Porting GCC to the IBM S/390 platform

Hartmut Penner *Ulrich Weigand*
IBM Deutschland Entwicklung GmbH
Schönaicher Str. 220, 71032 Böblingen, Germany
`{hpenner, uweigand}@de.ibm.com`

Abstract

IBM's mainframe architecture S/390 is the living architecture with the longest heritage, defined in a time when assembler programming was predominant and compilers were in their childhood. Hence in porting GCC to S/390 we had to cope with certain architecture features that were difficult or impossible to model in GCC's architecture-independent framework. These include 31-bit addressing mode, instruction-dependent address formats, limited availability of address displacements and immediate literals, and the condition code handling. These problems notwithstanding, the S/390 back end matured over the last couple of years to make GCC a stable and competitive compiler for the S/390 platform. In this paper we want to share how we managed to handle most of the mentioned architecture features. We also want to point out areas that promise room for further improvement in the back end itself and suggest middle-end modifications that would benefit our platform in particular.

1 Introduction

1.1 From System/360 to zSeries

In the early 1960s IBM defined the System/360 architecture. This architecture was designed to serve for a whole family of systems. The difference the distinguished systems of that family had was the way the instruction set was implemented. The System/360 architecture defined 16 32-bit general purpose registers, 4 64-bit floating point register and a 24-bit address space. Shortly afterwards, virtual addressing was added to the architecture. In 1970, System/370 was introduced, providing an enhanced instruction set. Around 1982 370/XA brought 31-bit addressing, in 1988 370/ESA introduced support for multiple address spaces. In the 1990s the ESA/390 architecture was introduced; subsequent machines added over time the relative branch instructions as well as the IEEE floating-point instruction set.

In 2000 the first IBM eServer zSeries machine came out, introducing a major architecture update. The z/Architecture remained upward compatible to ESA/390, but provided full 64-bit support, extending the general purpose register size to 64-bit and adding a 64-bit addressing mode in addition to the traditional 24-bit and 31-bit modes. This means in particular that both 64-bit and 31-bit applications can run under a 64-bit operating system (if that provides the required support). However, it is also possible to operate a zSeries machine in ESA/390 mode in order to run legacy 31-bit operating systems.

1.2 GCC S/390 port history

Within the S/390 firmware development we were searching in 1997 for a C compiler fulfilling specific requirements. We needed a compiler that could be link-compatible to the internally used pl.8 compiler, which was developed at IBM Research a decade ago. Also it should provide the ability to use embedded assembler code. One of the authors was asked to look into the then existing System/370 port of GCC, to evalute whether this could be adapted for the intended use. This port was not very stable at this time, but it could easily be shown that it could be used as a base. Since in firmware development there is no reason for backward-compatibility, we decided to set a certain level of architecture as given, and started internally with a S/390 port, producing only code for latest CMOS based systems.

When work on the upcoming Linux for S/390 port started in 1998, the compiler port developed by the firmware team could be used for the Linux port. The success of this new operating system proved to be beneficial for GCC on S/390 as well, since the Linux development team was then rapidly driving the efforts to develop the GCC port further to use the ELF linkage format and eventually to exploit the 64-bit z/Architecture. In 2001, the S/390 GCC port was finally donated to the Free Software Foundation, with the authors in charge as maintainers, one of us (Hartmut Penner) representing the S/390 hardware, the other (Ulrich Weigand) the Linux for S/390 constituency.

2 Architectural overview

Before going into details of the GCC back end implementation, we will start by giving a short overview of the relevant features of the zSeries architecture as well as the ABI used by the Linux for zSeries port.

2.1 zSeries instruction set

The zSeries architecture as a typical CISC architecture provides an extensive instruction set. It has a full set of I/O related instructions, dealing with a channel based I/O subsystem. For system programming there exists a full set of instructions which enables operation systems to retrieve all information about the system running on and do communication with a service element. The `START INTERPRETATIVE EXECUTION` instruction provides efficient virtualization capabilities, with the possibility to define very precisely which instructions are to be intercepted. Many of these architectural facilities were defined over the last 40 years, putting all the experience of the years before into the definition. However, even though the above mentioned fields are very interesting, we want to concentrate in this paper on the small subset of instructions a compiler normally deals with. For a complete reference of the ESA/390 or z/Architecture see [1] or [2].

The zSeries architecture defines 16 general-purpose registers and 16 floating-point registers. Depending on the architecture mode, the general-purpose registers have a width of 32 or 64 bits. It is a classical 2-address architecture, where for most instructions the first source operand is also used as destination. Each instruction has a length of 2, 4, or 6 bytes, and up to now more than 30 instruction formats are defined. For most ALU operations there exist two instruction types, one using two register operands (RR), the other a register and a memory operand (RX). Logical operations are also available with two storage operands (SS) or a storage and a immediate operand (SI).

More formally, the general instruction set of the zSeries architecture usable by a compiler can be divided into following classes of instructions:

RR	r1 = r1 op r2
RX	r1 = r1 op [x+b+d]
RI	r1 = r1 op ch
RS	r1 = r1 op [b+d]
SI	[b+d] = [b+d] opl cb
SS	[b1+d1] = [b1+d1] opl [b2+d2]

where we use the following elements:

r	General or floating-point register
x	Index register (register `%r1`–`%r15`)
b	Base register (register `%r1`–`%r15`)
d	Displacement, 12-bit constant (0–4095)
cb	8-bit constant, unsigned
ch	16-bit constant, signed or unsigned
op	Arithmetical or logical operation
opl	Logical operation (including move)
[addr]	Content addr is pointing to

If running in zSeries architecture mode, an address is 24, 31, or 64 bits wide, depending on the addressing mode a program operates in. The S/390 architecture mode provided only the 24-bit and 31-bit addressing modes. Here, the most significant bit of a 32-bit address is sometimes used to distinguish between 24-bit and 31-bit bit mode in 'mixed' environments. The displacement for address generation in the instruction itself is only 12 bits. Together with the fact that this displacement is unsigned, this causes some problems for defining a ABI and implementing a efficient compiler, especially when dealing with large stack frames, a downward growing stack, large GOT tables, etc. The impact of this for implementing the compiler will be shown later.

In order to provide conditional execution, zSeries uses a 2-bit condition code as part of its program status word. Most non-move or non-branch instructions, depending on the result of their operation, set this condition code. The actual value a specific instruction sets is defined for each instruction individually, and only to a certain extend a clear classification can be made. The architecture has branch instructions that decide whether a branch is taken depending on whether the current condition code equals one of the values provided in the form of a 4-bit branch condition mask as part of the instruction.

2.2 Linux for zSeries ABI

The Linux port on S/390 and zSeries uses a variant of the ELF ABI. For a full definition of the architecture-dependent parts see [3] and [4]; the following gives a short overview of the most important features. While the processor architecture does not define a stack, the ABI chooses by convention the general purpose register `%r15` for use as stack pointer. The stack grows downwards; the low 96 bytes (160 bytes on 64-bit) are reserved as register save area for use by called subroutines. Registers `%r0`–`%r5` are clobbered across function calls, while `%r6`–`%r15` are saved. Parameters are passed in registers and a parameter area on the stack.

Apart from the stack pointer `%r15`, the following general purpose registers may be used for special purposes: `%r14` holds the function call return address, `%r13` is used to point to a per-function literal pool, `%r12` points to the Global Offset Table in position-independent code, and `%r11` is used as frame pointer in functions that perform dynamic stack allocation (otherwise, the stack pointer is used as frame pointer as well).

The following short "hello world" example shows a typical 31-bit routine. Comments under each line give the semantics of the instruction, using the abbreviated syntax used by GCC in its scheduling printouts.

```
  stm  %r6,%r15,24(%r15)
# {[%r15+24]=%r6;[%r15+28]=%r7;...}
  bras %r13,.L2
# {%r13=.L1;pc=.L2}
.L1:
.LC0:  .long   .LC2
```

```
.LC1:  .long   printf
.L2:
  ahi  %r15,-96
# {%r15=%r15-96;clobber %cc}
  l    %r2,.LC0-.L1(%r13)
# {%r2=[%r13+.LC0-.L1]}
  l    %r14,.LC1-.L1(%r13)
# {%r14=[%r13+.LC1-.L1]}
  basr %r14,%r14
# {pc=%r14;%r14=pc+2}
  lm   %r6,%r15,120(%r15)
# {%r6=[%r15+120];%r7=[%r15+124];...}
  br   %r14
# {pc=%r14}
```

Note how the function prolog saves registers, sets up the literal pool pointer and allocates a new stack frame. The function proceeds to load the address of the `printf` routine as well as the address of the string constant from the literal pool and performs the call. The epilog simply restores all saved registers (thereby resetting the stack pointer and removing the current stack frame) and returns to the caller by branching to the address provided in `%r14`.

3 GCC and the zSeries architecture

While most of the features of the zSeries architecture can be easily modelled using the standard mechanisms available to a GCC back end, we have found some that require extra effort to implement correctly. This section describes how we addressed these issues in the current S/390 back end: literal handling, 31-bit addressing mode, and instruction-dependent address formats.

3.1 Literal handling

Literals, i.e. values determined at compile time, play an important role in most functions generated by a compiler. They include constant values of various types (e.g. integer, floating point, or string constants) provided in the source code as well as address constants generated by the compiler itself, used to reference code or data labels.

However, the original S/390 architecture did not provide instructions that could use literal values as immediate operands. While it was possible to load an immediate integer in the range 0–4095 into a register using the `LOAD ADDRESS` instruction, all other literal values required loading from memory.

On the other hand, accessing a memory location to load a literal from requires to express the address of that location first. Similarly, branch instructions need to be able to reference the branch target address. Again, the original S/390 architecture did not provide instructions that could use immediate address constants, neither as absolute nor as pc-relative values. The only way to specify an address, for any purpose, was to use the standard effective address generation mechanism that computes a target address as the sum of the contents of a base register, an index register, and an immediate displacement in the range 0–4095.

To overcome these restrictions, the usual coding conventions for S/390 applications required to reserve one general purpose register to always hold the address of the start of the routine currently executing. This way, targets for branches within the routine could be expressed via immediate displacement relative to that function base register, and by placing a pool of literal constants immediately adjacent to the routine's code section, the same mechanism could be used to load literals from memory.

The obvious disadvantage of this method is that it requires the total size of a routine's code section plus its literal pool not to exceed a single page (4 KB), to ensure every address within both code and literal pool remains addressable via the function base register. When these limits are exceeded, a function has to be split into multiple fragments, each consisting of up to 4096 bytes of code and literals required by the fragment. On every branch between two different fragments, the base register has to be reloaded to point to the beginning of the current fragment. This can incur significant runtime overhead.

Fortunately, over time several extensions to the S/390 architecture were implemented that provide relief to those constraints. With the second generation of S/390 machines, starting from 1992, the *relative and immediate instruction* facility provided a set of instructions that allow on the one hand the use of immediate integer constants with several operations, and on the other hand the use of pc-relative addressing modes for a number of branch instructions. However, due to the requirement that the new instructions fit within the overall scheme of S/390 instruction types, these literals were limited in range. This means that only integer values in the range of -32768–32767 are allowed as immediate operands, and pc-relative branch targets can only specific a range of up to 64 KB before or after the current instruction.

With the advent of the 64-bit z/Architecture in 2000, the latter restriction was once again loosened: the new *relative long* instructions accept pc-relative targets in a range of up to 4 GB before or after the current instruction. The `LOAD ADDRESS RELATIVE LONG` instruction finally allows the use of pc-relative addressing for other accesses besides branches.

Now, how does the GCC back end cope with those restrictions? We have chosen to support in the S/390 back end only processors that provide the relative and immediate instruction facility. This means that we can use the pc-relative branch instructions for all intra-function branches as long as the code section of the routine does not exceed 64 KB. While we also use immediate operands wherever possible, it is still necessary to maintain a literal pool for constants exceeding the allowed range. This pool is addressed via a base register (usually `%r13`) pointing to the start of the literal pool. To set up the pool base register, we use a `BRANCH RELATIVE AND SAVE` instruction, followed by the literal pool itself. Executing that instruction transfers control to the instruction following the pool, while at the same time loading the pool start address into the base register.

Address literals whose use cannot be avoided via pc-relative instructions are placed into the literal pool. However, if we are generating position-independent code for use in Linux shared libraries, we do not want to place absolute addresses into the literal pool, as those would require relocations to be applied by the dynamic loader to the text segment. This is undesirable as it prevents that page from actually being shared across multiple processes using the same library. To solve this issue, we instead place the *offset* from the start of the literal pool to the required address into the pool. Every user of that pool entry needs to add the pool start address back to that offset, which can usually be done implicitly as part of normal address generation, using the offset loaded into an index register together with the pool register as base register.

This method works fine as long as the routine's code size does not exceed 64 KB and its literal pool size does not exceed 4 KB. For the vast majority of routines these conditions hold true, which is why we have chosen to optimize for this common case. However, the compiler certainly has to be able to cope with cases where either or both of these limits are exceeded.

If a routine's code section exceeds 64 KB, determining whether the target of any particular branch within the function is out of range or not is nontrivial, as this recursively depends on the sizes of other branch instructions that lie in between. Fortunately, this analysis is performed by GCC's *branch shortening* pass, which we are able to use unmodified for our target. We simply need to provide GCC common code with information about the length of each instruction via the `length` attribute.

Once the branch shortening pass has determined which branches cannot be implemented via a pc-relative branch instruction, our machine-dependent reorganization pass replaces each of those out-of-range branches by a branch using a register as target, preceeded by an instruction loading the branch target address from the literal pool into that register:

```
l   %r14,.LCtarget-.Lpool(%r13)
br  %r14
```

As previously mentioned, when generating position-independent code, we place an offset to the branch target label into the literal pool instead:

```
l   %r14,.LCtarget-.Lpool(%r13)
b   0(%r14,%r13)
```

This replacement is simple and incurs only relatively low overhead. However, if the literal pool overflows its maximum size of 4096 bytes, things get much more difficult. Fortunately, this happens only extremely rarely; the cases where we have seen this to occur typically involve extremely large routines, unlikely to be found in source code written by hand, but sometimes occurring as a result of automatically generated code.

However, if literal pool overflow does occur, we still need to handle it correctly. What we do then is to partition the function into smaller *chunks*, each requiring a partial literal pool whose size does not exceed 4096 bytes.

At every transition between different chunks, we insert instructions to reload the pool base register with the start of the literal pool of the current chunk. Those reload instructions thus need to be inserted before the first instruction of every chunk as well as after every code label that is being branched to from an instruction located outside the chunk. Unfortunately, performing this reload operation is difficult, as we cannot use a pc-relative instruction to do so, we cannot use any arithmetical operations as those would clobber the condition code register which might be live at the point the reload is inserted, and we cannot even load anything from the literal pool because we do not know to *which* pool chunk the base register currently points—the same label might be the target of instructions residing in multiple different chunks. We solve this problem by using the following sequence of instructions:

```
basr  %r13,0
la    %r13,.Lchunk-.(%r13)
```

which resets the pool base register to the current instruction address, and adds the offset from there to the current pool chunk start address using a `LOAD ADDRESS` instruction to avoid clobbering the condition code. This technique unfortunately imposes further requirements on the pool chunks: every pool chunk must be placed within the function text section, following the corresponding code chunk, and the size of that code chunk must not exceed 4096 bytes to avoid overflowing the range of the LA instruction.

Once we've succeeded in dividing the function into chunks and inserting the pool base register reload instructions, we can then proceed to replace all references to the normal constant pool by explicit references to the current pool chunk, assuming the base register is set up properly. Position-independent code provides an additional challenge, however. Recall that in this scenario we are using offsets relative to the pool start address instead of absolute address literals. Now, when we've split the pool into multiple chunks, which pool chunk are those references supposed to be relative to? We've initially tried to set up things so that every offset is always relative to the chunk where it resides. Unfortunately this does not work, as due to constant propagation it is possible for an offset to be loaded into a register in a completely different chunk from where that register is finally used. Thus we've decided to keep the master literal pool present, even it is empty after all constants have been distributed to pool chunks, so that its start address can remain to serve as anchor for address literal offsets. To make this work, every *explicit* use of the literal pool base register `%r13` needs to be replaced by another register holding the master anchor address. That address can be computed on the fly using the current pool chunk address and an offset from the start of that chunk to the anchor; this offset is by convention always stored at the very start of each pool chunk:

```
l     %r14,0(%r13)
la    %r14,0(%r14,%r13)
```

Two final obstacles remain before literal pool splitting can be considered a general solution. The first is the fact that literal pool splitting introduces additional instructions at various points throughout the instruction stream. This can cause branch splitting information to become invalid, as some branches that were originally in-range can now exceed their allowed ranges. On the other hand, branch splitting works by placing branch target addresses into the literal pool, which can cause the pool to overflow. To solve this interdependency, we iterate branch splitting and attempting to split the literal pool until both operations succeed simultaneously. This is guaranteed to always terminate, as every branch that we decided to split at any one point will remain split forever, and thus the number of unsplit branches is strictly decreasing throughout this iterative process.

The final obstacle is that we require a temporary register for both branch splitting and literal pool splitting (for the case of anchor reloading). Fortunately, the live ranges introduced are very short, and span just the newly added instructions together with the immediately following instruction from the existing instruction stream. However, at this point in the code generation process (after reload), all registers might in fact be live at the point where we need to insert additional code. Thus, we currently reserve one register (`%r14`) for use for those purposes. Note that the ABI defines `%r14` to hold the function return address, which means it is always clobbered across function calls, but apart from that restriction the register would be free for arbitrary use inside a routine. We are not doing that, however, in order to have this register available for use in branch splitting and literal pool splitting. The only problem with that is that the decision whether we need to use `%r14`—and thus need to save and restore the register in the function prolog/epilog code—can be made only during machine-dependent reorg, long after the function prolog and epilog code was generated. Therefore, we always generate code to save and restore registers `%r13` and `%r14`, and remove that code during machine-dependent reorg once it has proven to be unnecessary.

Up to now, we have exclusively discussed code generation for S/390 machines in 31-bit mode. On z/Architecture machines, many of the problems described in this section disappear due to the availability of the *relative long* family of instructions. First of all, the `BRANCH RELATIVE LONG` instructions allow pc-relative branches within the range of 4 GB. By restricting the maximum allowed size of any single executable or shared object to 4 GB, we can thus use those instructions for nearly every branch. (The only occasion where we still might need branch splitting is in the case of `BRANCH ON COUNT` instructions, which lack a relative-long variant.)

Also, the `LOAD ADDRESS RELATIVE LONG` instruction allows us to directly load arbitrary address literals, without requiring literal pool entry, in a position-independent manner. This means that we never need to handle offsets relative to the literal pool base, and the whole issue of reloading the anchor register after pool splitting disappears. Also, as we can use `LARL` to load the literal pool start address, literal pools no longer need to reside in the text section, but can be moved to the read-only data section. This also simplifies inserting pool base reload instructions in the case of literal pool splitting. However, the core problem that the literal pool cannot exceed 4096 bytes remains.

The solution described in this section allows GCC to correctly handle every valid source code, even if it causes code or literal pool sizes to exceed their optimum limits. However, there is still a lot of room for improvement to optimize the code that is generated once that overflow happens. We are currently working on some minor improvements. In particular, we'll remove the whole complex of pool anchor reloading for position-independent code by representing address literals as offsets relative to the gobal offset table (like on other platforms) instead of relative to the literal pool. This requires some new relocation types to be implemented in binutils first. Once this is done, we can try to finally make register `%r14` available for regular use. This would require that every branch instruction reserves one register to be used for branch splitting if necessary, but even so overall register pressure should benefit.

The major problem with optimizing literal pool overflow situations, however, is to determine how to split the function into chunks. An optimal solution here would try to minimize the frequency of inter-chunk branches at run time. To try to tackle that problem will require control flow data including basic block boundaries and branch probabilities; unfortunately GCC currently no longer maintains that information at the point in time where literal pool splitting has to be performed (in machine-dependent reorg).

3.2 31-bit addressing mode

For historical reasons, the S/390 architecture does not have a 32-bit addressing mode, but uses 31-bit addressing. This means that while base and index registers used in address generation are regular 32-bit registers, the most significant bit is ignored when computing the effective address. (Note that this does not apply on zSeries in 64-bit addressing mode; most of the problems discussed in this section disappear in that environment.)

For the compiler, this causes two issues that need to be considered. As for every 31-bit address there are two equally valid 32-bit pointer representations, one with the high bit set and one with the high bit cleared, care must be taken when comparing pointer values for equality. To simplify this process, GCC tries to always represent pointers using the representation with the high bit cleared. However, some machine instructions store address values with the high bit set; most importantly the `BRANCH AND SAVE` family of instructions does so. A `BAS` instruction transfers control to another address, and at the same time stores the current instruction address (with the high bit set) into a register. GCC uses those instructions for two purposes: to implement function calls, and to set up the literal pool. Since both the call return address and the literal pool start address are normally used only for compiler-internal purposes, GCC does not bother to normalize these values by clearing the high bit. However, in some cases these values are visible externally, and extra care needs to be taken:

- The call return address can be retrieved by doing a stack backtrace, e.g. via the function `__builtin_return_address`. This will yield values with the high bit set, which the caller needs to normalize; this is handled by the `__builtin_extract_return_address` function. However, as this built-in does nothing on most platforms, we have seen several cases where applications didn't work on S/390 because they forgot to use it.
- The literal pool start address is used as anchor to compute the addresses of local variables in position-independent code. As these can be externally visible, the compiler needs to make sure this address computation will normalize the resulting pointer. This is done by using an `UNSPEC` operation that enforces the use of `LOAD ADDRESS` (instead of, say, a normal 32-bit addition operation) to perform the calculation. The `LA` instruction will always return a 31-bit value with the high bit cleared.

The second main problem caused by the 31-bit addressing mode is that address generation is a distinctly different operation from regular addition. As mentioned above, the `LOAD ADDRESS` instruction performs a 31-bit addition operation, adding the values of base and index register and an immediate displacement, and returning a 31-bit value. The `ADD` instruction, in contrast, performs a full 32-bit addition operation. The decision whether to use `ADD` or `LOAD ADDRESS` needs to take into account a number of issues:

- We *must not* use `LOAD ADDRESS` to perform integer addition, as the high bit of the result is not computed.
- Where the result of an addition operation is used as address, we can use `LOAD ADDRESS`, and it is in fact often the preferred method to minimize pipeline stalls.
- Some passes of the compiler (reload) insert address computation operations into the instruction stream, making the implicit assumption that they do not clobber the condition code. We *must* use `LOAD ADDRESS` in these cases.
- In some cases, in particular when computing local addresses in position-independent code (see above), we rely on the property that `LOAD ADDRESS` clears the high bit, so we must not use regular addition instead.

This has been a problematic area during the development of the S/390 back end; we have tried various ways of simultaneously meeting all these requirements, not always completely successfully. As an example for the difficulties involved, consider the question whether there should be an `LA` pattern that accepts all RTL instructions of the form `(set (reg) (plus (reg) (reg)))`. If this pattern exists, there is the danger that it might be incorrectly used to implement an integer addition. If it does not exist, there is the danger of reload failures as reload will create such instructions anyway. The current S/390 back end tries to solve this as follows:

- The `add` instruction patterns accept insns that explicitly clobber the condition code.

- The `la` instruction patterns accept insns that do not clobber the condition code, provided that it is safe to assume the result is being used as an address. This assumption can be made if one of the registers involved is the literal pool base register, the global offset table base register, or is known to point into the stack frame (stack register, frame register, argument pointer register etc.). The instruction will also accept addresses using an `UNSPEC` to enforce clearing the high bit.

- A second set of `forced_la` patterns accept all syntactically valid load address insns, without employing the sanity check mentioned above. Those use a special pattern that will never be accidentally generated by other parts of the compiler (e.g. combine), so that those patterns will only match in case they were explicitly generated by the S/390 back end.

- When reload tries to load a `plus` expression that would not be accepted by a regular `la` pattern, this is handled via the secondary input reload mechanism. This means that the `reload_insi` expander is called, which in turn will compute the address using `forced_la` patterns if necessary. That way, reload will never fall back to generating add operations by itself.

- To optimize for using `LA` where possible, a set of peephole2 patterns tries to transform `add` instructions into `la` instructions. This is only done when considered profitable.

A completely different option to solve the 31-bit addressing mode problems might be to employ the `PSImode` mechanism to explicitly represent a 31-bit data type. However, we have tried this solution and found that it typically generated less efficient code due to superfluous `SImode <-> PSImode` conversions inserted at various points by the middle end. Improving the `PSImode` support might make this option viable at some point in the future, though.

3.3 Instruction specific address formats

A fundamental assumption of GCC, in particular the reload pass, used to be that memory addresses are represented in the same format in all instructions. This means that if a particular RTL expression represents a valid address for one instruction, it is supposed to be valid for *all* other instructions as well. The most important place where this assumption is made is the `find_reloads` routine. This routine is supposed to check whether an RTL instruction matches the constraints imposed by the insn pattern, and if it doesn't, determine the most efficient way to modify the instruction stream by inserting additional reload insns to correct the problem. In doing so, `find_reloads` first tries to make sure that all memory addresses mentioned in the instruction are valid. This pass is performed in the same way for all instructions, and does not even look at the constraint string. This means there is no way to impose different conditions as to whether a memory address is valid or not, depending on which instruction is involved.

Unfortunately, the S/390 architecture uses two different formats to specify memory addresses in instructions. The most general address format allows to to specify a base register, an index register, and a displacement (in the range of 0–4095). These are added up to compute the effective address. Some other instructions, however, do not allow the use of an index register; instead, they compute the effective address simply as the sum of a base register and the displacement. (The two formats are commonly called X and S instruction operands, respectively.) However, the back end has only two choices when asked to validate an address RTX: either to never accept addresses with index register, or to always accept them. The first option causes very inefficient code to be generated, while the second option can potentially cause invalid operands for S-type instructions to be produced.

We have tried various ways of coping with this problem, but with limited success. It is possible to try to avoid invalid S-operands by checking for their presence in the instruction predicate of affected instruction patterns. However, this is not reliable, as an address operand that initially does not use an index register can be modified into one that does by the reload pass, e.g. due to register elimination or displacement overflow. While we could in addition to the predicate use a constraint letter to check for valid S-operands, this does not solve the problem: if a non-standard constraint does not match, reload will not know how to fix the problem, causing compilation to abort. We were able to overcome this by relying on undocumented—and arguably incorrect—behaviour of reload when interpreting the 'o' constraint; but this hack was not only fragile, it also didn't allow full flexibility in generating efficient code.

We finally solved this issue by introducing two new features to the reload pass, starting with GCC version 3.3—the `EXTRA_MEMORY_CONSTRAINT` and `EXTRA_ADDRESS_CONSTRAINT` target macros. These were inspired by the way reload was able to handle *offsettable* memory constraints. A memory operand is called offsettable, if it stays a valid memory operand when a small additional displacement is added to the address, so that every byte of the object comprising the operand can be addressed. As an example, the RTX

```
(mem:DI (plus:SI (reg:SI 1 %r1)
                 (const_int 4092)))
```

is a valid memory operand on S/390, but it is not an offsettable operand, because only the initial four bytes of the `DImode` operand are addressable before the displacment exceeds the maximum value of 4095. In some cases, instructions cannot accept non-offsettable operands, and GCC allows to specifc this using the 'o' constraint letter. If, after reload has performed all required modifications, a memory address marked with that constraint turns out to be non-offsettable, reload will generate a load-address operation to reload the address into a single register; this register can then be used as offsettable memory operand.

The `EXTRA_MEMORY_CONSTRAINT` target macro now allows the back end to specify other classes of memory operands that require similar treatment by reload. By declaring that a constraint letter describes an extra memory constraint, the back end promises that `EXTRA_CONSTRAINT`, when called to verify whether an expression satisfies this constraint, will:

- accept only memory operands, and
- accept all memory operands whose address consists of one single base register.

This allows the reload pass to handle such operands correctly: if a memory operand does not pass the `EXTRA_CONSTRAINT` check, reload is able to fix the problem by loading the address into a base register. Similarly, the `EXTRA_ADDRESS_CONSTRAINT` target macro allows the back end to define constraints that work like the standard 'p' constraint to denote address operands, but accepts only a subset of all valid addresses (again including all those that consist of solely a base register so that reload can fix the operand up if required).

The `EXTRA_MEMORY_CONSTRAINT` macro is used by the S/390 back end to define the 'Q' constraint to handle S-operand instructions; this allows the use of these instructions without abusing reload, and also provides flexibility to mix S-operand instructions with others in the same instruction pattern, choosing the best alternative depending on the specific situation. The `EXTRA_ADDRESS_CONSTRAINT` macro could be used by the S/390 back end to implement the full range of options to specify the count operand for shift instructions (this is not currently implemented yet, however).

4 Performance considerations

The previous section described issues relating to correctness of the generated code which required special handling. However, for GCC to be a competitive compiler on the zSeries platform, we need to not just generate correct, but also efficient code. This section details two areas where we found we could achieve significant performance benefits by exploiting specific features of the zSeries architecture: condition code handling and instruction scheduling.

4.1 Condition code handling

The S/390 architecture uses a *condition code* to implement conditional branches. The condition code consists of two bits stored in the program status word. Various arithmetical, logical, and comparison instructions set the condition code, while branch instructions make use of it to decide whether the branch is to be taken or not. As opposed to many other platforms, the S/390 condition code is not composed of single bits with specific semantics. Instead, the two bits of the condition code combine to represent a condition code value in the range 0–3. Branch instructions use a 4-bit branch condition mask to decide whether branching is performed. The current condition code selects one of the four mask bits, and if this bit is one, the branch is taken. The relationship between the condition code value and the mask position is given by the following table:

Condition Code	Mask Position Value
0	8
1	4
2	2
3	1

For example, the instruction `bcr 12,%r1` branches to the address given in register `%r1` if the current condition code is either 0 or 1. (The GNU assembler also accepts mnemonics instead of explicit mask values; as this branch typically represents a *less-or-equal* decision, it can equivalently be written as `bler %r1`.)

However, the numerical values 0–3 the condition code can assume have no fixed meaning. Instead, every instruction that sets the condition code is free to define the semantics of the condition code values it may set. In early versions of the S/390 back end we therefore used only the condition codes set by explicit comparison instructions (which are very regular), and completely ignored that other instructions may set the condition code as side effect of some other operation. This works, but can obviously cause code to be generated that is significantly less efficient. In particular, some important instructions the S/390 architecture provides (e.g. `TEST UNDER MASK`) could not be exploited at all.

To improve this situation, we have rewritten the condition code handling parts of the S/390 back end to use an explicit `CCmode` register to represent the condition code (instead of using `cc0`). The various different semantics that instructions can impose on the condition code values are represented via different machine modes of that register. The following list tries to give an overview of the typical uses of the condition code:

- Comparison operations (signed)

0	Operands equal
1	First operand low
2	First operand high
3	Operands unordered (floating point)

This condition code semantics is represented by the `CCSmode` mode. It is used by instructions like `COMPARE`; some other instructions (e.g. `LOAD AND TEST`, `SHIFT RIGHT SINGLE`) set their condition code according to this mode as well, assuming an implied comparison of their single operand against zero.

• Logical comparison operations (unsigned)

0	Operands equal
1	First operand low
2	First operand high
3	n/a

This condition code semantics is represented by the `CCUmode` mode. It is used by the `COMPARE LOGICAL` family of instructions.

• Arithmetical operations

0	Result zero; no overflow
1	Result less than zero; no overflow
2	Result greater than zero; no overflow
3	Overflow

This is used by the `ADD` and `SUBTRACT` instructions. Unfortunately, due to the fact that the case of signed arithmetic overflow is signalled via condition code 3, and in that case no comparison of the result against zero is performed, in most cases we cannot use the condition code set by those instructions. However, if one of the operands is a compile-time immediate constant, we may be able to determine at compile-time that if the operation overflows, the result *must* always be greater or less than zero, respectively. Those situations are represented by the `CCAPmode` and `CCANmode` modes. (Note that some languages, like C, guarantee that arithmetic on signed data types must not overflow. Unfortunately, this information is lost at the RTL level. Having some means to pass this fact to the back end would enable us to make use of the `ADD` condition code in many more cases.)

• Logical operations

0	Result zero; no carry
1	Result not zero; no carry
2	Result zero; carry
3	Result not zero; carry

This is used by `ADD LOGICAL` and in slightly modified form by `SUBTRACT LOGICAL`; we represent these cases by the `CCL1mode` and `CCL2mode` modes. We use the logical variants of the add and subtract operations in cases where the result of the operation is compared against zero, and we are not sure whether overflow happens. They can also be used to implement carry propagation for multi-word additions.

• Zero test

0	Result zero
1	Result not zero
2	n/a
3	n/a

The logical operations (`AND`, `OR`, `EXCLUSIVE OR`) use these condition code semantics, which we represent by `CCTmode`. What is important here is that some of the condition code modes mentioned above can also be used to implement a test against zero (e.g. `CCSmode, CCUmode`). We therefore implement such tests using a virtual condition code mode `CCZmode` that is allowed to match against all such modes, using a semantics of condition code 0 if result equals zero, and condition code nonzero if the result is nonzero.

The condition codes described above are all used by a number of different instructions, and share a certain amount of regularity. However, other instructions use the condition code in completely different ways. As an example we describe here an important instruction of the S/390 architecture, `TEST UNDER MASK LOW`, and how we can make use of this instruction within the GCC framework. `TEST UNDER MASK LOW` takes the low 16 bits of a register operand and compares them bit-for-bit against a mask provided as immediate operand. The sole effect of the instruction is to set the condition code, depending on whether the operand bits selected by the mask are ones or zeros:

0	Selected bits all zeros; or mask bit all zeros
1	Selected bits mixed, and leftmost is zero
2	Selected bits mixed, and leftmost is one
3	Selected bits all ones

This instruction is very useful to generate efficient code for a number of frequently used bit-test operations. The following `if` statement, for example:

```
if ((flags & 0x80) &&
    !(flags & 0x4))
```

can be translated into a single `TEST UNDER MASK LOW` operation followed by a conditional branch:

```
# Mask selects both 0x80 and
# 0x04 bits for testing
tml  %r1,0x84
# Branch if leftmost bit is one,
# and the other zero
brc  2,.Lxxx
```

Starting with GCC 3.3, the S/390 back end is in fact able to generate this optimal code sequence. This is made possible by the fact that the combiner pass notices the two subexpressions of the `if` clause can be combined into

```
if ((flags & 0x84) == 0x80)
```

The S/390 back end now uses the `SELECT_CC_MODE` macro to inform combine that it is possible to implement this particular comparison operation using the `CCT2mode` mode, causing the following (simplified) instruction sequence to be emitted:

```
(set (reg:CCT2 33 %cc)
     (compare:CCT2
       (and:SI (reg/v:SI 40)
               (const_int 132 [0x84]))
       (const_int 128 [0x80])))

(set (pc)
     (if_then_else
       (ne (reg:CCT2 33 %cc)
           (const_int 0 [0x0]))
       (label_ref 18)
       (pc)))
```

These in turn later generate the assembler code shown above. Note that the use of `CCT2mode` causes the branch instruction to use condition code 2 for equality (and all other condition codes for inequality); this is very different from how most other branches are handled.

Overall, the `CCmode` facilities of the GCC middle end allow to make use of the S/390 condition codes in many important cases; no changes outside the S/390 back end were necessary to exploit them. However, we have noticed some areas where common code changes would be required to further improve the generated code. One of these is to allow a condition code computed by one instruction to be reused across multiple branches; the sequence

```
if (x == 5)
  ...
else if (x < 5)
  ...
```

currently performs two distinct comparison operations, although the optimal implementation would use a single `COMPARE` to set the condition code, followed by two branch instructions evaluating it.

4.2 Instruction scheduling

The time required to run a certain program depends on the number of instructions and the time each specific instruction takes. Besides that, in most modern implementations of computer architectures, dealing with a pipelined and/or superscalar processor implementation, the cycles an instruction takes as part of an instruction stream depends heavily on the issue order. For some architectures (e.g. VLIW) an inappropriate scheduling of instructions will lead to a significant performance decrease.

Also on the recent z900 machines, some of the single-cycle instructions will in fact take from 1 to 5 cycles, depending on the order this instruction is issued within an instruction stream. The reason for this can easily be seen if we take a close look at the single-issue pipeline all instructions are executed on. (See [5] for a more detailed description of the z900 pipeline.)

After instruction fetching, the instruction pipeline consists of 6 stages. This pipeline is designed so as to ensure that register-memory (RX) instructions perform the best way possible.

DC Decode instruction, latch registers for address generation.

AA Address generation, by adding base, index register and displacement from instruction text.

C1 Cache access, TLB access.

C2 Send memory data to execution unit.

E1 Execute.

WR Writeback result to register file.

Regardless whether an instruction actually uses a memory operand or not, latching of base and index registers is done in the decode stage. Likewise, the address generation stage as well as the C1 and C2 stages are used for all instructions, even though they would be required for memory operands only. Together with the fast L1 data cache, this enables register-memory instructions to be as fast as register-register instructions.

Due to the single cycle E1 stage for most simple instruction, true data dependency does not cause a pipeline stall. This leads to a theoretical cpi of 1 for most compiler generated instructions, assuming an infinite cache. Also, since this pipeline is short, the penalty for mispredicted branches is comparatively small.

The main instruction-issue related problem left by this design is the address generation interlock (AGI). If a register used in the AA stage (e.g. base register) is changed in an instruction shortly before, the pipeline will be stalled for up to 4 cycles. This is due to the fact that the AA stage needs to wait for the WR stage to update the register needed.

(Please see Figure 1.)

This AGI lets most applications suffer a performance degradation in the double-digit percentage range. If we look at code examples like the PLT code generated for ELF shared libraries, the impact is even bigger. Over the last generations of S/390 systems attempts to reduce this impact led to building certain kinds of bypasses into the pipeline. Especially the *load* and *load address* type instructions, which generate all their side-effects in the early stages of the pipeline and which are frequently used in pointer intensive code, got those bypasses. The result of a *load address* type instruction is generated in the AA stage and ready after C1, and can be bypassed with a 1 cycle delay to the AA stage of a directly following instruction.

```
                 0  1  2  3  4  5  6  7  8  9  10 11
ar r2,r3         DC AA C1 C2 E1 WR
l  r2,0(0,r2)       DC             AA C1 C2 E1 WR
ar r4,r2                           DC AA C1 C2 E1 WR
```

Figure 1: Address Generation Interlock, first example

```
                 0  1  2  3  4  5  6  7  8  9  10 11
la r2,0(r2,r3)   DC AA C1 C2 E1 WR
l  r2,0(0,r2)       DC    AA C1 C2 E1 WR
ar r4,r2                  DC AA C1 C2 E1 WR
```

Figure 2: Address Generation Interlock, second example

(Please see Figure 2.)

The result of a *load* type instruction is ready after the C2 stage and can be bypassed with a 2 cycle delay to the AA stage of a directly following instruction.

(Please see Figure 3.)

All other instructions suffer a 4 cycle penalty if setter and user are issued back to back. To avoid this, we use in the recent GCC implementation the new DFA based scheduler.

To describe the behavior of the pipeline, we only need to define the last two stages. Down below we shortly show part of description of the z900 pipeline.

```
(define_automaton "z_ipu")
(define_cpu_unit "z_e1"   "z_ipu")
(define_cpu_unit "z_wr"   "z_ipu")

(define_insn_reservation "z_la" 1
  (and (eq_attr "cpu" "z900")
       (eq_attr "type" "la"))
  "z_e1,z_wr")

(define_insn_reservation "z_load" 1
  (and (eq_attr "cpu" "z900")
       (eq_attr "type" "load"))
  "z_e1,z_wr")

(define_insn_reservation "z_int" 1
  (and (eq_attr "cpu" "z900")
       (eq_attr "atype" "reg"))
  "z_e1,z_wr")

(define_insn_reservation "z_agen" 1
  (and (eq_attr "cpu" "z900")
       (eq_attr "atype" "agen"))
  "z_e1,z_wr")
```

The 4-cycle hazard of the pipeline due to AGI, the 1-cycle bypass for the *load address* type instructions and the 2-cycle bypass for *load* type instructions are described using the `define_bypass` construct.

```
(define_bypass 5 "z_int,z_agen"
        "z_agen,z_la,z_load" "s390_agen_dep_p")

(define_bypass 3 "z_load"
        "z_agen,z_la,z_load" "s390_agen_dep_p")

(define_bypass 2 "z_la"
        "z_agen,z_la,z_load" "s390_agen_dep_p")
```

With all this in place, GCC does a good job scheduling within a basic block. The places where we still see for certain code a non-optimal scheduling are as follows:

At the beginning of a basic block, the state of the DFA is reset. With GCC 3.4, the second scheduling pass is placed after basic block re-ordering. Since the reordering will lead to a high probability that a basic block is entered from the immediately preceding basic block,

```
                  0  1  2  3  4  5  6  7  8  9  10 11
l  r2,0(0,r3)     DC AA C1 C2 E1 WR
l  r2,0(0,r3)        DC       AA C1 C2 E1 WR
ar r4,r2                      DC AA C1 C2 E1 WR
```

Figure 3: Address Generation Interlock, third example

this could be used to improve scheduling. Instead of resetting the state at the beginning of the basic block, the state from the end of the last basic block scheduled could be used as initial state.

This uncovers another problem with the current way the DFA is defined. The `define_bypass` mechanism only influences `insn_cost`, which is used to set up the priority a insn is scheduled with. Also `insn_cost` is used to find out when a insn is ready, depending on the instructions already scheduled in the current basic block. However, this information is not actually part of the state of the DFA itself, and due to that the detection of AGI hazards cannot be achieved solely by looking at this state.

If GCC will use more and more of the DFA-based algorithms for scheduling, like global scheduling, the DFA should be built to model all resources. In our specific case, in order to detect AGIs, this needs to include the general register file. To model the AGI behaviour, we need to define a RR type instruction allocating the source register in the E1 stage and allocating the destination register in the AA, C1, C2, E1, WR stages. A RX type instruction allocates the address registers in the AA stage, the source register in the E1 stage and the destination register in the AA, C1, C2, E1, WR stages. In case of a *load* type instruction the destination register is only allocated in the AA, C1 and C2 stage, for a *load address* type instruction in the AA and C1 stage. Having this in place, the DFA would be sufficient for detecting the AGI hazard.

This all would need some kind of new syntax, in order to refer to the registers an instruction is using. Also, it would definitely not work before register allocation, since the number of states and transition could not be handled. Even after register allocation, it remains to be seen whether the the number of states and transisiton is managable. In our case, each instruction may use up to 16 registers, and will use up to two for addressing.

5 Conclusion

GCC on the IBM mainframe is a mature compiler that is in widespread use as the system compiler for all Linux on zSeries distributions. The efficiency of the generated code is competitive with other compilers for our platform.

However, there is still room for improvement. We will continue to work on the S/390 back end in order to fully exploit all features the architecture provides. We also remain committed to add support for future generations of the zSeries processor as soon as those become available.

References

[1] *ESA/390 Principles of Operation*, IBM Document Number SA22-7201-07, 2000. `http://publibfp.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/dz9ar007`

[2] *z/Architecture Principles of Operation*, IBM Document Number SA22-7832-01,

2000.
`http://publibfp.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/dz9zr001`

[3] *LINUX for S/390 ELF Application Binary Interface Supplement*, IBM Document Number LNUX-1107-00, 2001.
`http://oss.software.ibm.com/linux390/docu/l390abi0.pdf`

[4] *LINUX for zSeries ELF Application Binary Interface Supplement*, IBM Document Number LNUX-1107-00, 2001.
`http://oss.software.ibm.com/linux390/docu/lzsabi0.pdf`

[5] E.M. Schwarz et al. *The microarchitecture of the IBM eServer z900 processor*, IBM Journal of Research and Development Vol. 46 No 4/5, 2002.