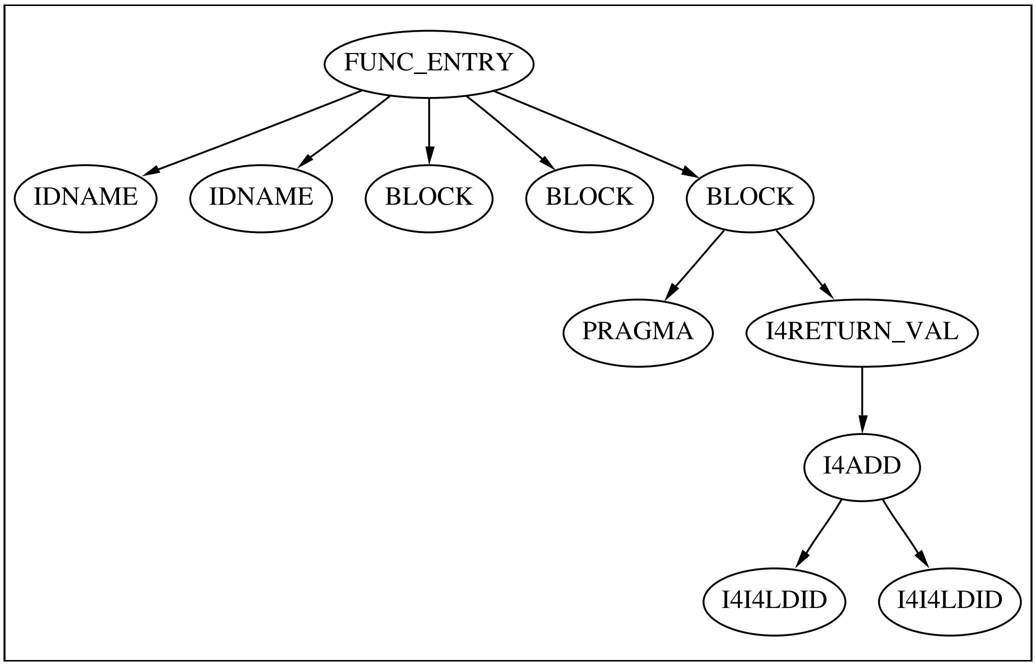
```
Usage: ir_b2a [-st] [-v] <Binary IR> [<ASCII IR>]
       (will write to stdout if ascii file not given)
       -st option will also print out the symbol table
       -opcodes option will print out opcodes as seen
       -lines option will print out line numbers
       -global_local <.G file> option will use separate global table
       -sym <.G file> is the same as -global_local
```

El archivo con la representación intermedia lo hemos obtenido utilizando la opción de compilación -keep. En este caso no se han aplicado optimizaciones.

En la salida generada por este comando todas las sentencias pertenecientes a un mismo bloque (BLOCK) se imprimen en el orden de ejecución. Las expresiones aparecen en notación postfija, y las construcciones de control de flujo estructurado en notación prefija. Para facilitar la lectura también se introducen palabras clave que no corresponden a ningún operador WHIRL, como por ejemplo END_BLOCK, ELSE y LOC. Esta última indica qué línea del programa fuente corresponde a una determinada porción de código en WHIRL.

Programa Ejemplo

Figura 3.1: Árbol PU suma()




```
int suma( int a, int b )
{
  return a + b;
}

int main( void )
{
  int m[5] = {1,2,3,4,5};
  int i, r;

  r=0;
  i=0;
  while (i<5)
  {
    r = suma( r, m[i] );
    i++;
  }

  printf( "%d\n", r );

  return 0;
}
```

La figura 3.1 muestra la estructura del árbol generado para la función suma.

Los dos nodos IDNAME corresponden a los parámetros de entrada. El primero de los dos bloques sin descendientes está destinado a contener la lista de pragmas que se deben considerar al compilar la función. El segundo es para los pragmas que se deben tener en cuenta en el punto donde la función es llamada. Finalmente encontramos el cuerpo de la función, con la definición de un pragma por un lado, y el retorno del resultado de la suma de los parámetros de entrada por el otro.

A continuación mostramos la traducción a ASCII del código WHIRL correspondiente a suma().
En $< 2, 1, a >$

> el 2 indica el nivel de anidación de la tabla de símbolos de la función y
>
> el 1 es el índice de la variable a dentro de la tabla de símbolos.

En $T < 4, .predef\_I4, 4 >$

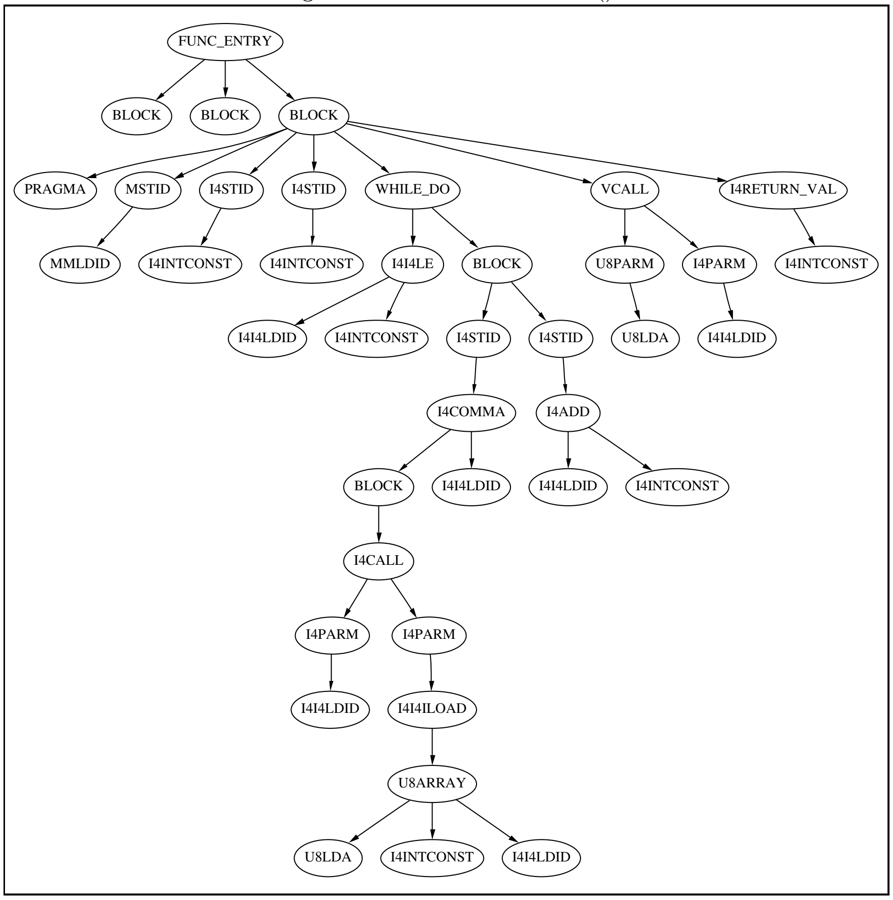
> el primer 4 es el índice correspondiente al tipo dentro de la tabla de tipos,
>
> .predef_I4 representa el nombre (existen nombres predefinidos para los tipos más frecuentes) y
>
> el último 4 indica la alineación para este uso en concreto del tipo. Todos los tipos se guardan en una tabla.

WHIRL función suma()

```
 LOC 0 0 source files:  1       "/users2/scratch/esantama/whirl/exemple.c"
 LOC 1 1 int suma( int a, int b )
 LOC 1 2 {
FUNC_ENTRY <1,20,suma>
 IDNAME 0 <2,1,a>
 IDNAME 0 <2,2,b>
BODY
 BLOCK
 END_BLOCK
 BLOCK
 END_BLOCK
 BLOCK
 PRAGMA 0 120 <null-st> 0 (0x0) # PREAMBLE_END
 LOC 1 3   return a + b;
   I4I4LDID 0 <2,1,a> T<4,.predef_I4,4>
   I4I4LDID 0 <2,2,b> T<4,.predef_I4,4>
  I4ADD
 I4RETURN_VAL
 END_BLOCK
 LOC 1 4 }
 LOC 1 5
```

La figura 3.2 muestra el árbol de main(). A continuación sigue su representación en ASCII.

Figura 3.2: Árbol PU main()



WHIRL función main()

```
 LOC 1 6 int main( void )
 LOC 1 7 {
FUNC_ENTRY <1,21,main>
BODY
 BLOCK
 END_BLOCK
 BLOCK
 END_BLOCK
 BLOCK
 PRAGMA 0 120 <null-st> 0 (0x0) # PREAMBLE_END
 LOC 1 8   int m[5] = {1,2,3,4,5};
  MMLDID 0 <2,2,m.init> T<29,.anonymous.1,4>
 MSTID 0 <2,1,m> T<29,.anonymous.1,4>
 LOC 1 9   int i, r;
 LOC 1 10
```

```
LOC 1 11   r=0;
 I4INTCONST 0 (0x0)
I4STID 0 <2,3,r> T<4,.predef_I4,4>
LOC 1 12   i=0;
 I4INTCONST 0 (0x0)
I4STID 0 <2,4,i> T<4,.predef_I4,4>
LOC 1 13   while (i<5)
WHILE_DO
  I4I4LDID 0 <2,4,i> T<4,.predef_I4,4>
  I4INTCONST 4 (0x4)
 I4I4LE
BODY
 BLOCK
LOC 1 14   {
LOC 1 15     r = suma( r, m[i] );
   BLOCK
     I4I4LDID 0 <2,3,r> T<4,.predef_I4,4>
    I4PARM 2 T<4,.predef_I4,4> #  by_value
       U8LDA 0 <2,1,m> T<30,anon_ptr.,8>
       I4INTCONST 5 (0x5)
       I4I4LDID 0 <2,4,i> T<4,.predef_I4,4>
      U8ARRAY 1 4
     I4I4ILOAD 0 T<4,.predef_I4,4> T<31,anon_ptr.,8>
    I4PARM 2 T<4,.predef_I4,4> #  by_value
   I4CALL 126 <1,20,suma> # flags 0x7e
   END_BLOCK
   I4I4LDID -1 <1,19,.preg_return_val> T<4,.predef_I4,4>
  I4COMMA
 I4STID 0 <2,3,r> T<4,.predef_I4,4>
LOC 1 16     i++;
   I4I4LDID 0 <2,4,i> T<4,.predef_I4,4>
   I4INTCONST 1 (0x1)
  I4ADD
 I4STID 0 <2,4,i> T<4,.predef_I4,4>
 END_BLOCK
LOC 1 17   }
LOC 1 18
LOC 1 19   printf( "%d\n", r );
  U8LDA 0 <1,23,(4_bytes)_"%d\n\000"> T<35,anon_ptr.,8>
 U8PARM 2 T<32,anon_ptr.,8> #  by_value
  I4I4LDID 0 <2,3,r> T<4,.predef_I4,4>
 I4PARM 2 T<4,.predef_I4,4> #  by_value
VCALL 126 <1,22,printf> # flags 0x7e
LOC 1 20
LOC 1 21   return 0;
 I4INTCONST 0 (0x0)
I4RETURN_VAL
END_BLOCK
```

3.4 Manipulación de los subárboles WHIRL

En esta sección comentamos brevemente algunas funciones para la creación y manipulación de nodos whirl. Veremos también como podemos recorrer un subárbol de la representación intermedia.

3.4.1 Acceso a los campos de datos

El compilador define los nodos WHIRL mediante una combinación de unions y structs de C (ver archivo wn_core.h). Define también un conjunto de funciones que facilitan el acceso a los campos de datos de los distintos tipos de nodo. Estas funciones se acompañan del prefijo WN:

> WN_yyy (WN *) devuelve el valor del campo yyy del nodo que recibe como parámetro.
>
> WN_set_yyy(WN *, tipo_valor) actualiza el valor del campo yyy.

Por ejemplo, la función WN_kid_count (const WN *) devuelve el número de hijos de un nodo. Sólo algunos campos disponen de una función de actualización.

Los diferentes nodos WHIRL ofrecen diferentes campos de datos. Es responsabilidad del programador utilizar estas funciones de acceso de forma correcta.

3.4.2 Manipulación de elementos

El archivo wn.h define distintas funciones para llevar a cabo la creación y eliminación de nodos WHIRL. Desde la función genérica:

```
WN *WN_Create(
    OPERATOR opr,
    TYPE_ID rtype,
    TYPE_ID desc,
    mINT16 kid_count
)
```

hasta funciones especializadas para los distintos tipos de nodo, como por ejemplo:

```
WN *WN_CreateDO(
    WN *index,
    WN *start,
    WN *end,
    WN *step,
    WN *body
)
```

Se definen también funciones de eliminación y copia.

3.4.3 El mecanismo de mapeo

El mecanismo de mapeo es un sistema muy utilizado por el compilador para asociar información adicional a los nodos WHIRL.

Podemos crear un nuevo mapeo de la siguiente forma:

```
LNO_Info_Map = WN_MAP_Create(&LNO_default_pool);
```

El parámetro LNO_default_pool indica en qué zona de memoria se debe solicitar espacio cuando sea necesario (el compilador implementa su propia gestión de la memoria dinámica).

Hecho esto podemos asociar información adicional a un nodo utilizando el mapeo que acabamos de crear. La función WN_MAP_Set() nos permite añadir dicha información, que recogeremos mediante la función WN_MAP_Get(). Podemos aprovechar un mismo mapeo para almacenar información distinta según el tipo de nodo.

```
WN_MAP_Set(LNO_Info_Map,wn,(void *)array);
DO_LOOP_INFO *dli = (DO_LOOP_INFO *) WN_MAP_Get(LNO_Info_Map,wn);
WN_MAP_Set (LNO_Info_Map, array_wn, aa);
ACCESS_ARRAY *array = (ACCESS_ARRAY *) WN_MAP_Get(LNO_Info_Map,wn);
```

Para terminar, se libera el mapeo cuando éste ya no es necesario.

```
WN_MAP_Delete(LNO_Info_Map);
```

3.4.4 Recorrido mediante llamadas recursivas

En ocasiones será necesario recorrer un subárbol WHIRL y actuar en función de los operadores encontrados. El siguiente código muestra una posible implementación de un recorrido en profundidad:

```
void foo(WN* wn_tree)
{
    OPCODE opcode = WN_opcode(wn_tree);

    switch (opcode) {
      case OPC_IF: ...
      case OPC_DO_WHILE: ...
      ...
    }

    if (opcode == OPC_BLOCK) {
      // recorre sentencias con la funcion WN_next()
      for (WN* wn = WN_first(wn_tree); wn != NULL; wn = WN_next(wn))
        // llamada recursiva
        foo(wn);
    } else {
      // llamada recursiva para cada uno de los hijos
      for (INT i = 0; i < WN_kid_count(wn_tree); i++)
        foo(WN_kid(wn_tree, i));
    }
}
```

Para conocer que clase de nodo representa wn_tree se ha utilizado la función WN_opcode(). Otra posibilidad sería utilizar la función WN_operator(). La diferencia entre un operador y un opcode radica en que el segundo incorpora información de tipo. Por ejemplo, existe un operador ADD que combinado con los distintos tipos de suma posibles genera dieciocho opcodes distintos:

operador	opcodes
OPR_ADD	OPC_I4ADD
	OPC_I8ADD
	OPC_U4ADD
	OPC_U8ADD
	OPC_F4ADD
	OPC_F8ADD
	OPC_FQADD
	OPC_F10ADD
	OPC_F16ADD
	etc.

El archivo opcode_gen_core.h contiene la relación completa de OPCODEs.

3.4.5 Recorrido mediante iteradores

Otra posibilidad al recorrer un subárbol WHIRL es la utilización de iteradores, por ejemplo los definidos en wn_util.h o, como en el ejemplo que sigue, los definidos en lwn_util.h para la manipulación de subárboles WHIRL durante la etapa de LNO.

La siguiente función visita todos los bucles contenidos en loop y devuelve la profundidad respecto a loop del bucle más interno.

```
Num_Inner_Loops(WN* loop)
{
  INT max_inner_loops = 0;

  // Recorrido:
  for (LWN_ITER* itr = LWN_WALK_TreeIter(WN_do_body(loop));
       itr;
       itr = LWN_WALK_TreeNext(itr)) {
    WN* wn = itr->wn;

    if (WN_operator(wn) == OPR_DO_LOOP) {
      // Se ha encontrado un bucle.
      INT num_inner_loops = 1;
      // Iteramos siguiendo los apuntadores al nodo padre
      // hasta encontrar el bucle loop:
      while ((wn = LWN_Get_Parent(wn)) != loop) {
        if (WN_operator(wn) == OPR_DO_LOOP) {
          num_inner_loops++;
        }
      }

      // Actualizamos profundidad maxima:
```

```
      if (num_inner_loops > max_inner_loops) {
        max_inner_loops = num_inner_loops;
        if (max_inner_loops >= MAX_INNER_LOOPS) {
          return MAX_INNER_LOOPS;
        }
      }
    }
  }
  return max_inner_loops;
}
```

3.5 Archivos

Los archivos con la definición de rutinas y estructuras de datos de WHIRL se encuentran en el directorio osprey1.0/common/com. Destacamos los siguientes:

wn_core.h Estructuras de datos y rutinas básicas.

wn.h Rutinas para crear y manipular nodos de WHIRL.

opcode_core.h Nombres de los opcodes, operadores, tipos asociados con los opcodes, etc.

wn_map.h Implementa el mecanismo de mapeo.

wn_util.h Rutinas para el recorrido, inserción y copia de nodos.

wn_tree_util.h Rutinas para recorrer el árbol.

wn_pragmas.h Soporte para OPC_PRAGMA.

Capítulo 4

Tablas de símbolos

La tabla de símbolos de WHIRL se compone en realidad de un conjunto de distintas tablas. Algunas de ellas son globales (visibles desde cualquier unidad de programa) y otras son locales (específicas de una unidad de programa). Basándonos en este criterio podemos clasificarlas en tres grupos:

1. sólo globales: PU_TAB, TY_TAB, FLD_TAB, TYLIST_TAB, ARB_TAB, TCON_TAB, BLK_TAB, INITV_TAB, STR_TAB, TCON_STR_TAB.
2. sólo locales: LABEL_TAB, PREG_TAB.
3. globales y locales (existen instancias de estas tablas tanto a nivel global como local): ST_TAB, INITO_TAB, ST_ATTR_TAB.

La figura 4.1 muestra como se distribuyen estas tablas entre las distintas funciones de un programa. Por una parte tenemos un conjunto de tablas globales visibles desde cualquier punto y, por otra parte, vemos como cada una de las funciones del programa tiene su propio conjunto de tablas locales. Todas estas tablas son generadas por la etapa de front-end. Como comentaremos en la sección 4.4 el back-end va a añadir algunas estructuras de datos adicionales.

A continuación pasamos a describir brevemente cada una de estas tablas. Para un descripción detallada consultar la referencia [7].

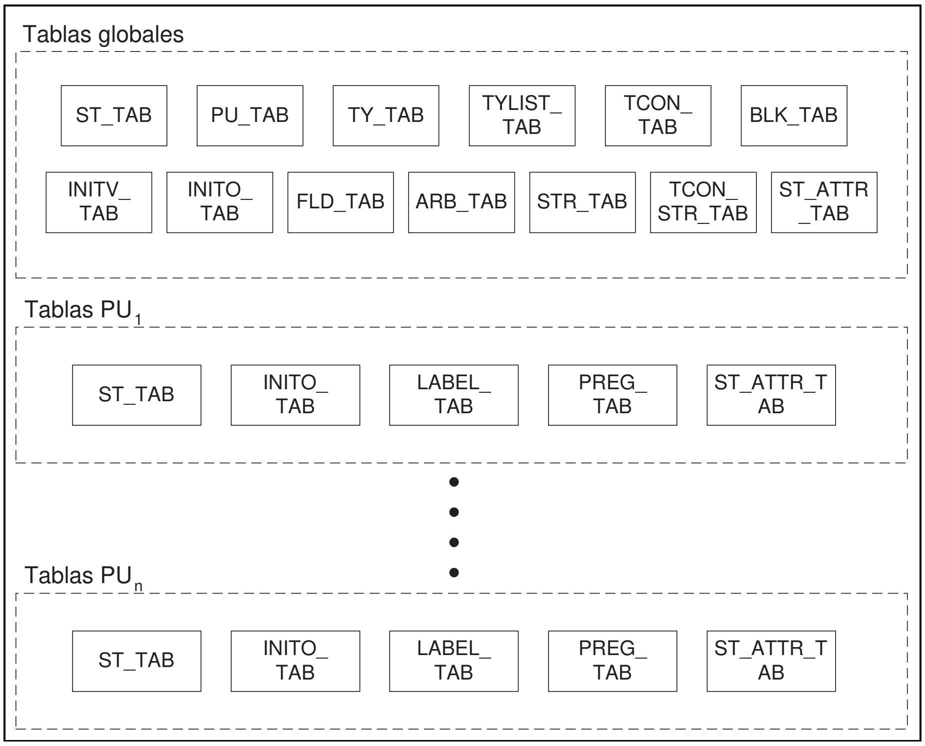
4.1 Tablas que pueden ser globales o locales

ST_TAB Cualquier símbolo que tenga un nombre ocupa una entrada en esta tabla. También habrá una entrada para cada valor que resida en memoria (constantes de tipo coma flotante y strings). Las entradas de esta tabla se denominan STs. Un ST_IDX identifica un símbolo del programa de forma unívoca. Un ST_IDX está compuesto por las siguientes partes:

nivel	8 bits de menor peso
índice de ST_TAB	24 bits de mayor peso

El campo "nivel" indica cual de las ST_TAB visibles desde la función en curso contiene el símbolo indicado por ST_IDX. Una función puede acceder a su propia ST_TAB, a las de las funciones que la contienen (si se trata de una función anidada) o a la ST_TAB global.

Figura 4.1: Tablas de símbolos generadas por los front-ends



INITO_TAB Cada entrada indica el valor inicial de un objeto de datos inicializado. Un campo indica de qué variable se trata mediante su índice dentro de la ST_TAB (ST_IDX) y otro campo apunta a una entrada de INTV_TAB, que es donde se encuentran los valores.

ST_ATTR_TAB Cada entrada asocia un determinado atributo con un elemento de la ST_TAB. Estos atributos son de uso poco frecuente y por este motivo se guardan fuera de la ST_TAB. Se utilizaría esta tabla, por ejemplo, para indicar que una variable tiene un registro físico dedicado, en cuyo caso almacenaríamos aquí el identificador de dicho registro. En la mayoría de casos esta tabla estará vacía.

4.2 Tablas sólo globales

PU_TAB Cada entrada representa un procedimiento (Program Unit) que aparece en el archivo fuente, ya sea como prototipo o como definición de función. El índice de esta tabla (PU_IDX) puede ser utilizado como identificador de PU.

TY_TAB Las entradas de esta tabla se denominan TYs y representan un tipo de dato del programa. Hacen referencia a las tablas FLD_TAB, TYLIST_TAB, ARB_TAB, o PU_TAB para especificar el tipo de forma completa. Un TY_IDX identifica de forma unívoca cualquier tipo de dato de alto nivel que aparezca en el programa. Los 24 bits de más peso indexan esta tabla, los 8 restantes aportan información adicional (por ejemplo alineación).

FLD_TAB Cada entrada especifica un campo dentro de una struct o union. La entrada correspondiente en TY_TAB apunta al primer campo, el resto siguen de forma consecutiva hasta encontrar un flag indicando el último.

TYLIST_TAB Cada entrada especifica el tipo de un parámetro en la declaración del prototipo de una función. La entrada de la función en TY_TAB apunta al elemento de TYLIST_TAB que corresponde al tipo devuelto, siguen el resto de parámetros. Cada entrada contiene un TY_IDX.

ARB_TAB Cada entrada contiene información sobre una de las dimensiones de un array. Las distintas dimensiones de un array se guardan en entradas consecutivas y se marca la última. Para arrays de C/C++ la primera dimensión corresponde a la dimensión más a la izquierda. Para arrays de Fortran la primera corresponde a la dimensión más a la derecha.

TCON_TAB Esta tabla almacena los valores de las constantes que no son de tipo entero. Para el caso de las constantes tipo string incluye una referencia a TCON_STR_TAB.

BLK_TAB Cada entrada contiene información sobre la distribución de un bloque de datos, que corresponde a un pedazo de memoria contigua del programa. Las variables se disponen respecto a los bloques de datos. Es creada por el back end.

INITV_TAB Cada entrada describe el valor inicial de un componente escalar de un objeto de datos inicializado. Los valores iniciales de tipos de datos complejos se describen mediante un árbol de entradas, cuya raíz viene determinada por el INITV_IDX de una de las entradas en INITO_TAB.

STR_TAB Las cadenas de caracteres de nombres de símbolos, tipos, etiquetas, etc. se guardan aquí.

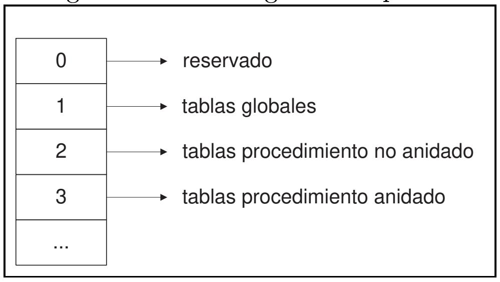
TCON_STR_TAB Todos los literales tipo string definidos en el programa se almacenan en esta tabla.

4.3 Tablas sólo locales

LABEL_TAB Aquí se guarda información asociada con cada etiqueta de WHIRL utilizada en la PU (índice a la STR_TAB para el nombre, flags y tipo).

PREG_TAB Aquí se guarda información asociada con cada pseudo-registro utilizado en la PU. Puesto que los pseudo-registros 0–71 están reservados para pseudo-registros dedicados hardware, los generados por el compilador empiezan en el 72. Así pues el índice de esta tabla es el número de pseudo-registro menos 71 (la entrada 0 está reservada).

Figura 4.2: Vector global Scope_tab



4.4 Otras estructuras de datos

Las estructuras de datos que aparecen en esta sección realmente no forman parte de la tabla de símbolos de WHIRL, pero están relacionadas:

FILE_INFO No es una tabla, se trata de una clase con información adicional que no encajaba en ninguna de las tablas anteriores. Contiene flags y atributos que hacen referencia al archivo que se está tratando.

BE_ST_TAB Esta tabla es creada y utilizada sólamente por el back-end. Cada entrada de esta tabla corresponde a una entra en la ST_TAB y se accede con el mismo índice. Incluye información adicional sobre los símbolos. Las entradas contienen un campo para almacenar flags, y un campo para guardar un apuntador a una estructura de datos utilizada por las rutinas de entrada/salida de Fortran.

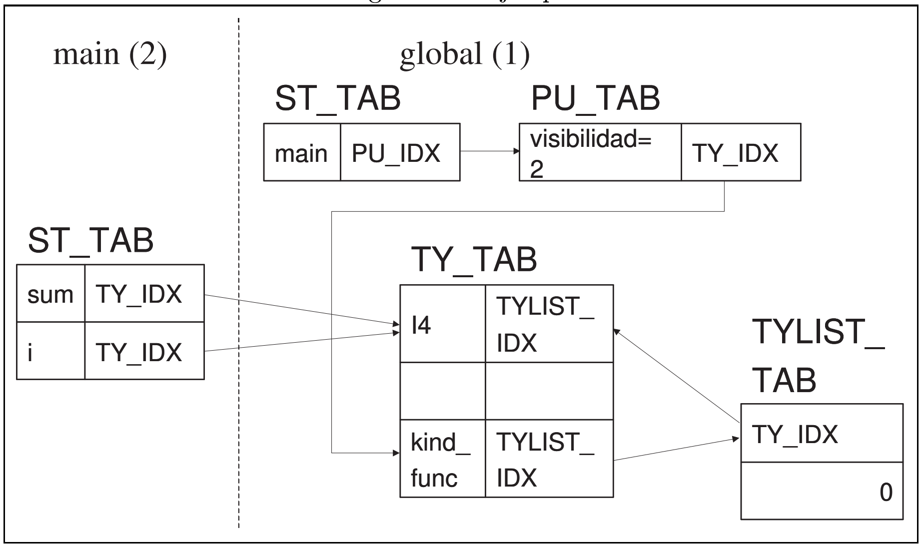
4.5 Visibilidad (SCOPE)

El conjunto de tablas visibles en un determinado momento depende del contexto. Un procedimiento anidado, por ejemplo, puede tener acceso a tres tablas ST_TAB (o más, dependiendo del nivel de anidación): su ST_TAB local, la de su procedimiento padre y la global. Existe un array global (Scope_tab) cuyos elementos especifican las tablas visibles desde cada PU. El elemento 0 de este array está reservado. El elemento 1 se refiere a las tablas de símbolos globales, el 2 a las locales, el 3 a las locales de un procedimiento anidado y seguiríamos en función del nivel de anidamiento.

Cada elemento de Scope_tab contiene un apuntador al área de memoria donde se encuentran las tablas locales (el compilador utiliza su propio mecanismo de gestión de la memoria dinámica) , un apuntador a la ST de la PU (la entrada que corresponde a la PU dentro de la ST_TAB) y apuntadores a las tablas de símbolos locales (no necesitamos ningún apuntador para las que sólo pueden ser globales).

Los índices predefinidos GLOBAL_SYMTAB y CURRENT_SYMTAB permiten acceder a la entrada correspondiente a las tablas de símbolos globales y a la entrada correspondiente a las tablas locales respectivamente. A continuación mostramos un pequeño código de ejemplo que muestra el uso de estos índices:

Figura 4.3: Ejemplo




```
// in a nested PU, check if an ST_IDX refers to a local
// variable in the parent PU
ST_IDX st_idx = WN_st_idx (wn);
if (ST_IDX_level (st_idx) != GLOBAL_SYMTAB &&
    ST_IDX_level (st_idx) < CURRENT_SYMTAB) {
    ...
}
```

4.6 Ejemplo

La figura 4.3 muestra un ejemplo con algunas tablas de símbolos y sus relaciones. En la parte derecha aparecen algunas tablas globales, cuyo índice en el vector Scope_tab es 1, y en la parte izquierda aparece una ST_TAB, que corresponde a la función principal (*main*) del siguiente programa en C:

```
int main(void)
{
  int i=0, sum=0;

  while (i<5) {
    sum = sum + i;
    i++;
  }

return sum;
}
```

En la ST_TAB global aparece una entrada para la función *main* acompañada de un índice que apunta a la PU_TAB, donde hay una entrada para cada función.

En la PU_TAB vemos que el ámbito de visibilidad de *main* (la posición de sus tablas locales en el vector Scope_tab) es 2, y encontramos también un apuntador a la tabla de tipos. El tipo de la función se define mediante la ayuda de otra tabla, la TYLIST_TAB. Dentro de la TYLIST_TAB se especifica el tipo de la función mediante una lista. El primer elemento es el tipo del resultado, en este caso un entero I4, a continuación siguen los parámetros, en el ejemplo no hay ninguno, y finalmente se marca el final de la lista con un 0.

Respecto a la ST_TAB local de *main*, ésta contiene dos entradas correspondientes a las variables *sum* e *i*. Ambas variables son de tipo entero de 4 bytes y contienen un apuntador a la entrada correspondiente en la tabla de tipos.

4.7 Acceso a las tablas de símbolos

La siguiente tabla muestra como acceder a los elementos de las tablas de símbolos.

Clase	Tipo índice	Tabla	Acceso a las entradas	Coste acceso
ST	ST_IDX	St_Table	ST_IDX idx; ST& st = St_Table[idx];	high
PU	PU_IDX	Pu_Table	PU_IDX idx; PU& pu = Pu_Table[idx];	low
TY	TY_IDX	Ty_Table	TY_IDX idx; TY& ty = Ty_Table[idx];	low
FLD	FLD_IDX	Fld_Table	FLD_IDX idx; FLD& fld = Fld_Table[idx];	low
ARB	ARB_IDX	Arb_Table	ARB_IDX idx; ARB& arb = Arb_Table[idx];	low
TYLIST	TYLIST_IDX	Tylist_Table	TYLIST_IDX idx; TYLIST& tylist = Tylist_Table[idx];	low
TCON	TCON_IDX	Tcon_Table	TCON_IDX idx; TCON& tcon = Tcon_Table[idx];	low
INITO	INITO_IDX	Inito_Table	INITO_IDX idx; INITO& inito = Inito_Table[idx];	high
INITV	INITV_IDX	Initv_Table	INITV_IDX idx; INITV& initv = Initv_Table[idx];	low
LABEL	LABEL_IDX	Label_Table	LABEL_IDX idx; LABEL& label = Label_Table[idx];	medium
PREG	PREG_IDX	Preg_Table	PREG_IDX idx; PREG& preg = Preg_Table[idx];	medium
	STR_IDX	Str_Table	STR_IDX idx; char *str = &Str_Table[idx];	very low

Para acceder a los elementos se debe utilizar un índice del tipo adecuado (XX_IDX), especialmente cuando este va ser pasado como parámetro. También se puede utilizar un apuntador para un acceso más rápido, como en el siguiente ejemplo:

```
void foo (TY_IDX ty_idx)
{
    const TY& ty = Ty_Table[ty_idx];

    if (TY_kind(ty) == KIND_STRUCT &&
        TY_is_union (ty) &&
```

```
        TY_no_ansi_alias(ty)) {
        ...
    }
}
```

Sólo en el caso de la ST, debido al elevado coste de acceso mediante un índice, se utilizan punteros como método estándar.

4.7.1 Acceso a los campos de datos

Para cada tipo de elemento se definen funciones para acceder a sus campos. En general:

XX_yyy(const XX& xx) devuelve el campo yyy de la clase XX, especificada por xx.

Set_XX_yyy (XX& xx, type_of_yyy val) asigna el valor de yyy a val.

Para consultar y manipular flags:

XX_flag1 (const XX& xx) devuelve TRUE si XX_FLAG1 está activado.

Set_XX_flag1 (XX& xx) pone XX_FLAG1 a TRUE.

Clear_XX_flag1 (XX& xx) pone XX_FLAG1 a FALSE.

4.7.2 Creación e inicialización de un elemento

Para cada clase XX, se definen las funciones New_XX() y XX_Init(). Por ejemplo:

```
TY_IDX ty_idx;

// New_TY returns an uninitialized TY entry,
// and set the index in the reference parameter ty_idx
TY& ty = New_TY (ty_idx);

// The init routine takes values of most commonly used fields, and
// zero out the remaining fields.
// Always initialize the entry right after calling New.
TY_Init (ty, 4 KIND_SCALAR, MTYPE_I4, Save_Str ("foo"));

// Other attributes must be set after calling TY_Init
Set_TY_align (ty_idx, 4);

// Should always return TY_IDX instead of TY&
return ty_idx;
```

4.7.3 Iteradores

Para cada clase XX, se define un iterador unidireccional. La función Make_xx_iter permite crear un iterador que apunte a un elemento determinado. Por ejemplo:

```
FLD_IDX fld_idx = TY_fld (ty);

// Create an iterator pointing to the entry given by fld_idx
FLD_ITER fld_iter = Make_fld_iter (fld_idx);
do {
    // operator* dereference the iterator
    FLD& fld = *fld_iter;
    ...
} while (!FLD_last_field (*fld_iter++));
```

El número total de entradas en la tabla se puede obtener con la función XX_Table_Size.

4.7.4 Recorrido rápido de todas las entradas

El iterador anterior es adecuado para recorrer un número de elementos limitado. Si vamos a recorrer toda la tabla podemos utilizar una solución más eficiente como la que presentamos en el siguiente ejemplo:

```
// Function object that defines the operation
struct clear_no_se {
    // First argument is the array index of the current entry,
    // ignored in this case
    // Second argument is the pointer to the current entry
    void operator() (UINT32, const PU* pu) const {
        Clear_PU_no_side_effects (*pu);
    }
};

void Clear_all_pu_side_effect ()
{
    // Pass in an anonymous function object
    For_all (Pu_Table, clear_no_se());
}
```

Podríamos conseguir el mismo efecto pasando un apuntador a función en lugar de un objeto función, pero sería una solución menos eficiente.

Si vamos a abandonar el recorrido antes de llegar al final podemos utilizar la función For_all_until(). En este caso, el objeto función debe devolver TRUE cuando se satisface la condición de finalización.

4.8 Archivos

Los siguientes archivos se encuentran en osprey1.0/common/com.

symtab.h Es el único archivo que se debe incluir (#include). También contiene funciones para acceder a distintos miembros de las tablas de símbolos.

symtab_defs.h Definiciones y declaración de tipos y variables. Se declaran las siguientes variables globales (se encuentran en symtab.cxx):

```
// declaration of global tables
extern FILE_INFO        File_info;
extern PU_TAB           Pu_Table;
extern SYMBOL_TABLE     St_Table;
extern TY_TAB           Ty_tab;
extern TYPE_TABLE       Ty_Table;
extern FLD_TAB          Fld_Table;
extern TYLIST_TAB       Tylist_Table;
extern ARB_TAB          Arb_Table;
extern STRING_TABLE     Str_Table;
extern TCON_TAB         Tcon_Table;
extern INITV_TAB        Initv_Table;
extern INITO_TABLE      Inito_Table;
extern PREG_TABLE       Preg_Table;
extern ST_ATTR_TABLE    St_Attr_Table;
extern LABEL_TABLE      Label_Table;
// some BLK_TAB items are really local,
// but make global to ease management
extern BLK_TAB          Blk_Table;

// global variables

extern SYMTAB_IDX Current_scope;
extern PU* Current_pu;
```

symtab_idx.h Definición de tipo de los distintos índices y funciones para su manipulación.

symtab_access.h Funciones para leer/escribir directamente sobre un campo de cada una de las estructuras de las tablas de símbolos.

symtab_utils.h Funciones adicionales para manipular las tablas de símbolos.

symtab_verify.h Verifica que el contenido de las tablas de símbolos sea consistente.

symtab_compatible.h Funciones para mantener la compatibilidad con un formato anterior de las tablas de símbolos.

irbdata_defs.h Definición de las clases INITO e INITV.

irbdata.h Funciones de acceso y funciones adicionales para manipular INITO e INITV.

strtab.h Interfaz externa de la STR_TAB.

Capítulo 5

Distribución fuentes

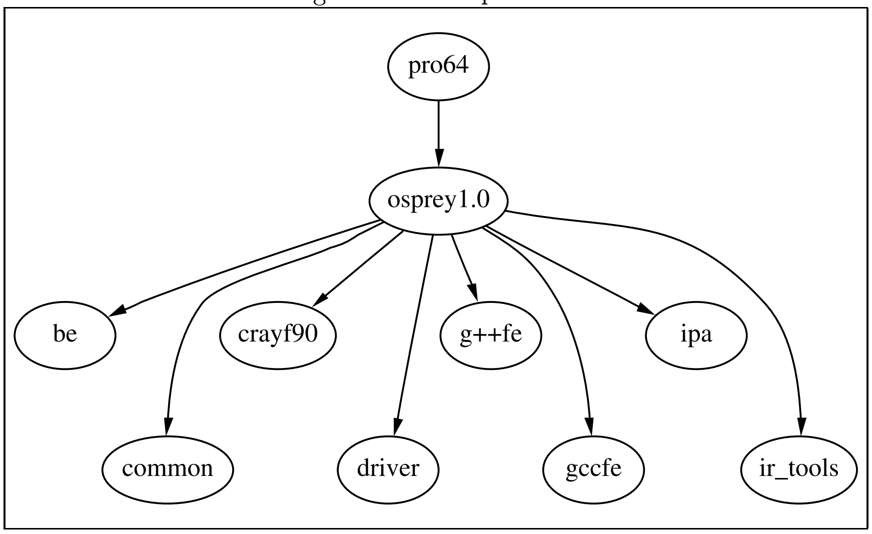
Después de descargar y descomprimir la distribución del compilador, nuestro sistema contendrá una cantidad considerable de directorios y subdirectorios. El objetivo de este capítulo es dar una visión general de lo que podemos encontrar en ellos. Veremos primero qué directorios principales se han creado y, a continuación, cuales son concretamente los componentes que conforman el compilador y las librerías nativas de IA-64. Finalmente apuntaremos algunos detalles sobre la organización de los Makefiles.

Este capítulo se basa en la estructura de directorios de la distribución 0.13 del compilador Pro64.

5.1 Directorios

Componentes Directorios con el código fuente de los principales componentes del compilador.

Figura 5.1: Componentes



be Componentes del back-end excepto análisis interprocedural.

common Archivos de uso común necesarios para generar distintos componentes. Contiene, por ejemplo, las definiciones de las estructuras de datos para almacenar la representación intermedia del programa y funciones para su manipulación.

crayf90 Front-end para el lenguaje F90.

driver Driver del compilador.

g++fe Front-end para el lenguaje C++.

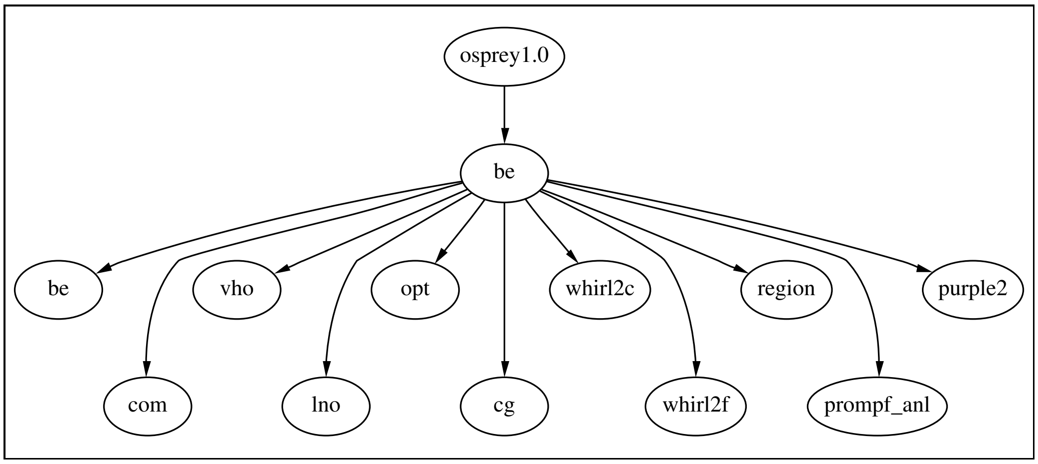
gccfe Front-end para el lenguaje C.

ipa Análisis inter-procedural.

ir_tools Herramientas para convertir los archivos binarios de representación intermedia en archivos ASCII.

Back-end Componentes que forman parte del back-end, excepto análisis interprocedural.

Figura 5.2: Back-end



be Driver de la etapa de back-end. Genera un ejecutable que es llamado una vez finaliza el front-end.

com Archivos comunes que van a ser utilizados por los distintos componentes.

vho Traduce VH WHIRL a H WHIRL y realiza algunas optimizaciones de muy alto nivel.

lno Optimizador y paralelizador de bucles.

opt Optimizador global.

cg Generador de código.

whirl2c Conversión de WHIRL a lenguaje c.

whirl2f Conversión de WHIRL a lenguaje Fortran.

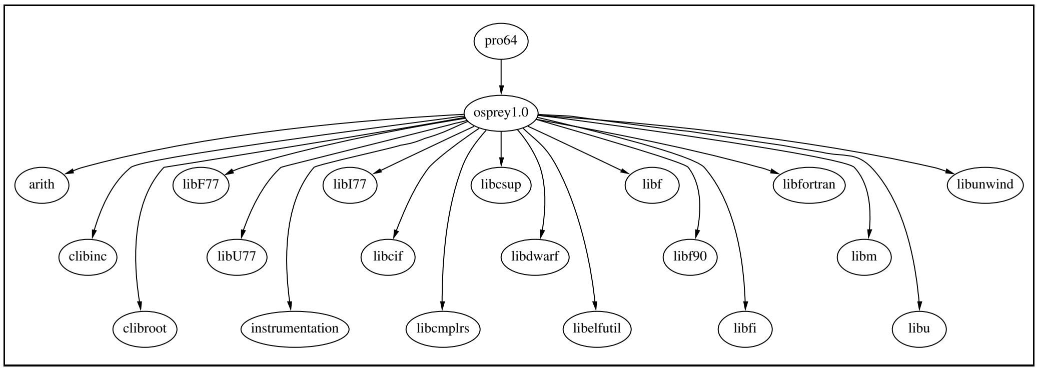
region Código relacionado con la manipulación de regiones.

prompf_anl Código para generar información destinada a Pro MPF. Pro MPF es una herramienta de SGI que no es de código abierto y no se distribuye con el compilador.

purple2 Directorio vacío.

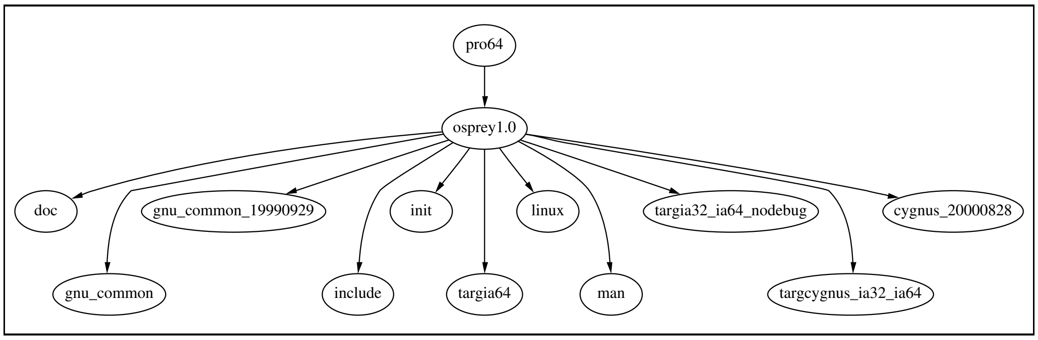
Librerías Directorios con librerías, ya sean necesarias para la compilación y ejecución del propio compilador, o bién utilizadas por éste durante la fase de enlace para generar nuevos binarios.

Figura 5.3: Librerías



Otros Incluimos en este apartado el resto de directorios.

Figura 5.4: Otros



doc Contiene un archivo de texto con información acerca de como depurar el compilador.

gnu_common Incluye fuentes necesarias para compilar los front-end de C y C++, y también para generar distintas librerías.

gnu_common_19990929 Contiene versiones distintas de algunos headers presentes en gnu_common.

include Cabeceras.

init Aparentemente, código para generar un archivo objeto con las funciones de inicialización para programas fortran.

targia64 Makefiles para construir de forma individualizada los componentes nativos de IA-64.

linux Cabeceras, definición de reglas para el gmake, herramientas, etc. para compilar sobre Linux. Debemos tener en cuenta que el compilador proviene del entorno de máquinas SGI.

man Páginas man.

targia32_ia64_nodebug Makefiles para generar un cross-compiler. Se ejecuta sobre IA-32, pero genera código para IA-64.

targcygnus_ia32_ia64 "(...) building the Cygnus tools for ia32→ia64".

cygnus_20000828 Aparentemente, código con partes del compilador de Cygnus, por ejemplo, el enlazador (ld).

5.2 Componentes y librerías nativas

Los directorios nombrados en la sección anterior contienen el código necesario para poder generar cada uno de los componentes básicos del compilador. El archivo README.src contiene un listado con todos estos componentes acompañados con una breve descripción, la ubicación de los binarios que se van a generar y como generarlos. Por este motivo, aquí nos limitaremos simplemente a nombrarlos.

- Compiler Drivers
- C Front-End
- C++ Front-End
- Fortran 90 Front-End
- Backend
- Code Generator
- Global Optimizer
- Loop Nest Optimizer
- Stand-Alone Inliner
- Interprocedural analysis optimizer
- Interprocedural analysis optimizer (local summary phase)
- Itanium(TM) processor target description module
- WHIRL to C converter
- WHIRL to Fortran converter

- WHIRL to ASCII converter

Las librerías nativas de IA-64 son la siguientes:

- Fortran90 runtime library
- Fast Fortran I/O runtime library
- SGI math library (vector version)
- libgcc.a
- lib.cat
- lib.exp

Se proporciona el binario de lib.cat y lib.exp.

5.3 Makefiles

En esta sección comentaremos brevemente como están organizados los Makefiles.

Al mismo nivel que el directorio osprey1.0 se encuentran los siguientes makefiles: Make.cross y Make.NUE.lib. El primero de ellos lanza la compilación de todos aquellos componentes que, en un entorno compuesto por una máquina de 32 bits con NUE, deben ser compilados generando objetos de 32 bits, dando lugar a un cross-compiler. Make.NUE.lib lanza la compilación de las librerías nativas de IA-64. Después de ejecutar estos Makefiles el script INSTALL completará la instalación.

Observando el contenido de los Makefiles anteriores veremos que, para cada componente que debe ser creado, se ejecuta el comando cd para ir al directorio de build de dicho componente seguido del comando gmake. Los directorios de build són los siguientes:

targcygnus_ia32_ia64 Corresponde a las herramientas de Cygnus necesarias para el compilador. En él encontraremos un README con más información.

targia32_ia64_nodebug Contiene subdirectorios, cada uno con su correspondiente Makefile, para compilar los componentes que van a ser ejecutados en 32 bits, pero que generarán código para IA-64.

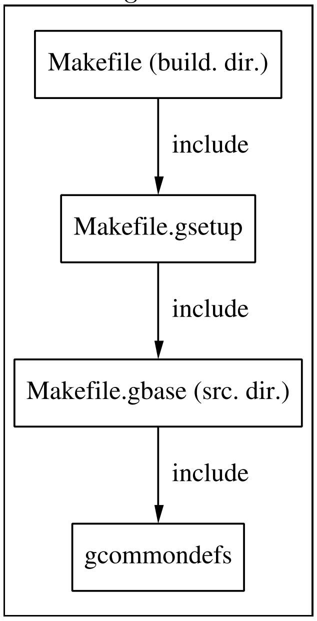
targia64 Contiene los subdirectorios y Makefiles para generar las librerías nativas de IA-64.

Los Makefiles que encontraremos en los directorios de build tienen la siguiente estructura: en primer lugar definen una serie de parámetros para llevar a cabo la compilación, y a continuación se realiza un include del archivo Makefile.gsetup.

Makefile.gsetup verifica los parámetros y, a su vez, realiza un include de otro Makefile: Makefile.gbase. Cada componente tiene un archivo denominado Makefile.gbase junto a sus fuentes. Makefile.gsetup conoce su ubicación a través de uno de los parámetros.

En Makefile.gbase encontraremos en primer lugar un include de otro Makefile que contiene reglas y definiciones que se van a utilizar en la compilación de todos los componentes. Finalmente, contiene las dependencias que definen cómo se debe compilar el componente en cuestión.

Figura 5.5: Organización Makefiles



Capítulo 6

Conclusiones y trabajo futuro

6.1 Conclusiones

El objetivo de este documento es facilitar la toma de contacto con los compiladores *Open64* y *ORC*. Los compiladores *Open64* y *ORC* son fruto de sendos proyectos de código abierto, ambos basados en el código fuente del compilador *Pro64* de SGI.

En primer lugar, se ha procedido a una descripción de los componentes que conforman los compiladores. Se ha visto que cada uno de ellos corresponde a una etapa de la compilación y se han enumerado algunas de las optimizaciones realizadas durante las distintas etapas.

A continuación se ha descrito la representación intermedia utilizada por el compilador. Esta representación intermedia se denomina WHIRL y se utiliza como interfaz común entre los distintos componentes. Un programa se representa en WHIRL mediante un conjunto de árboles sintácticos y tablas de símbolos.

Para terminar, hemos visto como están organizadas las fuentes del compilador en la estructura directorios.

6.2 Trabajo futuro

Actualmente existe poca documentación sobre el compilador y la mayor parte está disponible sólo en forma de presentación multimedia (tipo PowerPoint), de modo que resulta difícil conocer los detalles de implementación.

En el futuro, nos proponemos profundizar en los distintos componentes del compilador, empezando por el optimizador de bucles, y generar documentación que permita conocer de forma detallada qué optimizaciones se llevan a cabo en cada etapa y como están implementadas.

Bibliografía

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

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

[3] http://oss.sgi.com/projects/pro64/. Web oficial Pro64.

[4] http://oss.sgi.com/projects/pro64/mail/pro64-support/threads.html. Archivo mensajes lista de soporte de Pro64.

[5] http://www.software.hp.com/ia64linux. Página de descarga del Native User Environment.

[6] *WHIRL Intermediate Language Specification.*

[7] *WHIRL Symbol Table Specification.*

[8] Jim Dehnert. *Pro64: Performance Compilers For IA-64*, June 2000.

[9] Guang R. Gao, J.N. Amaral, J. Dehnert, and R. Towle. *Tutorial on the SGI Pro64 Compiler Infrastructure.* Disponible en http://www.cs.ualberta.ca/~amaral/Pro64/index.html.

[10] Roy Ju, Sun Chan, Fred Chow, Xiaobing Feng, and William Chen. *Open Research Compiler (ORC): Beyond Version 1.0*, September 2002. Tutorial PACT-2002.

[11] Roy Ju, Sun Chan, Chengyong Wu, Ruiqi Lian, and Tony Tuo. *Open Research Compiler (ORC) for Itanium Processor Family*, December 2001. Tutorial MICRO34.