UP CLOSE GCC 4.0

The compiler at the heart of open source is heading for a new release. GCC fan and sometime contributor Biagio Lucini talks to leading developers for an exclusive preview.

Nothing we do with open source would be possible without the compiler collection GCC. It may be mastered only by an inner circle of C++ gurus but it affects us all. It's GCC that allows your distributor to build the system you're running right now, and every improvement to it results in shorter execution times and smaller binaries.

GCC is where the magic takes place, and that's why we're paying close attention to the major forthcoming release of GCC 4.0, a benchmark for the project. Mailing lists talk of faster optimisation, improved security and cool hacks. Given GCC's chequered history, we were keen to find out if these new additions are the result of harmonious exploration – or acrimonious forks.

That history began in 1984, when Richard Stallman wrote the first chunk of GCC, the C front-end. In the same year the GNU project officially began, and it's no surprise that GCC is at the heart of it: it's hard to imagine how you could provide freely modifiable software without providing a way to convert the modifications into executable code.

Three years later, in 1987, Stallman decided to expand the front-end into a fully-blown compiler, beginning GCC's journey to the version 4.0 we're awaiting. Architectural limitations of this first release series were overcome in 1992 with the publication of version 2.0, which also added support for C++. GCC was beginning to be adopted as the official compiler on several software platforms (including Linux), and its 2.7 manifestation received special praise.

Fork ahead

Through the nineties, GCC development remained in the firm hands of the Free Software Foundation (FSF), which was more focused on stabilising than on improving the compiler. As a consequence, third-party patches aimed at simplifying the building process on some architectures or adding functionalities were very often

CODE BREAKERS
Watch out for these GCC breakages

Because GCC standards are pretty complicated, they haven't always been implemented; particularly in early versions. Recent releases have been more standards-compliant, but this means the old bad code is now breaking with updates. In fact, version 3.0 showed signs that certain features would break. The good news is that the level of breakages in this latest update is lower than the transition from 3.3 to 3.4. Here are three to look out for:

NEW FORTRAN FRONT-END Don't expect all of your code to be parsed as before.

JAVA ABI Breaks binary compatibility of Java applications, pretty much as happened with C++ from 2.95 to 3.0.

VARIABLE TRACKING This new feature requires the user to upgrade to GDB 6.1.

rejected. But because GCC was still GPL software, users could choose to apply the patch set they liked best. This gave rise to a dangerous spread of unofficial versions, with the risk that a serious fork would slow the development of the official version. To avoid this, in 1997 some leading GCC developers breathlessly decided to fork the project themselves.

This was the birth of EGCS (pronounced eggs). Among the declared objectives of EGCS were improvements in the C++ area and the addition of Fortran 77 support (g77). The project was very successful and many vendors included EGCS side by side with GCC in their distributions.

Within a few years the superiority of EGCS over GCC became striking, leading the FSF to give its official blessing to the development model at the root of EGCS in late 1999. EGCS, which was itself undergoing forks such as the PGCC project (aimed at building fast executables on Pentium-class machines), became GCC 2.95. One of the differences between the development process of EGCS and the previous GCC was that the open model of EGCS was tailored to make forks useless, and projects like PGCC slowly died out, being either reabsorbed or superseded by EGCS.

Storm in a red hat

Despite that, the story of forks was far from over. About a year after the adoption of EGCS as the official GCC, Intel released the Itanium, a promising new architecture with the potential to become a leading platform in the middle- to high-end server sector. Red Hat was faced with a problem: it wanted to provide out-of-the-box support for the new IA64 architecture; the official version of GCC at that time (2.96) did not support the Itanium, but the upcoming version of the GNU compiler collection (still in heavy development) would.

Keen to provide a unified base system across all supported platforms, Red Hat made the decision to provide as its official compiler a heavily patched version of what should have become GCC 3.0. By itself this would not have been a big deal, but it turned out that that this compiler (which Red Hat named GCC 2.96 without permission from the FSF) failed in building the Linux kernel. Even worse, the so-called GCC 2.96 was binary incompatible with both the stable and the development versions of GCC. Users assumed the FSF had released a buggy program that was unable to compile the kernel and that broke binary compatibility.

The GCC team reacted promptly, issuing an official statement in which they clarified their position on GCC 2.96 and blamed the poor performance on Red Hat. Even Red Hat tried to explain its actions and resolved some of the problems by providing an alternative compiler based on EGCS (known as KGCC, a compiler meant to be used to recompile the kernel). Alas, the fiasco was by then irreversible. Red Hat insisted on this dual compiler approach (followed closely by other vendors including Mandrake) for about a year, until GCC 3.0 was officially released. That said, Red Hat has been and still is one of the major contributors to GCC; today, some of the leading GCC developers are Red Hat staff.

Truly open at last

GCC 3.0 was the natural result of the efforts started with EGCS. The focus was still on stability, but improvements were no longer renounced, even if sometimes they could have broken compatibility. In fact, GCC 3.0 broke binary compatibility for C++ code, since it contained a major improvement in the form of a new Application Binary Interface (ABI) for that language. It took another minor release for the ABI to stabilise, but the neat result was a more standards-compliant and predictable compiler.

"USERS ASSUMED THAT THE FSF HAD RELEASED A BUGGY PROGRAM."

Throughout the 3.x series, developers have continued to improve and stabilise the set of features introduced in GCC 3.0. Although most of the work has centred on C++, support for the other officially-included languages (Objective C, »

INSTRUCTIONS FOR THE IMPATIENT
How to set up GCC 4.0 for immediate use

As with many open source projects, you can obtain GCC via anonymous CVS. For this, you need CVS installed on your system. Once you've made sure you have it, open a terminal and perform the following operations:

```
mkdir /tmp/gcc
cd /tmp/gcc
export CVS_RSH=ssh
cvs -d :pserver:anoncvs@gcc.gnu.org:/cvs/gcc -z 9 co -P gcc
```

This will create a new directory, gcc inside /tmp/gcc. It's now time to build the sources. If you are interested only in the C, C++ and Fortran front-ends, you can proceed as follows:

```
mkdir build
cd build
../gcc/configure --prefix=/opt/gcc --enable-languages=c,c++,f95
--enable-shared --enable-threads=posix --disable-checking --enable-long-long
--enable-__cxa_atexit --enable-clocale=gnu --disable-libunwind-exception
make bootstrap
```

and as root:

```
make install
```

This will install the compiler in /opt/gcc. The location has been chosen in such a way that no conflict is generated with the existing GCC installation, since you will need the old GCC for compiling new kernel modules and so on.

The last step is to tell the system where to look for GCC. Type

```
export PATH=/opt/gcc/bin:$PATH
export LD_LIBRARY_PATH=/opt/gcc/lib:/opt/gcc/libexec:$LD_LIBRARY_PATH
```

into a terminal and invoke GCC or equivalent commands.

The command gcc -v should now contain as the last line of the output something like 'gcc version 4.0.0 20050223 (experimental)', where the date refers to the CVS version you have checked out. Remember that those settings will be lost when you quit the shell. Of course, you can make GCC 4.0 your default compiler, but until your distribution migrates to it this is highly inadvisable.

WHAT IS SSA?

A framework for better optimisation. It will improve your life!

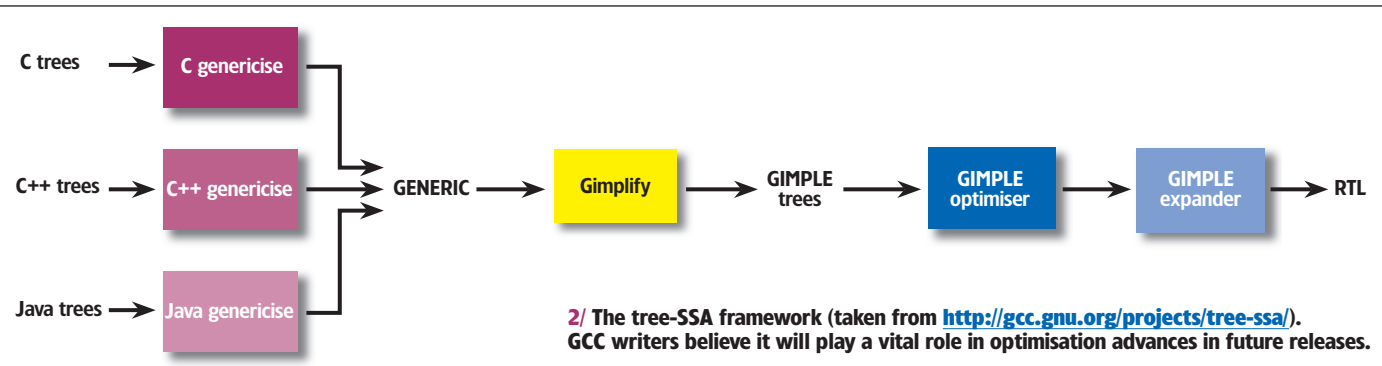
When writing code, it's common to reuse names of dummy variables. Take, for instance, the code snippet:

```
a = 3;
b = f(a);
a = 4;
```

The a that appears at line 3 has nothing to do with the a at lines 1 and 2. What the Single Static Assignment does is to give a different name to logically independent variables, so each newly referenced variable must have a new name. In the SSA representation, the same code becomes:

```
a1 = 3;
b1 = f(a1);
a2 = 4;
```

The scopes of the variables are now clearly exposed. This representation offers a powerful tool for analysing dependencies among different portions of a program, which is the starting point for effective optimisations.



2/ The tree-SSA framework (taken from http://gcc.gnu.org/projects/tree-ssa/). GCC writers believe it will play a vital role in optimisation advances in future releases.

Fortran 77, Ada and Java) has been vastly improved. As a result of the the language–independent infrastructure being revised, the generated code is generally faster than the corresponding 2.x executables, and support for more architectures has been added (there are few platforms to which GCC 3.x has not been ported).

Having learned its lesson with EGCS, GCC now welcomes new ideas – and the transformed open nature of the development process is a large factor in GCC's success and swift development. CVS access is restricted to a few trusted developers and, as GCC is still the property of the FSF, all contributors need to sign a copyright transfer form to donate their code to the project. But there's plenty of room for developers who want to experiment with new constructs within the framework of GCC.

Everyone can contribute patches by sending them to **gcc-patches@gcc.gnu.org**. These will be peer reviewed, and if they're considered correct, adherent to GCC coding conventions and useful to the community, they will be checked into the main tree.

Patches that require heavy modification of the architecture undergo a stricter review process. First, the main code is forked. Then an unofficial distribution maintained in the form of a CVS branch of the main repository is started, to be periodically synchronised with mainline. Being experimental software, the criteria for code that's checked into a branch are less strict than those for mainline additions. If and when the branch proves to do useful work without destabilising the compiler, it will be merged with mainline. Otherwise it will have been just an interesting exercise.

Many of GCC's major projects began life in one of these branches. The projects are overseen by the steering committee, a group of leading developers who decide what direction GCC should follow. It includes developers from different companies and institutions (such as David Edelsohn, a K42 researcher at IBM, Jeff Law of Red Hat and Gerald Pfeifer, who works on Itanium at SUSE), with the aim of balancing different or even opposing needs within the user community.

Before a new version is released, its source code undergoes three different stages. In Stage One the project is under heavy development and major modifications can be accepted. In Stage Two only stabilisation of the approved features can be performed. Any major revision will go to a branch, which will be the basis for the version following the next one. At Stage Three the known bugs are fixed. The final check consists of analysing the results obtained by running the compiler on the provided test suite: there must be no regression with respect to the previous version before the compiler can be tagged with the release number.

The person responsible for this process is the release manager. Since version 3.0, the release manager for GCC has been Mark Mitchell *(see Q&A, right)*.

High hopes

GCC is now at version 3.4.3, expected to be the last release in the successful 3.x series before the coming of version 4.0, which is at Stage Three in its development at the time of writing (and the chances are that it will be out by the time you read this).

The big jump in the release number reflects a major development: the adoption of a new optimisation framework that makes use of the Single Static Assignment (SSA) transformations. Once the framework matures, it will provide faster and better generated code and be the basis for further optimisation. The initial SSA implementation is largely

just a framework for the future, but the next few releases of GCC will include optimisations (tweaks, basically) based on this initial release.

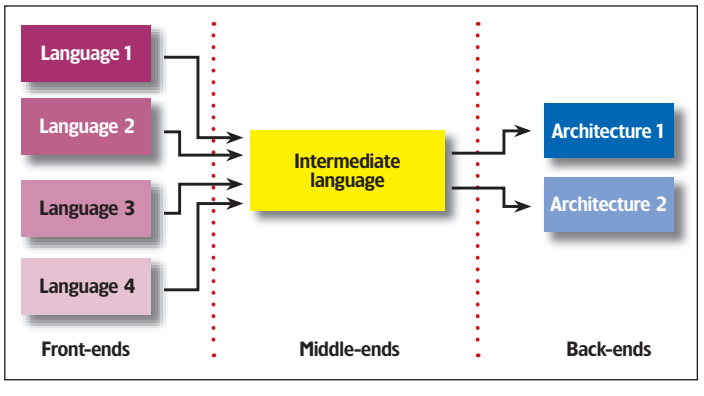
To understand why the new optimisation framework will make such a difference, we have to take a step backward and talk about compilers in general. A compiler is a software program that transforms a text file written according to well-defined lexical and syntactical rules specified by the programming language into machine executable code. The compilation process comprises a parsing part, in which the source is validated; an optimisation part, in which the code is restructured for improving its performance; and a generation part, in which the executable is built. Technically, we refer to them respectively as the front-end, the middle-end and the back-end.

The three components do not have to be kept distinct, but if they aren't, to support x languages on y different architectures one would need to write x times y different compilers. As readers of Paul Hudson's *LXF* series on compilers will know, the clever way to reduce the work is to make sure that the middle-end is logically separated from the front-end and the back-end. If the middle-end also makes use of a representation of the source code that is not language-specific, front-ends of different languages can share it.

In the same way, it's possible to interface several back-ends to the same middle-end. For a compiler that follows this structure, to support x languages on y architectures you would need x + y separate projects emitting or accepting code according to the rules dictated by the middle-end. **Fig 2** represents the structure of such a compiler.

In principle, old versions of GCC have followed that structure, with the front-end emitting abstract syntax trees (ASTs) and the intermediate language being Register Transfer Language (RTL). Unfortunately for fans of smooth compiling, the ASTs generated by each front-end differ, and the RTL representation is not well suited for high-level optimisations. Each front-end has to know about optimisations, which – apart from causing duplication of efforts – means the quality of the generated code is dependent on the language and optimisation processes in the particular front-end.

What's the answer? The new tree-SSA framework, which will offer a language-independent infrastructure for optimisations, sitting as it does between the front-ends and the RTL *(see What Is SSA? box, left)*.



2/ An ideal compiler that supports four languages on two different architectures.

»

MARK MITCHELL: *GCC GUARDIAN*

As GCC's release manager, Mark Mitchell has the heavy responsibility of overseeing new additions to the collection. We ask him if the project is feeling the heat from rival IBM compilers.

CV

Based: Granite Bay, California
GCC role: Release manager, Steering Committee member
Day job: Founder, CodeSourcery consultancy
On GCC 4.0: "It's going to be a bit of a smorgasbord."

LXF: How have you been involved in GCC's development?
MM: I've enjoyed working on compilers and programming languages for a long time: in fact, my elementary school computer teacher was a wonderful woman who was very interested in programming languages. So I think I was doomed to like compilers from about age five!

My biggest role is release manager. I decide when it's time to officially release a new version of the compiler. I also help steer what changes go into the compiler at which points in the development cycle and I try to facilitate high-level technical conversations about the desirability of particular changes.

Historically, I've done a lot of development of the G++ compiler. I still do some of that, but now I'm working more on other things, including managing CodeSourcery's rapid growth. I can get a lot more done by helping others than by trying to do it all myself!

LXF: What are the goals of GCC?
MM: It depends a lot on who you ask. One of the challenges is that the goals of the various stakeholders are not uniform. Some people want to see releases very frequently so that improvements are always available to people. The distribution vendors want to see releases that contain the features their customers need on a schedule that works for them. Some people want maximum backwards compatibility with older versions of the compiler. Some people want strict conformance with language standards.

"I THINK I WAS DOOMED TO LIKE COMPILERS FROM ABOUT THE AGE OF FIVE."

It's a pretty diverse set of goals, and sometimes the goals are incompatible.

LXF: How is GCC developed?
MM: GCC is developed by a pretty large team. Most of the major contributors are now being paid for their efforts, which is somewhat different from five or ten years ago. But there's still a tremendous amount of volunteer effort as well. I don't want to name particular organisations because I'll probably leave somebody out, and I don't want to be accused of promoting particular interests. In general, the major contributors are software development businesses (like CodeSourcery), GNU/Linux distribution vendors, operating system vendors and hardware vendors.

The development model has come out of years of evolution. It's a balance between freeform development and a strictly top-down model. The GCC Steering Committee sets some high-level policies, but most technical decisions are being made by the individual maintainers. There's a lot of back-and-forth between the developers to work out how best to solve problems. We use peer review to check each other's work and decide on designs.

LXF: What can the end user expect from GCC 4.0?
MM: It's going to be a bit of smorgasbord. The reason for the major version number change [from 3 to 4] is that GCC 4.0 will contain the tree-SSA infrastructure. There are some programs that run a lot faster with GCC 4.0.

I think that GCC 4.1 will demonstrate even more of an across-the-board win. Frankly, replacing most all of the optimisers in GCC with brand-new technology, and having it (a) work, and (b) not generate worse code is a huge achievement!

GCC 4.0 also contains a Fortran 95 front-end. It's not as polished as C or C++ at this point, but it's coming along very nicely. The C++ front-end is substantially faster when compiling without optimisation. As always, there is support for more chip variants, newer versions of operating systems, and tons of bugfixes.

LXF: Has the availability of the Intel compilers had any impact on the development goals of GCC?
MM: I believe that competition is great for GCC. People say a lot of things, positive and negative, about the Intel compilers. I'm not going to do that; I've not examined them closely enough to say for sure. I'm confident that there exist programs for which those compilers generate better code, and that will push GCC to improve as well.

TIGHTER SECURITY Version 4.0 gives you added protection

Security-minded readers will be pleased to hear that GCC 4.0 addresses a common exploit known as buffer overflow. This is where an attacker passes a huge string or number to a sick program, gaining access to memory areas and often taking on root privileges.

The answer is to perform sanity checks for possible buffer overflows in any line of code – but unfortunately this isn't done by default. Version 4.0's solution comes in the form of the -D_FORTIFY_SOURCE switch. When enabled, sanity checks will be performed by the compiler, and if there is the possibility of an overflow, more secure library functions will be called instead of the default ones. For this reason, you'll need the glibc library (version 2.3.4 or later) or a patch for it.

One of the biggest advantages of this method is that the check can be performed with no or very little run time overhead. There are two levels of fortification: -D_FORTIFY_SOURCE=1 is the standard, while -D_FORTIFY_SOURCE=2 gives even more security, at the expense of possible failures of some conforming programs. Read more at http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html.

Before code can be converted to the SSA form, two preliminary steps are needed, which go under the names of GENERIC and GIMPLE. GENERIC was introduced to overcome a thorny problem: though the middle-end expects input in the form of a common intermediate language from the front-ends, it turns out that there are inconsistencies between the intermediate language that each front-end emits.

To avoid heavy intervention at the front-ends, GENERIC was written to translate trees emitted by the front-ends into a common language. Still, this is not enough: SSA acts on simple instructions; hence, lines such as

```
a = b + c*d;
```

need to be simplified as follows:

```
e = c*d;
a = b + e;
```

so that each assignment operation consists of the reduction of two variables by a single operand. This operation is known as gimplification, and the step as GIMPLE. The step which follows consists of a rewriting using SSA rules. Once the code is in the SSA form it's straightforward to implement some high-level optimisation procedures before the code is passed to RTL for further lower-level optimisations (see **Fig 1**).

Among the optimisations that have been implemented are eliminating unreachable code, constant propagation and a sketched autovectorisation. Some of those optimisations were possible within the old framework, but the new SSA scheme generally outperforms it *(for more, see Diego Novillo Q&A, below)*.

Fortran news

Although tree-SSA is without doubt the biggest addition to GCC, version 4.0 will have many other improvements that catch the eye. Among them, the addition of Fortran

DIEGO NOVILLO: *SSA MAESTRO*

Much of the buzz surrounding GCC 4.0 is being generated by the new tree-SSA infrastructure, which promises fast, language-independent optimisation. *Linux Format* talks to its creator.

CV

Based: Toronto, Canada
GCC role: Tree-SSA creator
Day job: GCC developer at Red Hat
On GCC 4.0: "We can now implement optimisations that were impossible on RTL."

LXF: How did you get involved in GCC?
DN: I am originally from Argentina and came to Canada in 1993 to do a PhD in Computer Science at the University of Alberta. I started getting involved with compilers and ended up developing techniques for analysing and optimising concurrent programs.

In 1999 I came into contact with Cygnus and started working for the GCC team. Until then I only knew about GCC by name – I had played with it a little bit during my research, but not to any serious extent. After graduation, I relocated to Toronto and kept working on GCC (now as part of Red Hat, since [Cygnus was] acquired in late 1999).

LXF: What does it mean in practical terms to be the maintainer of a branch of GCC?
DN: The work isn't much different to what you do on mainline. Perhaps the major liability is merging changes from mainline into the branch. It's a delicate balance you have to strike – if you merge too often, you are bound to make the branch too unstable, particularly if mainline is in Stage 1, ie open to major changes. If you let too much time pass between merges, you may spend quite a few hours fixing merge problems, particularly if the branch is too active, like tree-SSA used to be.

Branches are not much different to mainline In terms of contributions either. First and foremost, you have to make sure that everyone contributing to the branch has all their FSF copyright paperwork in order.

As far as stability goes, branches also operate in stages. Initially, you allow just about any change that is reasonable, and as you are getting ready to merge into mainline you start clamping down. The tree-SSA branch was pretty flexible initially, but in the months prior to the final merge, I would not allow any patch that broke bootstraps on the 5 or 6 architectures I was testing. Even if the patch was not at fault, we would remove it and ask the author to figure it out.

LXF: Can you explain what tree-SSA is?
DN: Basically, it is an overhaul of GCC's optimisation infrastructure. With it, we can now implement optimisations like vectorisation and software pipelining that were difficult or impossible to implement on RTL. It also separates the front-ends from the back- and middle-ends so that adding new languages to GCC won't be nearly impossible anymore. Before, every front-end had intimate ties with the back-end and the internal interfaces were slim or non-existent.

As with any other internal infrastructure overhaul, these major changes typically mean little to the user. But in this case, the two major visible changes will be the inclusion of Fortran 95 and mudflap [a technology for checking run-time errors]. The new optimisations will probably help some users. For instance, the new scalarisation capabilities are likely to help C++ code with lots of short-lived small objects that were demoted to memory too early in previous versions of GCC. Also, the autovectorisation passes may come in handy for some codes.

I don't expect GCC 4.0 to do the job across the board, but the new architecture will certainly help us improve and maintain it a lot better than before.

LXF: How do you see the future of tree-SSA and of GCC in general?
DN: GCC is becoming a pretty good compiler and it's quickly assimilating modern optimisation techniques that were previously only seen in commercial compilers: vectorisation, for instance. Expect several sophisticated loop transformations to start popping up in subsequent versions of GCC.

We are also starting to add intermodule optimisations – optimisations that can work across function calls and even file boundaries. Explicit concurrency in the form of OpenMP [a shared-memory API] or something along those lines is also likely in the mid- to long term. Dynamic languages like Java will also benefit from the new architecture. People will be able to implement analyses like escape analysis and devirtualisation.

LXF: Do you plan to work on other innovative projects for GCC?
DN: I'm very interested in GOMP, an implementation of OpenMP. In the short term, I'm working in several propagation optimisations to help analyses like mudflap reduce the amount of memory-bound instrumentation. I'm also interested in reducing bounds and type checking for Java.

"SOPHISTICATED LOOP TRANSFORMATIONS WILL POP UP IN NEW VERSIONS."

95 support in the form of gfortran (short for the GNU Fortran 95 project) will be welcomed by the many scientists and engineers who use this programming language – Fortran has never been one of GCC's strong points.

Gfortran (**http://gcc.gnu.org/fortran**) is a good example of the benefits of a more open development model. It was forked from the original g95 project (still under heavy development at **http://g95.sf.net**) because the maintainer of g95 liked to keep very tight control. The developers of what is now gfortran argued for tighter integration with GCC and bet on tree-SSA succeeding when it was still an experimental project. Their bravery is about to be rewarded – like any other GCC subproject, gfortran is now the property of the FSF *(for more, see Paul Brook Q&A, page 64)*.

Even at this early stage of development, gfortran has the potential to fill the gap between Fortran and the other languages supported by GCC, and has been reckoned mature enough to replace the ageing g77 front-end, although there is still some work to be done. In particular, the compatibility with Fortran 77 is still far from perfect. For this reason, Linux distro vendors are expected to provide a port of g77 alongside the new gfortran.

Symbol clearout

The slow start-up time of essential software like OpenOffice.org, *Mozilla*, KDE and Gnome is a common gripe among Linux users. With GCC 4.0 this should be greatly speeded up – provided that software developers make use of the new features. The key is the new GCC visibility patch. This offers you the possibility of deciding which ELF symbols should be exported (ELF is the format of Linux executables) and which should remain private. In older software this feature required a substantial amount of monkeying to make it work. New projects are encouraged to use visibility options right from their inception.

With a careful choice of private symbols, the loading time of a library can be sharply reduced. It also gives the added benefits of up to 20% reduction in the size of executables, better scope for the optimiser to improve the code and reduced likelihood of symbol crashing. The advantage of using the visibility features should be pondered on a case-by-case basis; however, any large C++ library making heavy use of templates is expected to benefit considerably from them. That said, it is for C++ programs only – KDE and OpenOffice.org are already taking advantage of this, but Gnome – being C-based – has not and will not.

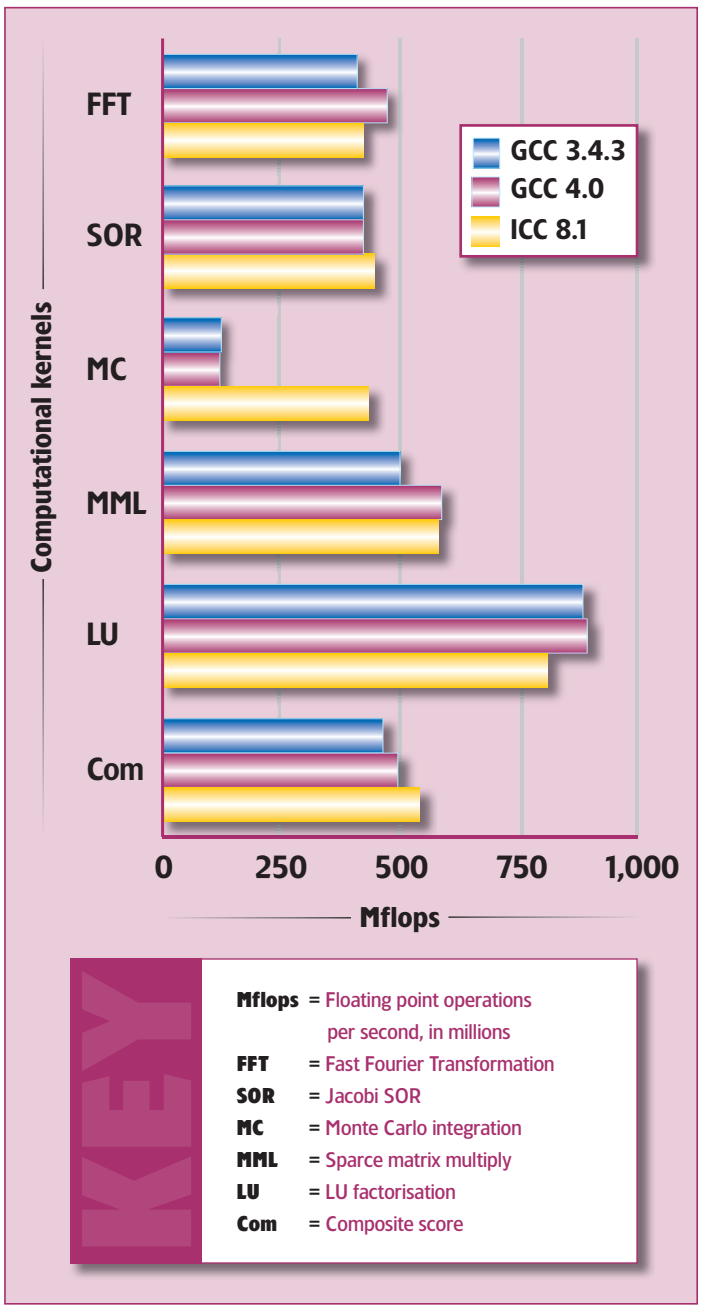
Among other improvements in version 4.0, we're excited by the (promised) much faster C++ parser, the new ABI for Java and the implementation of some mathematical functions on the IA32 and x86-64 architectures as inline intrinsics, for the benefit of number crunchers. A complete list of all the features of GCC 4.0 can be found at **http://gcc.gnu.org/gcc-4.0/changes.html**.

More speed

Of course, everyone wants a fast compiler and everyone expects a new release of a compiler to be faster than the previous one. However, there is no universal consent about the meaning of the word 'faster'. Maintainers of large software repositories for which speed is not critical would prefer a compiler that focuses on improving the compilation time, while people who deal with performance-critical software would rather benefit from shorter execution times (especially if they are buying CPU time, which is fairly likely among number crunchers).

Whatever your background, we're sure that you want benchmarks for GCC 4.0, and we are not going to disappoint you. However, the usual caveat that the only benchmark that should really matter to you is the one based on your code still applies. We should also point out that CVS versions of the compiler are very different from stable versions, even if they have the same release number, so you should take the benchmark results as a very rough estimate, with the understanding that the stable

»



3/ Floating-point performance of GCC 3.4.3, GCC 4.0 and ICC 8.1 as measured by the benchmark suite SciMark2, which was developed at the US National Institute of Standards and Technology to compare processing speeds of programs written in both C and Java.

PAUL BROOK: *FORTRAN VISIONARY*

Together with Steven Bosscher, Paul Brook made it his mission to have a Fortran 95 front-end as a part of the official GCC distribution. We asked Paul where the project's at today.

CV

Based: Yorkshire, England
GCC role: Gfortran maintainer
Day job: ARM processor work at CodeSourcery
On GCC 4.0: "It should help close the gap between gfortran and commercial compilers."

LXF: How long have you been working on GCC?
PB: I've been involved with GCC since I left university in 2002, and have been working for CodeSourcery on GCC for just over a year. I'm joint maintainer of the GCC ARM back-end and Fortran front-end, and spend most of my time working on these.

LXF: Why do you believe that GCC must support Fortran 95?
PB: Fortran is still quite widely used for computationally-intensive numerical simulations, particularly in academic institutions. It is quite common for new code to be written in Fortran 95, then combined with legacy Fortran 77 libraries.

Support for Fortran 95 is essential if GCC is to remain a viable alternative in this area. GCC's free availability and portability to a large number of hardware and OS platforms make it particularly attractive for a user wanting to develop an application on a local workstation, then migrate it to a high-performance cluster.

LXF: How did you get the idea of adding F95 support to GCC?
PB: My final year project at university involved modifying a fluid simulation code written in Fortran 95. I was frustrated by the lack of a free Fortran 95 compiler, which meant I was restricted to working on a few university machines.

After finishing university I joined the g95 project. At that time g95 could parse most Fortran 95 source, but had no real code generation capabilities. Like most recent university graduates I had quite a bit of spare time, so wrote the code to glue g95 and GCC together.

LXF: Why did you decide to fork g95?
PB: The original g95 author likes to keep very tight control of the project, ensuring that all code meets his personal standards and ways of doing things. We felt that it was important to have a more open development environment, and to work more closely with the rest of the GCC community. Our initial goal was to integrate gfortran into the main GCC CVS repository, making it part of official GCC releases.

LXF: Is there any cooperation among the two Fortran implementations of GCC? For instance, are you exchanging code for the libraries?
PB: No, not much. In practice the two projects have diverged sufficiently that most changes do not transfer easily. There has also been some difficulty obtaining up-to-date versions of the g95 source code.

LXF: What needs to be done to consider the implementation complete?
PB: Gfortran should still be considered beta quality. Most Fortran 95 language features have been implemented, and some large applications (eg the SPEC CPU2000 benchmarks) can be successfully compiled. However, there are still many bugs, and many of the language extensions supported by g77 aren't yet implemented.

I'll consider gfortran done when the few remaining corners of Fortran 95, and most of the extensions supported by g77, are working. GCC 4.0 will be the first GCC release to include gfortran. We expect that by then gfortran will be usable for many purposes, though it may not be suitable as a production compiler or as a direct replacement for g77.

LXF: Do you have any idea of how gfortran compares in terms of performance with commercial implementations such as Intel's?
PB: For Fortran 77 code gfortran should generate code that is at least as good as g77, and comparable to many commercial compilers. For some complex Fortran 95 code we generate code that is significantly slower than commercial compilers. Most of the work on gfortran is concerned with correct implementation of missing features: there's a lot of work left to do to improve performance. Having said that, gfortran uses the same optimisers as GCC and G++, so any improvements to these will benefit gfortran. GCC 4.0 will contain many new optimisations, like autovectorisation. These should help close the gap between gfortran and commercial compilers.

« version will be no worse than the experimental one.

The same applies to gfortran, which at the moment runs at about half the speed of the Intel Fortran Compiler version 8.1 in our self-developed Fortran 90 benchmark suite (we could not compare directly with GCC 3.4, since Fortran 90/95 support is a new feature of GCC 4.0).

With all this in mind, we tested the performance of the code generated by GCC 4.0 CVS with the SciMark2 benchmark suite (**http://math.nist.gov/scimark2**), designed for gauging the speed of floating-point operations, and did the same with GCC 3.4.3 and the Intel C Compiler release version 8.1. For the GNU compilers we used the optimisation flags

'gcc -O3 -funroll-loops -D__NO_MATH_INLINES -ffast-math -march=opteron -mfpmath=sse,387 -ftree-vectorize -onestep -fomit-frame-pointer -finline-functions -static'

except for the **-ftree_vectorize** option, which is specific to tree-SSA (other tree-SSA optimisation options are automatically activated by the -O3 switch). For ICC we used:

'-O3 -tpp7 -xW -ipo -align -Zp16 -static'.

Without the static option, which would have hidden the features we were interested in. The compilation time on 4.0 was on average about 10% slower than on 3.4.3, and the size of the executable was about 2% larger. The generated code was then executed on a dual AMD Opteron 244 processor machine with 4GB of RAM. Measured performances are plotted in **Fig 3** (for details about the various tests, refer to the home page of the benchmarks).

GCC 4.0 overperforms its predecessor in most tests, often by a wide margin. Even more excitingly, GCC 4.0 now runs neck and neck with the Intel compiler, and outperforms it by a significant margin in at least two tests. Still, at the moment a tedious optimisation bug (a wrong move of floating-point variables through integer registers) affects the performance of GCC 4.0. As this bug will be fixed before the official release, expect the official version to perform much better than in our tests. We don't expect you to have a dual Opteron on your desks, so we repeated the tests on a Pentium IV 1.7 GHz with 768 MB of RAM, which threw up roughly the same results.

The tests confirmed our hopes that GCC 4.0 will be a great release. But the GCC developers have no time to bask in the glory, since they are already working on new features and additions. GCC still lags behind commercial competitors in the high-performance computing market, and we expect this gap to be filled pretty soon. The GOMP project (**http://gcc.gnu.org/projects/gomp**), aimed at providing support for the powerful OpenMP parallel instruction extensions, is an initial step in that direction. LXF

ACKNOWLEDGEMENTS

Thanks to Vladimir Marakov, Paolo Bonzini, Uros Bizjak and especially Richard Guenther for discussing optimisation flags in GCC 4.0.