Context threading on the RubyVM

Koichi Sasada
Cookpad Inc.

Nagoya Ruby Kaigi 04
Lightning talk
Koichi Sasada
http://atdot.net/~ko1/

• A programmer
  • 2006-2012 Faculty
  • 2012-2017 Heroku, Inc.
  • 2017- Cookpad Inc.

• Job: MRI development
  • Core parts
    • VM, Threads, GC, etc
Notice

• This talk is based on the knowledge of computer science, especially interpreter development, CPU architecture and C language. Only one Ruby code here.

• This talk is about the virtual machine development.
Summary

• Introducing “Context threading” to improve VM performance with extension by “tailcall” technique.

• Now we can not observe performance improvements (slightly slows down), but we need to investigate more.
Background
VM instruction dispatch technique

- Token Threading

```c
while (1) {
    insn = *pc;                    // fetch
    switch (insn) {               // dispatch
        case Insn_A: do_a(); break; // execution
        case Insn_B: do_b(); break;
        case Insn_C: do_a(); break;
    ...
```
Background
VM instruction dispatch technique

• Direct Threading w/ GCC extension (label as value)

Insn_A:
  do_a();    // execution
  goto *pc;  // fetch and dispatch
Insn_B:
  do_a();    // execution
  goto *pc;  // fetch and dispatch
Background

Issues

• Indirect branch can hurt “branch prediction”
• Missing branch prediction may have a performance impact.

Insn_A:

do_a();    // execution
goto *pc;  // fetch and dispatch

// Branch target is decided by a pointer
// difficult to predict branch prediction
Context threading


• Remove most of indirect branch to improve branch prediction performance
Context threading

• Basic idea: Use call instruction (= subroutine threading)
  • Bytecode: [A, B, C, C, A]
    → Generate native “call” sequence
    NOTE: there are more techniques on CT, but eliminate them here.

• Advantage:
  • Similar to JIT idea. But JIT needs machine code knowledge.
  • CT only needs limited knowledge (call instruction).
  • “call A” instruction is not indirect branch because the destination is determined. We can increase instructions number more.
  • The call/return pair has been optimized by CPU (call stack cache).
Context threading
Problem

• How to eliminate parameter setup code?
  • The sequence should be [call A, call B, …]
  • Instructions should communicate each other with parameters
  • How to setup function parameters (rdi, rsi, … on x86_64)?
  • Original CT (subroutine threading) only support a labels in a function (calling labels directly)
    → Maybe we don’t need to setup function parameters.

• Disadvantage:
  • We can’t add new instructions outside of the function.
  • The setup time of the function can be grow.
  • “perf” only shows the function’s time.
NOTE: No indirect branch

Generated native code

call A
call B
call C
call C
call A

call A(*ec, *cfp) {
    // A body, dirty parameter regs
tail(ec, cfp); // setup regs and jump
}

NOTE: Most of case, tailcall will be “jump” CPU insn

tail(*ec, *cfp) {
    // empty → A function only has “ret”
}
Measurement

\[ i = 0 \]

while \( i < 100\_000\_000 \) # 100M iterations
  \[ i = i + 1 \]
end

# Impl. is not completed.
Result

<table>
<thead>
<tr>
<th>Execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct threading (Current)</td>
</tr>
<tr>
<td>Context threading (Proposal)</td>
</tr>
</tbody>
</table>

Congratulations! Your Ruby is fast!!

- [Small benchmark] → NO prediction misses on recent CPUs.
- So many prologue/epilogue code than my expect.
- “call/return” pair is expensive than my expect.
Remaining issues

- Memory management
  - We need to manipulation page protection (allowing execution) so that we can’t use “malloc/free” library functions.
  - On x86_64 CPU, “call” instruction should be 32 bit relative address so that code are should be near to instruction functions (A, B, ⋅⋅⋅).

- Verbose VM virtual registers manipulation
  - Stack caching can have an affinity because we can pass TOS values with function parameters.

- Not only “call”, but other asm is needed to improve more.
  - Maintenance issue.
  - Portability issue.
Summary

• Introducing “Context threading” to improve VM performance with extension by “tailcall” technique.
• Now we can not observe performance improvements (slightly slows down), but we need to investigate more.
Thank you for your attention

Context threading on the RubyVM

Koichi Sasada
Cookpad Inc.
<ko1@cookpad.com>