ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences

智能软件研究中心 Intelligent Software Research Center

V8后端代码生成：常量池及其实现

智能软件研究中心 邱吉

qiuji@iscas.ac.cn

2021/05/14

提纲

- 常量池是什么
- 常量池用途
- V8的RISCV64后端为什么需要常量池
- V8的常量池实现
    - 相关文件和数据结构
    - 常量池产生流程和关键函数
- 一个例子

常量池是什么？

- 在程序指令流中插入的，保存程序编译时和链接时的常量的数据结构
- ARM编译器最先使用了literal pool：
  - ARM的指令只有12bi的立即数域
  - 当指令所需的立即数超过12bit的表示范围时，在每4K边界的范围内，插入一个查找表来保存该立即数，并将程序中的立即数加载指令（LDR Rd, =const）改成PC-relative的load指令
  - 查找表：
    - 输入：PC
    - 输出：立即数
  - 用途一：保存程序所需的编译时常量，避免使用多条指令进行拼接： c=a+0x12345678
  - 用途二：保存程序所需的较大的跳转偏移 jump 0x12345678，这个偏移可能要在链接时才能计算得到

常量池用途

- 用途一：保存程序所需的编译时常量，避免使用多条指令进行拼接：：c=a+0x12345678
  - before：li rd,0x12345678->lui/andi/lsl/andi…
  - after: load rd, offset(pc)
- 用途二：保存程序所需的较大的跳转偏移 jump 0x12345678，这个偏移可能要在链接时才能计算得到
  - before: li rd, 0x12345678, jump rd (位置相关)
  - after: jump offset(pc) (offset和pc可以在链接时决定)

V8的RISCV64后端为什么需要常量池-1

- RISCV指令的立即数域：

31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

- lui：imm20
- auipc：imm20
- load/store： imm12
- addi/slti(u)/xori/ori/andi: imm12
- beq/bne/blt(u)/bge(u): imm12

https://riscv.org/technical/specifications/
Volume 1, Unprivileged Spec v. 20191213

V8的RISCV64后端为什么需要常量池-2

伪指令li的几个例子：在64bit地址空间下，大部分li需要4～8条指令

立即数位数	所需的加载指令	数量
imm<12bit	addi rd, x0, imm	1
12bit<imm<32bit	lui rd , imm[31:12] addi rd,x0, imm[11:0]	2

imm>32bit
general case
with temp register
	lui rd , imm[31:12]
	addi rd,x0, imm[11:0]
	Lui rtemp, imm[63:42]
	addi rtemp, rtemp, imm[41:32]
	slli rtemp.rtemp,32
	add rd,rd,rtemp
		6

立即数位数	所需的加载指令	数量

li_constant	lui(rd, (imm + (1LL << 47) + (1LL << 35) + (1LL << 23) + (1LL << 11)) >>48); // Bits 63:48
	addiw(rd, rd, (imm + (1LL << 35) + (1LL << 23) + (1LL << 11)) << 16 >>52); // Bits 47:36
	slli(rd, rd, 12);
	addi(rd, rd, (imm + (1LL << 23) + (1LL << 11)) << 28 >> 52); // Bits 35:24
	slli(rd, rd, 12);
	addi(rd, rd, (imm + (1LL << 11)) << 40 >> 52); // Bits 23:12
	slli(rd, rd, 12);
	addi(rd, rd, imm << 52 >> 52); // Bits 11:0
		8

li_ptr	int64_t a6 = imm & 0x3f; // bits 0:5. 6 bits
	int64_t b11 = (imm >> 6) & 0x7ff; // bits 6:11. 11 bits
	int64_t high_31 = (imm >> 17) & 0x7fffffff; // 31 bits
	int64_t high_20 = ((high_31 + 0x800) >> 12); // 19 bits
	int64_t low_12 = high_31 & 0xfff; // 12 bits
	lui(rd, (int32_t)high_20);
	addi(rd, rd, low_12); // 31 bits in rd.
	slli(rd, rd, 11); // Space for next 11 bis
	ori(rd, rd, b11); // 11 bits are put in. 42 bit in rd
	slli(rd, rd, 6); // Space for next 6 bits
	ori(rd, rd, a6); // 6 bits are put in. 48 bis in rd
		6

V8的RISCV64后端为什么需要常量池-3

- 在32bit定长指令的编码下，立即数编码域是有限的，加载机器指针长度的立即数，需要多条机器指令
- RISCV64这个问题尤其明显：li伪指令大部分情况下需要产生4条以上的机器指令
- 对比：MIPS的立即数域有16bit，可以减少一些指令
- 常量池可以在编译器时将立即数域提前放在指令流中，在使用这个立即数的地方，使用auipc/ld指令对来完成常量加载，节省指令，这也是代码重定位的基础

V8常量池涉及的文件和数据结构-1

- 文件
  - src/constant-pool.cc/h
  - src/codegen/riscv64/assembler-riscv64.cc
- 相关的Class和成员变量
  - ConstantPoolKey: 记录Key值, (value32/value64, Reloc mode， pc)
  - ConstantPool: std::multipamp<ConstantPoolKey, int> entries;
  - Assmebler::ConstantPool constpool_

V8常量池插入的流程:三大步

汇编过程中记录常量entry和pc信息，输出加载常量的模版 → 遇到特殊指令或结束编译单元的代码生成时，进行检查 → 满足条件则Emit常量池，并逐个patch加载常量模版到合适的地址

RecordEntry() → Check() → EmitAndClear()

V8常量池的记录：RecordEntry: 对li和jr指令记录

```
void TurboAssembler::li(Register rd, Operand j, LiFlags mode) {
…   int count = li_estimate(j.immediate(), temps.hasAvailable());
    int reverse_count = li_estimate(~j.immediate(), temps.hasAvailable());
    if (FLAG_riscv_constant_pool && count >= 4 && reverse_count >= 4) {
      // Ld a Address from a constant pool.
      RecordEntry((uint64_t)j.immediate(), j.rmode());
      auipc(rd, 0);
      // Record a value into constant pool.
      ld(rd, rd, 0);
    } else {...
```

```
void MacroAssembler::JumpToInstructionStream(Address entry) {
…
    RecordEntry(entry, RelocInfo::OFF_HEAP_TARGET);
    RecordRelocInfo(RelocInfo::OFF_HEAP_TARGET, entry);
    auipc(kOffHeapTrampolineRegister, 0);
    ld(kOffHeapTrampolineRegister, kOffHeapTrampolineRegister, 0);
  Jump(kOffHeapTrampolineRegister);
…
```

V8常量池插入的检查:

时机：无条件跳转指令之后&编译单元结束之时

check point	routine
TurboAssembler::Jump()	cc_always： ForceConstantPoolEmissionWithoutJump(); EmitConstPoolWithJumpIfNeeded();
TurboAssembler::Ret()	ForceConstantPoolEmissionWithoutJump();
TurboAssembler::Branch()	EmitConstPoolWithJumpIfNeeded();
TurboAssembler::BranchLong()	EmitConstPoolWithJumpIfNeeded();
TurboAssembler::BranchShortHelper()	cc_always: EmitConstPoolWithJumpIfNeeded();
Assembler:GetCode()	ForceConstantPoolEmissionWithoutJump();

V8常量池插入的重点函数 :检查时机例一 TurboAssembler:Jump() && Ret()

```
void TurboAssembler::Jump(Register target, Condition cond, Register rs,
                          const Operand& rt) {
  if (cond == cc_always) {
    jr(target);
    ForceConstantPoolEmissionWithoutJump();
  } else {
    BRANCH_ARGS_CHECK(cond, rs, rt);
    Branch(kInstrSize * 2, NegateCondition(cond), rs, rt);
    jr(target);
  }
}
```

```
void TurboAssembler::Ret(Condition cond, Register rs, const Operand& rt) {
  Jump(ra, cond, rs, rt);
  if (cond == al) {
    ForceConstantPoolEmissionWithoutJump();
  }
}
```

V8常量池插入的重点函数：检查时机例二

Assembler:GetCode()

```
void Assembler::GetCode(Isolate* isolate, CodeDesc* desc,
                        SafepointTableBuilder* safepoint_table_builder,
                        int handler_table_offset) {
...
  ForceConstantPoolEmissionWithoutJump();
...
}
```

V8常量池插入的重点函数：启动check

```
void ForceConstantPoolEmissionWithoutJump() {
  constpool_.Check(Emission::kForced, Jump::kOmitted);
}

void ForceConstantPoolEmissionWithJump() {
  constpool_.Check(Emission::kForced, Jump::kRequired);
}

void EmitConstPoolWithJumpIfNeeded(size_t margin = 0) {
  constpool_.Check(Emission::kIfNeeded, Jump::kRequired, margin);
}

void EmitConstPoolWithoutJumpIfNeeded(size_t margin = 0) {
  constpool_.Check(Emission::kIfNeeded, Jump::kOmitted, margin);
}
```

V8常量池插入的重点函数：check是否应该开始Emit

```
void ConstantPool::Check(Emission force_emit, Jump require_jump,
                         size_t margin) {
  if (IsBlocked()) {
    // Something is wrong if emission is forced and blocked at the same time.
    DCHECK_EQ(force_emit, Emission::kIfNeeded);
    return;
  }
  if (!IsEmpty() && (force_emit == Emission::kForced ||
                  ShouldEmitNow(require_jump, margin))) {
     int worst_case_size = ComputeSize(Jump::kRequired, Alignment::kRequired);
    int needed_space = worst_case_size + assm_->kGap;
    while (assm_->buffer_space() <= needed_space) {
      assm_->GrowBuffer();
    }
    EmitAndClear(require_jump);
  }
   SetNextCheckIn(ConstantPool::kCheckInterval);
}
```

V8常量池插入的重点函数：插入常量池

ConstantPool::EmitAndClear()

```
Label after_pool;
if (require_jump == Jump::kRequired) assm_->b(&after_pool);


assm_->RecordComment("[ Constant Pool");
EmitPrologue(require_alignment);
if (require_alignment == Alignment::kRequired) assm_>Align(kInt64Size);
EmitEntries();
assm_->RecordComment("]");


if (after_pool.is_linked()) assm_->bind(&after_pool);
```

mark

load x0,x0,size

pool guard

j 0

entry1

0x12345678deadbeef

entry2

0x12345678

V8常量池插入的重点函数：EmitEntries

```
void ConstantPool::EmitEntries() {
  for (auto iter = entries_.begin(); iter != entries_.end();) {
    DCHECK(iter->first.is_value32() || IsAligned(assm_->pc_offset(), 8));
    auto range = entries_.equal_range(iter->first);
    bool shared = iter->first.AllowsDeduplication();
    for (auto it = range.first; it != range.second; ++it) {
      SetLoadOffsetToConstPoolEntry(it->second, assm_->pc(), it->first);
      if (!shared) Emit(it->first);
    }
    if (shared) Emit(iter->first);
    iter = range.second;
  }
}
```

V8常量池插入的重点函数：Emit

```
void ConstantPool::Emit(const ConstantPoolKey& key) {
  if (key.is_value32()) {
    assm_->dd(key.value32()); //dump raw double bytes
  } else {
    assm_->dq(key.value64());// dump raw quad bytes
  }
}
```

V8常量池插入的重点函数： SetLoadOffsetToConstPoolEntry

```
void ConstantPool::SetLoadOffsetToConstPoolEntry(int load_offset,
                                                 Instruction* entry_offset,
                                                 const ConstantPoolKey& key) {
  Instr instr_auipc = assm_->instr_at(load_offset);
  Instr instr_ld = assm_->instr_at(load_offset + 4);
  // Instruction to patch must be 'ld rd, offset(rd)' with 'offset == 0'.
  DCHECK(assm_->IsAuipc(instr_auipc));
  DCHECK(assm_->IsLd(instr_ld));
  DCHECK_EQ(assm_->LdOffset(instr_ld), 0);
  DCHECK_EQ(assm_->AuipcOffset(instr_auipc), 0);
  int32_t distance = static_cast<int32_t>(
      reinterpret_cast<Address>(entry_offset) -
      reinterpret_cast<Address>(assm_->toAddress(load_offset)));
  int32_t Hi20 = (((int32_t)distance + 0x800) >> 12);
  int32_t Lo12 = (int32_t)distance << 20 >> 20;
  CHECK(is_int32(distance));
  assm_->instr_at_put(load_offset, SetAuipcOffset(Hi20, instr_auipc));
  assm_->instr_at_put(load_offset + 4, SetLdOffset(Lo12, instr_ld));
}
```

auipc rd, imm20
ld rd, imm12(rd)

V8常量池：一个例子

- ./d9 --print-code --code-comments --riscv-debug ./sunspider/3d-morph.js 2>&1 |tee logcstp.txt
- log内容：

```
[ Constant Pool
0x7f9de4c46154   574  00803003      constant pool begin
(num_const = 8)    ;; constant pool #ld x0,  0x8(x0)
0x7f9de4c46158   578  0000006f      constant #jal 0(x0)
0x7f9de4c4615c   57c  cccccccc      constant
0x7f9de4c46160   580  f37bebd5      constant
0x7f9de4c46164   584  3fcacee9      constant
0x7f9de4c46168   588  54442d18      constant
0x7f9de4c4616c   58c  400921fb      constant
0x7f9de4c46170   590  ac4258f5      constant
0x7f9de4c46174   594  bfca9cd9      constant
          ]
```

总结

- 常量池是在程序指令流中插入的，保存程序编译时和链接时的常量的数据结构
- 常量池用于减小代码尺寸和重定位
- RISCV64指令集的I-type指令只有12bit imm域，汇编li伪指令时，某些立即数需要4条以上机器指令实现，需要使用常量池进行优化
- 目前的V8 RISCV64后端已经实现了常量池

谢 谢

欢迎交流合作

2020/05/14