LUAJIT FOR AARCH64 AND MIPS64 PLATFORMS

RT - RK
COMPUTER BASED SYSTEMS

Stefan Pejić
Đorđe Kovačević

CISCO™

LuaJIT

- LuaJIT 2.0.4 (release)
  - No MIPS64 support
  - No ARM64 support
- LuaJIT 2.1 (development branch)
  - ARM64 interpreter
  - ARM64 JIT (as of November)
  - MIPS64 interpreter (as of May)
  - MIPS64 hard-float JIT (patch submitted)
  - MIPS64 soft-float JIT (coming soon)

GC64 mode

- LuaJIT 2.0.4 supports only 32 bit GC references (suboptimal for 64 bit architectures)
- LuaJIT 2.1 introduces GC64 mode
  - 47 bit pointers + 17 (13+4) bit tags
- At first, available only in interpreter mode
  - x64 and ARM64
- In JIT mode first available for x64

32 bit GC references (!LJ_GC64)

- 64 bit objects (NaN tagged)
  - 32 bits for type
  - 32 bits for pointers (with an exception)

```
                    ---MSW---.---LSW---
primitive types |  itype  |          |
lightuserdata   |  itype  |  void *  |  (32 bit)
lightuserdata   |ffff|    void *     |  (64 bit)
GC objects      |  itype  |  GCRef   |
int (LJ_DUALNUM)|  itype  |   int    |
number          -------double------
```

64 bit GC references (LJ_GC64)

- 64 bit objects (NaN tagged)
  - 17 (13+4) bits for type
  - 47 bits for pointers

```
                        ------MSW------.------LSW------
primitive types     |1..1|itype|1.................1|
GC objects/lightud  |1..1|itype|--------GCRef--------|
int (LJ_DUALNUM)    |1..1|itype|0..0|-----int-------|
number               ------------double-------------
```

ARM64 GC reference handling

- Pointer extraction:

```
#define LJ_GCVMASK       (((uint64_t)1 << 47) - 1)
and x0, x0, #LJ_GCVMASK
```

- Pointer tagging:

```
movn x0, #~LJ_TTAB
add x1, x1, x0, lsl #47
```

- Typecheck:

```
asr x0, x1, 47
cmn x0, #-LJ_TTAB
bne target
```

MIPS64 GC reference handling

- Pointer extraction:

```
dextm r1, r2, 0, 14
```

- Pointer tagging:

```
li r1, LJ_TTAB
dinsu r2, r1, 15, 31
```

- Typecheck:

```
dsra r1, r2, 47
daddiu r1, r1, -LJ_TTAB
bnez target
```

JIT port

- Implement missing pieces in interpreter (vm_*.dasc)
- Add missing instructions (lj_target_*.h)
- Implement emitters for different instruction types (lj_emit_*.h)
- Implement IR to machine code transformation (lj_asm_*.h)
- Implement disassembler (dasm_*.lua)

Interpreter (vm_*.dasc)

- Hot loop detection, exits, stitching, etc.

```
vm_exit_handler:
...
ldr CARG1, [sp, #64*8]
add CARG3, sp, #64*8
mv_vmstate CARG4, EXIT
stp xzr, CARG3, [sp, #62*8]
sub CARG1, CARG1, lr
ldr L, GL->cur_L
lsr CARG1, CARG1, #2
ldr BASE, GL->jit_base
sub CARG1, CARG1, #2
ldr CARG2w, [lr]
st_vmstate CARG4
str BASE, L->base
```

```
ubfx CARG2w, CARG2w, #5, #16
str CARG1w, [GL, #GL_J(exitno)]
str CARG2w, [GL, #GL_J(parent)]
str L, [GL, #GL_J(L)]
str xzr, GL->jit_base
add CARG1, GL, #GG_G2J
mov CARG2, sp
bl extern lj_trace_exit
ldr CARG2, L->cframe
ldr BASE, L->base
and sp, CARG2, #CFRAME_RAWMASK
ldr PC, SAVE_PC
str L, SAVE_L
b >1
```

Target-specific definitions (lj_target_*.h)

- Registers, instructions, instruction fields, etc.

Instruction field encoding:

```
#define A64F_D(r)       (r)
#define A64F_N(r)       ((r) << 5)
#define A64F_A(r)       ((r) << 10)
#define A64F_M(r)       ((r) << 16)
#define A64F_IMMS(x)    ((x) << 10)
#define A64F_IMMR(x)    ((x) << 16)
#define A64F_U16(x)     ((x) << 5)
#define A64F_U12(x)     ((x) << 10)
#define A64F_S26(x)     (x)
```

Instruction encoding:

```
typedef enum A64Ins {
 ...
 A64I_ADDx = 0x8b000000,
 A64I_ANDx = 0x8a000000,
 A64I_CMPx = 0xeb00001f,
 A64I_LDRw = 0xb9400000,
 A64I_STPw = 0x29000000,
 A64I_FCVT_F32_F64 = 0x1e624000,
 A64I_FCVT_F64_F32 = 0x1e22c000
 ...
} A64Ins;
```

Instruction emitter (lj_emit_*.h)

```
static void emit_dnm(ASMState *as, A64Ins ai, Reg rd, Reg rn, Reg rm)
{
  *--as->mcp = ai | A64F_D(rd) | A64F_N(rn) | A64F_M(rm);
}

static void emit_branch(ASMState *as, A64Ins ai, MCode *target)
{
  MCode *p = --as->mcp;
  ptrdiff_t delta = target - p;
  lua_assert(((delta + 0x02000000) >> 26) == 0);
  *p = ai | ((uint32_t)delta & 0x03ffffffu);
}
```

Instruction emitter (lj_emit_*.h)

```
/* Get/set from constant pointer. */
static void emit_lsptr(ASMState *as, A64Ins ai, Reg r, void *p)
{
  /* First, check if ip + offset is in range. */
  if ((ai & 0x00400000) && checkmcpofs(as, p)) {
    emit_d(as, A64I_LDRLx | A64F_S19(mcpofs(as, p)>>2), r);
  } else {
    Reg base = RID_GL; /* Next, try GL + offset. */
    int64_t ofs = glofs(as, p);
    /* Else split up into base reg + offset. */
    if (!emit_checkofs(ai, ofs)) {
      int64_t i64 = i64ptr(p);
      base = ra_allock(as, (i64 & ~0x7fffull), rset_exclude(RSET_GPR, r));
      ofs = i64 & 0x7fffull;
    }
    emit_lso(as, ai, r, base, ofs);
  }
}
```

IR assembler (lj_asm_*.h)

```
static void asm_intmin_max(ASMState *as, IRIns *ir, A64CC cc)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
  Reg right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
  emit_dnm(as, A64I_CSELw|A64F_CC(cc), dest, left, right);
  emit_nm(as, A64I_CMPw, left, right);
}

static void asm_min_max(ASMState *as, IRIns *ir, A64CC cc, A64CC fcc)
{
  if (irt_isnum(ir->t))
    asm_fpmin_max(as, ir, fcc);
  else
    asm_intmin_max(as, ir, cc);
}

#define asm_max(as, ir)         asm_min_max(as, ir, CC_GT, CC_HI)
#define asm_min(as, ir)         asm_min_max(as, ir, CC_LT, CC_LO)
```

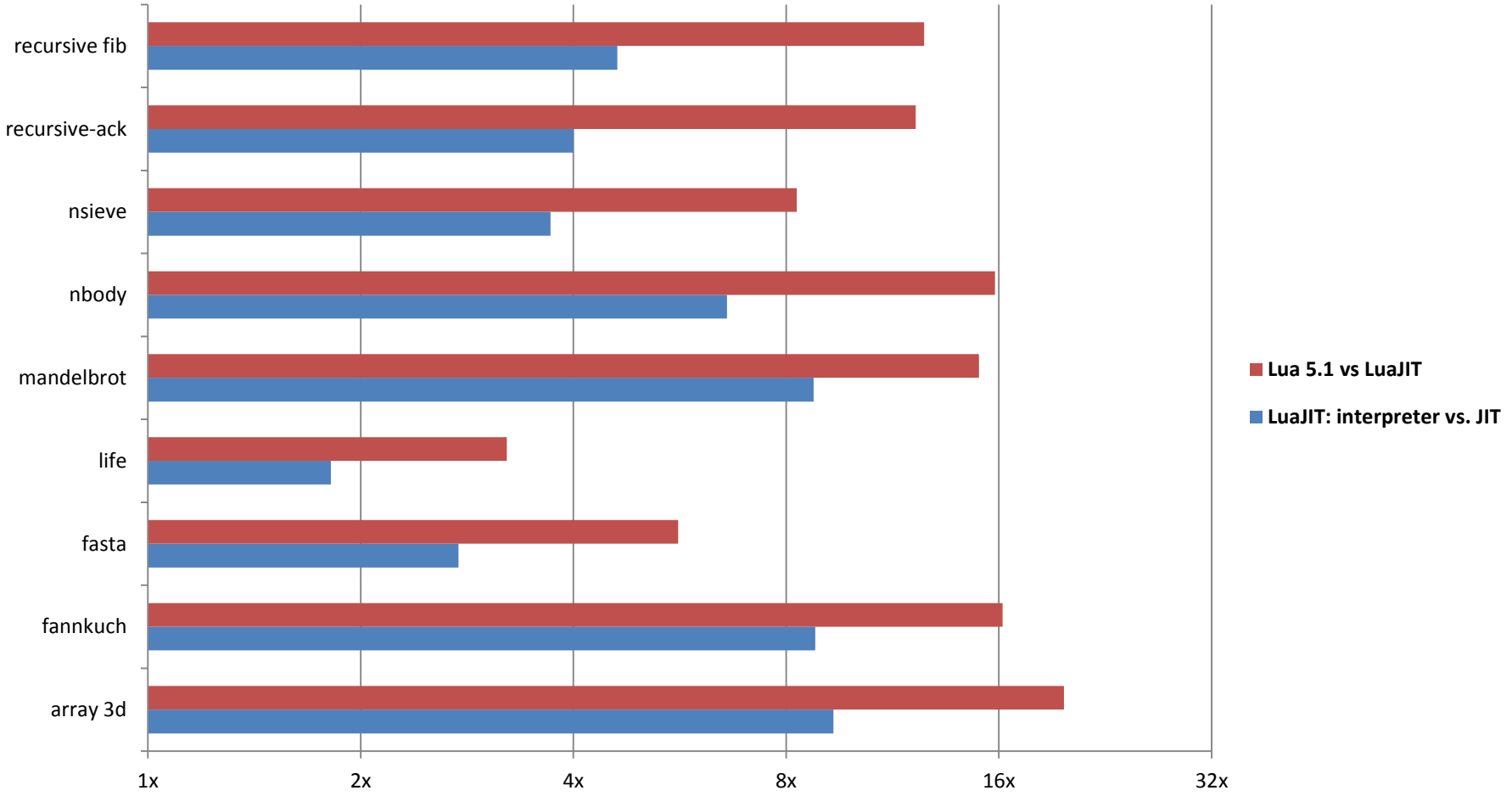
Optimizations

- Fusing multiple instructions into one
  - add + mul → madd
  - and + cmp + b.cc → tbz/tbnz
  - cmp(0) + b.cc → cbz/cbnz
  - and + shift → ubfm
  - or + shr + shl → extr/ror
- Loading constants

MIPS64 JIT

- MIPS64 hard-float
  - Similar to ARM64
- MIPS64 soft-float
  - JIT currently doesn’t support 64 bit soft-float architectures
  - Disable splitting 64 bit IRs into multiple 32 bit IRs for soft-float cases
  - Handle floating-point arithmetic

Performance on ARM64



Contact

Đorđe Kovačević: djordje.lj.kovacevic@rt-rk.com

Stefan Pejić: stefan.pejic@rt-rk.com