Diseño del compilador de la máquina virtual inmortal iVM

Eladio Gutiérrez[1], Sergio Romero[2], Oscar Plata[3]

Resumen— **Conservar la información digital durante mucho tiempo es difícil, incluso cuando se utiliza un medio de almacenamiento pasivo duradero, como una película fotográfica almacenada en las condiciones adecuadas. En dicho soporte, se pueden combinar descripciones analógicas, legibles por humanos, con información digital codificada. Sin embargo, después de cientos de años, es posible que los formatos utilizados para representar y codificar esta información se hayan olvidado, y es posible que cualquier código fuente sobreviviente no se pueda compilar y ejecutar simplemente. Explicar cómo interpretar los datos almacenados en un formato complejo corre el riesgo de cometer errores hoy y futuros malentendidos.**

El proyecto de máquina virtual inmortal (Immortal Virtual Machine, iVM) introduce una máquina abstracta con una descripción formal muy sencilla. Si junto con la información digital, se preservan la propia descripción de la máquina, y los binarios necesarios para decodificar la información codificados para la máquina abstracta, las generaciones futuras podrán decodificar y presentar la información simplemente implementando esta máquina.

Este artículo pone el foco en el desarrollo del compilador de C para la máquina virtual, que ha sido desarrollado sobre la infraestructura de compilación GCC.

Palabras clave— **Preservación de información digital, Immortal Virtual Machine (iVM), Compilador GCC**

I. El proyecto

El proyecto Immortal Virtual Machine (iVM) [1] tiene como objetivo crear una solución de preservación de información que garantice el acceso a los datos digitales actuales en el futuro (durante cientos de años). Con este objetivo, se propone una máquina virtual abstracta para reconstruir los datos preservados. Los datos se almacenan en un medio de preservación utilizando la tecnología desarrollada por la empresa Piql AS [2], donde la información se codifica digitalmente en fotogramas en una cinta de película fotográfica de alta resolución [3]. En la figura 1 se muestra este flujo de preservación de la información.

La solución de máquina virtual inmortal iVM, se basa en tres elementos:

1. una máquina abstracta extremadamente simple de describir formalmente (que denominaremos iVM),
2. descripciones independientes y tecnológicamente neutrales de la máquina, preservadas en forma analógica y dirigidas a futuros desarrolladores,
3. un compilador (de C) dirigido a esta máquina.

[1]Dpto. de Arquitectura de Computadores, Universidad de Málaga, e-mail: eladio@uma.es

[2]Dpto. de Arquitectura de Computadores, Universidad de Málaga, e-mail: sromero@uma.es

[3]Dpto. de Arquitectura de Computadores, Universidad de Málaga, e-mail: oplata@uma.es

La función de la máquina abstracta es implementar los decodificadores de formato que se almacenan en binario junto con los datos que se conservarán. Esto da como resultado contenidos autoejecutables que pueden producir resultados consistentes (imágenes, sonidos, ...) en los dispositivos de salida. De esta manera se propone un formato de almacenamiento de la información en la cinta de película como el mostrado en la figura 2. Dos requisitos importantes de esta máquina abstracta es que debe tener una fácil descripción formal y debe ser fácil de implementar por los futuros desarrolladores.

Uno de los puntos clave de este enfoque es la disponibilidad de un compilador que se encarga de generar el código iVM a partir de los códigos fuente de los decodificadores, asumiendo que estos decodificadores están escritos en un lenguaje muy consolidado como C. En la figura 3 se destaca el papel del compilador en la generación del contenido preservado.

De las muchas infraestructuras de compiladores de C disponibles, ha sido necesario elegir la más adecuada, y seleccionar una de código abierto ha sido un factor decisivo importante. Entre las opciones disponibles, varias razones nos llevaron a adoptar GCC (GNU Compiler Collection) [4], como son su popularidad entre la comunidad y un API de desarrollo bastante estable.

Este artículo describe los aspectos más relevantes del diseño del compilador y su uso. La misión del compilador es proporcionar una representación ensamblador de los programas C, que eventualmente un ensamblador convertirá a binario. El lenguaje ensamblador sirve como interfaz entre el compilador y el ensamblador. En realidad, el compilador desarrollado es un compilador cruzado de C, para ser compilado y ejecutado en las computadoras actuales.

En lo sucesivo, el término IVM64 se refiere a la arquitectura de la máquina virtual abstracta a la que se dirige el compilador.

II. La máquina virtual

La máquina virtual inmortal es un procesador abstracto de 64 bits (IVM64) diseñado con el objetivo de ser fácilmente descrito de manera formal, para que los futuros desarrolladores puedan recrearla y ejecutar el software necesario para extraer la información preservada en el carrete de film junto con el propio software [5].

IVM64 es una máquina de pila de 64 bits que está formada únicamente estos elementos:

- una memoria direccionable por bytes de tamaño arbitrario

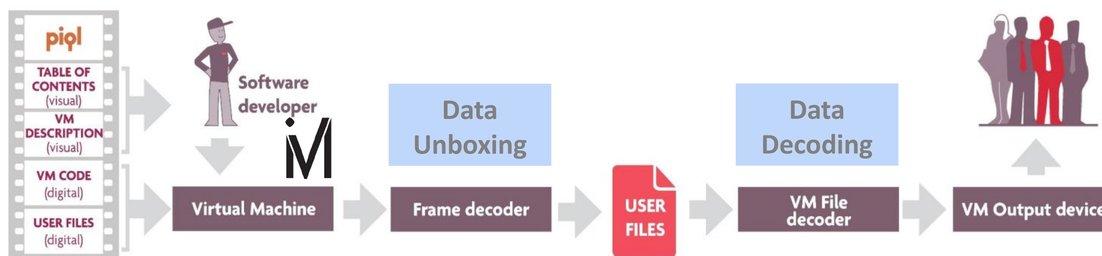

Fig. 1: Flujo de codificación y decodificación de la información en la película fotográfica.

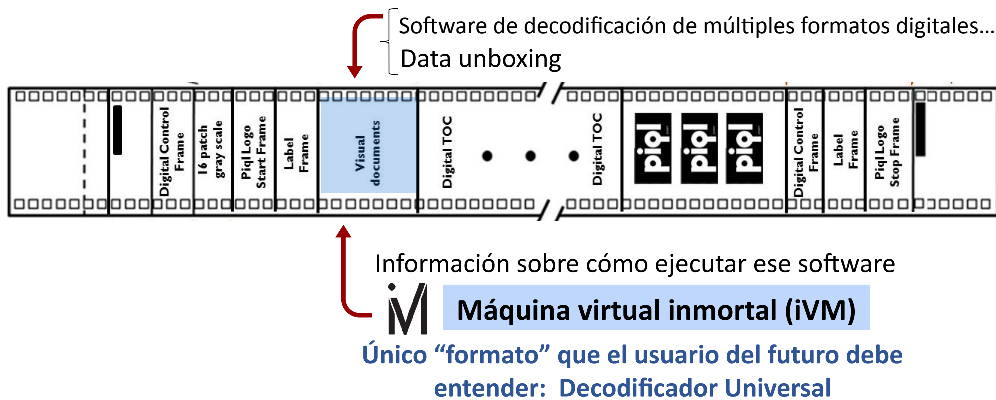

Fig. 2: Formato de la información en la película fotográfica.

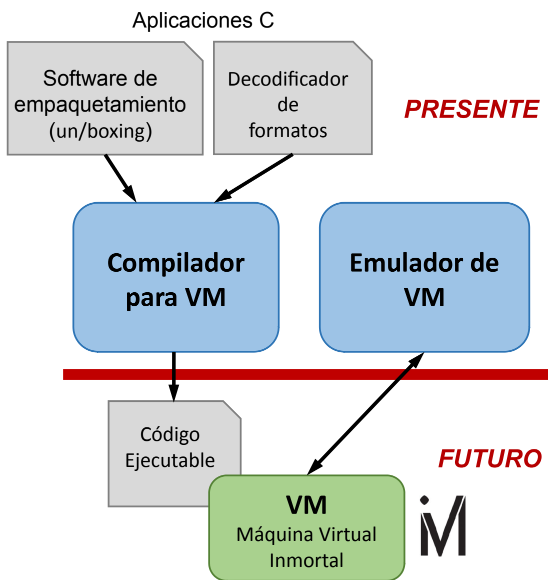

Fig. 3: Papel del compilador en la generación del contenido preservado.

- dos registros de propósito específico: el contador de programa (PC) y el puntero de pila (SP)
- un bit de detención (*halt bit*)

Inmediatamente después de encender la máquina, el registro PC apunta a la primera posición de memoria (dirección 0), SP apunta a la última y el bit de detención se restablece a 0. En esta máquina: la pila crece hacia direcciones decrecientes, los registros tienen un ancho de 64 bits, la memoria es direccionable por bytes y los datos se almacenan siguiendo un esquema *little-endian*. La ejecución se detiene con la instrucción `exit` que activa el bit de detención.

Desde el punto de vista del desarrollo de un compilador de C, es necesario saber qué debe producir el compilador. Como decisión de diseño se adoptó que el compilador de C para IVM64 se encargara de generar código ensamblador para la máquina IVM64, de tal forma que este código ensamblador deberá ser traducido posteriormente a binarios IVM64 en términos de las instrucciones nativas de la máquina abstracta.

El compilador de C implementado es en realidad un compilador cruzado, que se compila y se ejecuta en un equipo actual (por ejemplo, un PC basado en x86-64) para producir un código ensamblador IVM64.

Aparte del compilador, se requiere, una herramienta de ensamblado externa para ensamblar la salida del compilador, es decir, para traducir la salida al binario nativo IVM64. En el contexto de este proyecto, esta herramienta se denominó *la implementación* de la máquina virtual.

Además, un desarrollador actual también necesitará otra herramienta para simular el binario después de ensamblarlo, con fines de prueba y depuración.

Hay que mencionar que para facilitar el diseño se diferenció lo que son las instrucciones máquina de lo que es la sintaxis ensamblador. El conjunto de instrucciones nativo IVM64 tiene una correspondencia directa con el código binario. Por otro lado, la sintaxis ensamblador que se usa en los códigos es una sintaxis más rica que facilita la generación de código[1], o la escritura de código ensamblador a mano.

El compilador de C nunca genera código binario, sino ensamblador. Se puede encontrar una descrip-

[1] Algunos de los mnemónicos del ensamblador tienen una correspondencia directa con las instrucciones nativas (p. ej., `add`), pero otros son en realidad pseudoinstrucciones (p. ej., `div_u`). Además, algunas instrucciones nativas no tienen mnemónicos específicos, por ejemplo, `get_sp` debe expresarse como `push! &0`. Otros mnemotécnicos son meros alias, como `return` que equivale a `jump`.

ción más completa tanto de la sintaxis del ensamblador como del conjunto de instrucciones IVM64 en [6], [7]. La tabla I(ver [7]), resume la arquitectura del conjunto de instrucciones (ISA) de IVM64.

Nos referiremos al compilador cruzado de C para IVM64 como `ivm64-gcc` [8].

III. El ensamblador

Los códigos ensamblador para IVM64, como los generados por `ivm64-gcc`, no se expresan directamente en función de instrucciones nativas. En su lugar, se usa una sintaxis más rica introducida por [9], que incluye: directivas, mnemónicos, etiquetas, abreviaturas, operadores y expresiones. Podemos ver una descripción semiformal EBNF de esta sintaxis en la tabla II.

IV. El compilador

El papel del compilador es proporcionar una representación en ensamblador de los programas C, que eventualmente el ensamblador convertirá a binario. y será almacenado en el medio digital para su conservación. El lenguaje ensamblador sirve como interfaz entre el compilador y el ensamblador.

A. Infraestructura de compilación

Uno de los puntos clave en el diseño del compilador de C ha sido la adopción de una infraestructura de compilador de código abierto como sistema base. Se eligió GCC (GNU Compiler Collection) por varias razones: es un compilador ampliamente utilizado con una gran comunidad de soporte, posee un API de diseño muy estable entre diferentes versiones, y es un compilador completo de C.

El núcleo del compilador GCC C (programa `cc1`) está a cargo de traducir el lenguaje C al lenguaje ensamblador de la arquitectura destino. La figura 4 esboza la estructura de `cc1`. Se utilizan dos representaciones intermedias en diferentes etapas de la compilación: la representación GIMPLE independiente de la arquitectura para los pasos del árbol sintáctico y la denominada representación RTL (*Register Transfer Level*) para los pasos que dependen de la arquitectura destino. La traducción del formato GIMPLE a RTL se conoce como *expansión*. La expansión da lugar a una secuencia de expresiones RTL (*RTL expressions* o RTX), que puede verse como una aproximación de lo que será el código ensamblador objetivo. La sintaxis RTX está basada en notación LISP.

La representación interna RTL hace uso del concepto de *registro*. La fase de expansión considera un número infinito de registros cuando emite el primer RTL, previo al resto de transformaciones y optimizaciones que siguen. Estos registros simbólicos se denominan *pseudoregistros*. Los pseudoregistros deben mapearse en un momento dado a registros reales (arquitecturales) de la arquitectura de destino. Esta operación se conoce como *asignación de registros*, que se lleva a cabo mediante el paso de *reload*. Durante el paso *final*, la secuencia RTX se convierte a sentencias ensamblador para IVM64.

Añadir una nueva arquitectura destino (*target)* al compilador implica proporcionar una *descripción de la máquina* que especifica las características de la arquitectura. En esencia, la descripción de la máquina GCC define patrones RTL válidos, cumpliendo ciertas restricciones, como los modos de direccionamiento, así como proporciona las instrucciones ensamblador que se generarán.

B. Esquema de uso de la pila (stackification)

Uno de los principales retos a la hora de diseñar el *backend* de GCC para la arquitectura de la máquina abstracta ha sido la falta de registros. No hay registros de propósito general ni *Frame Pointer* (FP). Este hecho requiere definir unos registros a efectos de compilación (realente no están presentes en la arquitectura) y mapear de alguna manera dichos registros en la pila (*stackification*) en la fase de *reload*. Para ello se ha seguido un enfoque similar al de [11]. Al principio, se asume que la arquitectura dispone de un conjunto de registros de propósito general. Estos registros se utilizan como registros arquitecturales durante la fase de asignación de registros, pero finalmente se asignan a posiciones de pila cuando en la fase final se genera el código ensamblador.

La figura 5 muestra estos registros definidos a efectos del compilador. El PC es para GCC un registro implícito y por eso se ha omitido de esta lista. De especial importancia es el registro FP, que el compilador utiliza hasta la fase de asignación de registros. En dicha fase es eliminado (ya que no es un registro real en la máquina IVM64). Básicamente, FP se expresa en función de SP, de acuerdo con ciertas reglas de eliminación establecidas por la descripción de la máquina.

El registro TR es un registro instrumental que representa la parte superior de la pila (*top-of-the-stack* (TOS)). Escribir en él implica realizar una acción *push* sobre una palabra en la pila. Leerlo implica una acción *pop*, sacando una palabra de la pila. Este no es un registro de propósito general, pero se usa para expresar cálculos temporales que requieren mover datos desde/hacia la pila y, en consecuencia, solo debe manejarse siguiendo ciertas reglas.

Finalmente, se definen 16 registros de propósito general[2], denominados AR, X1, X2, ... X15. El registro AR es el empleado por las funciones para devolver un valor de retorno 64 bits (o menor).

La figura 6 muestra el bloque de pila asignado inmediatamente después de la llamada a una función determinada, incluidos los registros mapeados en pila. Obsérvese que es necesario llevar cuenta de la posición de SP para ubicar la posición actual del primer registro de propósito general. En una función, SP puede cambiar de dos formas: (1) explícitamente, por ejemplo, cuando se inserta un argumento en la pila (SP se decrementa previamente), y (2) implícitamente, por ejemplo, cuando se opera en TR (en ese caso, la maquinaria GCC no tiene conocimiento sobre la modificación SP).

[2]Este número se decidió experimentalmente.

Hex	Mnemonic	Comment	Immediate	Pop	Explicit effects	Push
00	EXIT	Stop execution	–	–	$T := 1$	–
01	NOP	No operation	–	–	–	–
02	JUMP	Jump to address	–	a	$PC := a$	–
03	JZ_FWD	Jump forward on zero	$(1)d$	x	$PC := PC + \text{if}(x = 0, d, 0)$	–
04	JZ_BACK	Jump backward on zero	$(1)d$	x	$PC := PC - \text{if}(x = 0, d + 1, 0)$	–
05	SET_SP	Set stack pointer	–	a	$SP := a$	–
06	GET_PC	Get program counter	–	–	–	PC
07	GET_SP	Get stack pointer	–	–	–	SP
08	PUSH0	Push literal zero	–	–	–	0
09	PUSH1	Push 1 immediate octet	$(1)x$	–	–	x
0A	PUSH2	Push 2 immediate octets	$(2)x$	–	–	x
0B	PUSH4	Push 4 immediate octets	$(4)x$	–	–	x
0C	PUSH8	Push 8 immediate octets	$(8)x$	–	–	x
10	LOAD1	Load 1 memory octet	–	a	–	load(1, a)
11	LOAD2	Load 2 memory octets	–	a	–	load(2, a)
12	LOAD4	Load 4 memory octets	–	a	–	load(4, a)
13	LOAD8	Load 8 memory octets	–	a	–	load(8, a)
14	STORE1	Store 1 memory octet	–	a, x	store(1, a, x)	–
15	STORE2	Store 2 memory octets	–	a, x	store(2, a, x)	–
16	STORE4	Store 4 memory octets	–	a, x	store(4, a, x)	–
17	STORE8	Store 8 memory octets	–	a, x	store(8, a, x)	–
20	ADD	Add	–	y, x	–	add(x, y)
21	MULT	Multiply	–	y, x	–	mul(x, y)
22	DIV	Divide	–	y, x	–	div(x, y)
23	REM	Find remainder	–	y, x	–	rem(x, y)
24	LT	Less than	–	y, x	–	$\text{if}(x < y, -1, 0)$
28	AND	Bitwise "and"	–	y, x	–	and(x, y)
29	OR	Bitwise "or"	–	y, x	–	or(x, y)
2A	NOT	Bitwise "not"	–	x	–	not(x, y)
2B	XOR	Bitwise "exclusive or"	–	y, x	–	xor(x, y)
2C	POW2	Binary power	–	x	–	pow2(x)
F9	PUT_BYTE	Put byte	–	x	putbyte(x)	–
FA	PUT_CHAR	Put character	–	c	putchar(c)	–
FB	ADD_SAMPLE	Put audio sample	–	r, l	addsample(l, r)	–
FC	SET_PIXEL	Put pixel	–	b, g, r, y, x	setpixel(x, y, r, g, b)	–
FD	NEW_FRAME	Output frame	–	r, h, w	newframe(w, h, r)	–
FE	READ_PIXEL	Get pixel	–	x, y	$z :=$ readpixel(x, y)	z
FF	READ_FRAME	Input frame	–	–	$(c, r) :=$ readframe()	c, r

Tabla I: Repertorio de instrucciones IVM64 nativas (ver [7]).

```
⟨assembly⟩    ::= ⟨import⟩∗ ⟨statement⟩∗
⟨import⟩      ::= "IMPORT" ( ⟨id⟩ "/" )+ ⟨id⟩
⟨statement⟩   ::= ⟨id⟩ ":" | "EXPORT" ⟨id⟩ | ⟨id⟩ "=" ⟨expression⟩ | ⟨data⟩ | "space" ⟨expression⟩ | ⟨instruction⟩ "!"∗ ⟨expression⟩∗
⟨expression⟩  ::= ⟨positive numeral⟩ | ⟨id⟩ | ( "-" | "∼" | "$" | "&" ) ⟨expression⟩ | "(" ⟨operator⟩ ⟨expression⟩+ ")"
⟨operator⟩    ::= "+" | "*" | "&" | "|" | "^" | "=" | "<u" | "<s" | "<=u" | "<=s" | ">=u" | ">=s" | ">u" | ">s"
                | "«" | "»u" | "»s" | "/u" | "/s" | "%u" | "%s"
⟨instruction⟩ ::= "exit" | "push" | "set_sp" | "jump" | "jump_zero" | "jump_not_zero" | "call" | "return"
                | "load1" | "load2" | "load4" | "load8" | "sigx1" | "sigx2" | "sigx4" | "sigx8"
                | "store1" | "store2" | "store4" | "store8" | "add" | "sub" | "mult" | "neg" | "and" | "or" | "xor" | "not"
                | "pow2" | "shift_l" | "shift_ru" | "shift_rs" | "div_u" | "div_s" | "rem_u" | "rem_s"
                | "lt_u" | "lt_s" | "lte_u" | "lte_s" | "eq" | "gte_u" | "gte_s" | "gt_u" | "gt_s"
                | "read_frame" | "read_pixel" | "put_char" | "put_byte" | "new_frame" | "set_pixel" | "add_sample"
⟨data⟩        ::= ("data1" | "data2" | "data4" | "data8") "[" ⟨expression⟩∗ "]"
```

Tabla II: Sintaxis ensamblador para IVM64 (ver [10]).

C. Expansión de instrucciones

Durante la fase de expansión (ver figura 4), la representación del árbol sintáctico interno (GIMPLE) se expande a través de un conjunto de reglas RTL, denominadas canónicas, que son de obligada inclusión en el archivo de descripción de la máquina. Es precisamente durante esta fase que se utiliza el registro instrumental TR para expresar aquellas operaciones temporales realizadas en la parte superior de la pila.

Obsérvese que TR es un registro especial y se debe indicar a GCC que no lo utilice de forma general. Se debe evitar realizar transformaciones que den lugar a programas erróneos, por ejemplo, debido a una pila desbalanceada. Con este objetivo, en la descripción de la máquina se han definido algunas operaciones sobre TR usando la cláusula RTL `unspec`, que impone ciertas restricciones al compilador al transformar estas expresiones RTL.

Por ejemplo, consideremos la transferencia `dst` ← `TR - TR`. ¿Qué significa? Quizás su signifi-

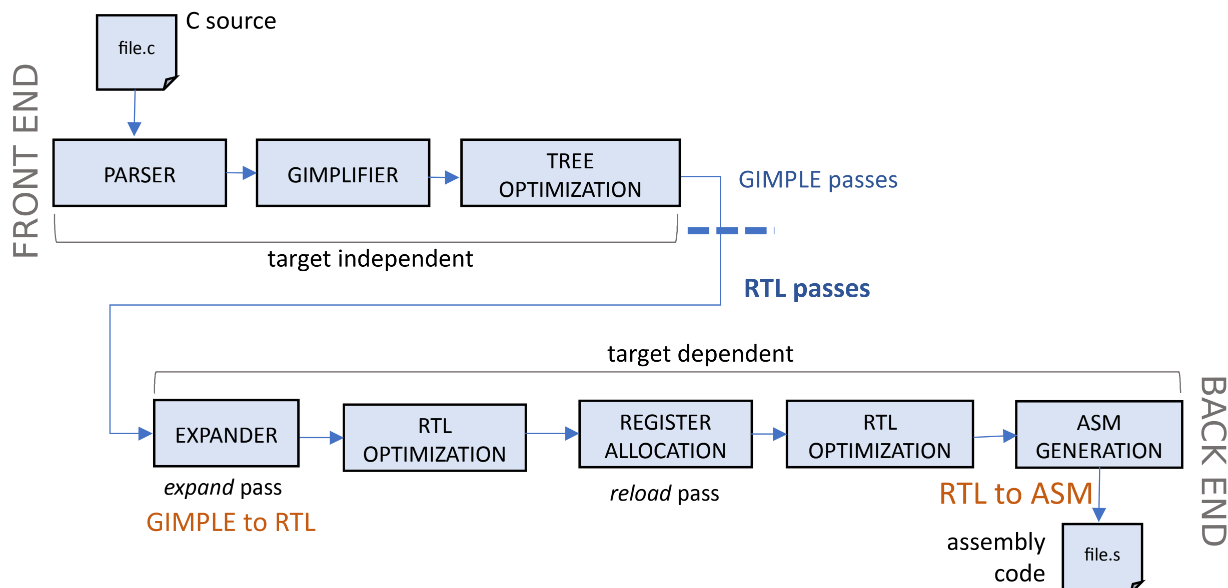

Fig. 4: Descripción general de las etapas de `cc1` (algunas fases como las de optimización pueden implicar múltiples pases)

#	Nombre	Función
0	SP	Stack Pointer
1	FP	Frame Pointer
2	TR	Top of Stack Register
3	AR	Primer registro de propósito general / Valor de retorno
4-18	X1, ... X15	Otros registros de propósito general

Fig. 5: Registros definidos a efectos del compilador. Salvo SP el resto no son registros de la arquitectura.

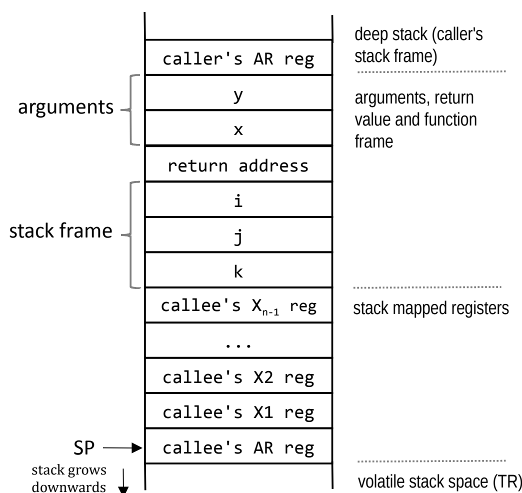

Fig. 6: Ejemplo del bloque de pila inmediatamente después del prólogo de una función.

cado no sea `dst` ← `0`, como probablemente inferiría la maquinaria GCC. En este caso, el propósito del desarrollador es expresar sacar dos operandos de la pila, restarlos y escribir el resultado en `dst`. El registro TR debe usarse sólo para operaciones y transformaciones permitidas en la descripción de la máquina, pero para nada más. Estos patrones RTL definen acciones básicas como acciones *push* (`TR` ← `operando`), *pop* (`dst` ← `TR`) y operaciones aritméticas (`TR` ← `TR op operando`), entre otros.

En el Apéndice se incluye un ejemplo que ilustra el proceso de expansión.

D. *Toolchain de compilación*

Uno de los objetivos al diseñar el compilador de C ha sido mantener el *toolchain* lo más convencional posible, de tal manera que los proyectos de C existentes puedan ser portados y construidos con el mínimo esfuerzo (por ejemplo, consideremos el esfuerzo de adaptar archivos *Makefile*, *scripts* de configuración, etc.).

En el flujo de compilación estándar, `gcc` actúa como un *driver* de compilación que invoca otros programas en secuencia: el preprocesador de C (`cpp`), el compilador de C en sí mismo (`cc1`), el ensamblador (`as`) y el enlazador (`ld`). En primer lugar, `cpp` traduce el archivo fuente C de entrada (por ejemplo, `prog.c`) a un archivo intermedio ASCII (`prog.i`), que `cc1` traduce a un archivo ensamblador (`prog.s`). En segundo lugar, el ensamblador traduce el archivo (`prog.s`) a un archivo objeto reubicable (`prog.o`). En tercer lugar, el enlazador *linker* (`ld`) combina uno o varios archivos de objetos (incluidos los de las librerías) en un solo archivo ejecutable.

En este proyecto no se definió ningún formato de fichero objeto binario enlazable. Como resultado, fue necesario adaptar el flujo de compilación estándar anterior. Esto condujo al desarrollo de un *toolchain* de compilación que funciona completamente en formato ensamblador, delegando la generación binaria a herramientas externas, después del proceso de enlazado. Con este propósito, se definieron los siguientes tipos de archivos:

- *Objeto ensamblador:* Es el resultado de aplicar el programa `as` a la salida de `cc1`. La extensión `.o` se usa para estos objetos. Básicamente, es el mismo archivo ensamblador generado por `cc1`, pero

agregando un sufijo único a los símbolos locales (etiquetas o macros que no se declaran como globales con la cláusula `EXPORT`).
El objetivo de este proceso de cambio de nombre es garantizar que los símbolos locales de un objeto no entren en conflicto con otros símbolos locales (con el mismo nombre) de otro objeto al combinar varios objetos durante el proceso de vinculación.

- *Ejecutable ensamblador:* El resultado de combinar varios *objetos ensamblador*, incluidos los que provienen de bibliotecas. Este proceso de vinculación lo realiza el programa `ld`.

La figura 7 muestra el flujo de compilación. Los programas `as` y `ld` se han implementado como *scripts* de *shell*. Téngase en cuenta que el script `as` al que nos referimos aquí no es la herramienta de externa de ensamblado a cargo de generar el binario.

El script `ld` desarrollado tiene la capacidad de crear un verdadero ejecutable por medio de un encabezado *shebang* agregado a la salida del ensamblador final. Esta función es especialmente útil para probar la salida del compilador en las plataformas actuales, donde se ha compilado el compilador.

De esta forma, la salida final del compilador, el *ensamblador ejecutable*, es una concatenación de: (1) un encabezado *shebang*, que permite su ejecución; (2) el archivo de inicialización del *run-time* de C, `crt0.o`, que se coloca al principio; (3) todos los archivos de objetos de los fuentes C que se compilan (que en realidad son archivos ensamblador); y (4) las bibliotecas C estándar y otras bibliotecas proporcionadas en la línea de comandos de compilación.

E. Comentarios sobre el desarrollo del compilador

El *backend* del compilador para la máquina abstracta IVM ha resultado en un compilador cruzado de C robusto y muy eficiente. La versión de GCC sobre la que se ha desarrollado ha sido la 10.2.0. Cabe mencionar que el desarrollo de este *backend* ha supuesto un gran esfuerzo de validación y trabajo experimental, no solo con las aplicaciones específicas sino con muchos otros benchmarks (ver Apéndice -E).

Hay que mencionar que para llevar a cabo un testeo eficaz del compilador, hemos desarrollado algunas herramientas complementarias como:

- un emulador de código binario IVM64 eficiente [12] escrito en C,
- soporte IVM64 en el virtualizador Qemu [13] y
- un generador de sistemas de archivos en RAM [14].

Como comentario final, indicar que se está trabajando en extender el sistema de compilación para C++, para admitir otros renderizadores de formato escritos en ese lenguaje.

V. Trabajos relacionados

Un buen punto de partida al portar GCC a una nueva arquitectura es analizar los *targets* ya existentes, puesto que GCC se ha portado a una gran cantidad de arquitecturas, tanto más modernas como otras más antiguas. La principal referencia sobre la migración de GCC a un nuevo procesador es [4]. Sin embargo, se pueden encontrar algunas pautas en otras referencias como [15], [16]. Centrándonos en máquinas puramente de pila, el número de *targets* es escaso, aunque podemos destacar dos de ellos: los procesadores Thor [11] y ZPU [17].

El procesador Thor es una CPU de 32 bits desarrollada para aplicaciones aeroespaciales. Es un proyecto relativamente antiguo (gcc 2.7, 1995), donde se desarrolló un *backend* de GCC para dicha arquitectura. El procesador es una máquina puramente de pila, sin más registros que el PC y el SP, aunque la arquitectura es un poco más compleja que la IVM64 propuesta (recuérdese que la simplicidad de descripción era uno de los objetivos). El proyecto Thor aporta algunas ideas valiosas de cómo manejar la ausencia de registros en este tipo de máquinas.

ZPU es una CPU simple de 32 bits de cero operandos. El código fuente del *toolchain* completo para GCC está disponible en [17] (gcc 3.4.2, 2004), incluyendo el compilador, el ensamblador, el enlazador y otros programas auxiliares. A diferencia de Thor, que hace el mapeo de operandos en pila lo antes posible (en fase de expansión), el *backend* de ZPU lo pospone al pase final.

VI. Conclusiones

Diseñar un *backend* de compilador de C para una máquina puramente de pila, como la arquitectura IVM64, plantea paradójicamente algunos desafíos, a pesar de ser una máquina sencilla, ya que la mayoría de las principales infraestructuras de compilación están destinadas a máquinas basadas en registros. Portar el compilador GCC a la arquitectura IVM64, ha dado lugar a un compilador cruzado C robusto y eficiente, que cumple todas las especificaciones contempladas en el proyecto iVM.

El compilador ha sido probado y validado no sólo con aplicaciones específicas del proyecto, sino también con un amplio conjunto de *benchmarks*. El código generado destaca en eficiencia, tanto por el número de instrucciones ejecutadas como por el tamaño binario. Además del propio *backend*, también se portó la librería estándar de C y se desarrollaron algunas otras utilidades auxiliares como un emulador rápido y un generador de sistemas de archivos en memoria.

Agradecimientos

El presente trabajo ha sido financiado mediante el proyecto “Immortal Virtual Machine – Solving the problem of file format and infrastructure obsolescence”, EUREKA Eurostars, número de referencia: E!12494, 2018-2021.

En el proyecto iVM han colaborado las siguientes instituciones: Piql (empresa, Noruega), Tedial (empresa, España), el Centro de Computación de Noruega, el Museo Nacional de Noruega y la Universidad de Málaga.

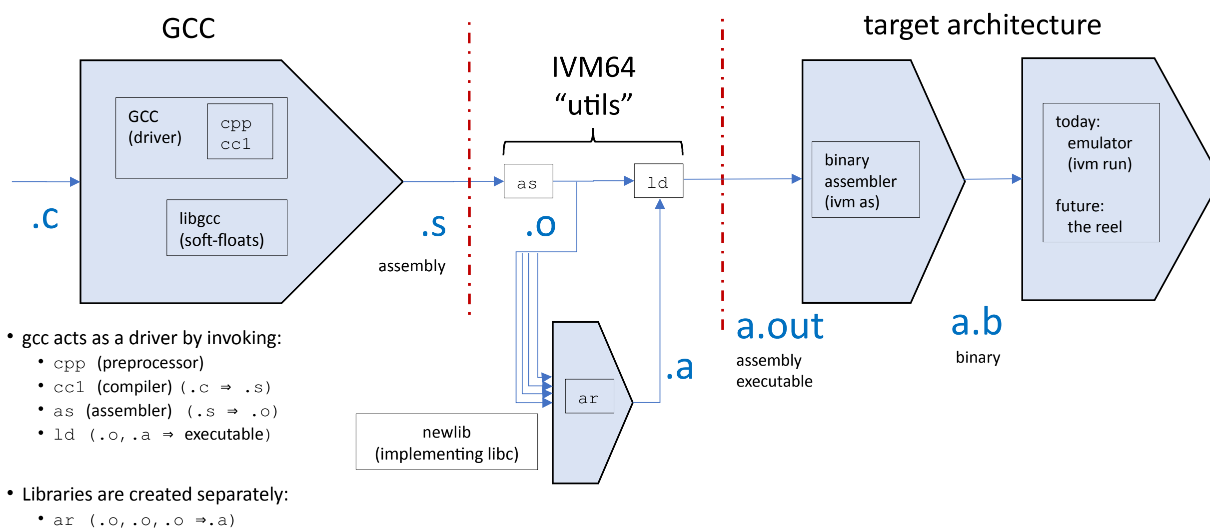

Fig. 7: *Toolchain* de compilación para la arquitectura IVM64.

Apéndice

Este anexo complementa algunos aspectos del diseño del compilador discutidos en el texto principal.

A. *Expansión de instrucciones*

Vamos a ilustrar la expansión de una regla RTL con un ejemplo. Considere esta regla canónica para la división aritmética con 3 operandos: `operando0` ← `operando1/operando2`. Su expansión implica tres pasos: un *push* del primer operando, dividir por el segundo y sacar de la pila el resultado. Esto se traduce en tres operaciones en TR: *push* (`TR` ← `operand1`), operación aritmética (`TR` ← `TR / operando2`), y *pop* (`operando0` ← `TR`). El patrón RTL correspondiente sería como este en la descripción de la máquina:

```
(define_expand "udivdi3"
 [(set (match_operand:DI 0 "" "")
       (udiv:DI (match_operand:DI 1 "" "")
                (match_operand:DI 2 "" "")))]
  ""
{
  ivm_expand_push(operands[1], mode);
  rtx udiv_rtx = gen_rtx_UDIV(mode, TR_REG_RTX(mode),
                              operands[2]);
  emit_insn(gen_rtx_SET(TR_REG_RTX(mode), udiv_rtx));
  ivm_expand_pop(operands[0], mode);
  DONE;
})
```

Aquí los RTL `unspec_push_tr` insertan un operando y `unspec_pop_tr` extraen el resultado. Una vez que aparece el resultado, el contenido del registro TR no está definido. Por esta razón, la función `unspec_pop_tr` necesita *clobber*. Así es como las expresiones de árbol sintáctico GIMPLE basadas en tres direcciones se traducen en varias instrucciones de un operando, con la ayuda del registro TR instrumental. Para este ejemplo, las expresiones RTL emitidas por la expansión para una transferencia como `AR` ← `X1/X2`, con tres registros de propósito general, serían:

```
(set (reg:DI TR_REGNUM)
     (unspec_volatile:DI
          [(reg:DI TR_REGNUM)
           (reg:DI X1_REGNUM)] UNSPEC_PUSH_TR))
(set (reg:DI TR_REGNUM)
     (udiv:DI (reg:DI TR_REGNUM)
              (reg:DI X2_REGNUM)))
(set (reg:DI AR_REGNUM)
     (unspec_volatile:DI
           [(reg:DI TR_REGNUM)] UNSPEC_POP_TR))
(clobber (reg: DI TR_REGNUM))
```

Finalmente, después de todos los pases de compilación, las expresiones RTL resultantes deben imprimirse como instrucciones ensamblador. Las reglas para generar el ensamblador también deben definirse en el archivo de descripción de la máquina. Por ejemplo, el siguiente patrón RTL describe cómo escribir el ensamblador correspondiente a la acción `TR` ← `TR / operand2`:

```
(define_insn "udivdi1"
[(set (match_operand:DI 0
         "tr_register_operand" "=r,r")
      (udiv:DI (match_operand:DI 1
                  "tr_register_operand" "r,r")
               (match_operand:DI 2
                  "arithmetic_operand" "i,rm"
                  )))]
     ""
     "@
     div_u! %2
     load8! %2\;div_u"
)
```

Aquí se define qué instrucciones ensamblador se escribirán al encontrar una expresión RTL que coincida con esta regla. La cadena `tr_register_operand` constituye un predicado, una función que solo es verdadera para el registro TR. Las letras entrecomilladas representan restricciones asociadas a los modos de direccionamiento `i` para inmediato, `r` para registro y `m` para memoria). Los predicados y las restricciones específicas de la máquina objetivo también forman parte de la descripción de la máquina.

B. *ABI*

El ABI (*Application Binary Interface*) es una pieza clave para la integración de módulos de programa escritos en C y código ensamblador escrito a mano. En esta sección se incluye un resumen del ABI diseñado.

Diseño de datos. Los datos enteros pueden tener cuatro tamaños: 1, 2, 4 u 8 bytes de longitud. Pa-

```
    int
    foo  (unsigned char a,
          unsigned int b,
          unsigned long c)
    {
        return a & b & c;
    }

    main()
    {
        return foo(1, 10, 100);
    }
```

```
    EXPORT  foo
foo:
    set_sp! &-16  # prologue:
                  # allocate stack mapped GPRs
    load1! &17    # get arguments
    load4! &19
    load4! &21
    and
    and
    store4! &18 # write return value
                # on the last argument
    set_sp! &16 # epilogue: free stack mapped regs.
    return

    EXPORT  main
main:
    set_sp! &-16 # prologue

    push! 100  # push arguments
    push! 10
    push! 1

    call! foo   # call

    store8! &3  # caller pops return value into AR
    set_sp! &2  # caller frees remainder arguments

    load4! &0   # get AR
    store4! &18 # write return value on the last arg.

    set_sp! &16 #epilogue
    return
```

Fig. 8: Ejemplo del convenio de llamada (código C en la parte superior; abajo código ensamblador generado).

ra enteros con signo, los tipos correspondientes son `char`, `short`, `int` y `long`. Existen los correspondientes tipos sin signo del mismo tamaño. Para números reales, el tipo `float` tiene una longitud de 4 bytes, mientras que el tipo `double` tiene una longitud de 8 bytes. El compilador admite dos tipos de datos adicionales de 128 bits, que se asignan a 2 ubicaciones consecutivas: `long128` y `complex double`. En el caso de estructuras, se debe tener en cuenta que se puede agregar relleno (*padding*) entre los miembros de la estructura, o al final del tipo agregado, pero nunca al principio, antes del primer miembro.

Convenio de llamadas. Se utiliza un convenio único para todas las funciones, independientemente del número de argumentos (sin argumentos, un número fijo de argumentos o un número variable de argumentos). El convenio de llamada sigue la regla *caller-pops-arguments*, de modo que la función llamante está a cargo de mover el valor de retorno a su destino y liberar los argumentos.

Un ejemplo del código ensamblador generado para un código C que llama a una función se muestra en la figura 8.

C. Librerías

El compilador GCC debe ir acompañado de las librerías estándar de C para producir códigos totalmente funcionales. Parte del desarrollo del compilador ha incluido el portar a la arquitectura IVM64 dos de estas librerías claves:

- La libería *libgcc*, que incluye una colección de rutinas para la emulación de punto flotante (`libgcc.a`). Es necesario ya que la máquina abstracta solo admite aritmética de enteros de forma nativa, por lo que la aritmética de punto flotante es emulada por software.
- La libería *newlib* [18], [19], que proporciona la librería estándar de C que incluye: el archivo de inicio del *run-time* de C (`crt0.o`), la librería estándar de C (`libc.a`) y la libería matemática (`libm.a`).

El archivo de inicio `crt0.o` se enlaza por defecto como el primer objeto del programa ejecutable, a menos que se defina un punto de entrada específico (indicador de línea de comando `-e`). Hay que indicar que el diseño de `crt0` depende mucho de cómo la implementación de la máquina abstracta construye e inicializa el código binario. El `crt0` provisto con newlib es responsable de: (1) proporcionar un punto de inicio predeterminado a través de la etiqueta global `_start`, (2) inicializar una variable global que apunta al comienzo del *heap*, (3) inicializar los argumentos de `main()`: `argc` y `argv`, (4) llamar a la función `main(int argc, char* argv[])`, y (5) al salir, dejar el valor de retorno de `main()` en la pila.

Como la arquitectura IVM64 carece de registro FP, se ha incluido en la librería `libc` una implementación software de la función `alloca()`, y que está basada en una implementación portable de dominio público [20].

La *toolchain* diseñada admite dos tipos de librerías, estática (`.a`) y dinámica (`.so`). Los términos *estático* y *dinámico* realmente se refieren al formato de la librería, más que a cómo se cargan. Todo el proceso de compilación es estático, es decir, todas las funciones utilizadas se incluyen en el binario final, ya que no existe un cargador dinámico. El script `ld` separa diferentes objetos incluidos en una biblioteca para procesarlos junto con los otros objetos provenientes de fuentes C. Además el script `ld` puede reducir el tamaño de la salida seleccionando solo aquellos objetos de librerías que realmente se utilizan en los fuentes de C que se están enlazando.

D. Optimizaciones

El código ensamblador generado por el compilador se beneficia de tres tipos de optimizaciones: (1) los pases de optimización proporcionados por GCC, (2) transformaciones *peephole* específicas diseñadas para el target IVM64, y (3) algunas optimizaciones de más bajo nivel a nivel ensamblador. De forma predeterminada, el compilador se ha configurado para utilizar el nivel de optimización `-O2`, que ofrece un buen equilibrio entre velocidad (instrucciones ejecu-

tadas) y tamaño. Las transformaciones *peephole* son un elemento clave de la optimización. Los *peepholes* se aplican como otro paso de optimización de GCC, pero son dependientes del *target*. Para la máquina abstracta se ha definido un amplio conjunto de estas transformaciones, lo que permite la generación de un código sumamente óptimo.

E. Validación

Además de las aplicaciones específicas relacionadas con este proyecto, como la librería *unboxing* de extracción de la información de los fotogramas de película que forman el medio de preservación, y los renderizadores de formato soportados (PDF, JPEG, TIFF,...), el compilador ha sido probado con múltiples conjuntos de prueba, que incluyen:

- el conjunto de tests para C, distribuido junto con GCC. Se puede iniciar después de compilar el compilador con `make check-gcc-c`. Esta es una extensa colección de pruebas básicas (+85000), que incluyen los *torture tests*, pruebas especialmente diseñadas para estresar al compilador,
- la *testsuite* de TCC [21] con casi cien códigos que chequean características básicas del lenguaje C,
- una colección del *benchmarks* provenientes del compilador LLVM [22]: *Olden, Prtdist, VersaBench, Fhourstones, lemon, llubenchmark, mafft, nbench, oggenc, spiff, viterbi, SMG2000, McGill, Shootout, Stanford* y *CoyoteBench*.

Uno de los retos al probar programas reales ha sido la falta de sistemas de archivos para la máquina abstracta. Un enfoque para abordar esta limitación es incluir los archivos en estructuras de datos de memoria en el código fuente de C, de tal manera que el acceso a los archivos se emule mediante el acceso a dichas estructuras. El problema de este enfoque es que implica modificar el código fuente. Por ello, para estos casos se desarrolló un generador de sistema de archivos sin carpetas en memoria [14]. Este generador proporciona un archivo C con el contenido del sistema de archivos, junto con las primitivas básicas de archivo de bajo nivel (`fopen, fclose, fread, fwrite, ...`). De esta forma, el archivo generado puede enlazarse con el resto de ficheros fuentes, que así pueden acceder a los archivos sin necesidad de modificaciones. Estas funcionalidad se basa en la capacidad del enlazador `ld` de reemplazar las primitivas de acceso a los archivo en `libc` por estas nuevas que se proporcionan junto con el sistema de archivos.

Referencias

[1] "Immortal Virtual Machine - solving the problem of file format and infrastructure obsolescence," EUREKA Eurostars project, reference number: E!12494, 2018-2021.

[2] Jedrzej Sabliński, Alfredo Trujillo, et al., "Piql. long-term preservation technology study," *Archeion*, , no. 122, 2021.

[3] Bjørn H Brudeli, "A holistic approach to digital preservation," in *SMPTE 2014 Annual Technical Conference Exhibition*, 2014, pp. 1–11.

[4] Richard M Stallman and the GCC Developer Community, "GNU Compiler Collection internals (for gcc version 10.2.0)," *Free Software Foundation*, 2020.

[5] Ivar Rummelhoff, "Formal specification of VM and I/O devices and description validation (Eurostars Programme, iVM project)," Tech. Rep., Norwegian Computing Center, Oslo, Norway, 2021.

[6] Thor Kristoffersen, "A guide to building the Immortal Virtual Machine," Tech. Rep., Norwegian Computing Center, Oslo, Norway, 2020.

[7] Thor Kristoffersen, "The Immortal Virtual Machine Instruction Set Architecture," Tech. Rep., VirtuMa Solution 3.3 Part a, 2020.

[8] Eladio Gutierrez, Sergio Romero, and Oscar Plata, "The IVM64 compiler," 2021, `https://github.com/immortalvm/ivm-compiler`.

[9] Ivar Rummelhoff, "IVM implementation (v0.37)," 2020, `https://github.com/immortalvm/ivm-implementations`.

[10] Ivar Rummelhoff, Eladio Gutiérrez, Thor Kristoffersen, Ole Liabø, Bjarte M Østvold, Oscar Plata, and Sergio Romero, "An abstract machine approach to preserving digital information," *IEEE Access*, vol. 9, pp. 154914–154932, 2021.

[11] Harry Gunnarsson and Thomas Lundqvist, "Porting the GNU C compiler to the Thor microprocessor," M.S. thesis, Saab Ericsson Space AB, Sweden, 1995.

[12] Eladio Gutierrez, Sergio Romero, and Oscar Plata, "Yet another (fast) IVM emulator," 2020, `https://github.com/immortalvm/yet-another-ivm-emulator`.

[13] Sergio Romero, Eladio Gutierrez, and Oscar Plata, "A QEMU port for the IVM64 architecture," 2021, `https://github.com/immortalvm/qemu-ivm`.

[14] Eladio Gutierrez, Sergio Romero, and Oscar Plata, "IVM filesystem generator," 2020, `https://github.com/immortalvm/ivm-fs`.

[15] Krister Walfridsson, "Writing a GCC back end," 2017, `https://kristerw.blogspot.com/2017/08/writing-gcc-backend_4.html` (retrieved Dec 1, 2020).

[16] "Writing GCC Machine Descriptions," 2010, `http://www.cse.iitb.ac.in/grc/intdocs/gcc-writing-md.html` (retrieved Dec 1, 2020).

[17] "ZPU - The worlds smallest 32 bit CPU with GCC toolchain," 2009, `https://opencores.org/projects/zpu` (retrieved on Dec 1, 2020).

[18] Corinna Vinschen and Jeff Johnston, "The Red Hat newlib C library," 2021, `https://sourceware.org/newlib/` (retrieved on Mar 1, 2021).

[19] Jeremy Bennett, "Howto porting newlib: A simple guide," 2010, `https://www.embecosm.com/appnotes/ean9/ean9-howto-newlib-1.0.html` (retrieved on Mar 1, 2021).

[20] D A Gwyn, "Alloca – a (mostly) portable public-domain implementation," 1986.

[21] Fabrice Bellard, "Tiny C Compiler," `http://bellard.org/tcc` (retrieved on Mar 1, 2021).

[22] "LLVM testsuite," `https://github.com/llvm/llvm-test-suite`.