Contributions to the GNU Compiler Collection

D. Edelsohn
W. Gellerich
M. Hagog
D. Naishlos
M. Namolaru
E. Pasch
H. Penner
U. Weigand
A. Zaks

The GCC (GNU Compiler Collection) project of the Free Software Foundation has resulted in one of the most widespread compilers in use today that is capable of generating code for a variety of platforms. Since 1987, many volunteers from academia and the private sector have been working to continuously improve the functionality and quality of GCC. Some of the compiler's key components were, and continue to be, developed at IBM Research laboratories. We review several of IBM's contributions to the compiler, including a code generator for the IBM zSeries® processor and a front end for a PL/I-like language used for systems software programming. We also cover many optimizations, including the interblock instruction scheduler, software pipeliner, and vectorizer. These contributions help improve the overall performance of code generated by GCC, and in particular, enhance the IBM RISC (reduced instruction set computer) architecture and the zSeries processors. This paper includes a report on our general experience with GCC in both open source and proprietary software environments and reviews the quality and performance of GCC-generated code.

The GNU Compiler Collection (GCC) is an optimizing compiler for the GNU project that is capable of generating code for a variety of platforms and that supports a number of languages, computer architectures, and operating systems.[1–3] It is one of the most visible aspects of the Free Software Foundation (FSF) GNU project. The goal of the GNU Project is to create a UNIX**-style operating system composed of free software.

The GCC compiler can be configured to generate code for more than 30 different computer architectures. Many architecture configurations are designed to support multiple operating systems, including the GNU system and GNU/Linux**. The primary set of languages available with the compiler are C, C++, Fortran, Java**, Objective C, and Ada. Runtime libraries for the languages are also included in the compiler suite. Support for additional languages is currently in various stages of development.

The GNU Project includes an assembler, linker, and other object file tools, commonly called "binutils," the GDB debugger, and glibc (GNU C library).

Together, these components provide a software development environment or "tool chain."

GCC structure

GCC was one of the first components of the GNU Project. Richard Stallman initially tried to extend the Pastel compiler, developed by the

■ GCC evolved through the efforts of a worldwide group of developers, including members of industry and academia and independent consultants ■

Lawrence Livermore National Laboratory, but needed to rewrite the compiler from scratch due to technical limitations of the Pastel compiler.

The compiler was initially targeted at the common microprocessors of the late 1980s, such as the Motorola 68000, and was ported to other CISC (complex instruction set computer) processors, such as the Intel 80386. GCC initially parsed source code one statement at a time, focusing on local optimizations. One of the important optimizing phases from the earliest versions of GCC is a phase called combine that operates as a generalized peephole optimizer, reducing multiple instructions into single, more powerful instructions (see "GNU back end").

Recent improvements have expanded the compiler's view of the program to focus on one function at a time, the translation unit, or the whole program. These changes allow more aggressive optimizations, including inter-procedural analysis. Other recent enhancements include the addition of a Static Single Assignment (SSA) design with basic SSA-based scalar optimizations, high-level loop transformations, and vectorization.

The compiler phases for GCC 4.0 first parse the input program into an intermediate representation called GENERIC. GENERIC is expanded and lowered into an SSA form called GIMPLE. The compiler optimizes the SSA form and then removes the SSA names. The program statements are translated to a different intermediate language called register transfer language (RTL), which directly corresponds to the instruction set of the target processor (i.e., the "target instruction set"). RTL optimizations that require details about the target processor instructions are applied, such as instruction scheduling, software pipelining, and register allocation.

RTL is designed to correspond to valid target instructions. The RTL instruction codes themselves are independent of the target, but the subset of codes used for each target match the machine instructions of the target. Other than missing register numbers and memory offsets, RTL transformations are intended to convert a valid instruction stream into another valid instruction stream (i.e., sequence of instructions).

After all optimizations have been applied, the RTL instruction stream is output as a file in assembly language appropriate for the target system. GCC does not have an integrated assembler and does not generate an object file directly for any target. An external assembler, possibly the GNU Assembler, creates an object file, and an external linker, possibly the GNU Linker, binds the executable or shared object. The operating system may use the GNU C Library to provide an interface to system services.

The GCC compiler is written in the C language, and the source code is composed of files common to all targets and files with specific information about the target architecture, target system, and target file format—the latter referenced as the machine description. Some of the files in the machine description affect the way the common parts of the compiler behave (e.g., the size of data types, size of registers, register allocation order, etc.). Other files are used by programs within the GCC build process to create machine-generated files that interface with the common parts of GCC to describe the target instructions and output format.

GCC development

GCC evolved through the efforts of a worldwide group of developers, including members of industry, academia, and independent consultants. As with many other free-software and open-source projects, the hierarchy of developers strives to achieve a meritocracy. A core set of developers provides most of the technical leadership, and a steering committee provides the political leadership and interface to the FSF.

Developers collaborate in a decentralized fashion with informal collaboration, setting design goals and avoiding duplicated effort. The majority of communications and technical decisions occur in public forums such as mailing lists and chat rooms. The GCC source code is available in revision control systems on publicly accessible servers.

All GCC developers are required to have copyright assignments on file with the FSF. After that documentation is on file, changes offered by a developer for inclusion in GCC can be considered. Patches are mailed to public mailing lists and reviewed for coding style, design, and implementation correctness by senior developers with authority to approve patches for various components. Documentation for the GCC project explains the development plan and other criteria of the project. The coding style follows the GNU coding conventions and GCC extensions.

GCC includes an extensive and growing test suite to help maintain the quality of the compiler. All patches are supposed to be tested with the complete test suite, and authors are expected to certify that a proposed patch did not generate any new test suite failures.

To maintain the quality criteria for GCC, releases should create no test suite regressions on important target platforms. Because of the large number of GCC targets (architectures, operating systems, file formats, etc.), some regressions do occur. The lack of complete coverage testing and unit testing in the current design is one of the major limitations in the GCC testing procedures.

GCC legal issues

Free software, a concept originated by Richard Stallman to encompass the GNU Project, refers to the freedom of users and developers to use, modify, redistribute, and distribute modified versions of the software.[4] Free software commonly refers to software distributed under the terms of one of the GNU General Public Licenses (GPL). Open-source software refers to a broader set of possible licenses.

Although the GPL applies to the GCC and the GNU tool chain, building an application using the GCC does not affect the software license of the application itself. Proprietary applications can be built using GCC.

Use of GCC

Use of GCC has become pervasive throughout the software industry because of its flexibility. It is able to generate applications for many proprietary and open-source UNIX** operating systems, as well as OpenVMS**, z/OS,* Microsoft Windows**, VxWorks**, and others.

GCC has been available for AIX* on the POWER* platform and MVS* on the S/390* platform for over ten years. In addition to its use by IBM customers on AIX and in software enablement for embedded processors, GCC has also been used for many research projects and prototypes; for example, experimental work with the PowerPC* instruction set and the 64-bit XCOFF file format.

Customers frequently use GCC instead of proprietary compilers because of its portability. GCC itself provides language extensions, but the extensions are consistent across all systems; therefore, customers do not have to worry that they will use a compiler feature that locks them into a particular system. The GPL ensures that the customer always has access to the source code of the compiler and libraries to perform any development or maintenance. A customer's decision to use GCC often depends on a few primary factors, including performance, portability, and service.

Overview

In this paper, we describe several of our contributions to GCC. IBM has made additions to GCC which encompass all phases of the compiler—the front end, optimizations in tree and RTL intermediate representations, and the back end. The specific details of each contribution are outside the scope of this paper; the interested reader is referred to the actual code and documentation, which is freely available at http://gcc.gnu.org. This paper does not cover all contributions to GCC made by IBM developers, but rather describes some projects in an attempt to focus on our experience with GCC and its limitations and potential. In the following sections we describe a new front end, some optimizations, and a new back end.

PL8 FRONT END

This subsection describes the development of firmware for the PL8 and IBM zSeries* systems, and the technical issues arising from this effort.

PL8 and IBM zSeries firmware development

The term "firmware" refers to the software layer between hardware and the operating system. Firmware functionality includes I/O path management,

■ GCC includes an extensive and growing test suite to help maintain the quality of the compiler ■

I/O load balancing, recovery from hardware and firmware errors, and some system management functions, which, in other computer systems, are typically implemented in operating-system layers.

Firmware development requires low-level programming, as firmware has many interfaces to hardware registers and to assembler-written routines. The firmware implements low-level services that require accessing specific addresses and dealing with individual bits or words smaller than a byte. PL8, which basically is a subset of PL/I, supports these requirements by use of appropriate declarations, which is considered a strength of the language.[5] The language has been used for firmware development since the early 1980s with an old compiler that has not been maintained for years. However, there have been significant enhancements made to the zSeries architecture, including additions to its instruction set, improved pipeline structures, and an extension to 64 bits. Some firmware internal structures were strongly geared to 64-bit implementation, which the original PL8 compiler could not provide. The original compiler was also inherently tied to the library and build environment on VM/CMS as its only execution platform.

PL8-front-end technical issues

Given GCC's modular structure and the fact that GCC already had a back end generating S/390 code (see "The zSeries back end" and Reference 6), an obvious approach was to implement PL8 again as an additional GCC front end. The language was extended to support 64-bit data types, and its rules concerning memory layout were adapted. The GCC framework also suggested a few language modifications.

In contrast to most other GCC front ends, the PL8 front end is well-suited for two-pass compiling. This is because PL8 allows forward references to declarations. The two-pass approach also simplifies certain other translations. The first pass does lexical and syntactic analysis, which is implemented using the compiler-generating tools Flex and Bison, respectively. Its output is a front-end internal representation of the input program which is an attributed syntax tree.[7] Tree nodes are implemented as records with fields containing data or pointers to other tree nodes. Whenever possible, the GCC predefined tree nodes are used to represent PL8 constructs. For example, this is done for if, do while, and do until statements. More elaborate statements, such as select and PL8 counting loops, have no direct correspondence to any existing nodes; they are thus first translated into front-end specific nodes, as are most of PL8's declarations, namely the attributes based, offset, and redefines.

In pass two the compiler starts working on the data structures generated by pass one and does a few semantic checks. In this pass the compiler also does some optimizations. These include type compatibility checks to verify that variables are assignable. Implicit type conversions are inserted where the PL8 language definition allows the assignment of variables with different types. Range checks are generated for array accesses, and for all accesses to based variables through offsets. Pass two also carries out some optimizations, such as constant folding[8] and an elimination of range checks, which deletes a check if it can determine at compile time that an index will never be out of the valid range.

Finally, the PL8-specific nodes are translated into GCC-defined tree nodes and are passed to the GCC "middle end" (i.e., second phase).[5] The PL8 front end is approximately 50 KLOCs (thousand lines of code) in size.

Compiler validation

Validation for the new front end was performed in several steps. First, a regression test package with almost 3500 test cases was run automatically. It consisted of test cases systematically developed by experienced PL8 programmers, test cases used for the original PL8 compiler, and test cases derived from compiler problems. The second step was to run all zSeries firmware test cases, using a stable firmware version, with the new compiler. The front-end sources were also subject to a formal code

review and were analyzed by a static code-checking tool.

During final system test, only five compiler problems were found. So far, no field problems are attributed to the PL8 compiler.

TREE OPTIMIZATIONS

The GCC uses different internal representations (IRs) at different phases of compilation. The tree representation is a language-independent and machine-independent IR that preserves high-level language constructs, a property which makes it suitable for a range of compiler optimizations. Until recently, however, almost all optimizations in the GCC took place at a lower level IR—the RTL. This situation is gradually changing since the recent introduction of the new tree-SSA framework.[9] This framework includes further simplification of the tree IR into a three-address language (GIMPLE), and an implementation of SSA on top of it.[10]

The introduction of tree-SSA simplifies and encourages the development of many optimizations and analyses, thereby providing the required infrastructure for the development of a vectorizer in the GCC. On January 1, 2004, we submitted the first implementation of a basic vectorizer to the GCC, based on tree-SSA utilities,[11] and it is now part of the GCC mainline version 4.0. Additional capabilities are constantly being developed on the "loop-nest-optimizations" branch.[12] In this section, we describe our work on the GCC vectorizer, and in particular, the issues that arise due to the multi-platform nature of the GCC.

Vectorization

To take advantage of vector hardware such as AltiVec** and MMX**/SSE[13,14] (multimedia or streaming SIMD [single instruction, multiple data] extensions to general-purpose instruction set architectures), programs can be written using explicit vector operations (e.g., using Altivec intrinsics[15] or the Fortran90 operations on whole arrays). These vector operations work on multiple elements in parallel, in contrast to the standard scalar operations that operate on individual elements, one after the other. The transformation of these scalar operations into an equivalent vector form is referred to as vectorization[16] and can be applied manually or automatically by the compiler.

Opportunities for applying vectorization are usually found in loops, where operations from different iterations can execute in parallel (exploiting data parallelism across loop iterations). Applications in many domains have an abundance of natural parallelism in the computations they perform. If this parallelism can be leveraged to exploit the vector capabilities of the target architecture, the performance of these applications can be considerably improved. The level of parallelism that can be implemented depends on the size of the vectors supported by the target and the size of the data types operated upon in the application. In AltiVec, the vector size is 128 bits, which can accommodate four floating-point numbers, four integers, eight "shorts," or 16 characters. We refer to the number of elements that can be operated upon simultaneously as the "vectorization factor."

■ Use of GCC has become pervasive throughout the software industry due to its flexibility ■

The importance of automatic vectorization has increased in recent years, with the introduction of SIMD extensions to general-purpose processors, and with the growing significance of applications that can benefit from it. SIMD introduced some new difficulties for vectorizing compilers,[17] which are especially challenging in the context of GCC, as discussed next.

Vectorization components

Research into the area of vectorization is already quite mature.[16,18,19] The main focus of classic vectorization is the use of data dependencies and loop analyses to: (1) detect statements that can be executed in parallel without violating the semantics of the program, and (2) increase such occurrences by means of loop transformations.

While the analyses above deal with proving the theoretical correctness of applying vectorization, most other analyses and transformations employed by the vectorizer are low level and deal with machine-dependent cost and trade-off analysis, rather than general properties of the code itself. This is because the specific characteristics of the avail-

able architectural vector support can directly affect the vectorization transformation, and even determine whether it should be applied at all.

The vectorizer starts with a set of loop-level analyses, including analysis of data dependencies, data-access patterns, data alignment, loop-exit conditions, and analysis to determine if all the operations in a loop have a vector form supported on the target platform. This information is modeled through the machine model files. For simple generic operations, it is easy to query (even at the machine-independent tree-level IR) whether the operation is supported. However, this information is not so easily accessible for operations that do not have an equivalent scalar form (such as data permutations, reduction, and unaligned accesses). In these cases, we have to enhance the infrastructure to represent this information to the vectorizer.

■ The natural parallelism of many applications can be leveraged to exploit the capabilities of the target architecture and enhance performance ■

For loops that successfully pass the analysis stage, the vectorizer applies the actual vector transformation. This consists of "strip mining" the loop[20] by the vectorization factor and then replacing each scalar operation in the loop by its vector counterpart (using the machine model files). In many cases additional handling beyond the one-to-one substitution of statements is required. Constants and loop invariants require that vectors be initialized before the loop; other computations, such as reduction, require special "epilogue code" after the loop, and some operations (unaligned accesses, type conversion) require special data manipulation to take place between vectors.

The machine-dependent components of the vectorizer are mostly related to memory architecture limitations of vector machines. The memory architecture usually restricts vector data accesses to consecutive vector-size elements, aligned on vector-size boundaries. Gathering data from nonconsecutive or unaligned locations requires special mechanisms for data reorganization, which are costly and hard to use. These issues are especially true for SIMD systems because SIMD memory architectures are typically weaker than those of traditional vector machines. Moreover, SIMD architectures tend to be very different from one another, a fact that can be problematic for a vectorizer operating at a high-level machine-independent IR in a multiplatform compiler such as GCC. Some of these problems are elaborated next.

GCC implementation issues

The aspects of vectorization discussed earlier demand that low-level architecture-specific factors be considered throughout the process of vectorization. However, at the tree-level IR, it is not trivial to express low-level target-specific mechanisms, such as those that are used to reorganize unaligned data or pack or unpack data between vectors of different data types. These mechanisms need to be introduced into a high-level platform-independent tree IR, while allowing low-level platform-specific details to be hidden as much as possible, to be applicable to any platform, and to be as efficient as possible on each platform.

These properties are even more difficult to tackle in a multiplatform compiler such as GCC, due to the tendency of SIMD instruction-set architectures to be much less general-purpose and less uniform than traditional vector machines. Many specialized domain-specific operations are included, many operations are available only for specific data types but not for others, and often a high-level understanding of the computation is required in order to take advantage of certain functionality. Furthermore, these particular characteristics differ from one architecture to another.

Misalignment support is an excellent example of this situation. Different machines display different behavior upon an access to an unaligned location and offer different mechanisms for handling such accesses. For example, an efficient scheme that reuses data across iterations can be used for targets that can combine data elements from two vectors. AltiVec has such a capability; other SIMD platforms usually support this functionality only when the misalignment is known at compilation time. If it cannot be determined at that time, a less efficient scheme can be employed, using a special unaligned move

instruction (as is available in SSE), or a sequence of instructions (as in Alpha EV6,[21] for example). In order to accommodate different targets, we introduced a set of new generic tree codes[22] and target-hooks[23] that allow us to model and express the most efficient scheme for each target. These extensions to the tree IR are a result of close collaborations and long discussions with the GCC community.

There are also machine-independent issues that impact the effectiveness of the vectorizer, most notably, the presence of pointers and the limitations of aliasing analysis in GCC. Aliasing analysis in GCC is expected to improve in the near future; in the meantime, its limitations will be overcome by performing loop versioning, however, at the cost of a runtime dependency-test overhead.

Status and future work

We are in the early stages of developing vectorization optimization in GCC. The basic infrastructure is in place to support initial vectorization capabilities. These capabilities are demonstrated by the vectorization test cases, which are available as part of the GCC test suite and are updated to reflect new capabilities as they are added. Work is under way to extend these capabilities and to introduce more advanced vectorization features. The current development status can be found in Reference 18.

The domain of vectorizable loops can be described in terms of the forms, data references, and operations of the loops that can be supported. Currently, vectorization support in GCC handles innermost, single-basic-block loop forms and some cases of loops that contain if-then-else constructs. The data references must be consecutive and array-based or pointer-based and must not overlap (usually this means that pointers need to be annotated as "restricted"). Preliminary misalignment support is also available. Operations must not create a scalar cycle (no reduction or induction[24]), must all operate on the same data type, and must have a vector form that is expressible with existing tree codes.

There are many future directions for enhancing the vectorizer, including the addition of support for more data reference forms, runtime aliasing tests, multiple data types, reduction and induction operations, special idioms (such as saturation, minimum/maximum, dot product, etc.). Among the idioms that are of a particular interest are those representing operations that work on a block (or "chunk") of elements in parallel and can be optimized even when no vector support is available. This can be done by calling a library function (memset, for example) or using special string operations (the S/390 TRANSLATE operation, for example).

Longer term goals include vectorization of nested loops, exploiting data reuse, support for additional access patterns (e.g., strided, that is, a sequence of memory addresses separated by a constant distance) or permuted accesses (that is, accesses by the alphabetic order of each of the constituents for composite terms that require data manipulations), straight-line code vectorization, loop parallelization using threads, and more.

RTL SCHEDULING AND OPTIMIZATIONS

After performing optimizations at the high-level tree IR, GCC expands the code into the RTL, which directly corresponds to the target instruction set. The RTL level contains the details required by instruction scheduling, load and store operations, and register-allocation optimizations.

In this section, we present several RTL optimizations that we contributed to GCC, including interblock scheduling, dispatch group scheduling for out-of-order executing targets, modulo-scheduling of loops, and optimizations for "load-hit-store" events.

Interblock instruction scheduling

Prior to 1997, the original GCC scheduler supported scheduling only within basic blocks (intrablock scheduling). To take advantage of the newer superscalar architectures, more advanced scheduling techniques were required. This section describes the work done to extend the GCC instruction scheduler to support interblock scheduling, focusing on the design of the new interblock scheduler. This project was done in 1997, and the interblock scheduler has become part of GCC's standard distribution since then. For a general description of instruction scheduling, see References 25 and 26.

The design and implementation of the scheduler were influenced by the desire to reuse existing code and have the same code for intrablock and interblock scheduling. We also had to retain global information (e.g., debug notes) throughout scheduling and preserve the GCC compiling speed. To

achieve the goal of supporting interblock scheduling and code motion[26] (i.e., movement relative to other instructions), several design decisions were made:

1. Define the scope of interblock scheduling (the "region") to contain all blocks of innermost loops or entire loop-free functions.
2. Support speculative[27] and equivalent motions, but not duplicative motions, because of associated high compilation and development costs and questionable benefits.
3. Keep the scheme of activating the scheduler twice (before and after register allocation), using the interblock scope in the first invocation.
4. Reverse the scheduling order from a bottom-up order to a top-down order (as used in Reference 24), a requirement for the support of interblock moves (in particular, speculative moves).
5. Develop a visualization mechanism for step-by-step tracking of the scheduler and relevant modeling information.

At the time of this work, the GCC did not have the infrastructure needed to support advanced optimizations. The infrastructure had to be extended in the following directions:

1. *Building control-flow arcs*[28]—At the time, the available control flow information contained only the set of nodes (basic blocks).
2. *Computing block dominator*[29] *and reachability*[30] *information*—To identify and support possible (equivalent and speculative) motion opportunities (see, e.g., Reference 25).
3. *Identifying the regions, based on the control flow graph*—In particular, if all regions are set to contain a single basic block, the case is simplified to that of an intrablock scheduler (meeting the second requirement above).

In addition, the data dependency graph was extended to span multiple blocks, and the list-scheduling algorithm (e.g., ready list, heuristics) was extended to work with instructions from different blocks.

To perform interblock movement, several analyses were implemented that determine if a move is possible and whether it is speculative or equivalent. For speculative moves we determine the conditional execution probability, where to check and update life information,[31] and if loads are exception free (i.e., executing the load will not cause an exception). The high-level design of the interblock scheduler is shown in ***Figure 1***. The steps of computing flow-related information and data-dependency information are independent and can be executed either in order or in parallel. Similarly, the steps of computing the probability and the update blocks are independent.

Escape and update blocks

The scheduler may move an instruction from basic block S to basic block T only if T dominates S, to avoid code duplication. When considering a speculative motion from block S to block T of instruction I that defines register R, we must prevent I from interfering with another live range of R. (This is in addition to the standard restrictions imposed by data dependencies.) To do so, we first examine paths from T that avoid S and identify the first block in each such path from which S cannot be reached. These blocks are called "escape blocks." To prevent I from interfering with another live range of R, we check that R is not alive at the beginning of each such escape block. If this is true, we may move I speculatively, thereby extending the live range of R along the path from S to T. We then need to update the live information for all blocks on this path that are siblings of escape blocks (these blocks are called update blocks), to prevent subsequent speculative moves from interfering with this live range. Our computation of escape and update blocks also helps determine if two blocks are equivalent without requiring explicit post-dominance computations.

Analyzing whether loads are exception free

When a load instruction is moved speculatively, there is a risk of causing an "illegal memory access" exception speculatively. We therefore only move loads speculatively if they are known to be exception free, or if we can prove that the moved load causes an exception only after another load causes a similar exception. We implemented a mechanism that checks for certain types of exception-free and exception-related loads to support speculative movement of such loads. For example, loads of local variables from the stack or loads of global variable addresses from the table of contents could be considered exception free.

There is still room for improving the speculative motion of load instructions. We believe that this is

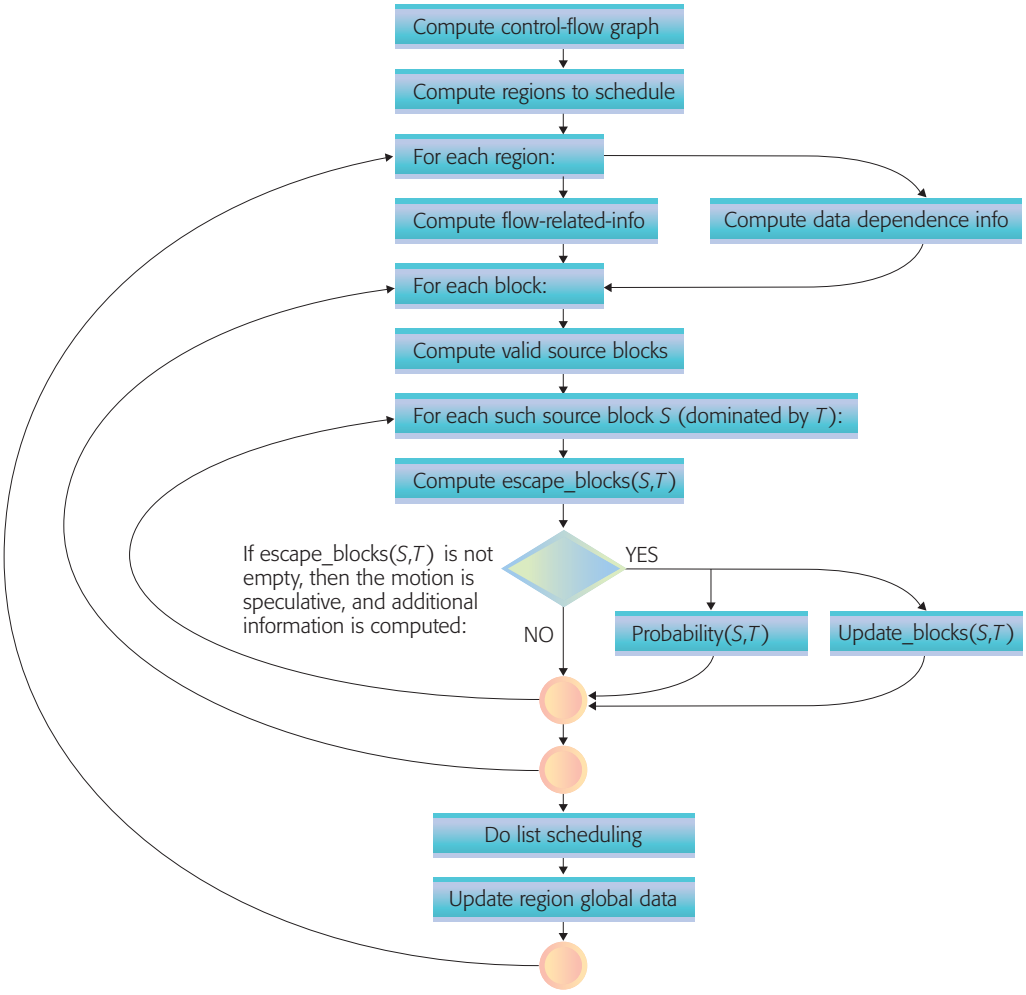

Figure 1
High-level design of interblock scheduler

very important because improved motion of loads will often enable subsequent code motion and may help hide memory latency. We therefore included a flag that enables aggressive (that is, nonconservative) speculative motion of all loads to obtain upper bounds on the potential improvement that can be gained by further improving the initial mechanism.

GCC is an evolving compiler. The infrastructure of GCC improved after the interblock scheduler was added, making the implementation of advanced optimizations much easier. Regarding scheduling, GCC now has a more general control-flow and loop-identification support, as well as a more powerful mechanism for describing the architecture[23] and a modulo-scheduling pass (see the section "Software pipelining and modulo scheduling").

Dispatch group scheduling

Advanced processors such as Power4* and PowerPC970 execute instructions out of order, while dispatching and completing them as groups in order. The compiler should consider the various hardware constraints of dispatch-group formation in order to maximize the instruction-level parallelism (ILP).

The GCC scheduler (described in the section "Interblock instruction scheduling") follows the classic approach of instruction scheduling, which models the delays and constraints of each instruction and schedule to avoid all stalls. This approach can be suboptimal for out-of-order multiple-issue

processors, and specifically for the Power4 and PowerPC970. The GCC scheduler has an option for maximizing the number of instructions issued per cycle,[32] but this option increases compiling time and did not suit our needs.

The major observations that guided our work on instruction scheduling for dispatch groups were that it is practically impossible to accurately model and predict the delays that will occur at runtime (due to out-of-order execution); that there is a better chance of predicting the grouping that will be formed (due to certain constraints); and that accurate emulation of dispatch groups can be very important for increasing the ILP.

In this section, we present three optimizations that enhance the GCC scheduler to handle and optimize dispatch group formation and out-of-order execution. We contributed the optimizations described in this section to the FSF during October 2003,[33–35] and they are part of the GCC 3.4 release. See Reference 32 for specific implementation and performance details of these optimizations.

Including dispatch group restrictions as a criterion during scheduling

The Power4 processor dispatches a group of up to four instructions (and a branch) in each cycle. Some instructions can only be assigned to certain dispatch slots. Specifically, there is a set of instructions that can only be dispatched as the first instruction in a dispatch group. When issued, such dispatch-slot-restricted instructions always open a new group, causing the termination of a previous dispatch group.

When the instruction scheduler chooses an instruction and decides to schedule it in the currently scheduled cycle, it is better to choose an instruction that must be first (if available and) if a new dispatch group is being opened. We used an existing target hook, adjust_priority, to modify the priority of instructions during scheduling according to these dispatch-group considerations.

Allowing early scheduling of stalled instructions

The GCC scheduler uses two queues to manage the instructions that are candidates for scheduling. Those that are still waiting for other (already scheduled) instructions to be completed are placed in the "stalled queue," and those that do not need to wait are placed in the "ready queue." The instructions in both queues can be scheduled in the current cycle, but instructions in the stalled queue will (according to the model) wait for data or some resource to become available.

The scheduler selects instructions for scheduling only from the ready queue. With time, instructions from the stalled queue become ready and move to the ready queue. If the ready queue is empty, the compiler closes the current dispatch group that it is modeling, even if there are vacant dispatch slots available, and proceeds to schedule instructions for the next group and cycle.

This scheme does not do a good job of modeling for processors where delays can occur between the dispatch and the execution of instructions. Optimizing the dispatch of instructions is important for group-formation considerations, and optimizing their execution is relevant for data-dependent latency considerations. For such targets, the stalled queue contains instructions that cannot be executed in the current cycle (because of data or resource delay), but they can and sometimes should be issued in the current cycle.

In order to optimize for execution and dispatch group utilization, we allow the scheduler to select instructions directly from the stalled queue. We limit this to cases where there are potentially vacant dispatch slots to fill and there is nothing better to fill them with (i.e., the ready queue is empty). This also improves the emulation of dispatch group formation; the resulting schedule better matches the groups that will be formed at runtime. Because some dependencies (which keep instructions in the stalled queue) may incur a high penalty if broken, we prevent premature selection from the stalled queue in such cases.

Null-operation insertion

The GCC scheduler models the formation of dispatch groups, but does not insert NOPS (null operations) to fill up vacant slots. In order to improve the synchronization of group boundaries between the compiler and hardware, we implemented a post-scheduling pass that scans the instruction stream and examines the dispatch group boundaries that the scheduler had marked. Vacant issue slots that are detected are padded with NOPS,

in order to force the scheduler's grouping on the processor dispatcher.

A naive implementation inserts too many NOPs, which have a significant negative effect on performance. Indeed, NOPs should be inserted very sparingly, and only in cases in which the presence of a dependency within a dispatch group (which NOPs can prevent) is truly problematic. We therefore tried to classify such costly dependencies and implemented a new scheme. The scheme scans the instruction stream right after scheduling, but this time, inserts NOPs only between instructions that have a costly dependency between them, in order to force these instructions into separate groups. This new scheme is much less intrusive and can be viewed as a fine tuning of the group boundaries to better match the processor behavior.

There are several parameters that tune the NOP-insertion mechanism, such as the definition of costly dependencies (we considered memory accesses true dependencies; see the section "Handling load-hit-store events"), and the number of NOPs to be inserted (the minimum number based on group emulation or regardless of this emulation).

Software pipelining and modulo scheduling

After we enhanced the instruction scheduler of the GCC to handle interblock code motion, as described in the section "Interblock instruction scheduling," the scheduler's main limitation became its inability to move instructions across loop-back arcs or iterations. *Modulo scheduling* is an instruction-scheduling technique focused on improving the schedules of loops by enabling instructions to transfer between iterations. We further enhanced the GCC instruction scheduler by implementing a Swing Modulo-Scheduler (SMS), which is an implementation of modulo scheduling designed to reduce register pressure.[36,37] SMS first orders the instructions according to the data dependencies in an alternating up-and-down order (hence its name)—first ordering instructions that are successors of already ordered instructions, then instructions that are predecessors of already ordered instructions, and so on. The scheduling phase then traverses the nodes in the given order, trying to schedule dependent instructions as close as possible and thus shorten live ranges of registers. This section describes our implementation of SMS in GCC.

SMS implementation in GCC

SMS[37,38] is performed immediately before the first interblock scheduling pass, and indeed, could be combined into one pass if so desired in the future.

■ The infrastructure of GCC improved after the interblock scheduler was added ■

The modulo-scheduling pass traverses the loops and generates a new schedule for each loop according to the following steps. First, the modulo scheduler builds a data dependency graph (DDG) that represents intra- and inter-loop dependencies. SMS then determines a fixed ordering of the instructions based on the DDG and uses it in repeated attempts to schedule the kernel of the loop. Finally, after a schedule for the kernel is constructed, SMS performs modulo variable expansion, generates prologue and epilogue code, and inserts a loop precondition if needed. SMS also marks the loop after scheduling it, to prevent subsequent rescheduling by the standard instruction-scheduling passes. Only the kernel is marked; the prologue and epilogue are subject to subsequent scheduling.

The main infrastructure contributions to GCC involved in our implementation of SMS were: (1) a new DDG for loops, (2) the ability to perform list scheduling in both directions, compared to top-down or bottom-up cycle scheduling, and (3) effective renaming of registers during scheduling as needed, by performing modulo variable expansion. We now elaborate on these contributions.

DDG generation

The existing representation of data dependencies in GCC does not meet the requirements for implementing modulo scheduling; it lacks support for interloop dependencies, and it is not easy to use. We decided to implement a DDG, which provides additional capabilities (i.e., loop-carried dependencies) and a modulo-scheduling-oriented API.

The DDG is built in several steps. We first construct the intraloop dependencies using the standard routines of the scheduler. Next, we calculate interloop register dependencies of distance one[39] by using the GCC flow analysis. Finally, we calculate interloop memory dependencies in a conservative

way. This is currently being developed and improved.

We provide several graph-theoretic utilities based on the DDG to support the node-ordering algorithm of SMS. These include finding strongly connected components, finding all nodes that lie on directed paths between two sets of nodes, and calculating the longest cycle (in terms of total latency divided by total distance) in a connected component.

List scheduling the kernel of the loop

SMS schedules the instructions (i.e., the nodes) for the kernel of the loop according to a precomputed order. For each node we calculate a scheduling window, that is, a range of cycles in which we can schedule the node similarly to already scheduled nodes. Use of previously scheduled predecessors (PSP) increases the lower bound of the scheduling window, whereas use of previously scheduled successors (PSS) decreases the upper bound of the scheduling window. The scheduling windows are related to instructions of the same iteration.

The scheduling window itself contains a range of a number of cycles equal to the Initiation Interval (II), at most. After computing the scheduling window, we try to schedule the node during some cycle within the window, while avoiding resource conflicts. If we succeed, we mark the node and its (absolute) schedule time. If we do not succeed in scheduling the given node within the scheduling window, we increment the value for II and start over again. If II reaches an upper bound, we quit and leave the loop without transforming it.

During the process of scheduling the kernel, we maintain a partial schedule that holds the scheduled instructions in a number of rows equal to II. When an instruction is scheduled in a cycle T (inside its scheduling window), it is inserted into row $T \bmod \mathrm{II}$ of the partial schedule. The instructions in the partial schedule may belong to different iterations. After all instructions are scheduled successfully, the partial schedule supplies the order of generating the instructions. Special care is needed when dealing with the start and end cycles of the scheduling window, as the order of instructions within these rows has to be considered.[39]

When modulo scheduling the kernel, we need to repeatedly check whether given instructions will cause resource conflicts if scheduled at a given cycle or slot of a partial schedule. The resource model based on the DFA (deterministic finite automaton) in GCC[32] works by checking a sequence of instructions, in their order. This approach is suitable for cycle-scheduling algorithms in which instructions are always appended at end of the current schedule. In order for SMS to use this linear approach, we generate a trace of the instructions, cycle by cycle, centered at the candidate instruction, and feed it to the DFA.[32] The major drawback of this mechanism is the increase in compilation time; our future plans include addressing this concern.

Modulo variable expansion

After all instructions have been scheduled in the kernel, some values defined in a given iteration and used in some future iteration must be stored so that they are not overwritten. Such values are overwritten when their life range exceeds a number of cycles which equals II. The defining instruction will execute more than once before the using instruction accesses the value. Life ranges of registers can exceed this number of cycles because register antidependencies[41] are removed from the DDG.

The problem of overwriting these values is solved by using modulo variable expansion, which we implemented by generating register copy instructions as follows:

$$R_n \leftarrow R_{n-1};\ R_{n-1} \leftarrow R_{n-2}\ ;\ldots;\ R_1 \leftarrow R_{\mathrm{def}}$$

where R_{def} is the register in the defining instruction, and n is determined according to the number of times the back edge of the newly scheduled kernel is crossed between the defining instruction and its appropriate use. Every register antidependency that is broken by code motion is fixed by using this register copying and renaming.

The SMS appears in Version 3.5 of GCC. We are continuing to work on several enhancements to improve its performance.

Handling "load-hit-store" events

In several cases, we observed that load instructions that follow stores to the same memory location cause delays and reduce performance. It is obvious that this sequence, called a “load-hit-store” event, could be avoided by simply copying the value from the stored register instead of loading it from memory, if such a register copy instruction were

available. However, in many cases the compiler did not recognize this possibility. There are two sources to this problem: redundant accesses to memory in the source code of the program and spill code[42] generated by the compiler to save and restore registers. We addressed both cases.

The first case, where redundant loads appear before register allocation, is handled by redundancy elimination optimization. Redundancy elimination removes redundant calculations of expressions by reusing previously calculated values that are stored in some register. The redundancy elimination pass of GCC did consider loading a calculation of an expression from memory, but did not consider store operations as expressions. Thus, GCC did replace a load following another load from the same memory location by a register copy, but did not replace a load following a store to the same location. We enhanced the redundancy elimination pass so that it would also consider stores as expressions, and hence replace subsequent loads from the same location with register copies.

The second case of load-hit-store events was due to poor register spilling (i.e., the reload pass in GCC).[43] We handled this case in two ways. First, we added a "cleanup" pass after the reload that removed such redundancies, similar to the first case. However, this solution is limited because it works with hard (that is, allocated) registers. We reused the existing redundancy elimination infrastructure and added a special consideration of register availability for the register moves that we generate. We also took care of partial redundancy elimination by adding loads on basic blocks that are less critical (according to profiling), provided we can replace loads from critical blocks by register moves.

Our second method of handling load-hit-store events after register allocation was to keep such loads away from the stores. We changed the instruction scheduler to add NOPs between a store and a subsequent load from the same location; this served to keep them in different dispatching groups (see "Dispatch group scheduling").

THE ZSERIES BACK END

The GCC back end for a particular processor describes the architecture and its implementation in a manner that allows the platform-independent optimization passes to generate correct and efficient code for the target. The framework provided for this purpose is powerful and flexible enough to allow GCC to currently support more than 30 major processor architectures, including many different types. The S/390 back end implements support for the S/390 and zSeries mainframe processors.

■ It is practically impossible to accurately model and predict the delays that will occur at runtime ■

History

In 1997, the S/390 firmware development group was searching for an ANSI C compiler with specific requirements, including link compatibility to the PL8 compiler and the possibility to write embedded assembler code. At this time, it was discovered that the existing S/370* MVS port by Jan Stein and Dave Pitts could be used as a starting point.

In 1998, when Linux for S/390 work was started, this compiler (and its linkage) was used as a base. Over time, the Linux development team had taken over as the driving force behind the S/390 back end, changing it to use the ELF (executable and linking format) linkage format, exploiting the 64-bit architecture, and finally donating the port to the FSF. Since then, two of the authors of this paper have been maintaining this back end, providing all necessary fixes and enhancements for the user community.

The zSeries architecture

The zSeries architecture is a typical CISC (complex instruction set computer) architecture. It provides an extensive set of assembler instructions (over 700 opcodes for the current model), including a sophisticated subsystem to perform I/O operations. These complex instructions tend to be implemented by internal firmware. The processor also provides efficient support for logical partitioning and virtual machines, for example by means of the SIE (START INTERPRETIVE EXECUTION) instruction. For the compiler back end, however, the simple instructions are the most important; these provide the means to move data between memory and registers, perform

the standard arithmetical and logical operations, and affect control flow via conditional and unconditional branches as well as subroutine calls.

Recent zSeries processors can be operated in two different modes of operation, Enterprise Systems Architecture (ESA/390) mode and z/Architecture mode.[44,45] The latter provides 64-bit general purpose registers and a complete set of instructions operating on them; otherwise, it is compatible with the later ESA/390 mode. The processor provides 16 general-purpose registers which are 32 or 64 bits wide, depending on the architecture mode, as well as 16 floating-point registers, which are 64 bits wide. Most instructions allow two operands, with the first source operand being used as the destination as well. As opposed to typical RISC architectures, zSeries allows memory operands to be used directly with nearly all operations; all arithmetic and logical instructions provide both a register-to-register (RR) and a memory-to-register (RX) form. Logical and move instructions are also available as memory-to-memory (SS) or immediate-to-memory (SI) operations. One important design goal of the zSeries processor microarchitecture is that RX and RR instructions execute with the same speed, as long the memory is already available in the level one (L1) cache.

The System/360* architecture originally provided a 24-bit address space, in which the high eight bits of 32-bit registers used in address generation were ignored. As 16MB of address space proved too small, the address space needed to be extended. For compatibility reasons, the 24-bit addressing mode still exists, and a new 31-bit addressing mode was introduced with the S/370 architecture. Certain instructions now use the top bit of a 32-bit value to decide whether to operate in 24- or 31-bit addressing mode. Even though Linux on zSeries never uses 24-bit addressing, we still have to handle some complications that come with 31-bit addressing as opposed to the 32-bit addressing that is common on many other platforms. The zSeries processors finally introduced 64-bit addressing as a third mode, extending the address space up to 16 exabytes. The GCC back end supports generating code for either 31-bit or 64-bit addressing modes. It can also be tuned to a specific operating environment, for example, to generate optimal code for the 31-bit ABI (application binary interface) when running on a z990 processor in z/Architecture mode. The target ABI, architecture mode, processor instruction set level, and target processor for tuning purposes can all be selected independently.

The Linux on zSeries ABI

The GCC back end must take care to generate code that is not only appropriate for the processor architecture but also interoperable by means of subroutine calls with other programs running on the target platform. The ABI defines all aspects of code generation that are required for interoperability. The GCC back end currently supports two ABIs, those used by Linux for S/390[46] and those used by Linux for zSeries.[47] Note that these ABIs differ significantly from the interfaces used with other operating systems on the mainframe, namely the traditional operating-system linkage and the high-performance XPLINK. The calling convention used on Linux passes arguments in up to five general-purpose registers and up to four floating-point registers; excess arguments or those with data types preventing register use are passed on the stack. As the processor architecture does not actually define the concept of a stack at the hardware level, the Linux ABI uses register r15 as a stack pointer by convention. Function prologue and epilogue code handles setting up the registers and stack frame as defined by the ABI.

The ABI details can have a significant impact on performance. For example, early releases of GCC on zSeries generated code in the function prologue that would explicitly maintain a stack back chain, that is, a pointer stored at the start of a function's stack frame that would give the address of the caller's frame. This results in a linked list of stack frames being maintained at runtime that can be used to generate a back trace listing for debugging purposes. In recent releases, however, we have eliminated this overhead by using DWARF-2[48] call frame information records to store details of the stack frame layout for each function in an extra data section of the executable image. Debugging tools can use this data to generate stack back traces without any runtime support by generated code. With GCC 4.0, we have also reduced the amount of stack space required per function call; this is helpful in environments like the Linux kernel code where stack size is restricted.

GCC back end

We now describe in detail some of the issues we encountered when implementing the zSeries back end. This discussion will approximately follow the

flow of the middle end optimization passes from RTL expansion to final assembler code generation, and describe the contributions of the zSeries back end to each stage. For more implementation details see Reference 6.

The initial transformation of the program into the lower-level RTL representation is performed with expanders[49] that encapsulate the representation of a predefined set of standard arithmetical, logical, and move operations. For example, the addsi3 expander generates RTL to add two 32-bit integer source operands and store the result in a destination operand. The sequence of RTL thus generated is then processed by several generic optimization passes, including passes for common subexpression elimination, jump threading and bypassing, dead-code elimination, and low-level loop optimizations. Note that while similar optimizations are already performed on the higher-level tree representation in GCC 4.0, these are still not completely redundant, as RTL expansion may have introduced new optimization opportunities. Cost functions defined by the back end are used to guide these algorithms towards instruction sequences that are particularly well suited to the target platform.

One optimization of special importance for the back end is the combine pass, which allows the use of assembler instructions that implement more complex operations than those directly available as expanders. The middle end tries various ways of combining multiple logically dependent RTL instructions into a single one; if the resulting RTL is accepted by a back-end insn[50] pattern, the replacement is performed. This is used to match zSeries fused multiply-and-add instructions, for example. The combine facility is also employed to make efficient use of the zSeries condition code, a two-bit value stored in the program status word that is set by comparison instructions; conditional branches depend on it to decide whether to take the branch. However, many arithmetical, logical, and other operations also set the condition code in addition to computing their results, which makes explicit use of comparison instructions superfluous in many cases. The zSeries instruction set also provides operations like TEST UNDER MASK that employ the condition code to implement frequently used bit-test operations very efficiently; by defining appropriate insn patterns, the zSeries back end is able to make full use of these platform features.

The ADD LOGICAL WITH CARRY and SUBTRACT LOGICAL WITH BORROW instructions introduced with z900 also allow using the condition code in a non-branch instruction. They are primarily intended to implement multiword addition and subtraction by allowing a carry or borrow from a low-order word to be automatically considered when operating on the next higher word. However, it is also possible to use these instructions to perform a restricted form of conditional execution; for example, the statement if (a < b) x++; can be implemented without using any conditional branch instruction, thus avoiding potentially expensive erroneously predicted branches. This transformation is performed by a platform-independent "if-conversion" pass that calls into a back-end conditional add expander to implement the details.

■ Instruction scheduling is a crucial optimization pass required to prevent expensive pipeline stalls ■

Up to this point, the program was kept in a high-level variant of RTL that makes some simplifying assumptions, most notably that the processor provides an unlimited supply of registers. At some point, it is necessary to transform the program into a stricter representation that respects actual machine constraints. This happens during the register allocation and reload passes, using register information and per-instruction constraints provided by the back end. This phase also ensures that all memory operands are accessed using proper address formats.

On zSeries, effective addresses may be formed, in general, by adding the contents of a base register, an index register, and an immediate 12-bit unsigned displacement. However, some instructions do not allow the use of an index register. Alternatively, the z990 processor introduced the long displacement facility, which allows use of a 20-bit signed displacement for selected instructions. This is one of the few areas where it proved necessary to enhance GCC platform-independent code in order to fully support zSeries, because the reload pass was unable to handle so many different address formats.

At this point, the back end inserts all function prologue and epilogue code as required by the ABI (see "The Linux on zSeries ABI"). The back end can cause further optimizations to be performed on the RTL sequence at this point by defining "splitters" and "peepholes." *Splitters* are used to break up one RTL instruction into a sequence of patterns. They may be used by the back end to delay exposing details of the processor until a later stage of the compilation, for instance to present a doubleword addition as a single pattern to early optimization passes, while splitting it up later into single word additions with carry. Splitters may also be required to ensure correct code is generated in some cases, for example, to handle instructions with restricted addressing modes like LOAD MULTIPLE on zSeries. *Peepholes*, on the other hand, allow the back end to merge a sequence of RTL instructions into a single pattern or a different sequence, possibly using additional scratch registers.

Instruction scheduling (see "RTL scheduling and optimizations") is a crucial optimization pass required to prevent expensive pipeline stalls, in particular as zSeries processors operate in order. This means that whenever some stage of the pipelined execution of an instruction depends on data resulting from a preceding instruction that has not yet been completed, the processor will stall until the data becomes available. The platform-independent scheduling algorithms use a detailed description of such pipeline dependency hazards of the particular target-processor microarchitecture; this is provided in the form of a finite-state machine by the back end. For zSeries, we currently define two such pipeline descriptions.

For the z900 processor, the central pipeline hazard is address generation interlock (AGI), triggered when the result of an operation is used as a base or index register to form an effective address of a subsequent instruction. As operand addresses are required early in the instruction pipeline, at least four other instructions need to be scheduled between the two to avoid an AGI stall. However, for some simple operations like load, the hardware provides an AGI bypass such that their results are available for address generation after only one or two cycles. The z990 is more complex: its superscalar microarchitecture[51] is able to execute up to three instructions in parallel. This adds new requirements to instruction scheduling, as an instruction that uses the results of a preceding one cannot run in parallel with it. Finally, the much improved floating-point unit[52] of the z990 features a pipelined execution that introduces another complex set of data dependencies for floating-point instructions.

After the final scheduling pass, the back end cleans up all remaining target-specific issues. For zSeries, this includes handling the limited ranges of branch instructions as well as a possible overflow of the pool holding literal constants. In ESA/390 mode, relative branch instructions can reach only a range of 64KB relative to the current instruction address; more distant targets can only be reached by register-indirect branches. As zSeries does not provide instructions to load arbitrary literal values as immediate operands, these need to be stored in a literal pool in memory. If that pool exceeds 4KB in size for any particular function (a condition that fortunately rarely ever occurs, but still needs to be handled), we need to split up the pool into multiple smaller ones and reload the register pointing to the pool base as required. After this is completed, the sequence of RTL instructions is translated into assembler source code as defined in the back-end instruction patterns.

PERFORMANCE

This section examines the performance of GCC by use of various benchmarks.

GCC improvements on zSeries

In this section we take a closer look at the performance improvements that have been observed for GCC on zSeries during the last five years. The following comparison is based on estimated SPECint2000** results. For details on SPEC**, see Reference 53, and for details on SPECint2000, see Reference 54. All runs have been compliant base runs according to the SPECInt2000 rules. As it is the purpose to present the development over time, all results have been normalized. The measurements for the 1999, 2000, 2001, and 2002 results have been executed on a z900, and the measurements for 2001, 2002, 2003, and 2004 on a z990. The overlapping measurements (2001, 2002) have been used to scale the 1999 and 2000 measurements to a z990. All measurements have been run in a LPAR environment (the zSeries version of logical partitioning—see Reference 55 for more details) with the respective Linux operating system of that time.

Figure 2 shows the relative performance development for the GCC compiler on zSeries. The 1999 result was obtained using the first published version of the GCC version 2.95.1 for S/390 using an IBM internal driver. As the development focus was on functionality only, there was a large potential for performance improvement. The 2000 result was measured on a SLES 1 (SUSE LINUX Enterprise Server 1) distribution using the compiler included (GCC 2.95.2). As shown in the figure, some performance improvements were made, and they were in the back end of the compiler exclusively. The 2001 result has been obtained using the SLES 7 distribution and the included system compiler (GCC 2.95.3). Again, the improvements have been implemented in the GCC back end.

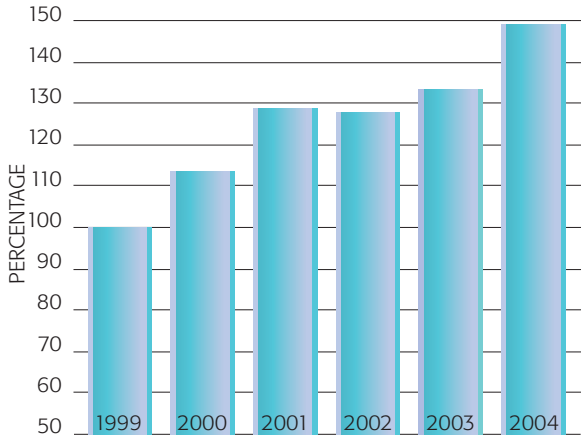

Figure 2
Relative performance of some GCC versions on the IBM zSeries platform

After those improvements, the transition to the new GCC 3.x family occurred. Initially, this resulted in slight performance degradation, as can be seen in the 2002 results. This was measured on a SLES 8 with the GCC 3.2 compiler included. However, with SLES 8 SP3, an optional GCC 3.3 compiler was shipped at the end of 2003. Using this compiler, the performance improved again. Here the main contributor was the improved scheduling for the z990 engine described in the section "The zSeries back end."

The final measurement was done on a GCC 3.4 compiler with an IBM internal driver from development. At this point, the profile-directed feedback was working on all SPECint** cases for zSeries, and so could be used for this run. This compiler will most likely be available in the next Red Hat distribution.

Overall, we have seen an improvement of 49 percent over the first results. The first steps towards better performance have been easy ones. We believe most of the back-end work has been completed, and more work is required in the middle end of the compiler. The inclusion of the tree-SSA[9] is a promising step in this direction.

Published results for SPECint benchmark using GCC

Not many results have been published on the SPEC CPU2000 Web site. This can be attributed to the fact that other compilers are producing better code for this benchmark than GCC. Generally speaking, the compilers provided by the processor vendors are still one step ahead of GCC. In fact, the only results available are those using the AMD Opteron and Athlon processors. We make three different comparisons with these compilers.

For the early Opteron results[56] in 2003, AMD used the same hardware with the Intel 7.0 compiler and the GCC 3.3 coming with SLES 8. Here GCC achieves 90 percent of the performance of the Intel compiler for base runs. In one case GCC outperforms the Intel compiler.

In 2004, AMD[57] used identical hardware with the Intel 8.0 compiler and GCC. One important difference this time was that for the base runs GCC produced 64-bit code; whereas, the Intel compiler produced 32-bit code. Generally, the 64-bit code is expected to be slower because it has a larger memory footprint. With that difference, GCC achieved 87 percent of the score of the Intel compiler for base runs and 90 percent for peak runs. However, there are now three base and two peak workloads where GCC is ahead.

SUN[58] used the PathScale EKO (Every Known Optimization) compiler suite[11] on hardware comparable to that used by AMD.[57] Here, GCC achieves 95 and 94 percent of the PathScale result for base and peak runs respectively. For this comparison, GCC is ahead on 4 cases for base (or 3 for peak) measurements.

PL8 compiler

Before the GCC PL8 code was allowed to generate the firmware code on the z990, it had to match the performance of the original PL8 compiler. That meant that the generated code had to perform at least as well as the code produced by the competitive compiler. It is reported in Reference 5 that this goal was achieved.

Again, this is an indication that the GCC compiler suite is capable of generating competitive code if the appropriate focus is put into development. Note also that the good performance of the PL8 code is also due to the fact that it inherited the performance gains of the GCC back end described in the section "The zSeries back end."

CONCLUSIONS

There are many benefits to our work on optimizations for GCC. Many existing and potential users of IBM platforms are using GCC, and this is an effective means to provide them with additional value and improve support for our platforms. The vibrant collaboration and synergy among compiler writers contributing to GCC from various affiliations is very helpful and supportive. The infrastructure of GCC can pose challenges for advanced optimizations, but it is being improved. The widespread usage of GCC across platforms and environments also helps in testing and debugging the compiler. Another benefit of GCC is its availability for research in academia and industry, an advantage we seek to exploit to continue providing state-of-the-art and innovative optimizations in GCC in the future.

The open-source approach and GCC's modular structure turned out to be of great value. Writing a zSeries back end immediately made all languages implemented by the GNU Compiler Collection available on that new platform. Writing a PL8 front end made that language available on all platforms supported by GCC. In both cases, existing code could be reused. Most of the contributions described in this paper have already been released as open-source software.

ACKNOWLEDGMENTS

The authors would like to thank many people from IBM and the GCC development community who were involved in the GCC efforts covered in the paper, including (in alphabetical order): Daniel Berlin, Doron Cohen, Zdenek Dvorak, Olga Golovanevsky, Mario Held, Richard Henderson, Vladimir Makarov, Devang Patel, and Sebastian Pop. We would also like to thank the many additional people from IBM who have contributed to GCC in areas not covered by this paper.

CITED REFERENCES AND NOTES

1. GCC Home Page—GNU Project—Free Software Foundation (FSF), http://gcc.gnu.org.
2. R. M. Stallmann, *Using and Porting the GNU Compiler Collection*, Free Software Foundation, Boston, MA (1999).
3. "GNU Compiler Collection Internals," Free Software Foundation, Boston, MA (2004), http://gcc.gnu.org/onlinedocs/gccint/index.html#Top.
4. *The Free Software Definition*, Free Software Foundation, http://www.gnu.org/philosophy/free-sw.html.
5. W. Gellerich, T. Hendel, R. Land, H. Lehmann, M. Mueller, P. H. Oden, and H. Penner, "The GNU 64-bit PL8 Compiler: Toward an Open Standard Environment for Firmware Development," *IBM Journal of Research and Development* **48**, No. 3/4, 543–556 (July 2004), http://www.research.ibm.com/journal/rd/483/gellerich.pdf.
6. H. Penner and U. Weigand, "Porting GCC to the IBM S/390 Platform," *Proceedings of the GCC Developer's Summit* (2003), pp. 195–213, http://zenii.linux.org.uk/~ajh/gcc/gccsummit-2003-proceedings.pdf/.
7. An "attributed syntax tree" is a syntax tree with additional attributes associated with its nodes.
8. "Constant folding" is an optimization technique to execute an operation at compilation time rather than at execution time if all operands are constant. For example, an assignment i = 2+3 would be replaced by i = 5 rather than actually generating an ADD machine instruction.
9. *SSA for Trees—GNU Project*, Free Software Foundation (FSF), http://gcc.gnu.org/projects/tree-ssa/.
10. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems* **13**, No. 4, 451–490 (October 1991).
11. D. Naishlos, "Autovectorization in GCC," *Proceedings of the GCC Developer's Summit* (2004), pp. 105–118, http://www.gccsummit.org/2004/2004-GCC-Summit-Proceedings.pdf.
12. *Loop Nest Optimizer—GNU Project*, Free Software Foundation (FSF), http://gcc.gnu.org/projects/tree-ssa/lno.html.
13. K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales, "Altivec Extension to PowerPC Accelerates Media Processing," *IEEE Micro* **20**, No. 2, 85–95 (March–April 2000).
14. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro* **16**, No. 4, 43–45 (August 1996).
15. AltiVec intrinsics constitute a set of C functions which the GCC compiler maps onto single AltiVec instructions. For

a listing of this set see: http://developer.apple.com/hardware/ve/instruction_crossref.html.

16. R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures—A Dependence-Based Approach*, Morgan Kaufmann, San Francisco, CA (2001).
17. G. Ren, P. Wu, and D. Padua, "A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions," *Proceedings of the 16th International Workshop of Languages and Compilers for Parallel Computing (LCPC2003)*, Lecture Notes in Computer Science **2958**, pp. 420–435 (October 2003).
18. R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems* **9**, No. 4, 491–542 (October 1987).
19. M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison Wesley, Reading, MA (1996).
20. "Strip mining" is a term used for an optimization technique to (partly) parallelize the execution of a loop. If a loop ranges from i = 1 to 1000, and the hardware is able to execute eight instances of the loop body at the same time, then strip mining would involve first executing the loop instances for i = 1 to i = 8 at the same time; next, the instances for i = 9 to i = 16; and so on.
21. *Exploring Alpha Power for Technical Computing*, Compaq Technology Brief, High Performance Technical Computing Group, Compaq Computer Corporation (2000), http://h18002.www1.hp.com/alphaserver/download/wp_alpha_tech_apr00.pdf
22. In the GCC's intermediate representation of trees, each tree has a code which indicates what type of tree it is.
23. A "target hook" is a mechanism within GCC that enables a target-independent pass to execute target-dependent operations or preferences.
24. "Reduction" refers to an operation that produces a scalar output from a vector input, for example, computing the sum or maximum value of vector elements. "Induction" refers to an operation that updates a scalar variable inside a loop, based on its values from previous iterations, incrementing or decrementing by a loop-invariant amount.
25. H. S. Warren, Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor," *IBM Journal of Research and Development* **34**, No. 1, 85–92 (January 1990), http://www.research.ibm.com/journal/rd/341/imbrd3401J.pdf.
26. D. Bernstein and M. Rodeh, "Global Instruction Scheduling for Superscalar Machines," *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, SIGPLAN Notices **26**, No. 6, 241–255 (June 1991).
27. A move of an instruction from position P1 to position P2 is speculative if there exist executions that pass through P2 but not P1. In such a case, the instruction will be executed because of its position at P2 but would not have been executed if it remained at P1.
28. A "control-flow arc" is an arc in a control-flow graph. This graph has a node for each basic block. Two nodes are connected by an arc if during any execution, the second block can be executed immediately after the first block.
29. Block B1 is a dominator of block B2 if any execution that reaches B2 must go through B1.
30. Block B2 is reachable from block B1 if there exists an execution that reaches B2 after B1.
31. A variable is "live" at every position between its definition and its use, also known as its "live range."
32. V. N. Makarov, "The Finite State Automaton Based Pipeline Hazard Recognizer and Instruction Scheduler in GCC," *Proceedings of the GCC Developer's Summit* (2003), pp. 135–150, http://zenii.linux.org.uk/~ajh/gcc/gccsummit-2003-proceedings.pdf.
33. D. Naishlos, *insn Priority Adjustments in Scheduler and RS6000 Port*, http://gcc.gnu.org/ml/gcc-patches/2003-10/msg00485.html.
34. D. Naishlos, *Scheduling of Queued insns*, http://gcc.gnu.org/ml/gcc-patches/2003-10/msg00698.html.
35. D. Naishlos, *Scheduling Tuning in RS6000 Port*, http://gcc.gnu.org/ml/gcc-patches/2003-10/msg01702.html.
36. Register pressure refers to the number of registers that are needed at certain positions in the code because their live ranges intersect.
37. E. Ayguade, M. Valero, J. Llosa, and A. Gonzalez, "Swing Modulo Scheduling: A Lifetime Sensitive Approach," *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)* (1996), pp. 80–87.
38. E. Ayguade, M. Valero, J. Llosa, A. Gonzalez, and J. Eckhardt, "Lifetime-sensitive Modulo Scheduling in a Production Environment," *IEEE Transactions on Computers* **50**, No. 3, 234–249 (2001).
39. This refers to a register dependency between an instruction in one iteration and an instruction (possibly the same one) in the next iteration. For example, the instruction r5 = r5 + 1 inside a loop creates an interloop register dependency of distance one.
40. M. Hagog and A. Zaks, "Swing Modulo Scheduling in GCC," *Proceedings of the GCC Developer's Summit* (2004), http://www.gccsummit.org/2004/2004-GCC-Summit-Proceedings.pdf.
41. This is also known as a "write-after-read" dependency. It is the dependency of an instruction that writes to a variable on a previous instruction that read from the variable.
42. "Spill code" is a set of instructions that store and load a variable into memory. This is needed in situations where there is no available register.
43. Because of the limited number of registers, values previously held in registers must at times be stored in memory, if the value is needed, and this process is referred to as "register spilling."
44. *ESA/390 Principles of Operation*, IBM Document Number SA22-7201-07 (2000), http://publibfp.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/dz9ar007.
45. *z/Architecture Principles of Operation*, IBM Document Number SA22-7832-01 (2000), http://publibfp.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/dz9zr001.
46. *LINUX for S/390 ELF Application Binary Interface Supplement*, IBM Document Number LNUX-1107-00 (2001), http://oss.software.ibm.com/linux390/docu/1390abi0.pdf.
47. *LINUX for zSeries ELF Application Binary Interface Supplement*, IBM Document Number LNUX-1107-00 (2001), http://oss.software.ibm.com/linux390/docu/lzsabi0.pdf.
48. DWARF is a format for debugging information in which additional information is inserted into the binary file produced by the compiler.
49. "Expanders" are operations that make modifications to GCC's intermediate RTL representation.
50. An "insn" is a pattern used internally by GCC and constitutes a formal description of a machine instruction.

The description is used to translate the program being compiled from a (more or less) target-independent form into real binary code.

51. T. J. Slegel, E. Pfeffer, and J. A. Magee, "The IBM eServer z990 Microprocessor," *IBM Journal of Research and Development* **48**, No. 3/4, 295–310 (July 2004), http://www.research.ibm.com/journal/rd/483/slegel.pdf.
52. G. Gerwig, H. Wetter, E. M. Schwarz, J. Haess, C. A. Krygowski, B. M. Fleischer, and M. Kroener, "The IBM eServer z990 Floating-Point Unit," *IBM Journal of Research and Development* **48**, No. 3/4, 311–322 (July 2004), http://www.research.ibm.com/journal/rd/483/gerwig.pdf.
53. SPEC—Standard Performance Evaluation Corporation, http://www.spec.org.
54. SPEC CPU2000 V1.2, http://www.spec.org/cpu2000.
55. I. G. Siegel, B. A. Glendening, and J. P. Kubala, "Logical Partition Mode Physical Resource Management on the IBM eServer z990," *IBM Journal of Research and Development* **48**, No. 3/4, 535–541 (July 2004), http://www.research.ibm.com/journal/rd/483/siegel.pdf.
56. Pathscale—Compiler Suite, http://www.pathscale.com/products1.html.
57. CINT2000 Result: Advanced Micro Devices ASUS SK8V Motherboard, AMD Opteron 150, http://www.spec.org/osg/cpu2000/results/res2004q2/cpu2000-20040503-02999.html, http://www.spec.org/osg/cpu2000/results/res2004q2/cpu2000-20040503-03003.html.
58. CINT2000 Result: Sun Microsystems Sun Java Workstation W2100z, http://www.spec.org/osg/cpu2000/results/res2004q3/cpu2000-20040628-03192.html.

Accepted for publication November 9, 2004.
Published online April 12, 2005.

David Edelsohn
IBM T.J. Watson Research Center, 1101 Kitchawan Road, Yorktown Heights, New York (edelsohn@us.ibm.com). Dr. Edelsohn received an A.B. degree in astronomy and physics from the University of California at Berkeley in 1988, an M.Sc. degree in astronomy from the California Institute of Technology in 1990, and a Ph.D. degree in physics from Syracuse University in 1996. He joined IBM Research in 1995 and developed the PowerPC port of the GCC. He is a member of the GCC Steering Committee.

Wolfgang Gellerich
IBM Deutschland Entwicklung GmbH, Schoenaicher Strasse 220, 71032 Boeblingen, Germany (gellerich@de.ibm.com). Dr. Gellerich studied computer science and chemistry at the University of Erlangen-Nuernberg and graduated in 1993 with a Master's degree in computer science. Until 1999, he was with the programming languages group of Stuttgart University where he received a Ph.D. degree. Dr. Gellerich joined the IBM development laboratories in Boeblingen in 2000. He was with the firmware development group, where his main responsibility was the development of GNU PL8, and he recently joined the compiler team focusing on code optimization for the IBM zSeries.

Mostafa Hagog
IBM Research Division, Haifa Research Laboratory, University Campus, Haifa, Israel 31905 (mustafa@il.ibm.com). Mr. Hagog received a B.Sc. degree in computer engineering in 1998 and an M.Sc. degree in electrical engineering in 2001 from the Technion–Israel Institute of Technology. He has been with the IBM Research Lab in Haifa since 2000. His fields of interest include compilers and code optimization technologies.

Dorit Naishlos
IBM Research Division, Haifa Research Laboratory, University Campus, Haifa, Israel 31905 (dorit@il.ibm.com). Ms. Naishlos received a B.Sc. degree in computer science from the Technion–Israel Institute of Technology in 1998 and an M.Sc. degree in computer science from the University of Maryland in 2000. Since 2001, she has been with the code optimization technologies group at the IBM Research Lab in Haifa.

Mircea Namolaru
IBM Research Division, Haifa Research Laboratory, University Campus, Haifa, Israel 31905 (namolaru@il.ibm.com). Mr. Namolaru received an M.Sc. degree in mathematics from the University of Bucharest in 1985, and an M.Sc. degree in computer science from the Technion–Israel Institute of Technology in 1992. Since 1992, he has been with the code optimization technologies group at the IBM Research Lab in Haifa.

Eberhard Pasch
IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (epasch@de.ibm.com). Dr. Pasch studied mathematics at the universities of Tuebingen and Massachusetts. He received a Master's degree in 1995 and a Ph.D. degree in 1998 from the University of Tuebingen. After joining IBM in 1999, he worked in Linux development, specializing on a variety of performance problems. He is now a Senior Technical Staff Member responsible for Linux architecture and performance.

Hartmut Penner
IBM Deutschland Entwicklung GmbH, Schoenaicher Strasse 220, 71032 Boeblingen, Germany (hpenner@de.ibm.com). Mr. Penner studied computer science at the University of Kaiserslautern and graduated in 1996 with an M.A. degree. In 1996, he joined IBM and worked in firmware development, commencing shortly after the development of the zSeries back end for GCC and working on the Linux port for zSeries.

Ulrich Weigand
IBM Deutschland Entwicklung GmbH, Schoenaicher Strasse 220, 71032 Boeblingen, Germany (uweigand@de.ibm.com). Dr. Weigand studied computer science at the University of Erlangen-Nuernberg and graduated in 1994 with a Master's degree. Subsequently, he was with the department of theoretical computer science at the University of Erlangen-Nuernberg, where he received a Ph.D. degree. Since 2000, he has been a member of IBM's Linux on zSeries development group, focusing on improving the zSeries back end in GCC. He is currently one of the official code maintainers of that back end.

Ayal Zaks
IBM Research Division, Haifa Research Laboratory, University Campus, Haifa, Israel 31905 (zaks@il.ibm.com). Dr. Zaks is a manager in the Code Optimization Technologies group. He received B.Sc., M.Sc., and Ph.D. degrees in mathematics and operations research from Tel Aviv University. He joined the IBM Haifa Research Lab in 1997 and initially worked on compiler back-end optimizations for the AS/400. Later he worked on developing an optimizing compiler for the eLite DSP, spending one year at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. ■