Addressing Mode Selection in GCC

Naveen Sharma
HCL Technologies Ltd.
naveens@noida.hcltech.com

Rakesh Kumar
HCL Technologies Ltd.
rakeshku@noida.hcltech.com

Abstract

GCC has no formal addressing mode selection mechanism. It uses target hooks to generate valid addressing modes for a target. However, a significant amount of high level information is destroyed while doing this, especially for targets lacking a rich set of addressing modes. This leads to poor aliasing, and subsequently poorer CSE, GCSE, and scheduling. Hence, an unoptimal object code. This paper proposes an abstraction over RTL to generate machine independent addressing modes to achieve better aliasing. The actual addressing modes of the target are exposed after the first scheduling pass, where they are selected based on current execution scenario. Inter block address inheritance is also done at this point. The idea can be extended to specify a general "mid-level" RTL for GCC.

1 Introduction

1.1 Addressing Modes

In simple terms, addressing modes specify the way instruction operands are chosen at run time. In most general purpose machines, an addressing mode can specify a location in memory, a register or a constant/literal. This paper talks about addressing modes mainly in context of memory loads and stores i.e. operations which move data between registers and memory. In both operations, the destination gets affected; the source is not changed. When discussing about memory, the effect of Memory Management Unit can be ignored, since we are concerned with the addresses generated by the compiler.

Architectures have wide variance of features while considering addressing modes of a machine. Every processor, based on its application domain, has its unique set of addressing mode features. This choice is usually a function of various parameters like register set, instruction size and alignment restrictions. The most common addressing modes for loads/stores on a typical RISC architecture are:

- Displacement addressing mode: It is used when data is at known offset from some base address in a register.

```
mov.l @(4, r1),r2
```
[1]

- Register Indirect addressing mode: It is used when memory address of the required data is taken from register.

```
mov.l @r1,r2
```

- Register Index addressing mode is used when the exact offset from a base address is not known.

```
mov.l @(r0,r1),r2
```

[1]The assembly snippets correspond to SH4 processor.

- Auto Increment/Decrement modes: They combine memory access and address arithmetic.

```
mov.l @r1+,r2 !post-inc
mov.l r1,@-r2 !pre-dec
```

Some processors also allow pc-relative loads, where effective address is relative to the current point of execution. The above mentioned modes can also occur with some restrictions e.g. the SH4 processor allows only 4-bit displacement in displacement addressing mode.

1.2 Addressing Mode Selection

The compiler owns the responsibility of producing optimized code that exploits the features of target processor. The optimal choice of addressing modes aims at reduced code size and increased performance. GCC traditionally uses `RTL` as its intermediate language. Although `RTL` representation is machine independent, the `RTL` actually generated for a target is machine dependent. This is because `RTL` is generated directly from information in the machine description file. The machine description contains the description of exact instruction set of the target. The `RTL` can therefore be described as low-level intermediate form(or target `RTL`). This form is not very suitable for several high level/mid-level optimizations. The `tree-ssa` work overcomes the difficulty to a large extent by defining a new high level intermediate form. But some sort of mid-level `RTL` would is desirable for effective optimizations by GCC's `RTL` optimizer. In one sense, the notion of infinite pseudo registers can be considered a mid-level `RTL` abstraction.

We propose that addressing modes can also be abstracted as part of mid-level `RTL`. The advantages would be:

- Several initial `RTL` optimization passes would be able to perform better.

- Addressing Mode Selection based on execution scenario is likely to be better as address arithmetic is reduced.

1.3 Address Inheritance Problem

Every addressing mode has its associated cost. This cost could be evaluated in terms of pipeline characteristics of the processor, the instructions involved in address arithmetic, or the cost imposed by the target design. The concept of address inheritance encourages the reuse of address calculations. It states that wherever possible, the side effects of address arithmetic instructions should be carried forward, so that there are no recalculations at the point of next load/store operation. It looks similar to `CSE/GCSE` but there is a subtle difference. `CSE/GCSE` look at exact expressions.They do not know the relationship between two expressions of the form `reg+k1` and `reg+k2`, where `k1, k2` are constants. The fact that these can possibly be derived from each other in target specific way is out of scope for their functionality.

Address inheritance can be viewed as function with two parameters—time and space. Spatially related addresses are those which do not alias and which can be accessed from the same base without address arithmetic. The temporal local addresses are those which do not alias and which are separated by minimum number of instructions. Both can assume two attributes—`near` and `far`. Spatially related addresses represents a range of addresses allowed in reg+displacement addressing mode. Temporal relation is determined by number of available registers and control flow graph.

Time	Space	Inherit(Yes/No)
near	near	Yes
near	far	Sometimes Possible
far	near	Register Pressure Issues
far	far	No

Architectures with relatively more number of registers are potential candidates for "far & near" combination, whereas architectures having more number of bits reserved for offset in displacement addressing mode are potential candidates for "near & far" combination.

2 Problem Description

The machine description files are used to generate target-IL at the time of RTL generation. As already emphasized, though RTL representation is machine independent, but its generation is machine dependent. GCC imposes the restriction that every pass should generate valid target-RTL. This strategy hampers the addressing mode optimization, since subsequent passes are more restricted.

2.1 The Current Scheme

In the current situation, GCC relies on several target macros. It uses GO_IF_LEGITIMATE_ADDRESS to verify all memory address related changes. During RTL generation, the macro LEGITIMIZE_ADDRESS, is used to break large offsets to valid target-RTL form. When using one addressing mode, GCC queries whether the chosen mode is too expensive for the target. It uses the target hook TARGET_ADDRESS_COST to compute the cost of an addressing mode. Targets define TARGET_ADDRESS_COST as simple heuristic values. The hook exhibits a limited form of cost model for addressing mode choice, but it is not a complete framework and certainly misses optimal choice in most cases.

```
{
  i = 234;
  a[i] = 12123;
  ...
  i = 290;
  a[i] = 12123;
  ...
  i = 236;
  a[i] = 12123;
  ...
  i = 228;
  a[i] = 12123;
  ...
}
```

Figure 1: Random Accesses in an Array

```
      mov.w .L3,r1
      mov.w .L4,r0
      mov.l r1,@(r0,r14)
      mov.w .L5,r0
      mov.l r1,@(r0,r14)
      mov.w .L6,r0
      mov.l r1,@(r0,r14)
      add #-32,r0
      mov.l r1,@(r0,r14)
.L3:
      .short 12123
.L4:
      .short 936
.L5:
      .short 1160
.L6:
      .short 944
```

Figure 2: Output without AMS for Figure 1

Figure 1 illustrates some aspects of addressing mode selection problem.

Figure 2 shows the code generated by current implementation of GCC. There are few points noteworthy here:

- The value of *i* is known at compile time,

```
1       mov.w   .L3,r1
2       mov.w   .L4,r2
3       add     r14,r2
4       mov.l   r1,@(24,r2)
5       mov.w   .L5,r0
6       mov.l   r1,@(r0,r14)
7       mov.w   .L6,r0
8       mov.l   r1,@(32,r2)
9       mov.l   r1,@r2
10  .L3:
11     .short 12123
12  .L4:
13     .short 912
14  .L5:
15     .short 1160
```

Figure 3: Output with AMS for Figure 1

but copy propagation could not take advantage of it due to target restrictions.

- Since GCC assumes *i* is not known at compile time, it has chosen register index addressing mode. It could have done better as Figure 3 shows.

- There are three related addresses[2] in the snippet, viz. `a[228], a[234], a[238]`; but GCC is unable to recognize the fact.

- Extra pc-relative loads are generated.

The optimal assembly for the above snippet is shown in Figure 3. The line numbers are not part of assembly; these are kept for further reference.

With this improvement, even in this trivial example, we get 4 bytes of code size reduction, lesser stress on `r0`, the only index register available on SH4, and one fewer PC-rel load.

[2]The notion of related addresses is explained in the next subsection.

The selection of the optimal addressing modes with minimal code size and minimal execution time depends on many parameters and is NP complete in general[Eckstein]. One important criteria for choosing appropriate mode is the execution scenario. The choice which seems to be best in one scenario may prove to be unoptimal in another execution sequence. For example in Figure 3, dual register indirect addressing mode is used in line 6. Note that `r0` suffices many needs on SH4, and it is generally advisable to avoid the use of `r0` wherever possible. Still, this mode is the best choice in this execution sequence. The other choice left is register-indirect which would generate the sequence

```
mov.w   .L5,r3
add     r14,r3
mov.l   r1,@r3
```

The former choice is better since it is saving one address arithmetic instruction. The above example shows the choice of addressing mode should be determined by the execution scenario. Hence, it should be decided flexibly, and not rigidly as done in GCC currently.

2.2 Address Inheritance in GCC

GCC implements address inheritance in limited form through two passes—`regmove` and `reload_cse`. `regmove` intents register to register copy elimination. As a side effect, it does the following transformation:

```
pX<-pA+N |   pX <- pA + N
...      |->   ...
pX<-pA+M |   pX <- pX + (M - N)
```

This transformation is an address inheritance transformation as the the address computed in `pX` is reused subsequently. `regmove` is ineffective in several cases because:

- CSE and GCSE both run before regmove, and they attempt to optimize address arithmetic prior to regmove. They pull address calculations near basic block boundaries where regmove cannot optimize them.
- regmove pass cannot see beyond basic blocks and is unable to propagate information across basic blocks.
- regmove is able to do the required transformation only for SImode accesses for SH4.

reload_cse is simple CSE pass over hard registers after reload. The functions of reload_cse include:

1. It eliminates no-op moves where two different registers are assigned to the same hard register, and then copied one to the other.
2. It detects cases where we load a value from memory into two registers, and changes it to simply copy the first register into the second register if memory is more expensive than registers.
3. It scans the operands of each instruction to see whether the value is already available in a hard register. If possible, it replaces the operand with the hard register.

2.3 Alias Analysis

Several passes need alias information for doing effective optimizations. Alias information is most important for passes like CSE, loop invariant code motion, instruction scheduling, and register allocator. GCC can successfully determine aliasing between two memory references if they

```
void foo (float *a, float *b)
  {
    a[17] = a[0] + a[18];
    b[17] = b[1] + a[18];
  }
```

```
mov    r4,r2
add    #72,r2
fmov.s @r2,fr2 !Load a[18].
mov    r4,r3
fmov.s @r4,fr1 !Load a[0].
add    #68,r3
fadd   fr2,fr1 !Add.
fmov.s fr1,@r3 !Store a[17].
fmov.s @r5,fr1 !Load b[0].
fmov.s @r2,fr2 !Load a[18] again.
fadd   fr2,fr1
fmov.s fr1,@r1 !Store b[17].
```

Figure 4: The Alias problem

- use distinct constant offsets from the same register
- one of them points to stack

For machines that do not have "reg + displacement" addressing mode, pointer arithmetic is necessary to compute a pointer to the desired address. GCC lacks the mechanism to determine aliasing between such computed pointers[Sanjiv]. Consider the code in Figure 4. The Figure 4 also shows the corresponding SH4 assembly with -O2 option.

Since SH4 doesn't support "reg + displacement" addressing mode for floats, GCC alias analysis mechanism fails. Hence CSE is unable to determine if a value can be retained in a register across a write and a[18] is loaded twice.

3 Solution Strategy

3.1 Designing an Abstraction over RTL

It is desirable to have some sort of abstraction to hide target addressing modes to eliminate problems highlighted in the previous sections. We initially pretend some standard high level addressing modes. The change to target's addressing mode is done in a separate pass after first scheduling pass. The scheduler can do better load/store scheduling with abstract modes. There is a new macro called `TARGET_USE_ABSTRACT_MODES`. If this is nonzero, this will force the front end to generate memory references with following abstractions.

- Infinite displacement (natural register size) for register+offset addressing mode.
- Dual register indexed mode with two general purpose pseudos—i.e., @(rm, rn)—is supported.
- Auto-ionic modes are disabled as they effect the scheduling adversely.

3.2 Addressing mode selection pass

The addressing mode selection pass (AMS) lowers mid-level `RTL` to a low-level `RTL` by imposing target's constraint on addressing modes. At the same time, it would generate the required arithmetic. Address inheritance is part of the functionality of the AMS pass.

Virtual Displacement Handling: The transformation of infinite virtual displacements to target specific displacements is done as follows. The pointer pseudo is given the following attributes:

1. The bias of a pointer is the value currently added to the base pointer.
2. The mode of a pointer is the mode in which the register is accessed or used.
3. The slack of a pointer is the maximum negative value, which can currently be added to the pointer and still properly address the memory references which have already been assigned to this pointer.

The algorithm also defines a `locked_pool` of pointer pseudos which contains bias values at a specific execution point. A locked register is a register which is usable for an address within the currently visible lookahead window without any bias changes. The look ahead window is normally a basic block. We also define `unlocked_pool` registers with each register's bias. An unlocked register is a register which is currently not usable for an address within the currently visible lookahead window without any bias changes. E.g., consider the reference sequence with addresses:

```
(plus:Pmode (fp,124))
(plus:Pmode (fp,120))
(plus:Pmode (fp,128))
```

where `fp` is the frame pointer. It can be any base register in general. Initially, a new pseudo (say `rn`) is created with a `<bias, slack>` value pair as `<124, 60>` for SH4. We can then access memory at (fp, 124) simply as `(rn, 0)` with displacement addressing mode. At second access, (fp, 120), we note that we can reuse `rn` with a bias change of 4. So we change the `<bias, slack>` value to `<120, 56>`. When an offset is not reachable with any pseudo in the locked pool, then a new pseudo (say `rn+1`) is created.

By applying the above reasoning the following output is generated for SH4(which has 60 byte valid displacement):

```
mov     #120,r2
```

```
add     r14,r2     !r14 is fp
mov.l   @(4,r2),r1 !fp+124
mov.l   @r2,r3     !fp+120
..
mov.l   r1,@(8,r2) !fp+128
```

With current framework GCC ends up generating code like this for SH4.

```
mov     r14,r2
add     #64,r2
mov.l   @(60,r2),r1
mov.l   @(56,r2),r2
...
mov     r14,r2
add     #124,r2
mov.l   r1,@(4,r2)
```

The address arithmetic is reduced in former case. To avoid creating too many pseudos during the process, some heuristics have been tried. Limiting pseudos to approximately half of the register set usually turns out be good. With a proper register rematerialization framework `new-ra`, limiting pseudos might become un-necessary.

General index register mode: We can tackle this mode in two ways depending on architecture features and register pressure. There can be weird limitations on use of index registers. While in common cases, the index registers form a `REGISTER_CLASS`, there may be cases like SH4 where `r0` is the sole legal index register. Excessive pressure build up on `r0` as ABI specifies it as a return value register too. So in many cases, it is simply desirable to convert from index register mode to register index mode. The register pressure estimation is still in experimental phases, and forms part of several other problems in compiler technology. We use very simple register pressure heuristics based on machine modes of register. The results would be updated once a general infrastructure for register estimation can be implemented.

Inter-Block Address Inheritance: The technique described needs to be extended to retain `<bias, slack>` values across basic blocks. Taking control flow graph into account is a difficult problem. For simplicity, we propagate the `locked_pool` information only to `fallthru` basic blocks. So some address calculations are saved across basic blocks. The overall strategy is still in investigative phase.

4 Conclusion

Implementation of the ideas presented here have confirmed the expected aliasing gains. The implementation has been tested for SH4 and IA-64. Preliminary benchmarking indicate that execution gains can be as high as 5-7%. However, some more work is required for the idea to work on CISC machines.

5 Acknowledgements

We owe thanks and credit to Toshiyasu Morita (Renesas Technologies). Much of this work is based on his ideas and insights. As always, GCC developers are so helpful. We specially thanks all global maintainers, whose insights often save us weeks of work.

Our special thanks to Mr. Sandeep Khurana at HCL Technologies for his invaluable inputs.

References

[GCCINT] GCC Internals Manual,
`http://gcc.gnu.org`

[SH4]
SH4 Programming Manual, Version 4.0
Renesas Technologies,
`http://www.renesas.com`.

[Eckstein] Erik Eckstein, Bernhard Scholz:
Addressing Mode Selection.

[Sanjiv] Sanjiv K. Gupta, Naveen Sharma:
Alias Analysis for Intermediate Code.
GCC Summit 2003