USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000

GCC 3.0: The State of the Source

Mark Mitchell Alexander Samuel
CodeSourcery, LLC
mark@codesourcery.com samuel@codesourcery.com

August 24, 2000

1 Introduction

The GNU Compiler Collection (GCC) is the most fundamental component of the GNU/Linux developer's toolchest. GCC, like the Linux kernel and the X windowing system, is a complex but important part of the GNU/Linux operating system. In fact, both the kernel and X are built with GCC, so, to a large extent, the speed and correctness of the entire system depends on GCC.

The next major release of GCC, GCC 3.0, will be released sometime late this year. This release of GCC will contain a number of features of considerable import to the community. In addition, the quality assurance processes for this release will be more stringent than in any previous release. Therefore, GCC 3.0 will likely be a more reliable compiler, and will be more capable of supporting the needs of an ever-expanding developer community, than previous releases of GCC.

Principal among the improvements will be a new, stable, industry-standard C++ application binary interface (ABI). This new C++ ABI will provide, for the first time, an assurance that C++ programs compiled with one version of G++ can be linked with C++ libraries compiled with a different version of G++, and, in fact, with libraries compiled with other compilers.

In addition, GCC 3.0 will contain much improved support for Java, a number of new optimizations and bug fixes, improvements in compile-time performance, and new infrastructure to support future optimization and enhancement.

Because much of our work has focused on the C++ front-end, this paper will focus on C++-related work. That orientation should not be construed as implying that the other improvements in GCC 3.0 are less important. While a perhaps disproportionate amount of this paper focuses on the C++ ABI, we have attempted to describe, in somewhat less detail, some of the other highlights of the upcoming release.

2 General improvements

Significant efforts have been made during this release cycle to make GCC easier to maintain and improve in the future. Although these infrastructure investments do not have immediate user-visible benefits, their import cannot be overestimated. A well-organized, well-designed compiler means better compilers down the road.

2.1 Runtime support library

GCC provides certain support functions (like support for arithmetic on numbers with more bits than are supported on the target hardware, and support for exception-handling) in a special library, traditionally called `libgcc.a`. On non-GNU/Linux systems, this kind of compiler support functionality is typically provided in the system C library. That is a workable solution if the same vendor produces both the C library and the compiler. When GCC runs on non-GNU systems, however, there is no way to control the contents of the C library, and GCC must therefore provide its own runtime support library.

Unfortunately, `libgcc.a` contains some global data. It is important that there be only one copy of this global data in a complete program. However, in earlier versions of GCC, `libgcc.a` was linked into every shared library created by GCC. That could result in multiple copies of the global data, and incompati-

ble support routines, in a complete executable, with the result that linking together shared libraries compiled with different versions of GCC did not always work.

In GCC 3.0, these problems will be solved by providing `libgcc` as a shared library. Libraries and executables will be linked against this shared library, which should prevent the kinds of incompatibilities described above.

2.2 Memory management

The memory management scheme used by the compiler itself was radically altered for the GCC 3.0 release. Memory allocated by the compiler is now garbage collected; previous releases used a complex system of memory pools. This change greatly reduced the number of memory-allocation bugs in the compiler, and simplified the implementation of new features.

Use of garbage collection, and other associated improvements associated with memory management, have dramatically reduced the memory footprint of the compiler in some cases. There have been improvements as great as 60% (from approximate 300 MB to approximately 100MB) when compiling some C++ programs.

2.3 Parsing whole functions

In early 2000, we converted the C++ front-end to produce a parse-tree for an entire function. (Previously, parse-trees were only available for individual statements.) G++ can now parse all of the code for a function before committing to the code shape for the function. This change has already lead to significant developments. The C++ front-end takes advantage of this representation to perform function inlining at a higher level, which allows more functions to be inlined with less memory usage. Future optimizations will take advantage of the fact that the parse tree for the complete function is available. For example, the "scatter-gather" optimization whereby a structure with several scalar members is treated simply as a collection of scalars can be implemented using the new representation.

Furthermore, the availability of a parsed representation of the entire program makes possible entirely new tools. A research group at Stanford is making use of the new representation to perform domain-specific error checking; they have, for example, detected bugs in the Linux kernel that could result in deadlocks. This analysis is only possible because the compiler is able to provide the error-checking tool with a representation of the entire program.

In addition, SGI was able to connect the G++ front-end to their IA64 back-end with relative ease by taking advantage of the new representation. They simply translated G++'s high-level representation for the program into a format understood by the SGI optimizers. In the future, it is likely that source browsers and other similar tools will also take advantage of these features. In fact, there has even been discussion of providing a plug-in interface that would allow the insertion of domain-specific optimization passes.

In the near future, we will convert the C front-end to use the same representation as the C++ front-end, which will yield all of the improvements listed above for C as well as C++.

2.4 Using the flow graph

Richard Henderson and others have made dramatic improvements to the optimization framework used by GCC. In particular, Richard's work allows GCC's optimization passes to access the flow graph for the function in a convenient way. (The flow graph shows where branches and loops can occur in the function.) Optimizations based on the flow graph are so-called "global" optimizations. The use of these techniques will allow the compiler to perform much more comprehensive optimizations than was previously possible.

GCC contains some experimental code to translate to and from single static assignment (SSA) form. SSA form is a means of representing the program that allows the compiler to perform many optimizations more easily. A compiler that takes advantage of SSA form can generate better code, and it can generate that code faster. In the future, we hope to convert many of GCC's high-level optimization passes to use this framework.

2.5 New optimizations

GCC 3.0 contains some powerful new optimizations. A new basic block reordering pass reorders generated code to improve cache performance based on either estimates of branch probabilities, or using output from profiled execution of the code. New exception-handling optimizations take advantage of throw specifications to eliminate unneeded exception handlers.

Perhaps most importantly, GCC 3.0 will contain a ground-up rewrite of the x86 back-end. Since so many GNU/Linux users run on x86, improvements in this area are clearly very important. The new back end describes the architecture much more accurately, and thereby allows GCC to generate considerably better code. There are also new optimizations targeting AMD's Athlon processor as well Intel's Pentium II and Pentium III processors.

2.6 Java

GCC 3.0 will include the GCJ Java compiler and the libgcj runtime support libraries. GCJ compiles Java code directly to native code just as the C, C++, and Fortran compilers do. In fact, GCJ uses the same object layout for Java objects that is used by GCC for C++ objects, so it is relatively easy to write programs that consist partly of C++ code and partly of Java code.

GCJ supports most Java language features present in version 1.1 of the JDK. As of yet, the library does not support many of the popular Java APIs available from Sun, but work is underway on the implementation of these APIs.

2.7 Bug tracking

The GCC project now makes use of the GNATS bug-tracking tool. That tool has made it a lot easier for users to report bug reports and for the GCC maintainers to respond to them. Geoff Keating set up an automated regression-testing tool that informs contributors when changes cause regressions in the GCC test suite. Increased reliance these automated tools will make it easier to make high-quality GCC releases.

3 C++ improvements

In recent releases of GCC, such as the GCC 2.95 release, GCC's support for modern C++ programming has increased dramatically. Our contributions have brought considerable improvements in ANSI/ISO conformance, increased the robustness of the compiler considerably, improved compile-time performance, introduced new optimizations, and improved error-reporting. The GCC 2.95 release was the first to compile complex expression-template programs, such as programs using the Blitz numerical programming library.

However, there has not been a stable C++ ABI. The ABI includes decisions made by the compiler as to how big objects will be, where data members will be located within an object, and the interfaces to run-time support functionality like exception-handling and run-time type identification. If the ABI changes between two versions of a compiler, then libraries created with the first version of the compiler cannot be linked with code compiled with the second version, and vice versa.

Frequent changes in the ABI therefore prevent people from distributing binary versions of C++ libraries. Distributors of free or open-source libraries suffer since it is harder for users to easily download the library. (Just because the source is available does not mean that users want to actually compile the source! The success of the various GNU/Linux distributions is proof of this fact; users are happy to be able to easily install the system without having to compile everything themselves.) Proprietary vendors suffer to an even greater degree; they do not wish to distribute source code, and they cannot easily provide object code that will work smoothly with various versions of the compiler.

The C++ ABI has changed frequently between previous GCC releases for a variety of reasons. To some extent, it has changed accidentally; changes to the compiler can result in inadvertent changes to the ABI. Until now, there have been no tests in the GCC test-suite that specifically test the C++ ABI, and for that reason it has been difficult to verify that changes to the compiler did not alter the ABI. The ABI has also changed out of necessity; as additional features mandated by the ANSI/ISO C++ standard have been implemented, ABI changes have been required in some cases.

The GCC maintainers have recognized the value in a stable C++ ABI for some time, and had in fact begun to work on implementing a new ABI. Fortuitously, a number of UNIX vendors and other interested parties joined forces to develop a C++ ABI for Intel's new IA-64 architecture. Their goal was to enable programs compiled with one compiler to link with libraries compiled with another compiler, or even to run a program compiled for one IA-64 operating system on another IA-64 operating system. (For example, a program compiled for the Monterey operating system being developed by IBM and SCO could be run on GNU/Linux, provided that the necessary support libraries on both systems adhere to the specified ABI.)

We participated in the effort to formulate the IA64 C++ ABI, and have now completed implementations of the ABI in G++ and in other compilers. Although the new ABI has only been formally specified for the IA-64 architecture, the ABI has been generalized to handle other architectures as well, since most of the design choices are equally applicable. So GCC 3.0, and subsequent releases of GCC, will use this same ABI on all architectures. The new ABI should greatly increase the ability of vendors to ship libraries that can be linked with versions of GCC that are released after the library was created.

In fairness, it is likely that some bugs remain in the GCC 3.0 ABI implementation. These bugs might necessitate minor changes in the future, but we expect that any such changes will not impact most programs, and should be resolved in the relatively near future.

3.1 Benefits of the new ABI

The new ABI has many benefits in addition to the stability that it will bring. In particular, the ABI committee worked hard to reduce the performance penalties long associated with some C++ features. Some programmers have avoided C++ altogether because of perceived performance problems. Others have formulated coding standards that prohibit the use of virtual functions, virtual base classes, or exception-handling in order to maximize performance. The committee took the implicit criticisms seriously, and attempted to design an ABI that would minimize the costs associated with these features. Many of the design decisions were based on existing practice in compilers available from EDG, HP, IBM, SGI, Sun and other vendor. Therefore, credit for these designs should be attributed not to the committee, but to the original designers of these optimizations.

Some of the changes in the new ABI are necessary simply to ensure correctness. For example, C++ has complex rules involving how virtual function calls should be dispatched during object construction and destruction. These rules could not be implemented using the G++ ABI on Linux, with the result that the GCC maintainers received frequent bug reports about which nothing could be done. The new ABI also contains a number of performance improvements, including dramatically shorter mangled names, reduced memory usage, and faster virtual function calls.

3.1.1 Mangled names

Many users reported mangled names of several kilobytes; in most cases the new ABI will reduce the length of these names to tens of characters. That change will reduce the size of object files, and make linking faster.

Much of the improvement stems from the observation that type signatures for template functions (which typically have the longest mangled names), often involve the same types. For example, the full signature for for `vector<vector<int> >` is:

```
vector<
  vector<int, std::allocator<int> >,
  std::allocator<std::vector<int,
                 std::allocator<int> > > >
::vector(void)
```

Here, the templates `vector` and `allocator` are repeated several times. If the element type was itself a template type, then there could be even more repetition.

Using the old ABI, the mangled name for this constructor is the following 173-character string:

```
__Q23stdt6vector2ZQ23stdt6vector2ZiZ
  Q23stdt9allocator1ZiZQ23stdt9allocator1Z
  Q23stdt6vector2ZiZQ23stdt9allocator1ZiRC
  Q23stdt9allocator1ZQ23stdt6vector2ZiZ
  Q23stdt9allocator1Zi
```

Using the new ABI, this same function is mangled as the more reasonable 39-character string:

```
_ZNSt6vectorIS_IiSaIiEESaIS1_EEC1ERKS2_
```

These compression techniques make for even greater savings when using more complex templates, such as those found in complex expression-template libraries. Part of the reason for the savings in this example is that the new mangling scheme also contains special abbreviations for some of the names in the standard library, including `std::string`. Since so many functions take parameters that involve these types, these abbreviations are very valuable.

3.1.2 Constructing virtual bases

The new ABI reduces the penalty for using virtual bases in several ways. Consider a constructor for a class that has virtual base classes. That constructor must decide whether or not to call the constructor for the virtual base subobject. The constructor for the virtual base class should not be called more than once for any single object. In the old ABI, the constructor took an additional parameter that told it whether or not to construct the virtual base classes. When a complete object was created, its constructor was called with the parameter set to a non-zero value. This value indicated that virtual base classes should be constructed. When the constructor for the complete object called base class constructors, the extra parameter was set to zero to indicate that constructors for virtual base classes should not be run again.

Unfortunately, setting up the parameters, and then checking their values, can take a substantial amount of time. The new ABI defines two different entry points for each constructor: one that constructs virtual bases, and one that does not. These may be alternate entry points into the same function, or they may be entirely separate functions; the choice is up to the compiler. Using alternate entry points, rather than a parameter, to distinguish between the constructors eliminates the run-time overhead of passing and checking the parameters.

3.1.3 Laying out virtual bases

As another example of the sorts of improvements provided by the new ABI, we consider the way in which virtual base classes are laid out. Virtual base classes present a problem in that their location, relative to the derived class that contains them, is not known when the program is compiled. For example, consider this fragment:

```
class V {
public:
  virtual void f();
  int i;
};

class D1 : virtual public V {
public:
  double d1;
};

void f (D1* d1) { d1->i = 3; }
```

The location of `i`, relative to `d1`, depends on the dynamic type of the object pointed to by `d1`. That this is the case stems from the fact that `V` is a virtual base; there is only one copy of `V` even in multiple classes in a hierarchy derived from `V`.

Traditionally, G++ (following the lead of the original AT&T implementation of C++), handled virtual bases by creating, in the derived class, a pointer to the virtual base. When `D1` is used as a base class, it is laid out like the following C structure:

```
struct D1 {
  double d1;
  V* vbase;
};
```

A complete object of type `D1` looks like:

```
struct D1_Complete {
  double d1;
  V* vbase;
  V v;
};
```

The `v` field is filled in with the address of the virtual base `v`. So, the expression `d1->i = 3` would have been implemented as `d1->vbase->i = 3`.

Now, consider what happens if several classes derived from `V`. For example, suppose we extend our example with:

```
class D2 : virtual public V { double d2; };
class D : public D1, public D2 {};
```

The layout for a complete object of type `D` would look like:

```
struct D_Complete {
  D1 d1base;
  D2 d2base;
  V vbase;
};
```

Note that both `D1` and `D2` contain pointers to `V`. Using this scheme, a complex hierarchy with many virtual bases, a large percentage of the space consumed by an object can end up devoted to pointers to virtual bases. The old ABI squandered not only space (because of all the pointers), but also time: initializing all of the pointers makes object construction and destruction unnecessarily costly.

In practice, hierarchies making use of virtual bases almost always make use of virtual functions. Therefore, each object already contains a "vtable pointer", i.e., a pointer to a virtual function table. Dynamic dispatch is implemented using the virtual function table; when the *i*th virtual function in a class is called, control is transferred to the address given by the *i*th entry in the virtual function table. In addition to the virtual function addresses, the new ABI stores offsets from derived classes to their virtual bases in the virtual function table. This mechanism eliminates the need to store pointers to virtual bases in the objects themselves, thereby eliminating both the space penalty, and the time penalty for initializing the objects.

3.2 C++ standard library

The implementation of the C++ standard library that shipped with GCC 2.95.2 and previous releases was woefully out-of-date. It provided no support for wide characters or locales, did not place names in the `std` namespace, did not support templatized I/O streams, and contained numerous other deficiencies.

Benjamin Kosnik and others have worked very hard over the last several years on a ground-up rewrite of the library. GCC 3.0 will be the first GCC release to use the new standard library, which is now mostly complete. Already, the deficiencies mentioned above have been repaired, and users will find that the new library conforms much more closely to the ANSI/ISO C++ standard.

3.3 C/C++ preprocessor

GCC 3.0 will include a new implementation of the C preprocessor, contributed by Zack Weinberg and others. The new preprocessor is faster than the old preprocessor. More importantly, the new preprocessor will allow direct integration with the compiler front-ends themselves. Presently, GCC first creates a temporary file containing the preprocessed source file. The front-ends then process this preprocessed file. The overhead of writing, and then reading, the temporary file is considerable.

In addition, the process of tokenization is needlessly duplicated between the preprocessor and the compiler front-ends. For example, when confronted with the string `f (71)` the preprocessor determines that the token stream consists of an identifier, an opening parenthesis, a numeric literal, and a closing parenthesis. The compiler must perform the same analysis.

The new preprocessor will be able to connect directly to the compiler front-ends, without using an intermediate file. In this way, the overhead of reading and writing the file will be eliminated, as will the redundant retokenization. In this way, the compile-time performance of GCC should be improved substantially. Initial measurements show as much as a 10% speedup when compiling some programs with the new preprocessor.

4 Future directions

The GCC community has already started talking about what will happen after GCC 3.0. In addition to the usual improvements, we are hoping to incorporate two new languages into the GNU Compiler Collection: Pascal and ADA. Unifying the development groups that have been working on these front-

ends with the core GCC development time should make it much easier for users of Pascal and ADA to obtain compilers that interoperate smoothly with the rest of GCC. On the other hand, since there has been little interest in Chill, it is possible that the Chill front-end will be dropped from GCC at some point in the future. The C and C++ front-ends may be combined into a single front-end in order to reduce code duplication and to make it easier to achieve consistent semantics between C and C++.

There have been serious discussions about making relatively major changes in the internal data structures used by GCC in order to offer new optimization opportunities. Presently, GCC uses two major representations of the source program: an abstract syntax tree representation that is close to the source language, and a register transfer language that is close to the eventual generated machine code. Unfortunately, there is no convenient intermediate representation: one which is simpler than the source language, but that still abstracts away from the machine representation.

This "representation gap" makes it difficult to implement many high-level optimizations in GCC. For example, it is difficult to implement many loop optimizations. Some of the algorithms that are implemented in GCC suffer both in implementation complexity and in the quality of the generated code by having to work on inconvenient representations of the program. Therefore, it is likely that structural changes will be made to accommodate a new representation.

5 Challenges

GCC has a lot going for it. It is a free, easily retargetable optimizing compiler supporting the most important available compiled programming languages. Unfortunately, there are some notable weaknesses in GCC as well. Presently, GCC's optimization is not as good as that provided by some commercial compilers. The error messages produced by GCC are not as helpful as they should be. Despite major improvements, there are still weaknesses in support for some language features in C++, Java, and Fortran. The compile-time performance of the compiler is not as good as that of the best commercial compilers. The documentation provided is not as good as it could be. There are substantial weaknesses in the supporting tools, including debuggers and profilers. Other important tools, like incremental linkers, source browsers, and integrated development environments, do not exist, are not of commercial quality, or not are not freely available. Quality assurance between releases has not been sufficient to ensure that users can easily upgrade from one version of the compiler to the next.

For these reasons, GCC will face threats in the future from commercial compilers. The success of GNU/Linux represents a new market for tools vendors; already, for example, there are two proprietary compiler products available for GNU/Linux. If these proprietary products are better than GCC, then many developers will use them. Many companies will find the cost of these products worth the price if they feel that these tools will enable faster, easier development.

Therefore, the community should continue to invest in GCC, by donating development effort, by providing hardware resources, and by funding future improvements in GCC. That up-front investment will ensure that GCC continues to improve, and that it remain the best available compiler for the GNU/Linux operating system.