OpenJDK11u for RV32 移植进展

曹贵 caogui@iscas.ac.cn

Table of Contents

背景

PLCT实验室 Lead OpenJDK社区RISC-V相关开发工作，RISC-V支持已于2022年3月Upstream到OpenJDK主线，并且先后backport到JDK21U、JDK17U等JAVA LTS版本；PLCT负责OpenJDK RISC-V相关代码的日常开发、测试、代码检视和架构看护；先后实现了RISC-V平台上的JAVA语言重要新特性，包括协程、VectorAPI，ForeignAPI等。所提供的方案属于业界主流实现，目前RISCV芯片厂商如RIVOS、平头哥、华为等均已投入人力参与开发和测试，支持自家芯片所实现的RISC-V新扩展特性等。

背景

目前 OpenJDK 上游已经完全支持 RISC-V 64位后端实现，包括主线版本，还有 JDK21U，JDK17U这些长期支持版本后端也都对 RISC-V 64位进行了实现。但是 RISC-V 32位的相关实现还不完整，近期 PLCT OpenJDK 团队完善了 JDK11U C2 for RV32G 的实现。有了 JIT C2的支持，使得 JDK11U 在 RISCV-V 32位平台的执行速度更快，性能更好。

https://mail.openjdk.org/pipermail/riscv-port-dev/2023-November/001212.html

OpenJDK11u linux/riscv32 port work update

曹贵 caogui at iscas.ac.cn
Thu Nov 16 09:20:43 UTC 2023

- Previous message (by thread): Fwd: [openjdk/jdk] 8319716: RISC-V: Add SHA-2 (PR #16562)
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Hello, I am Gui Cao from ISCAS (Institute of Software, Chinese Academy of Sciences). Recently we have finished porting of OpenJDK11U C2 JIT to Lir
mainline OpenJDK supports Linux/riscv64 platform, which won't run on the Linux/riscv32 platform because of the difference in register widths. So v
qemu for development and testing the port on Linux/riscv32 platform. We can run benchmark workloads such as SPECjbb2015, SPECjbb2005, SPECjvm2008,
when tested with qemu, and these benchmarks can pass the test normally.

Of course the work will go on:
1. There are still some jtreg test failures, we will continue to investigate the issues.
2. We will sync with latest openjdk jdk11u repo[2] after after passing the jtreg tests.
3. Add more C2 intrinsic optimizations to make it run faster.
4. We need to test the functionality and performance on real hardware.

Let me know if you are interested in this project or have any suggestions.

Best regards,
Gui Cao

[1] https://github.com/openjdk-riscv/jdk11u
[2] https://github.com/openjdk/jdk11u
```

开发过程及难度

OpenJDK 可以在编译的时候指定构建版本，可以指定为zero, core ,server, client, custom

zero: 可以构建一个纯C++实现的 OpenJDK 后端，不需要任何汇编支持

core: 构建一个模板解释器的后端，对每一个java字节码使用汇编指令实现

server: 构建服务端后端，构建 server 版本包括 client 版本

client: 构建一个客户端版本，启动速度相比server版本快，启动后的运行速度没有server版本快

custom: 可以指定版本，比如只构建一个server版本

开发过程及难度

模版解释器相关实现，首先需要实现这些 JAVA 字节码对应的汇编

templateTable_riscv32.cpp 9 | templateTable.cpp 4

src > hotspot > share > interpreter > templateTable.cpp > ...

```
259   //                                    interpr. templates
260   // Java spec bytecodes                ubcp|disp|clvm|iswd  in    out   generator            argument
261   def(Bytecodes::_nop                , ____|____|____|____, vtos, vtos, nop                , _           );
262   def(Bytecodes::_aconst_null        , ____|____|____|____, vtos, atos, aconst_null        , _           );
263   def(Bytecodes::_iconst_m1          , ____|____|____|____, vtos, itos, iconst             , -1          );
264   def(Bytecodes::_iconst_0           , ____|____|____|____, vtos, itos, iconst             ,  0          );
265   def(Bytecodes::_iconst_1           , ____|____|____|____, vtos, itos, iconst             ,  1          );
266   def(Bytecodes::_iconst_2           , ____|____|____|____, vtos, itos, iconst             ,  2          );
267   def(Bytecodes::_iconst_3           , ____|____|____|____, vtos, itos, iconst             ,  3          );
268   def(Bytecodes::_iconst_4           , ____|____|____|____, vtos, itos, iconst             ,  4          );
269   def(Bytecodes::_iconst_5           , ____|____|____|____, vtos, itos, iconst             ,  5          );
270   def(Bytecodes::_lconst_0           , ____|____|____|____, vtos, ltos, lconst             ,  0          );
271   def(Bytecodes::_lconst_1           , ____|____|____|____, vtos, ltos, lconst             ,  1          );
272   def(Bytecodes::_fconst_0           , ____|____|____|____, vtos, ftos, fconst             ,  0          );
273   def(Bytecodes::_fconst_1           , ____|____|____|____, vtos, ftos, fconst             ,  1          );
274   def(Bytecodes::_fconst_2           , ____|____|____|____, vtos, ftos, fconst             ,  2          );
275   def(Bytecodes::_dconst_0           , ____|____|____|____, vtos, dtos, dconst             ,  0          );
276   def(Bytecodes::_dconst_1           , ____|____|____|____, vtos, dtos, dconst             ,  1          );
277   def(Bytecodes::_bipush             , ubcp|____|____|____, vtos, itos, bipush             ,  _          );
278   def(Bytecodes::_sipush             , ubcp|____|____|____, vtos, itos, sipush             ,  _          );
279   def(Bytecodes::_ldc                , ubcp|____|clvm|____, vtos, vtos, ldc                ,  false      );
280   def(Bytecodes::_ldc_w              , ubcp|____|clvm|____, vtos, vtos, ldc                ,  true       );
```

开发过程及难度

JVM在调用 java 方法时，对于普通的 java 方法，是通过通过 zerolocals 调用的。然而 zerolocals 例程是通过 generate_normal_entry 该函数生成。一个例程我们可以理解为是一个函数，只不过该函数没有对应的源码，该函数是通过 generate_normal_entry 操作机器码动态生成出来的。 generate_normal_entry 函数代码如下：

```
address InterpreterGenerator::generate_normal_entry(bool synchronized) {
  // ... 省略压栈，生成固定桢等逻辑

  // 跳转到目标Java方法的第一条字节码指令，并执行其对应的机器指令
   __ dispatch_next(vtos);

  // ... 省略统计相关逻辑

  return entry_point;
}
```

开发过程及难度

模版解释器相关实现：字节码 iload_1 的汇编实现

```
----------------------------------------------------------------------
iload_1  27 iload_1  [0x3cce4240, 0x3cce42c0]  128 bytes

  // 栈顶缓存
  0x3cce4180: addi  s4,s4,-4
  0x3cce4184: sw    a0,0(s4)
  0x3cce4188: j 0x3cce41c0
  0x3cce418c: addi  s4,s4,-4
  0x3cce4190: fsw   fa0,0(s4)
  0x3cce4194: j 0x3cce41c0
  0x3cce4198: addi  s4,s4,-8
  0x3cce419c: fsd   fa0,0(s4)
  0x3cce41a0: j 0x3cce41c0
  0x3cce41a4: addi  s4,s4,-8
  0x3cce41a8: sw    zero,4(s4)
  0x3cce41ac: sw    a0,0(s4)
  0x3cce41b0: j 0x3cce41c0
  0x3cce41b4: addi  s4,s4,-4
  0x3cce41b8: add   a0,a0,zero
  0x3cce41bc: sw    a0,0(s4)

  // iload_1指令实现
  0x3cce41c0: lw    a0,0(s8)
```

```
// 指令跳转
0x3cce41c4: lbu   t0,1(s6)
0x3cce41c8: addi  s6,s6,1
0x3cce41cc: lui   t1,0x0
0x3cce41d0: addi  t1,t1,1024 # 0x00000400
0x3cce41d4: add   t1,t0,t1
0x3cce41d8: slli  t1,t1,0x3
0x3cce41dc: add   t1,s5,t1
0x3cce41e0: lw    t1,0(t1)
0x3cce41e4: jr    t1
```

开发过程及难度

在近期 JDK11U C2 的开发过程中遇到了不少问题，其中包括 Long 类型数据的操作，一个 Long 类型数据在 RISCV 64位平台只占用一个寄存器，但是在 RISC-V 32 位平台却需要两个寄存器保存，因此 Long 寄存器的分组，寄存器的运算，寄存器的保存和恢复等都需要进行特殊的处理。其次还有针对 Long 类型和 Double 类型的转换以及栈帧对齐等相关的问题。

```
void MacroAssembler::lShiftL_reg_reg(Register dst, Register src1, Register src2)
{
  mv(t0, src1);
  mv(dst->successor(), src1->successor());
  mv(dst, t0);
  // only the low 6 bits of rs2 are considered for the shift amount
  andi(src2, src2, 0x3f);

  Label blt_branch, done;
  addi(t0, src2, -32);
  bltz(t0, blt_branch);
  sll(dst->successor(), dst, t0);
  mv(dst, 0);
  beqz(zr, done);
  bind(blt_branch);
  mv(t1, 31);
  srli(t0, dst, 0x1);
  sub(t1, t1, src2);
  srl(t0, t0, t1);
  sll(dst->successor(), dst->successor(), src2);
  orr(dst->successor(), t0, dst->successor());
  sll(dst, dst, src2);

  bind(done);
}
```

```
// Long Addition
instruct addL_reg_reg(iRegLNoSp dst, iRegL src1, iRegL src2) %{
  match(Set dst (AddL src1 src2));
  effect(TEMP_DEF dst);

  ins_cost(ALU_COST * 5);
  format %{ "mv   t0, $src1.lo\n\t"
            "add  $dst.lo, $src1.lo, $src2.lo\n\t"
            "sltu t0, $dst.lo, t0\n\t"
            "add  $dst.hi, $src1.hi, $src2.hi\n\t"
            "add  $dst.hi, t0, $dst.hi\t#@addL_reg_reg" %}

  ins_encode %{
    __ mv(t0, as_Register($src1$$reg));
    __ add(as_Register($dst$$reg), as_Register($src1$$reg), as_Register($src2$$reg));
    __ sltu(t0, as_Register($dst$$reg), t0);
    __ add(as_Register($dst$$reg)->successor(), as_Register($src1$$reg)->successor(), as_Register($src2$$re
    __ add(as_Register($dst$$reg)->successor(), t0, as_Register($dst$$reg)->successor());
  %}
```

开发过程及难度

sharedRuntime_riscv32.cpp 是模版解释器和C2编译器相互转换的桥梁

```
// ---------------------------------------------------------------
AdapterHandlerEntry* SharedRuntime::generate_i2c2i_adapters(MacroAssembler *masm,
                                                            int total_args_passed,
                                                            int comp_args_on_stack,
                                                            const BasicType *sig_bt,
                                                            const VMRegPair *regs,
                                                            AdapterFingerPrint* fingerprint) {
  assert_cond(masm != NULL && sig_bt != NULL && regs != NULL && fingerprint != NULL);
  address i2c_entry = __ pc();
  gen_i2c_adapter(masm, total_args_passed, comp_args_on_stack, sig_bt, regs);

  address c2i_unverified_entry = __ pc();
  Label skip_fixup;

  Label ok;

  const Register holder = t1;
  const Register receiver = j_rarg0;
  const Register tmp = t2;  // A call-clobbered register not used for arg passing

  // -------------------------------------------------------------------------
  // Generate a C2I adapter.  On entry we know xmethod holds the Method* during calls
  // to the interpreter.  The args start out packed in the compiled layout.  They
  // need to be unpacked into the interpreter layout.  This will almost always
  // require some stack space.  We grow the current (compiled) stack, then repack
  // the args.  We  finally end in a jump to the generic interpreter entry point.
  // On exit from the interpreter, the interpreter will restore our SP (lest the
  // compiled code, which relys solely on SP and not FP, get sick).

  {
```

开发过程及难度

sharedRuntime_riscv32.cpp 是模版解释器和C2编译器相互转换的桥梁

```
int SharedRuntime::java_calling_convention(const BasicType *sig_bt,
                                           VMRegPair *regs,
                                           int total_args_passed,
                                           int is_outgoing) {
  assert_cond(sig_bt != NULL && regs != NULL);
  // Create the mapping between argument positions and
  // registers.
  static const Register INT_ArgReg[Argument::n_int_register_parameters_j] = {
    j_rarg0, j_rarg1, j_rarg2, j_rarg3,
    j_rarg4, j_rarg5, j_rarg6, j_rarg7
  };
  static const FloatRegister FP_ArgReg[Argument::n_float_register_parameters_j] = {
    j_farg0, j_farg1, j_farg2, j_farg3,
    j_farg4, j_farg5, j_farg6, j_farg7
  };

  uint int_args = 0;
  uint fp_args = 0;
  uint stk_args = 0;

  for (int i = 0; i < total_args_passed; i++) {
    switch (sig_bt[i]) {
    case T_BOOLEAN:
    case T_CHAR:
    case T_BYTE:
    case T_SHORT:
    case T_INT:
    case T_ARRAY:
    case T_OBJECT:
    case T_ADDRESS:
      if (int_args < Argument::n_int_register_parameters_j) {
        regs[i].set1(INT_ArgReg[int_args++]->as_VMReg());
      } else {
        regs[i].set1(VMRegImpl::stack2reg(stk_args++));
      }
      break;
```

```
void SharedRuntime::gen_i2c_adapter(MacroAssembler *masm,
                                    int total_args_passed,
                                    int comp_args_on_stack,
                                    const BasicType *sig_bt,
                                    const VMRegPair *regs) {
  // Cut-out for having no stack args.
  assert_cond(masm != NULL && sig_bt != NULL && regs != NULL);
  int comp_words_on_stack = align_up(comp_args_on_stack * VMRegImpl::stack_slot_size, wo
  if (comp_args_on_stack != 0) {
    __ sub(t0, sp, comp_words_on_stack * wordSize);
    __ andi(sp, t0, -8);
  }

  // Will jump to the compiled code just as if compiled code was doing it.
  // Pre-load the register-jump target early, to schedule it better.
  __ lw(t1, Address(xmethod, in_bytes(Method::from_compiled_offset())));

  // Now generate the shuffle code.
  for (int i = 0; i < total_args_passed; i++) {
    if (sig_bt[i] == T_VOID) {
      assert(i > 0 && (sig_bt[i - 1] == T_LONG || sig_bt[i - 1] == T_DOUBLE), "missing h
      continue;
    }

    // Pick up 0, 1 or 2 words from SP+offset.

    assert(!regs[i].second()->is_valid() || regs[i].first()->next() == regs[i].second(),
           "scrambled load targets?");
    // Load in argument order going down.
    int ld_off = (total_args_passed - i - 1) * Interpreter::stackElementSize;
    // Point to interpreter value (vs. tag)
    int next_off = ld_off - Interpreter::stackElementSize;

    VMReg r_1 = regs[i].first();
    VMReg r_2 = regs[i].second();
    if (!r_1->is_valid()) {
      assert(!r_2->is_valid(), "");
      continue;
```

使用方式

因为目前还没有 RISC-V 32 位可运行linux的物理机器，因此开发和测试都是使用qemu模拟环境(https://github.com/openjdk-riscv/jdk11u/wiki/Build-OpenJDK11-for-RV32G)。这里我们可以参考 OpenJDK11 For RV32G 的构建来完成环境的搭建和 OpenJDK11 的编译，完成编译后即可得到一个 OpenJDK11 For RV32G 的 C2 版本，通过这个版本，我们可以使用 C2 来加速代码的执行。

```
zifeihan@plct-c8:~/jdk11u$ cd build/linux-riscv32-normal-custom-release/jdk/bin/
zifeihan@plct-c8:~/jdk11u/build/linux-riscv32-normal-custom-release/jdk/bin$ time ./java -version
openjdk version "11.0.9-internal" 2020-10-20
OpenJDK Runtime Environment (build 11.0.9-internal+0-adhoc.zifeihan.jdk11u)
OpenJDK Server VM (build 11.0.9-internal+0-adhoc.zifeihan.jdk11u, mixed mode)

real    0m9.766s
user    0m20.471s
sys     0m0.410s
zifeihan@plct-c8:~/jdk11u/build/linux-riscv32-normal-custom-release/jdk/bin$
```

性能测试结果–SPECjbb2015 测试

JDK11U RV32 C2 中这两项指标和 BishengJDK RV64 C2 比较接近，JDK11U RV32 解释器这两项指标比较低。性能测试数据对比如下:

测试版本	SPECjbb2015-Composite max-jOPS	SPECjbb2015-Composite critical-jOPS
BishengJDK RV64 C2	5013	1544
JDK11U RV32 解释器	899	0
JDK11U RV32 C2	5258	1624

性能测试结果–SPECjbb2005 测试

在 Warehouses 3 场景下 JDK11U RV32 C2 得分是 JDK11U RV32 解释器的21.3倍，在 Warehouses 4 场景下JDK11U RV32 C2 得分是 JDK11U RV32 解释器的 21.9 倍，得分比 BishengJDK RV64 C2 稍微也高一点。

测试版本	Warehouses 3	Warehouses 4
BishengJDK RV64 C2	194684	255914
JDK11U RV32 解释器	9166	12247
JDK11U RV32 C2	195679	269303

性能测试结果–SPECjvm2008 测试

可以看到 JDK11U RV32 C2 相比 JDK11U RV32 解释器 OPS 至少提升一倍，在多数情况下提升了10–30倍左右。在某些情况下，OPS 比 BishengJDK 高一些，但是在大多数情况下，OPS 还是比 BishengJDK 低（BishengJDK 实现了更多的Intrinsic加速）

测试用例/版本	BishengJDK RV64 C2	JDK11U RV32 解释器	JDK11U RV32 C2	JDK11U RV32 C2 比 JDK11U RV32 解释器提升N倍
scimark.fft.small	181.82 ops/m	19.95 ops/m	616.99 ops/m	30
scimark.lu.small	368.13 ops/m	11.78 ops/m	346.04 ops/m	29
scimark.sor.small	326.39 ops/m	14.87 ops/m	331.28 ops/m	22
scimark.sparse.small	314.62 ops/m	11.24 ops/m	305.53 ops/m	27
scimark.monte_carlo	291.62 ops/m	6.29 ops/m	290.61 ops/m	46
serial	161.42 ops/m	6.45 ops/m	41.37 ops/m	6
sunflow	44.76 ops/m	6.55 ops/m	11.94 ops/m	1
xml.transform	192.32 ops/m	13.21 ops/m	120.77 ops/m	9

性能测试结果–Jtreg 测试

BishengJDK RV64 C2 测试结果如下：

jdk_tier1 测试结果，成功率：1798/1833 =98.0%
测试报告附件：jdk_tier1.zip

Results

Tests that passed	1798	plain view	grouped view
Tests that failed	22	plain view	grouped view
Tests that had errors	13	plain view	grouped view
Total	1833		

hotspot_tier1 测试结果，成功率：1060/1139 =93.0%
测试报告附件：hotspot_tier1.zip

Results

Tests that passed	1060	plain view	grouped view
Tests that failed	48	plain view	grouped view
Tests that had errors	31	plain view	grouped view
Total	1139		

JDK11U RV32 解释器 测试结果如下：

jdk_tier1 测试结果，成功率：1792/1830=97.9%
测试报告附件：jdk_tier1.zip

Results

Tests that passed	1792	plain view	grouped view
Tests that failed	27	plain view	grouped view
Tests that had errors	11	plain view	grouped view
Total	1830		

hotspot_tier1 测试结果，成功率：1012/1117=90.5%
测试报告附件：hotspot_tier1.zip

Results

Tests that passed	1012	plain view	grouped view
Tests that failed	68	plain view	grouped view
Tests that had errors	37	plain view	grouped view
Total	1117		

JDK11U RV32 C2 测试结果如下：

jdk_tier1 测试结果，成功率：1743/1831=95.1%
测试报告附件：jdk_tier1.zip

Results

Tests that passed	1743	plain view	grouped view
Tests that failed	73	plain view	grouped view
Tests that had errors	15	plain view	grouped view
Total	1831		

hotspot_tier1 测试结果，成功率：1013/1126=89.9%

测试报告附件：hotspot_tier1.zip

Results

Tests that passed	1013	plain view	grouped view
Tests that failed	71	plain view	grouped view
Tests that had errors	42	plain view	grouped view
Total	1126		

性能测试结果–测试报告链接

SPECjbb2015 测试报告链接: https://github.com/openjdk-riscv/jdk11u/issues/595

SPECjbb2005 测试报告链接: https://github.com/openjdk-riscv/jdk11u/issues/592

SPECjvm2008 测试报告链接: https://github.com/openjdk-riscv/jdk11u/issues/594

OpenJDK tier1 测试报告链接: https://github.com/openjdk-riscv/jdk11u/issues/596

未来路线图

1.通过上述测试报告可以看到， JDK11U RV32 C2 相比 JDK11U RV32 解释器 性能提升了不少，但是在某些场景和和 BishengJDK 还有一定的差距，因为 BishengJDK 实现了更多的 Intrinsic 加速，这些 Intrinsic 后续也都会进行实现和完善。

2.OpenJDK 的 JIT 包含了C2和C1两个实现，二者优化方向不同，目前 C2 的实现有了一些阶段性进展，但是 C1 还未实现，后续也会对JIT C1进行实现。

3.JDK11U RV32 C2 在测试 OpenJDK自带的 tier1 测试过程中还有一些bug需要修复，这也是目前正在进行的工作。

4.将JDK11U RV32 版本移植到最新的OpenJDK11版本。

JDK11U for RV32 的移植进展

谢谢大家!