GCC 4.1 – Features and Benchmarks

TEST FLIGHT

The new version of the GNU Compiler (GCC) has a fresh crop of optimizations and support for Objective C. The Recursive Descent Parser introduced in version 4.0 is now used for the C and Objective C derivatives. **BY RENÉ REBE**

GCC 4.1 has seen the light of the world with a delay of just one week [1]. Release Manager Mark Mitchell was forced to postpone the scheduled released data slightly to accommodate 128-bit floating point support for the PowerPC, as this feature is important for the future Glibc 2.4.

Manual Parser

One of the major changes is that GCC now uses the parser introduced with the latest version of C++ for C and Objective C. This manually written recursive descent parser (that is, it was not written with a traditional parser generator such as Bison or Yacc) is quicker, and is expected to be easier to maintain in the long run.

Apple added the ability to mix Objective C with C++ code to the GCC on MacOS X quite awhile back. Objective C is an alternative to C++ that offers object-orientation with a syntax that resembles that of Smalltalk, along with dynamic typing and introspection. On the downside, it lacks modern C++ constructs such as templates or namespaces. Objective C++ now gives you the ability to mix C++ features with Objective C, or simply to use C++ libraries with Objective C code.

The GCC developers have now integrated the Stack Smashing Protector (SSP) which was developed by IBM and has been available as an add-on patch for quite awhile now [2]. Programs built with SSP support change the order of variables on the stack to prevent inadvertent or malicious pointer manipulation. SSP lets programmers design functions to detect buffer overflows.

The GCC programmers have also considerably extended the Java library that implements major aspects of the Java API, including the AWT graphic toolkit and Swing. The Java applications alone take up more than a third of the ChangeLog [3]. GCJ and GNU Classpath support the compilation of the Eclipse developer environment, which was written in Java, without modifying the code.

The new optimizations, which are based on the Tree SSA infrastructure introduced in version 4.0 now work across function borders. This provides a more reliable scheme for detecting unused sections of code, candidates for inlining, and variables completely removed by optimizations. The developers have also improved automatic vectorization, which maps loops to vector units such as SSE (Intel/AMD) and Altivec (PowerPC) .

Legacy

Just as in previous versions of GCC, GNU extensions have been removed from standardized languages to ensure improved portability of the applications across different platforms and compilers. In this latest version of the GNU compiler, it appears that friend declarations in classes have somehow fallen foul of the cleanup:

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

Benchmarks

For the benchmarks, I again used the Openbench benchmark collection referred to in previous articles for this magazine. Openbench has now officially made the benchmark list on the GCC homepage [8].

I had to change a few things; I updated Botan again to the more recent version 1.4.12, and replaced the libmad test, which shows very similar results for different compilers, with the Lame open source MP3 encoder.

As many other open source developers are interested in a free benchmark with similar properties to SPEC, I anticipate that an initial reference version of Openbench will be frozen this year for future comparisons. Your comments and assistance are welcome.

In GCC 4.1, C++ programmers have to define a standards-compliant friend function outside of the class:

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

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

Overspecified namespaces, as used in many C++ programs, are another victim, as an analysis by Debian programmer Martin Michlmayr demonstrates [4].

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

The GNU Compiler now returns an error: *error: extra qualification 'b::' on member 'f'*. In this example, you need to remove the *b::* in front of the class methods.

Speed

As in my previous articles on the GNU Compiler, I again used Openbench to discover how the current compiler compares to its predecessors (see the "Benchmarks" box).

In response to a request by several readers, I measured the *-O0* time, although this makes the diagrams more dynamic. The new GCC versions compile far more quickly if optimization is disabled. *-O0* is particularly useful in the edit-compile cycle during development. The Athlon system I used previously has now been replaced by an AMD64 Turion64.

Listing 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 }
```

The most obvious effect is visible in the Botan and Tramp3d C++ tests: 25 percent quicker for *O2*, and over 40 percent for *O3* in the case of Botan. The legacy C benchmark results are a mixed bunch (see Figure 1).

If you take a look at the build times (Figure 2), you will see how much more work the compiler puts in if a user enables optimization. And the optimization effort is not always reflected as a positive influence on the benchmark runtimes. One thing that makes me optimistic is that at least the demanding Tramp3d C++ benchmark is compiled more quickly by version 4.1 than by the predecessor.

SSP

As mentioned earlier, the IBM Stack Smashing Protector lets programmers

Listing 2: mudflap-test.c

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

	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)

Figure 1: Runtimes for various compiler scenarios.

detect buffer underflows or overflows. To do so, it places a random value from */dev/urandom*, or if this is not available, a terminating string *\0\xFF\n* somewhere near the return address on the stack. If this value is changed when the program exits the function, you can assume that the return address has also been overwritten. This is a technique commonly used by hackers and malware. In this case, the program outputs a warning and quits. If you compile and run the program in Listing 1, you will see an error message similar to the following:

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

The canary makes it difficult for attackers to inject malevolent code in the form of scripts. The GCC command line parameter, *-fstack-protector*, enables this protection.

Mudflap

A technique introduced in GCC 4.0 takes this protection one step farther than SSP, enabling validation of pointer references in C and C++. During the build, this mechanism, known as Mudflap [5], instruments memory access to reflect the access type and the conditions the compiler detects at this time. For example, constant propagation could render some checks unnecessary. At runtime, the functions provided by the Libmudflap library validate this access, terminating the program in critical cases. Libmudflap also validates many standard C functions capable of overwriting memory, including *mem**, *str**, **put**, **get**, and many more. To use Mudflap, all files must be compiled with the *-fmudflap* flag and linked with *-lmudflap*. The small C test program in Listing 2 just accesses a memory position one byte beyond the boundary of the *a* array. In our test, Mudflap correctly pointed out the variable name that overstepped the boundary.

As Mudflap validates memory access in many cases, it can affect performance considerably in comparison to SSP, which is hardly noticeable. In this light, Mudflap is mainly useful for developers who are interested in a quick method of detecting potentially critical errors at an early stage of development.

Future

Version 4.1 of the GNU compiler comes with new optimizations and other important advances. GNU programmers, however, are already looking ahead to the next release. The features scheduled for integration with GCC 4.2 [6] include support for OpenMP [7] and extensions of the C, C++, and Fortran languages to support explicit parallelization. These projects will close the gaps that recently opened up between the free compiler collection and commercial compilers. ■

Info

[1] GCC homepage: *http://gcc.gnu.org/*

[2] Stack Smashing Protector SSP: *http://www.trl.ibm.com/projects/security/ssp/*

[3] Changes to GCC 4.1: *http://gcc.gnu.org/gcc-4.1/changes.html*

[4] Compiling Debian with GCC 4.1 – report on experiences: *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/*

	Botan	Bzip2	Gnupg	Gzip	Lame	OpenSSL	Tramp3d
3.4.0-00	106.80	2.23	16.46	1.06	14.90	74.05	28.12
4.0.0-00	94.02	2.35	17.09	1.12	15.04	76.13	35.63
4.1.0-00	95.98	2.38	17.14	1.16	15.25	74.85	35.46
3.4.0-01	156.91	2.87	23.48	1.56	19.09	88.40	50.41
4.0.0-01	141.38	3.77	27.05	2.01	21.84	98.03	110.11
4.1.0-01	183.61	4.55	29.16	2.17	23.51	100.46	87.50
3.4.0-0s	193.57	4.48	30.25	1.86	23.46	98.96	68.16
4.0.0-0s	60.30	3.90	31.77	2.19	23.77	104.91	43.36
4.1.0-0s	63.76	4.34	33.33	2.35	25.54	108.28	42.60
3.4.0-02	198.22	5.09	33.56	2.15	24.68	109.39	74.60
4.0.0-02	165.72	5.62	35.82	2.64	28.14	118.50	139.78
4.1.0-02	99.01	6.17	36.66	2.67	29.90	117.30	108.59
icc9.0-02		5.36	32.90	2.07	29.32	118.61	
4.0.0-02-loops	76.35	6.16	38.10	3.40	35.60	120.06	144.44
4.1.0-02-loops	103.32	7.06	41.98	3.64	38.54	125.88	115.81
4.0.0-02-rename-reg	168.43	5.45	35.96	2.48	28.57	115.84	134.40
4.1.0-02-rename-reg	236.07	6.13	37.66	2.70	30.16	118.95	109.20
4.0.0-02-tracer	171.01	5.49	35.02	2.54	28.54	116.25	136.12
4.1.0-02-tracer	238.54	6.17	37.42	2.77	30.45	117.98	110.14
4.0.0-02-vect	166.87	5.89	35.76	2.56	29.78	117.78	141.35
4.1.0-02-vect	234.50	6.43	38.24	2.82	31.78	121.03	112.31
3.4.0-03	93.44	5.79	36.13	2.40	26.88	112.72	73.52
4.0.0-03	78.25	6.13	37.01	2.83	30.80	116.92	142.56
4.1.0-03	104.89	7.05	40.57	3.22	35.95	123.19	117.28
4.0.0-03-loops	81.48	7.74	42.20	3.80	38.77	124.95	153.54
4.1.0-03-loops	109.67	7.97	47.05	4.18	43.08	130.84	123.62
4.0.0-03-rename-reg	180.12	6.25	38.18	2.88	30.91	119.53	144.09
4.1.0-03-rename-reg	248.32	7.15	41.61	3.25	36.08	124.97	117.92
4.0.0-03-tracer	178.12	6.26	38.90	2.97	31.95	118.46	145.33
4.1.0-03-tracer	252.00	7.17	41.32	3.28	36.19	123.72	119.48
4.0.0-03-vect	177.04	6.79	39.10	3.02	32.70	121.69	150.60
4.1.0-03-vect	247.40	7.45	43.59	3.37	37.73	127.30	121.95

(in seconds – smaller is better)

Figure 2: Build times for various compiler scenarios.