PowerPC GCC memory handling changes
by Michael Meissner
September 7th, 2018

meissner@linux.ibm.com

Current support

- PowerPC instructions
- ELF v2 ABI
- Fusion

Current support: PowerPC Instructions

- PowerPC has 2 basic memory instructions:
  - X-form: base register + index register
  - D/DS/DQ-form: base register + offset
    - D-form has a 16 bit offset
    - DS-form has a 16-bit offset, but the bottom 2 bits must be 0
    - DQ-form has a 16-bit offset, but the bottom 4 bits must be 0
- If the base register is 0, 0 is used instead of using the value in GPR register 0

Current Support: PowerPC instructions #2

- The **ADDIS** instruction allows you to create larger offsets. ADDIS adds a 16-bit signed value that is shifted left 16 bits.
- For example, to load up a byte value that is 0x123456 bytes from the base register:
  - **ADDIS** *rtemp*,*rbase*,0x12
  - **LDB** *rvalue*,0x3456(*rtemp*)

Current Support: ELF v2 ABI

- The 64-bit PowerPC ABI for little endian systems (ELF v2) uses a dedicated register (r2) to hold the current TOC address.
- The TOC region commonly includes data items within the .got, **.toc**, **.sdata**, and **.sbss** sections. In the medium code model, they can be addressed with 32-bit signed offsets from the TOC pointer register. The TOC pointer register typically points to the beginning of the .got section + 0x8000, which permits a 2 GB TOC with the medium and large code models.
- Static data and constants are put into sections addressable from the TOC pointer.

Current Suport: ELF v2 ABI #2

- Each external function that references static addresses has two entry points.
- The first entry point is for indirect calls. It is required that the function address being called is placed in register r12. The compiler emits 2 instructions to convert the function address into the TOC address in r2.
- The second entry point is for direct calls that are in the same TOC region. These calls assume that the r2 register is already setup and skip the instructions to set it up.
- The linker creates a PLT (program linkage table) to handle direct calls that use a different toc region or are in a shared library.

Current support: ELF v2 ABI #3

- There is a dedicated location on the stack (offset 24) to store the TOC pointer of the current function and to allow it to be restored.
- If you call a function directly, it is required that the TOC value in r2 be stored into the location before doing the call, and place a **NOP** instruction after the call.
- The linker may change the call to call a PLT stub that saves the TOC address, loads up the new TOC address and jumps to the function. It would convert the **NOP** to a **LD** *r2*,24(*r1*) to reload the TOC address.
- If you call an indirect function, you must place the function address into the *r12* register before moving it to the **CTR** register to do the jump and link instruction and save your TOC value.

Current support: Elf v2 ABI #4

- Example:
  - double add1 (double x) { return x + 1.0; }
- Generates:

add1:	addis	2,12,.TOC.-add1@ha	R2 ← TOC address, indirect entry point
	addi	2,2,.TOC.-add1@l	
	.localentry	add1,.-add1	
	addis	9,2,.LC0@toc@ha	R9 ← TOC + ((.TOC-.LC0)<<16) & 0xffff0000
	lfd	0,.LC0@toc@l	F0 ← 1.0
	fadd	1,1,0	
	blr		return
	.section	.rodata.cst8	
.LC0:	.long	0	1.0 constant
	.long	1072693248	

Current support: ELF v2 ABI #5

- Example:
  - extern long foo (void); long bar (void) { foo () + 1; }
- Generates:

bar:	addis	2,12,.TOC.-bar@ha	R2 ← TOC address, indirect entry point
	addi	2,2,.TOC.-bar@l	
	.localentry	bar,.-bar	Direct entry point
	mflr	0	Copy link register (return address) to r0
	std	0,16(1)	Store return address on stack
	stdu	1,-32(1)	Allocate stack frame
	bl	foo	Call foo
	nop		Linker may change NOP into LD 2.24(1)
	addi	1,1,32	Deallocate stack
	ld	0,16(1)	Load up return address
	addi	3,3,1	Do final add
	mtlr	0	Return to the caller
	blr		

Current support: Fusion

- Power8 added limited support for fusing the **ADDIS** and gpr unsigned load D/DS-form instructions:
  - The **ADDIS** instruction must appear immediately before the load;
  - The **ADDIS** instruction must set the GPR that will be loaded (this excludes register r0, since it can't be used as a base register);
  - The signed version of the load byte, half-word, and word instructions do not fuse with the ADDIS;
  - Load and update instructions do not fuse.

ADDIS	3,2,.LC0@toc@ha	Load high offset
LD	3,.LC0@toc@l(3)	Do the load

Current support: Fusion #2

- As a first implementation, GCC did fusion via peephole2 operations.
- It does not fuse if you are loading r0, or the floating point and vector registers.
- If you specify an -O2 or higher optimization, the compiler will convert a signed load half-word, or word operation into an unsigned load half-word or word, and then do an explicit sign extension (the machine does not have a load byte with sign extension instruction).
- Using a peephole2 can mean some combinations are missed due to optimizations done in the compiler.

Current support: fusion #3

- PowerPC GCC has support for an extended fusion operation, where an adjacent ADDIS to a D/DS-form load or store that uses the ADDIS;
- GCC only implemented extended fusion for loading and storing scalar floating point values. It did not implement additional fusion for loading or storing GPR registers;
- Unfortunately, fusion was dropped from the power9 hardware, but it may re-appear in future hardware.
- There was also a variant of fusion called TOC fusion that was recently deleted. Due to a code error, it was hard to enable, and on existing hardware, it didn't improve things.

Issues

- Some of the issues I've been thinking about since working on the PowerPC compiler include:
  - The TOC addressing is done too early;
  - Fusion;
  - Issues with constants.

Issues: TOC addressing

- During expand, GCC loads up each TOC address used into a separate pseudo register;
- Before register allocation, the TOC addresses are merged into the addresses, pretending that we have a single instruction that can do a load/store from a static variable;
- Register allocation then splits the load/store into an **ADDIS** followed by the load/store using that **ADDIS** instruction.
- This means that if there are separate uses of the TOC address, the compiler will generate multiple **ADDIS** instructions.

Issues: TOC addressing #2

- Example:
  - static long a[10]; long add (void) { return a[0] + a[1]; }
- Generates:

add:	addis	2,12,.TOC.-add@ha	Set up TOC pointer
	addi	2,2,.TOC.-add@l	
	.localentry	add,.-add	
	addis	3,2,.LANCHOR0@toc@ha	ADDIS #1
	ld	3,.LANCHOR0@toc@l(3)	
	addis	9,2,.LANCHOR0+8@toc@ha	ADDIS #2
	ld	9,.LANCHOR0+8@toc@l(9)	
	add	3,3,9	
	blr		

Issues: Fusion

- As I mentioned earlier, the current fusion support is done via peephole2 patterns.
- I’ve seen cases where the registers get reallocated late in the compilation, and it has caused a load to a base register that can be fused to be changed either to loading the non-base register r0 or to floating point and vector registers, and this has caused extra moves to be generated.

Issues: Constants

- One minor gripe that I've had over the years is the PowerPC port pretends its floating point instructions support constants directly, but in the reload passes, these constants get push out to the constant pool.
- I tend to think that pushing constants to the constant pool before the register allocation pass might allow some micro-optimizations to be done.
- I've also seen cases where due to pushing the constant during reload creates an address with a PLUS of two registers, and it generates a load with a single register instead of using an X-form form to incorporate the add automatically.

Future changes

- Optimize ADDIS instructions;
- Fusion changes;
- Optimize constants.

Future changes: ADDIS improvements

- I've been experimenting with a small pass before IRA that recognizes when there are multiple references to the same address within a basic block, and pushes out a load address insn, and re-adjusts the addresses.
- It helps with small gains (1%) on the 403.gcc benchmark in Spec 2006.
- I would like to expand it to recognize small loops that don't involve call instructions, and move the **ADDIS** out of the loop.
- I've tried moving the optimization earlier (before CSE), and the initial results were mixed. I tend to feel you don't want to optimize a single cycle insn to cause a spill in another register.

Future changes: Fusion

- I want to rewrite the fusion support so that before register allocation, the move is rewritten. I would then delete the peephole2 insns.
- I would adjust the fusion to the machines that currently support it, as well as keeping it flexible for future machines that will support fusion once again.

Future changes: Constants

- While the code ‘works’, I want to push constants to the constant pool earlier if it is a win or at least performance neutral.
- I want to investigate new ways to create constants in the floating point and vector registers that don’t involve loading the value from memory.