

GCC 4.1 - Características y Bancos de Pruebas

VUELO DE PRUEBA



La nueva versión del compilador de GNU (GCC) viene con un buen número de optimizaciones y soporte para Objective C. El analizador descendente recursivo presentado en la versión 4.0 se utiliza ahora para los derivados de C y Objective C. **POR RENÉ REBE**

GCC 4.1 ha visto la luz del mundo con un retraso de tan sólo una semana [1]. Mark Mitchell, el director de versiones, se vió forzado a postponer un poco la fecha de publicación para acomodarse al soporte de punto flotante de 128 bits de PowerPC, ya que esta característica es importante para el futuro de Glibc 2.4.

Analizador Manual

Uno de los mayores cambios introducidos es que ahora GCC usa el analizador presentado con la última versión de C++ para C y Objective C. Este analizador descendente recursivo escrito a mano (es decir, no se escribió con los generadores tradicionales de analizadores como Bison o Yacc) es más rápido y se espera que sea más sencillo de mantener con el tiempo.

Apple añadió la posibilidad de mezclar Objective C con código C++ usando GCC con MacOS X hace ya un tiempo. Objective C es una alternativa a C++ que ofrece orientación a objetos con una sintaxis que recuerda a Smalltalk, junto con escritura dinámica e introspección. Como contrapartida, carece de las construcciones modernas de C++ como las plantillas o los espacios de

nombre. Actualmente Objective C++ ofrece la posibilidad de mezclar las características de C++ con Objective C o simplemente permite utilizar las librerías de C++ con el código Objective C.

Los desarrolladores de GCC han integrado ahora el SSP (Stack Smashing Protector) que fue desarrollado por IBM y ha estado disponible como parche desde hace un tiempo [2]. Los programas compilados con el soporte SSP cambian el orden de las variables en la pila para impedir las manipulaciones de los punteros, ya sean inadvertidas o maliciosas. SSP permite a los programadores diseñar funciones que detecten los desbordamientos de los búfers.

Los programadores de GCC han extendido considerablemente las librerías de Java que implementan los aspectos principales de la API, incluyendo el conjunto de herramientas gráficas AWT y Swing. Sólo las aplicaciones Java toman más de un tercio de los Change-Log [3]. GCJ y GNU Classpath soportan la compilación del entorno de desarrollo Eclipse, que fue escrito en Java, sin tener que modificar el código.

Las nuevas optimizaciones, que están basadas en la infraestructura Tree SSA presentada en la versión 4.0,

funcionan a través de los límites de las funciones. Esto proporciona un esquema más fiable para detectar secciones de código sin uso, candidatos para código en línea y variables completamente eliminadas por las optimizaciones. Los desarrolladores han mejorado también la vectorización automática, que aplica bucles sobre vectores tales como SSE (Intel/AMD) y AltiVec (PowerPC).

Bancos de Pruebas

Para los bancos de pruebas, he vuelto a utilizar la colección de bancos de pruebas Openbench referenciados en artículos anteriores de esta publicación. Openbench actualmente dispone de una lista de bancos de pruebas en la web de GCC [8]. He cambiado unas cuantas cosas; he actualizado Botan a la versión más reciente 1.4.12 y he sustituido el test libmad, que muestra resultados muy parecidos para compiladores distintos, con el codificador de MP3 de código abierto Lame. Como muchos otros desarrolladores de código abierto están interesados en un banco de pruebas gratuito con propiedades parecidas a SPEC, propuse que una versión inicial de Openbench congelará este año para futuras comparaciones. Serán bien recibidos sus comentarios y sugerencias.

El Legado

Al igual que en las versiones anteriores de GCC, las extensiones de GNU han sido eliminadas de los lenguajes estandarizados para asegurarse una mayor facilidad de la portabilidad de las aplicaciones al gran abanico de diferentes plataformas y compiladores existentes. En la última versión del compilador de GNU, parece que las declaraciones friend de las clases tendrán que retocarse, no permitiéndose los siguiente:

```
struct a {
    friend void f() {
        ...
    }
};
```

En GCC 4.1, los programadores de C++ tienen que definir las funciones friend, de acuerdo con el estándar, fuera de la clase:

```
struct a {
    friend void f();
}

void f() {
    ...
}
```

Los espacios de nombre sobrespecificados, tal y como se utilizan en muchos programas C++, son otra víctima, como demuestra un análisis del programador de Debian Martin Michlmayr [4].

```
class b {
    void b::f();
};
```

El compilador de GNU devuelve un error: *error: extra qualification 'b::' on member 'f'*. En este ejemplo, hay que eliminar la *b::* de delante de los métodos de clase.

Velocidad

Como ya hiciera en mi artículo previo sobre el compilador de GNU, utilizo de nuevo Openbench para comparar el compilador actual con sus predecesores (véase el cuadro “Bancos de Pruebas”).

En respuesta a una pregunta de varios lectores, medí el tiempo *-O0*,

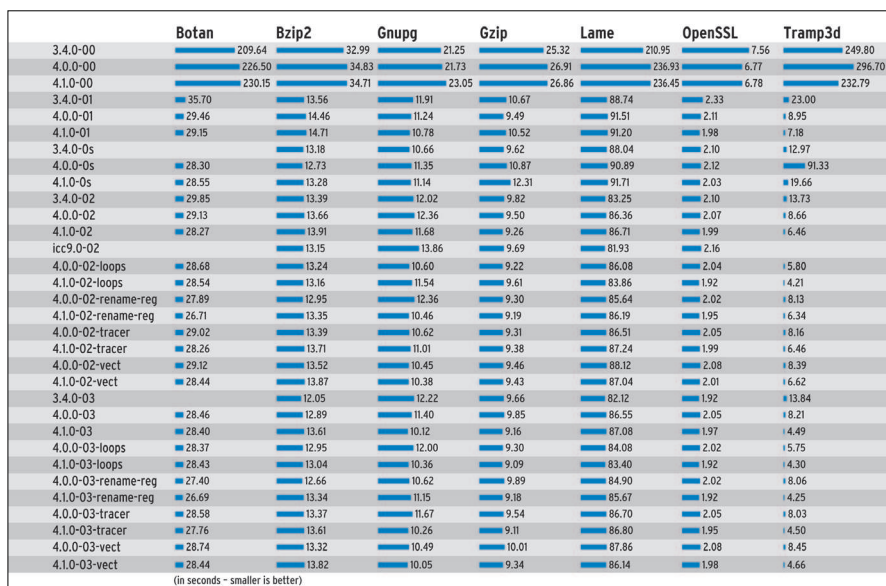


Figura 1: Tiempos de ejecución para varios tipos de compiladores.

aunque esto hace los diagramas más dinámicos. Las nuevas versiones del compilador GCC compilan mucho más rápidamente si las optimizaciones están deshabilitadas. *-O0* es particularmente útil en el ciclo edición-compilación durante el desarrollo. El sistema Athlon que utilicé anteriormente ha sido actualmente reemplazado por un AMD64 Turion64.

El efecto más obvio se hace visible en las pruebas Botan y Tramp3d C++: 25% más rápido con *O2* y sobre el 40% para *O3* en el caso de Botan. Los resultados del banco de prueba oficial de C son una mezcla de datos (véase la Figura 1).

Si se le echa un vistazo a los tiempos de compilación (Figura 2), se podrá ver cuánto tiempo más consume el compilador si un usuario habilita la optimización. El esfuerzo de optimización no se refleja siempre como una influencia positiva en los tiempos de los bancos de prueba. Algo que me hace optimista es que al menos el banco de pruebas de Tramp3d C++ se compila más rápidamente con la versión 4.1 que con su predecesora.

SSP

Tal y como se ha mencionado anteriormente, el SSP (Stack Smashing Protector) de IBM permite a los programadores detectar los desbordamientos de memoria. Para

ello, coloca un valor aleatorio tomado de */dev/urandom*, o si no está disponible, una cadena terminada con *\0\xff\n*, en algún lugar cerca de la dirección de retorno en la pila. Si este valor se ve modificado cuando el programa sale de la función, se puede suponer que la dirección de retorno ha sido sobrescrita. Esto es una técnica comúnmente utilizada por los hackers y el malware. En este caso, el programa produce un aviso y finaliza. Si se compila y ejecuta el programa del Listado 1, se verá un mensaje de error similar al siguiente:

```
***stack smashing detected ***
./ssp-test terminated
```

El *canario* hace que sea difícil para un atacante inyectar código malicioso en forma de scripts. El parámetro de la línea de comandos de GCC, *-fstack-protector*, habilita esta protección.

Listado 1: ssp-text.c

```
01 int f () {
02 char a [200];
03 char* b = a;
04 int i;
05 for (i = 0;
06 i < 201; ++i)
07 a[i] = i;
08 }
09
10 int main () {
11 f ();
12 }
```

	Botan	Bzip2	Gnupg	Gzip	Lame	OpenSSL	Tramp3d
3.4.0-00	209.64	32.99	21.25	25.32	210.95	7.56	249.80
4.0.0-00	226.50	34.83	21.73	26.91	236.93	6.77	296.70
4.1.0-00	230.15	34.71	23.05	26.86	236.45	6.78	232.79
3.4.0-01	35.70	13.56	11.91	10.67	88.74	2.33	23.00
4.0.0-01	29.46	14.46	11.24	9.49	91.51	2.11	8.95
4.1.0-01	29.15	14.71	10.78	10.52	91.20	1.98	7.18
3.4.0-0s		13.18	10.66	9.62	88.04	2.10	12.97
4.0.0-0s	28.30	12.73	11.35	10.87	90.89	2.12	91.33
4.1.0-0s	28.55	13.28	11.14	12.31	91.71	2.03	19.66
3.4.0-02	29.85	13.39	12.02	9.82	83.25	2.10	13.73
4.0.0-02	29.13	13.66	12.36	9.50	86.36	2.07	8.66
4.1.0-02	28.27	13.91	11.68	9.26	86.71	1.99	6.46
icc9.0-02		13.15	13.86	9.69	81.93	2.16	
4.0.0-02-loops	28.68	13.24	10.60	9.22	86.08	2.04	5.80
4.1.0-02-loops	28.54	13.16	11.54	9.61	83.86	1.92	4.21
4.0.0-02-rename-reg	27.89	12.95	12.36	9.30	85.64	2.02	8.13
4.1.0-02-rename-reg	26.71	13.35	10.46	9.19	86.19	1.95	6.34
4.0.0-02-tracer	29.02	13.39	10.62	9.31	86.51	2.05	8.16
4.1.0-02-tracer	28.26	13.71	11.01	9.38	87.24	1.99	6.46
4.0.0-02-vect	29.12	13.52	10.45	9.46	88.12	2.08	8.39
4.1.0-02-vect	28.44	13.87	10.38	9.43	87.04	2.01	6.62
3.4.0-03		12.05	12.22	9.66	82.12	1.92	13.84
4.0.0-03	28.46	12.89	11.40	9.85	86.55	2.05	8.21
4.1.0-03	28.40	13.61	10.12	9.16	87.08	1.97	4.49
4.0.0-03-loops	28.37	12.95	12.00	9.30	84.08	2.02	5.75
4.1.0-03-loops	28.43	13.04	10.36	9.09	83.40	1.92	4.30
4.0.0-03-rename-reg	27.40	12.66	10.62	9.89	84.90	2.02	8.06
4.1.0-03-rename-reg	26.69	13.34	11.15	9.18	85.67	1.92	4.25
4.0.0-03-tracer	28.58	13.37	11.67	9.54	86.70	2.05	8.03
4.1.0-03-tracer	27.76	13.61	10.26	9.11	86.80	1.95	4.50
4.0.0-03-vect	28.74	13.32	10.49	10.01	87.86	2.08	8.45
4.1.0-03-vect	28.44	13.82	10.05	9.34	86.14	1.98	4.66

(in seconds - smaller is better)

Figura 2: Tiempos de compilación para varios tipos de compiladores.

Mudflap

Una técnica presentada en GCC 4.0 lleva esta protección un paso más allá que SSP, habilitando la validación de las referencias a punteros en C y C++. Durante la compilación, este mecanismo, conocido como Mudflap [5], controla los accesos a memoria para reflejar el tipo de acceso y las condiciones que el compilador detecta en este momento. Por ejemplo, la propagación de constantes podría ocasionar comprobaciones innecesarias. En tiempo de ejecución, las funciones proporcionadas por la librería Libmudflap validan estos accesos, finalizando el programa en casos críticos. Libmudflap también valida muchas funciones C estándar capaces de sobrescribir la memoria, incluyendo *mem**, *str**, *put**, *get**, y muchas otras más. Para utilizar Mudflap, todos los ficheros deben compilarse con la opción *-fmudflap* y enlazados con *-lmudflap*. El pequeño programa en C de prueba del Listado 2 accede a una posición de memoria un byte más allá del límite del array *a*. En nuestra

prueba, Mudflap detectó correctamente el nombre de la variable que sobrepasó el límite.

Como Mudflap valida los accesos a memoria en muchos casos, puede afectar considerablemente al rendimiento en comparación con SSP, que apenas se hace notar. Ante esto, Mudflap es realmente útil para los desarrolladores que están interesados en un método rápido de detección de errores críticos potenciales en las etapas tempranas de desarrollo.

El Futuro

La versión 4.1 del compilador de GNU viene con nuevas optimizaciones y otros avances importantes. Los programadores GNU, sin embargo, ya están mirando hacia la siguiente versión. Las características previstas

para GCC 4.2 [6] incluyen el soporte para OpenMP [7] y extensiones para los lenguajes C, C++ y Fortran para soportar paralelización explícita. Estos proyectos cerrarán los huecos que recientemente han quedado abiertos entre los compiladores libres y los comerciales. ■

RECURSOS

- [1] Sitio web de GCC: <http://gcc.gnu.org/>
- [2] SSP (Stack Smashing Protector): <http://www.trl.ibm.com/projects/security/ssp/>
- [3] Cambios en GCC 4.1: <http://gcc.gnu.org/gcc-4.1/changes.html>
- [4] Compilación en Debian con GCC 4.1. basado en pruebas: <http://gcc.gnu.org/ml/gcc/2006-03/msg00740.html>
- [5] Mudflap: <http://gcc.fyxm.net/summit/2003/mudflap.pdf>
- [6] GCC 4.2: <http://gcc.gnu.org/wiki/GCC%204.2%20Projects>
- [7] OpenMP: <http://www.openmp.org/drupal/mp-documents/spec25.pdf>
- [8] Openbench: <http://www.exactcode.de/oss/openbench/Info>

Listado 2: mudflap-test.c

```
01 int main () {
02 char a [200];
03 char* b = a;
04 printf ("%c\n", b[200]);
05 }
```