2023

A Deep Dive Into Dispatching Techniques

Jonathan Müller

C++ now

Disclaimer

Do not optimize without running your own benchmarks first.

Disclaimer

Do not optimize without running your own benchmarks first.

My benchmarks: 2020 Apple Mac Mini (M1) running Asahi Linux and clang 14.

Dispatch Loop

```
while (…)
{
  switch (…)
  {
  case …: …
  case …: …
    …
  }
}
```

think-cell

Anti-Example: enum to string

```
enum class my_enum { … };

const char* to_string(my_enum e)
{
    switch (e)
    {
    using enum my_enum;

    case a:
        return "a";
    case b:
        return "b";

        …
    }
}
```

Anti-Example (taken from work): Subset of enum

```
switch (viewType)
{
case PowerPoint::ppViewSlideMaster:
case PowerPoint::ppViewTitleMaster:
case …:
    // Handle special view type.
    …
    break;

default:
    // Do nothing.
    break;
}
```

think-cell

Anti-Example (taken from work): Subset of enum

```
if (tc::is_subset(viewType,
        PowerPoint::ppViewSlideMaster | PowerPoint::ppViewTitleMaster | …))
{
    // Handle special view type.
    …
}
```

think-cell

Anti-Example (taken from work): Subset of enum

```
if (tc::is_subset(viewType,
        PowerPoint::ppViewSlideMaster | PowerPoint::ppViewTitleMaster | …))
{
    // Handle special view type.
    …
}
```

We're hiring: think-cell.com/cppnow

Example: Parsing a binary file

```
while (auto header = parse_header(reader))
{
    switch (header.type)
    {
    case header_type::integer:
        parse_integer(reader);
        break;
    case header_type::string:
        parse_string(reader, header.length);
        break;

        …
    }
}
```

Canonical example: Bytecode interpreter

```
while (*ip != bytecode::exit)
{
    switch (*ip)
    {
    case bytecode::add:
        …
    case bytecode::push:
        …

      …
    }
}
```

think-cell

Simple stack-based bytecode

think-cell

Simple stack-based *bytecode*

Bytecode: instructions are single byte op-codes or data.

```
enum class bytecode_op : std::uint8_t
{
    …
};

union bytecode_inst
{
    bytecode_op  op;
    std::uint8_t value;
    std::int8_t  offset;
};

using bytecode = std::vector<bytecode_inst>;
```

Simple *stack-based* bytecode

Stack-based: instructions modify value stack (*vstack*).

`bytecode_op::push`: push constant in next byte onto the vstack

```
a, b, c => a, b, c, 42
```

`bytecode_op::add`: pop two values from the vstack, push sum

```
a, b, c => a, (b+c)
```

think-cell

Example: Sum 3 numbers

```
push 1;
push 2;
add;
push 3;
add;
```

```
vstack: 1
vstack: 1, 2
vstack: 3
vstack: 3, 3
vstack: 6
```

Stack operations

How to use the same value multiple times?

Stack operations

How to use the same value multiple times?

`bytecode_op::dup`: duplicate the value on top of the vstack

```
a, b, c => a, b, c, c
```

think-cell

Stack operations

How to use the same value multiple times?

`bytecode_op::dup`: duplicate the value on top of the vstack

```
a, b, c => a, b, c, c
```

What if values on the vstack are in the wrong order?

Stack operations

How to use the same value multiple times?

`bytecode_op::dup`: duplicate the value on top of the vstack

```
a, b, c => a, b, c, c
```

What if values on the vstack are in the wrong order?

`bytecode_op::swap`: swap the two values on top of the vstack

```
a, b, c => a, c, b
```

Control flow

Interpreter maintains instruction pointer (*ip*).

- Normal instruction: increment ip past opcode plus data.
- `bytecode_op::jump`: increment ip by offset specified in next byte.
- `bytecode_op::jump_if`: increment ip by offset specified in next byte if top is non-zero.

think-cell

Function calls

For simplicity: only a single function allowed.

- Arguments: pushed onto the vstack before call.
- Return value: left on vstack after call.
- `bytecode_op::recurse`: save ip, set ip to beginning of bytecode.
- `bytecode_op::return_`: jump to saved ip.

ip saved in call stack (*cstack*).

think-cell

Example: Recursive fibonacci

```
fib(n) = n < 2 ? n : fib(n-1) + fib(n-2)
```

Example: Recursive fibonacci

```
fib(n) = n < 2 ? n : fib(n-1) + fib(n-2)
```

```
// if n < 2                              vstack: n
dup; push 2; cmp_ge;                     vstack: n, (n >= 2)
jump_if 3;                               vstack: n

// return n
return_;                                 vstack: fib(n)

// return fib(n-1) + fib(n-2)
dup; push 1; sub; recurse;               vstack: n, fib(n-1)
swap; push 2; sub; recurse;              vstack: fib(n-1), fib(n-2)
add; return_;                            vstack: fib(n)
```

think-cell

Interpreter State

```
using bytecode_ip = const bytecode_inst*;
```

`bytecode_ip ip` instruction pointer

`int* vstack_ptr` vstack pointer

`bytecode_ip* cstack_ptr` cstack pointer

`const bytecode& bc` bytecode

think-cell

Execute instructions

```
// push
*vstack_ptr++ = ip[1].value;
ip += 2;
```

Execute instructions

```
// push
*vstack_ptr++ = ip[1].value;
ip += 2;
```

```
// dup
auto top = vstack_ptr[-1];
*vstack_ptr++ = top;
++ip;
```

Execute instructions

```
// push
*vstack_ptr++ = ip[1].value;
ip += 2;
```

```
// dup
auto top = vstack_ptr[-1];
*vstack_ptr++ = top;
++ip;
```

```
// add (, sub, cmp_ge, …)
auto rhs        = *--vstack_ptr;
auto lhs        = *--vstack_ptr;
*vstack_ptr++ = lhs + rhs;
++ip;
```

think-cell

Execute instructions

```cpp
// push
*vstack_ptr++ = ip[1].value;
ip += 2;
```

```cpp
// dup
auto top = vstack_ptr[-1];
*vstack_ptr++ = top;
++ip;
```

```cpp
// add (, sub, cmp_ge, …)
auto rhs         = *--vstack_ptr;
auto lhs         = *--vstack_ptr;
*vstack_ptr++ = lhs + rhs;
++ip;
```

```cpp
// jump_if
auto condition = *--vstack_ptr;
if (condition != 0)
  ip += ip[1].offset;
else
  ip += 2;
```

think-cell

Execute instructions

```cpp
// push
*vstack_ptr++ = ip[1].value;
ip += 2;
```

```cpp
// dup
auto top = vstack_ptr[-1];
*vstack_ptr++ = top;
++ip;
```

```cpp
// add (, sub, cmp_ge, …)
auto rhs       = *--vstack_ptr;
auto lhs       = *--vstack_ptr;
*vstack_ptr++ = lhs + rhs;
++ip;
```

```cpp
// jump_if
auto condition = *--vstack_ptr;
if (condition != 0)
  ip += ip[1].offset;
else
  ip += 2;
```

```cpp
// recurse
*cstack_ptr++ = ip + 1;
ip = bc.data();
```

think-cell

Execute instructions

```
// push
*vstack_ptr++ = ip[1].value;
ip += 2;
```

```
// dup
auto top = vstack_ptr[-1];
*vstack_ptr++ = top;
++ip;
```

```
// add (, sub, cmp_ge, …)
auto rhs        = *--vstack_ptr;
auto lhs        = *--vstack_ptr;
*vstack_ptr++ = lhs + rhs;
++ip;
```

```
// jump_if
auto condition = *--vstack_ptr;
if (condition != 0)
  ip += ip[1].offset;
else
  ip += 2;
```

```
// recurse
*cstack_ptr++ = ip + 1;
ip = bc.data();
```

```
// return
ip = *--cstack_ptr;
```

think-cell

Interpreter

```
int execute(const bytecode& bc, int argument)
{
    int         vstack[vstack_size];
    bytecode_ip cstack[cstack_size];

    bytecode_ip ip  = bc.data();
    auto vstack_ptr = &vstack[0];
    auto cstack_ptr = &cstack[0];

    *cstack_ptr++ = &exit_instruction;
    *vstack_ptr++ = argument;

    return dispatch(ip, vstack_ptr, cstack_ptr, bc);
}
```

Missing piece

```
int dispatch(bytecode_ip ip, int* vstack_ptr, bytecode_ip* cstack_ptr,
             const bytecode& bc);
```

- Read `ip->op`.
- Execute appropriate body and increment `ip`.
- Repeat until exit instruction.

think-cell

Aside: Safety

Bytecode interpreters are prime candidates for remote code execution exploits.

Aside: Safety

Bytecode interpreters are prime candidates for remote code execution exploits.

NEVER start executing untrusted, unverified bytecode.

think-cell

Dispatch Technique #0: `switch`

Idea: switch over opcode

```
while (true)
{
    switch (ip->op)
    {
    case bytecode_op::push:
        …
    case bytecode_op::add:
        …
    case bytecode_op::exit:
        return *--vstack_ptr;


    }
}
```

Idea: switch over opcode

```
while (true)
{
    switch (ip->op)
    {
    case bytecode_op::push:
        …
    case bytecode_op::add:
        …
    case bytecode_op::exit:
        return *--vstack_ptr;
    default:
        __builtin_unreachable();
    }
}
```

Interlude: AArch64 Assembly

Registers

General purpose registers: `r0-r30`

- `x0-x30` 64-bit access
- `w0-w30` 32-bit access (lower half)

think-cell

Interlude: AArch64 Assembly

Registers

General purpose registers: `r0-r30`

- `x0-x30` 64-bit access
- `w0-w30` 32-bit access (lower half)

Addressing modes

- `[x0]` (indirect) address stored in `x0`
- `[x0, #42]` (offset) address stored in `x0` offset by 42

think-cell

Interlude: AArch64 Assembly

Registers

General purpose registers: `r0-r30`

- `x0-x30` 64-bit access
- `w0-w30` 32-bit access (lower half)

Addressing modes

- `[x0]` (indirect) address stored in `x0`
- `[x0, #42]` (offset) address stored in `x0` offset by 42
- `[x0, #42]!` (pre-increment) increment `x0` by 42, then address stored in `x0`
- `[x0], #42` (post-increment) address stored in `x0`, then increment `x0` by 42

think-cell

Interlude: AArch64 Assembly

Registers

General purpose registers: `r0-r30`

- `x0-x30` 64-bit access
- `w0-w30` 32-bit access (lower half)

Addressing modes

- `[x0]` (indirect) address stored in `x0`
- `[x0, #42]` (offset) address stored in `x0` offset by `42`
- `[x0, #42]!` (pre-increment) increment `x0` by `42`, then address stored in `x0`
- `[x0], #42` (post-increment) address stored in `x0`, then increment `x0` by `42`
- `[x0, x1, lsl 3]` (index) address stored in `x0` offset by `x1 << 3`

think-cell

Generated assembly

```
.loop:
    ldrb w8, [x0] ; w8 := ip->op
    cmp  w8, #0   ; w8 == bytecode_op::push?
    b.eq .push    ; then: goto push
    cmp  w8, #1   ; w8 == bytecode_op::add
    b.eq .add     ; then: goto add
    …
    b .exit       ; else: goto exit
```

```
.push:
    ldrb w8, [x0, #1]
    str  w8, [x1], #4
    add  x0, x0, 2
    b .loop               ; goto loop
```

```
.exit:
    ldur w0, [x1, #-4]
    ret                   ; exit loop
```

Actual generated “assembly”

```
if (ip->op < 4) // 0-3
{
    if (ip->op <= 1)  // 0-1
    {
        if (ip->op == 0)
            goto push;
        else
            goto add;
    }
    else // 2-3
    {
        …
    }
}
else
{
```

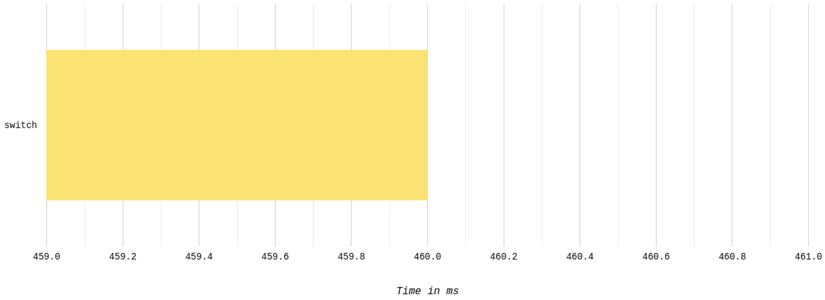
Benchmark

Measure the time for `fib(35)`.

think-cell

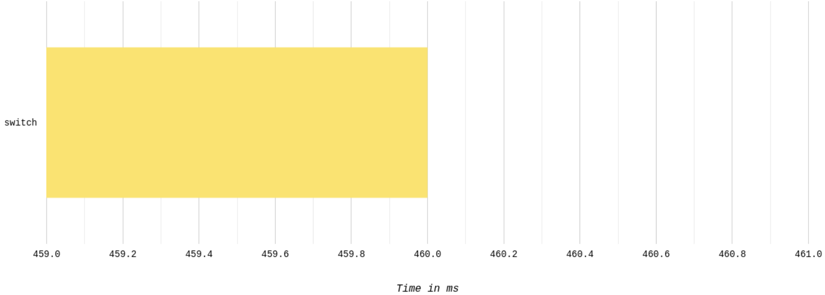
Benchmark

Measure the time for `fib(35)`.



Benchmark

Measure the time for `fib(35)`.



Is that fast?

think-cell

Aside: How to benchmark

1. Take multiple runs.
2. Report average and standard deviation.
3. Compare against some alternative implementation (!).

A command-line benchmarking tool.

```
▶ hyperfine --warmup 3 'fd -e jpg -uu' 'find -iname "*.jpg"'
Benchmark #1: fd -e jpg -uu
  Time (mean ± σ):     329.5 ms ±   1.9 ms    [User: 1.019 s, System: 1.433 s]
  Range (min … max):   326.6 ms … 333.6 ms    10 runs

Benchmark #2: find -iname "*.jpg"
  Time (mean ± σ):      1.253 s ±  0.016 s    [User: 461.2 ms, System: 777.0 ms]
  Range (min … max):    1.233 s …  1.278 s    10 runs

Summary
  'fd -e jpg -uu' ran
    3.80 ± 0.05 times faster than 'find -iname "*.jpg"'
▶ █
```

github.com/sharkdp/hyperfine

think-cell

How to Optimize

1. **Guess a problem**

Interlude: CPU instruction pipeline

Simplified

Assembly instruction execution happens in phases.

1. **Fetch:** fetch memory of the next instruction
2. **Decode:** figure out what the next instruction is
3. **Execute:** actually execute the instruction
4. **Write-back:** write the results into memory

think-cell

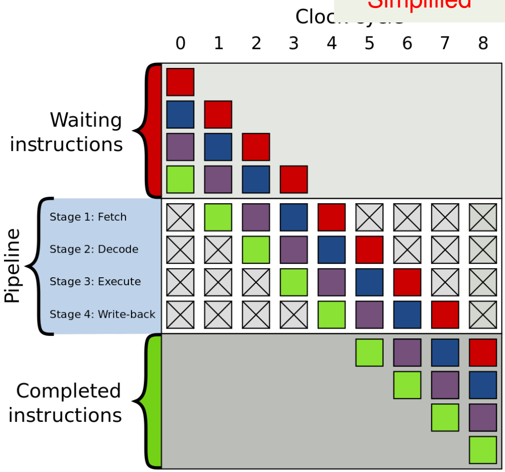
Interlude: CPU instruction pipeline

Assembly instruction execution happens in phases.

1. **Fetch:** fetch memory of the next instruction
2. **Decode:** figure out what the next instruction is
3. **Execute:** actually execute the instruction
4. **Write-back:** write the results into memory

This is done in parallel.

Simplified



Interlude: CPU branch prediction

Simplified

Idea: predict which branch is taken and start processing its assembly instruction.

- Correct prediction: efficient use of the pipeline.
- Incorrect prediction: pipeline has to be cleared, rolled back → expensive.

think-cell

Interlude: CPU branch prediction

Simplified

Idea: predict which branch is taken and start processing its assembly instruction.

- Correct prediction: efficient use of the pipeline.
- Incorrect prediction: pipeline has to be cleared, rolled back → expensive.

As such: remember history for branches to predict correctly.

Interlude: CPU branch prediction

Simplified

Idea: predict which branch is taken and start processing its assembly instruction.

- Correct prediction: efficient use of the pipeline.
- Incorrect prediction: pipeline has to be cleared, rolled back → expensive.

As such: remember history for branches to predict correctly.

But: we're executing different bytecode ops in each iteration.

How to Optimize

1. Guess a problem
2. **Measure to verify guess**

Branch misses

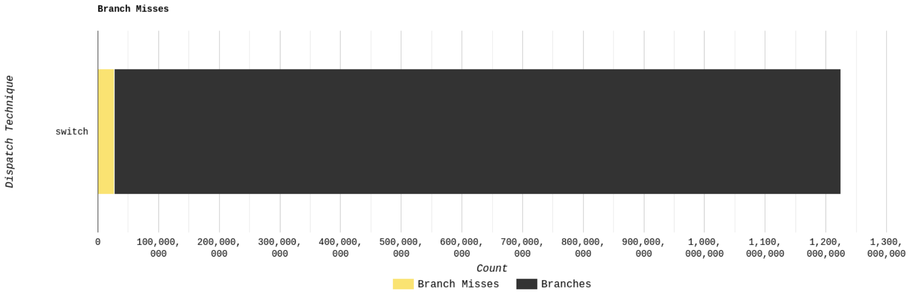
`perf stat`: query hardware performance counters

```
$ perf stat -e branches,branch-misses ./vm_switch.out
```

Branch misses

`perf stat`: query hardware performance counters

```
$ perf stat -e branches,branch-misses ./vm_switch.out
```

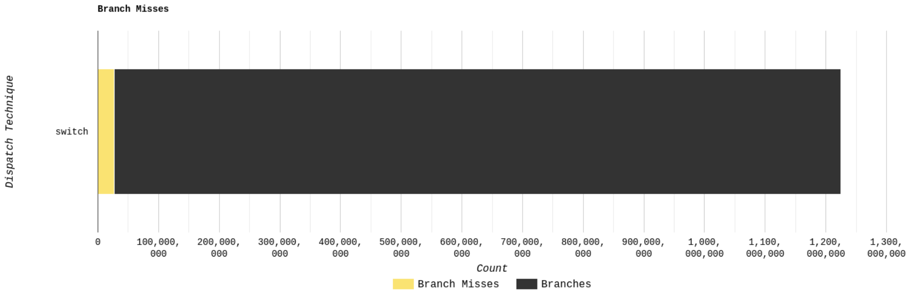


think-cell

Branch misses

`perf stat`: query hardware performance counters

```
$ perf stat -e branches,branch-misses ./vm_switch.out
```



Is that a lot?

think-cell

How to Optimize

1. Guess a problem
2. Measure to verify guess
3. **Workaround problem**

think-cell

Dispatch Technique #1: Call threading [1]

[1] It has nothing to do with threads.

Idea: Array of function pointers

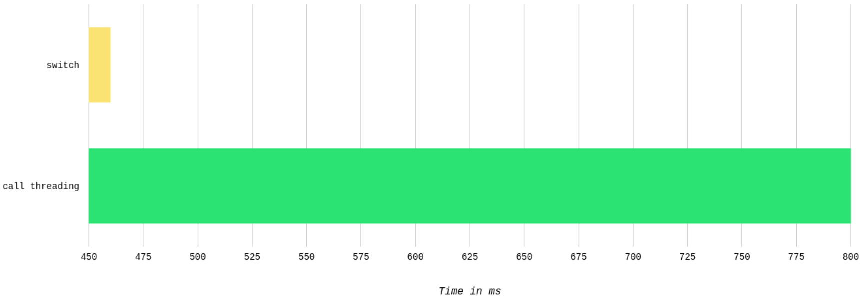
```
void do_execute_push(bytecode_ip& ip, int*& vstack_ptr,
                     bytecode_ip*& cstack_ptr, const bytecode& bc) { … }
void do_execute_add(bytecode_ip& ip, int*& vstack_ptr,
                    bytecode_ip*& cstack_ptr, const bytecode& bc) { … }
```

```
constexpr std::array execute_table
    = {&do_execute_push, &do_execute_add, …};

while (ip->op != bytecode_op::exit)
{
    execute_table[int(ip->op)](ip, vstack_ptr, cstack_ptr, bc);
}
```

Generated assembly

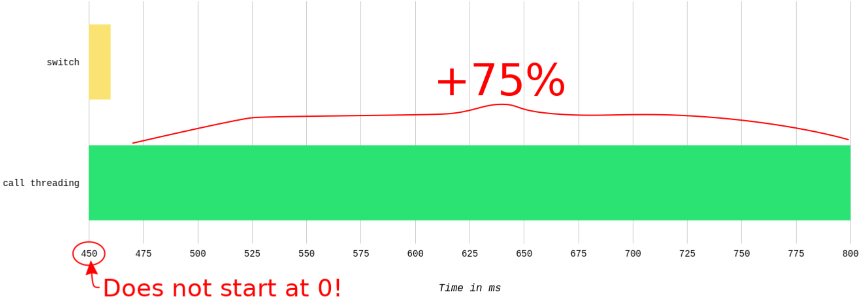
```
; setup omitted
.loop:
  add x0, sp, #24              ; x0 := &ip
  add x1, sp, #16              ; x1 := &vstack_ptr
  add x2, sp, #8               ; x2 := &cstack_ptr
  mov x3, x19                  ; x3 := bc

  ldr x8, [x20, w8, lsl #3]    ; x8 := &execute_table[int(ip->op)]
  blr x8                       ; x8()

  ldrb w8, [sp, #24]           ; w8 := ip->op
  cmp w8, #9                   ; w8 == bytecode_op::exit?
  b.ne .loop
; exit omitted
```

Benchmark

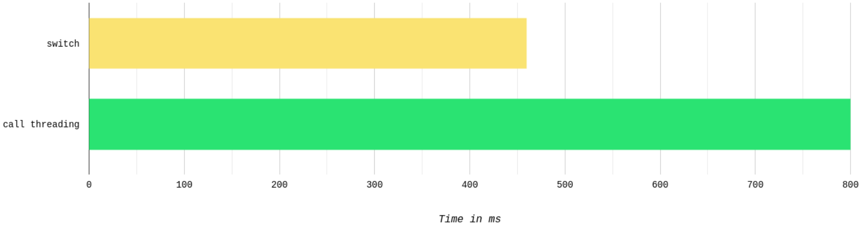


switch
call threading
450
475
500
525
550
575
600
625
650
675
700
725
750
775
800
Time in ms

Aside: How to create bad graphs

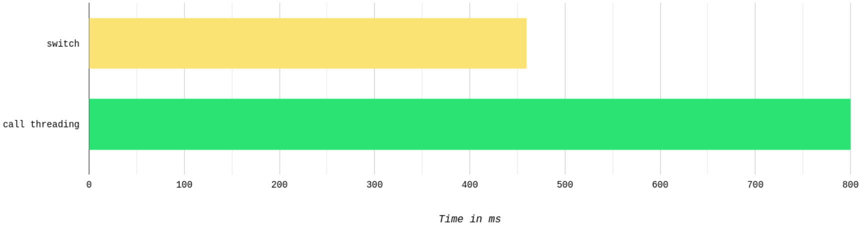


Benchmark



think-cell

Benchmark



Memory overhead.

Generated assembly of execute functions

Call-by-value

```
do_execute_push:
    ldrb    w8, [x0, #1]
    str     w8, [x1], #4
    add     x0, x0, #2
```

Generated assembly of execute functions

Call-by-value

```
do_execute_push:

    ldrb    w8, [x0, #1]


    str     w8, [x1], #4



    add     x0, x0, #2
```

Call-by-reference

```
do_execute_push:
    ldr     x8, [x0]
    ldrb    w8, [x8, #1]

    ldr     x9, [x1]
    str     w8, [x9], #4
    str     x9, [x1]

    ldr     x8, [x0]
    add     x8, x8, #2
    str     x8, [x0]
```

think-cell

Generated assembly of execute functions

Call-by-value

```
do_execute_push:

    ldrb    w8, [x0, #1]


    str     w8, [x1], #4



    add     x0, x0, #2
```

Call-by-reference

```
do_execute_push:
    ldr     x8, [x0]
    ldrb    w8, [x8, #1]

    ldr     x9, [x1]
    str     w8, [x9], #4
    str     x9, [x1]

    ldr     x8, [x0]
    add     x8, x8, #2
    str     x8, [x0]
```

CPU can only work on register values.

How to Optimize

1. Guess a problem
2. Measure to verify guess
3. Workaround problem
4. **Repeat 3 if necessary**

think-cell

Dispatch Technique #2: Token threading[2]

[2] Sometimes imprecisely referred to as indirect threading.

GNU Extension: Computed goto

Normal `goto`

- Label a statement:

```
label: foo;
```

- `goto` label:

```
goto label;
```

GNU Extension: Computed goto

Normal goto

- Label a statement:

```
label: foo;
```

- goto label:

```
goto label;
```

Computed goto

- Take address of label:

```
void* label_addr = &&label;
```

- goto label by dereferencing its address:

```
goto *label_addr;
```

Idea: Array of labels (jump table)

```
constexpr std::array execute_table = {&&do_execute_push, &&do_execute_add, …};

while (true)
{
      goto *execute_table[int(ip->op)];
do_execute_push:
      …
      continue;
do_execute_add:
      …
      continue;
      …
do_execute_exit:
      break;
}
```

Generated assembly

```
.loop:
    ldrb w9, [x0]                   ; w9 := ip->op
    ldr  x9, [x8, x9, lsl #3]       ; x9 := execute_table[w9]
    br   x9                         ; goto

.do_execute_push:
    …
    add  x0, x0, 2
    br .loop ; continue

.do_execute_add:
    …
    add  x0, x0, 1
    br .loop ; continue

    …
```

Actual generated assembly

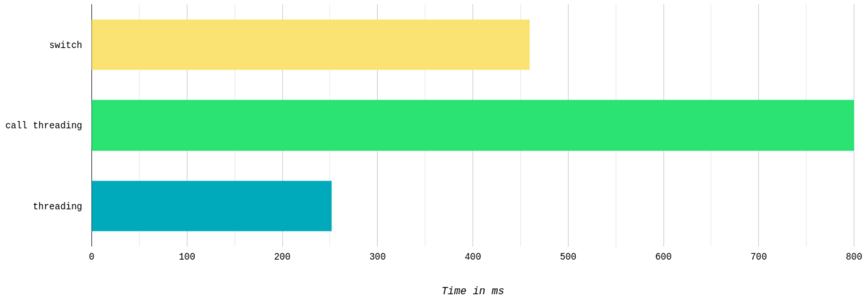
```
    ldrb w9, [x0]                   ; w9 := ip->op
    ldr  x9, [x8, x9, lsl #3]       ; x9 := execute_table[w9]
    br   x9                         ; goto

.do_execute_push:
    …
    ldrb w9, [x0, #2]!              ; ip += 2; w9 := ip->op
    ldr  x9, [x8, x9, lsl #3]       ; x9 := execute_table[w9]
    br   x9                         ; goto

.do_execute_add:
    …
    ldrb w9, [x0, #1]!              ; ip += 1; w9 := ip->op
    ldr  x9, [x8, x9, lsl #3]       ; x9 := execute_table[w9]
    br   x9                         ; goto
```

Canonical token threaded dispatch implementation

```
constexpr std::array execute_table = {&&do_execute_push, &&do_execute_add, …};
goto *execute_table[int(ip->op)];

do_execute_push:
    …
    goto *execute_table[int(ip->op)];

do_execute_add:
    …
    goto *execute_table[int(ip->op)];

    …
```

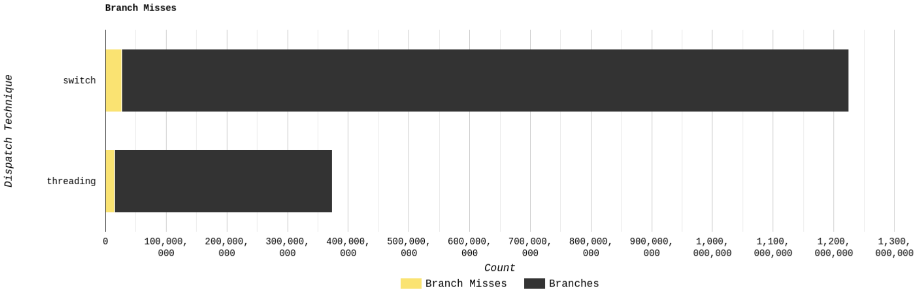
think-cell

Benchmark



switch
call threading
threading
0
100
200
300
400
500
600
700
800
Time in ms

That's still a branch



Duplicated dispatch code

`switch` dispatch:

- Single dispatch for all bytecode instruction handlers.
- Single location for branch prediction.
- Can only learn about common bytecode instructions.

Threaded dispatch:

- Separate dispatch after each bytecode instruction handler.
- Separate locations for branch prediction.
- Can learn what bytecode instruction usually follows.

think-cell

Let's figure out what's still slow

```
$ perf record ./vm_token_threading.out
```

Let's figure out what's still slow

```
$ perf record ./vm_token_threading.out

$ perf report
  Overhead  Shared Object              Symbol
  ........  .........................  ......................

    99.85%  vm_token_threading.out     [.] dispatch
     0.10%  ld-linux-aarch64.so.1      [.] _dl_lookup_symbol_x
     0.04%  ld-linux-aarch64.so.1      [.] do_lookup_x
     0.00%  ld-linux-aarch64.so.1      [.] copy_hwcaps
     0.00%  ld-linux-aarch64.so.1      [.] _dl_start
     0.00%  ld-linux-aarch64.so.1      [.] _start
```

Wishlist

- Separate functions for executing bytecode instructions.
- No memory overhead.

Dispatch Technique #2.5: Token-threaded dispatch with tail calls

think-cell

Idea: Each bytecode instruction handler calls next handle

```
constexpr std::array execute_table = {&do_execute_push, &do_execute_add, …};

int do_execute_push(bytecode_ip ip, int* vstack_ptr,
                    bytecode_ip* cstack_ptr, const bytecode& bc)
{
    …
    return execute_table[int(ip->op)](ip, vstack_ptr, cstack_ptr, bc);
}
int do_execute_add(bytecode_ip ip, int* vstack_ptr,
                   bytecode_ip* cstack_ptr, const bytecode& bc)
{
    …
    return execute_table[int(ip->op)](ip, vstack_ptr, cstack_ptr, bc);
}
```

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

`dispatch`

1 Push PC.

think-cell

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

`dispatch`
`do_execute_dup`

1. Push PC.
2. Jump to first execute.
3. Push PC.

think-cell

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

`dispatch`
`do_execute_dup`
`do_execute_push`

1. Push PC.
2. Jump to first execute.
3. Push PC.
4. Jump to second execute.
5. Push PC.

think-cell

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

`dispatch`
`do_execute_dup`
`do_execute_push`
…

1. Push PC.
2. Jump to first execute.
3. Push PC.
4. Jump to second execute.
5. Push PC.
6. ...

think-cell

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

`dispatch`
`do_execute_dup`
`do_execute_push`
…
`do_execute_exit`

1. Push PC.
2. Jump to first execute.
3. Push PC.
4. Jump to second execute.
5. Push PC.
6. …
7. Jump to final execute.
8. Push PC.

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

`dispatch`
`do_execute_dup`
`do_execute_push`
…

1 Push PC.
2 Jump to first execute.
3 Push PC.
4 Jump to second execute.
5 Push PC.
6 ...
7 Jump to final execute.
8 Push PC.
7 Pop PC and jump back.

think-cell

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

`dispatch`
`do_execute_dup`
`do_execute_push`

1. Push PC.
2. Jump to first execute.
3. Push PC.
4. Jump to second execute.
5. Push PC.
6. ...
7. Jump to final execute.
8. Push PC.
7. Pop PC and jump back.
7. Pop PC and jump back.

think-cell

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

`dispatch`
`do_execute_dup`

1. Push PC.
2. Jump to first execute.
3. Push PC.
4. Jump to second execute.
5. Push PC.
6. ...
7. Jump to final execute.
8. Push PC.
7. Pop PC and jump back.
7. Pop PC and jump back.
8. Pop PC and jump back.

think-cell

Call stack

Call pushes program counter (PC) and jumps to label, return pops and jumps back.

```
dispatch
```

1. Push PC.
2. Jump to first execute.
3. Push PC.
4. Jump to second execute.
5. Push PC.
6. ...
7. Jump to final execute.
8. Push PC.

7 Pop PC and jump back.

7 Pop PC and jump back.

8 Pop PC and jump back.

9 Pop PC and jump back.

Actual call stack

Actual call stack

dispatch

1. Push PC.

Actual call stack

`dispatch`
`do_execute_dup`

1. Push PC.
2. Jump to first execute.
3. Push PC.

think-cell

Actual call stack

`dispatch`
`do_execute_dup`
`do_execute_push`

1. Push PC.
2. Jump to first execute.
3. Push PC.
4. Jump to second execute.
5. Push PC.

think-cell

Actual call stack

dispatch
do_execute_dup
do_execute_push
…

1. Push PC.
2. Jump to first execute.
3. Push PC.
4. Jump to second execute.
5. Push PC.
6. …

think-cell

Actual call stack

`dispatch`
`do_execute_dup`
`do_execute_push`
...
Stack overflow

1. Push PC.
2. Jump to first execute.
3. Push PC.
4. Jump to second execute.
5. Push PC.
6. ...
7. Stack overflow.

think-cell

Tail call optimization

If a function ends with `return foo();`, just jump there without push.

Tail call optimization

If a function ends with `return foo();`, just jump there without push.

```
dispatch
```

1 Push PC.

Tail call optimization

If a function ends with `return foo();`, just jump there without push.

1. Push PC.
2. Jump to first execute.

Tail call optimization

If a function ends with `return foo();`, just jump there without push.

`dispatch`

`do_execute_push`

1. Push PC.
2. Jump to first execute.
3. Jump to second execute.

think-cell

Tail call optimization

If a function ends with `return foo();`, just jump there without push.

`dispatch`

...

1. Push PC.
2. Jump to first execute.
3. Jump to second execute.
4. ...

Tail call optimization

If a function ends with `return foo();`, just jump there without push.

dispatch

`do_execute_exit`

1. Push PC.
2. Jump to first execute.
3. Jump to second execute.
4. ...
5. Jump to final execute.

think-cell

Tail call optimization

If a function ends with `return foo();`, just jump there without push.

`dispatch`

1. Push PC.
2. Jump to first execute.
3. Jump to second execute.
4. ...
5. Jump to final execute.
6. Pop PC and jump back to caller.

think-cell

clang Extension: `[[clang::musttail]]`

https://clang.llvm.org/docs/AttributeReference.html#musttail

> *If a return statement is marked musttail, this indicates that the* ***compiler must generate a tail call*** *for the program to be correct, even when optimizations are disabled. This guarantees that the call will not cause unbounded stack growth if it is part of a recursive cycle in the call graph.*

think-cell

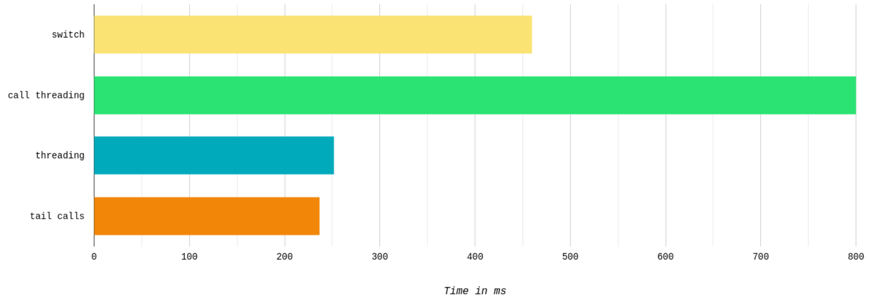
Idea: Each bytecode instruction handler tail calls the next handler

```
constexpr std::array execute_table = {&do_execute_push, &do_execute_add, …};

int do_execute_push(bytecode_ip ip, int* vstack_ptr,
                    bytecode_ip* cstack_ptr, const bytecode& bc)
{
    …
    [[clang::musttail]] return execute_table[int(ip->op)]
                                    (ip, vstack_ptr, cstack_ptr, bc);
}
int do_execute_add(bytecode_ip ip, int* vstack_ptr,
                   bytecode_ip* cstack_ptr, const bytecode& bc)
{
    …
    [[clang::musttail]] return execute_table[int(ip->op)]
                                    (ip, vstack_ptr, cstack_ptr, bc);
}
```

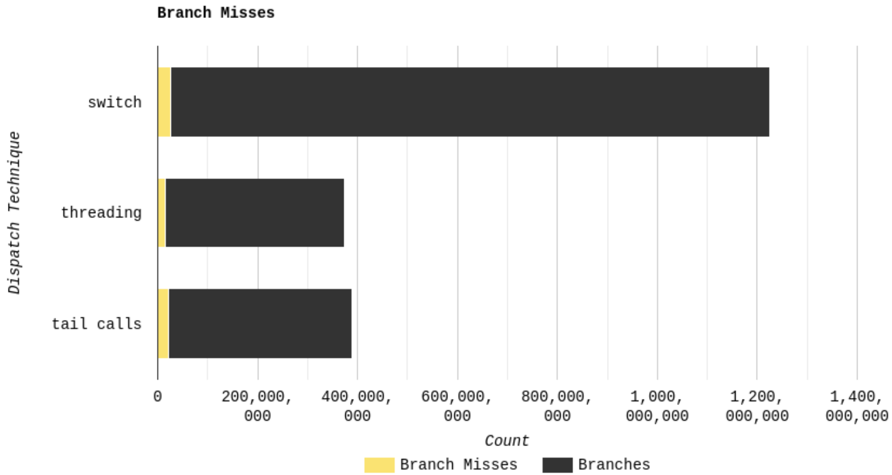
Generated assembly

```
dispatch:
    adrp x8, execute_table
    add  x8, x8, :lo12:execute_table ; x8 := execute_table
    ldrb w9, [x0]                    ; w9 := ip->op
    ldr  x9, [x8, x9, lsl #3]        ; x9 := x8[w9]
    br   x9                          ; tail call

do_execute_push:
    …
    adrp x8, execute_table
    add  x8, x8, :lo12:execute_table ; x8 := execute_table
    ldrb w9, [x0, #2]!               ; ip += 2; w9 := ip->op
    ldr  x9, [x8, x9, lsl #3]        ; x9 := x8[w9]
    br   x9                          ; tail call
```

Benchmark


switch
call threading
threading
tail calls
0
100
200
300
400
500
600
700
800
Time in ms

Branch misses



Let's figure out what's still slow

```
$ perf record ./vm_token_tail_call.out
$ perf report
  Overhead  Shared Object              Symbol
  ........  .........................  .......................

    15.97%  vm_token_tail_call.out  [.] do_execute_push
    15.21%  vm_token_tail_call.out  [.] do_execute_add
    12.74%  vm_token_tail_call.out  [.] do_execute_sub
    12.58%  vm_token_tail_call.out  [.] do_execute_dup
     9.91%  vm_token_tail_call.out  [.] do_execute_cmp_ge
     9.64%  vm_token_tail_call.out  [.] do_execute_jump_if
     9.30%  vm_token_tail_call.out  [.] do_execute_recurse
     8.98%  vm_token_tail_call.out  [.] do_execute_return_
     5.57%  vm_token_tail_call.out  [.] do_execute_swap
```

think-cell

Interlude: `register` keyword

Dear C compiler, please keep this variable in a register.

```
register bytecode_ip ip;
register int* vstack_ptr;
register bytecode_ip* cstack_ptr;

// Interpreter loop here.
```

think-cell

Interlude: register keyword

Dear C compiler, please keep this variable in a register.

```
register bytecode_ip ip;
register int* vstack_ptr;
register bytecode_ip* cstack_ptr;

// Interpreter loop here.
```

Modern compilers do it for you.

Interlude: `register` keyword

Dear C compiler, please keep this variable in a register.

```
register bytecode_ip ip;
register int* vstack_ptr;
register bytecode_ip* cstack_ptr;

// Interpreter loop here.
```

Modern compilers do it for you.

Except when they don't.

think-cell

Why LuaJIT's interpreter is written in assembly

Mike Pall, http://lua-users.org/lists/lua-l/2011-02/msg00742.html

> *We can use a direct or indirect-threaded interpreter even in C, e.g. with the computed 'goto &' feature of GCC. […] This effectively replicates the load and the dispatch, which helps the CPU branch predictors. But it has its own share of problems: […]* ***The register allocator can only treat each of these segments separately and will do a real bad job.*** *There's just no way to give it a goal function like "I want the same register assignment before each goto".*

Interlude: Calling convention

Simplified

Function call: jump to address.

How are arguments passed?

think-cell

Interlude: Calling convention

Simplified

Function call: jump to address.

How are arguments passed?

Calling convention for AArch64

- `x0` to `x7`: **Argument values**
- `x9` to `x15`: Local variables (caller saved).
- `x19` to`x29`: Local variables (callee saved).

Calling convention forces register assignment

Want something in a register? Pass it as argument.

```
int do_execute_push(bytecode_ip ip, int* vstack_ptr,
                    bytecode_ip* cstack_ptr, const bytecode& bc)
{
    …
}
```

```
do_execute_push:
    ldrb    w8, [x0, #1]
    str     w8, [x1], #4
    adrp    x9, execute_table
    add     x9, x9, :lo12:execute_table
    ldrb    w8, [x0, #2]!
    ldr     x4, [x9, x8, lsl #3]
    br      x4
```

Calling conventions

Standard calling convention

- AArch64: 8 registers for arguments
- x86_64 Linux: 6 registers for arguments
- x86_64 Windows: 4 registers for arguments

Calling conventions

Standard calling convention

- AArch64: 8 registers for arguments
- x86_64 Linux: 6 registers for arguments
- x86_64 Windows: 4 registers for arguments

We can use a custom calling convention!

Calling conventions

Standard calling convention

- AArch64: 8 registers for arguments
- x86_64 Linux: 6 registers for arguments
- x86_64 Windows: 4 registers for arguments

We can use a custom calling convention!

`[[gnu::regcall]]` : Pass as many arguments as possible in registers.

- AArch64: ignored
- x86_64 Linux: 12 registers for arguments
- x86_64 Windows: 11 registers for arguments

think-cell

The fast and slow path

Mike Pall, http://lua-users.org/lists/lua-l/2011-02/msg00742.html

> *If you write an interpreter loop in assembler, you can do much better:*
>
> - *Keep a fixed register assignment for all [bytecode] instructions.*
> - *Keep everything in registers for the fast paths. Spill/reload only in the slow paths.*
> - *Move the slow paths elsewhere, to help with I-Cache density.*

think-cell

Interlude: I-Cache

Simplified

- Assembly instructions are stored in memory.

Interlude: I-Cache

Simplified

- Assembly instructions are stored in memory.
- Memory access is slow.

think-cell

Interlude: I-Cache

Simplified

- Assembly instructions are stored in memory.
- Memory access is slow.
- Special cache for instructions: I-cache.

Interlude: I-Cache

Simplified

- Assembly instructions are stored in memory.
- Memory access is slow.
- Special cache for instructions: I-cache.
- But: don't pollute it with cold code.

think-cell

A bytecode instruction with a slow path

`bytecode_op::print42`: print the top value if it is 42

`x => x`

```
int do_execute_print42(bytecode_ip ip, int* vstack_ptr,
                       bytecode_ip* cstack_ptr, const bytecode& bc)
{
    if (vstack_ptr[0] == 42)
        std::puts("42");

    ++ip;
    [[clang::musttail]] return execute_table[int(ip->op)]
                                  (ip, vstack_ptr, cstack_ptr, bc);
}
```

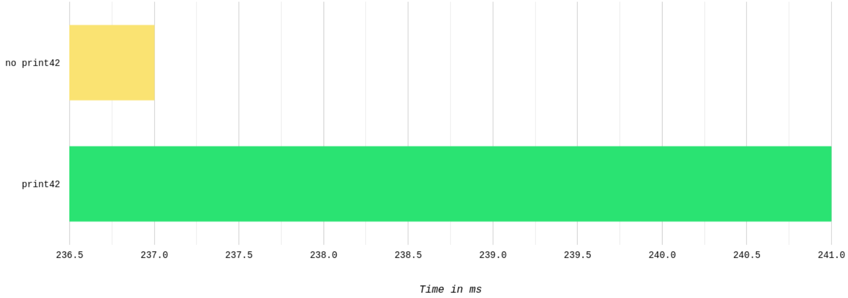
think-cell

Benchmark

`fib(35)`: once with initial `print42`, once without.

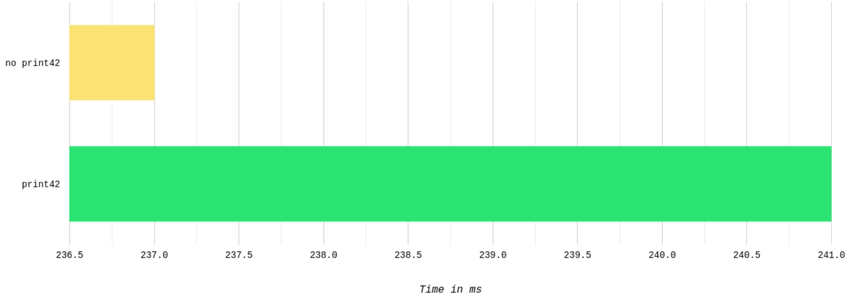
think-cell

`fib(35)`: once with initial `print42`, once without.



Benchmark

`fib(35)`: once with initial `print42`, once without.



4ms slower.

think-cell

Generated assembly

```
do_execute_print42:
    stp     x29, x30, [sp, #-48]!
    stp     x22, x21, [sp, #16]
    mov     x29, sp
    stp     x20, x19, [sp, #32]
    ldr     w8, [x1]
    mov     x19, x3
    mov     x20, x2
    mov     x21, x1
    mov     x22, x0
    cmp     w8, #42
    b.ne    .LBB1_2
    adrp    x0, .L.str
    add     x0, x0, :lo12:.L.str
    bl      puts
```

```
.LBB1_2:
    ldrb    w8, [x22, #1]!
    adrp    x9, execute_table
    mov     x0, x22
    add     x9, x9, :lo12:execute_table
    mov     x1, x21
    mov     x2, x20
    mov     x3, x19
    ldp     x20, x19, [sp, #32]
    ldp     x22, x21, [sp, #16]
    ldr     x4, [x9, x8, lsl #3]
    ldp     x29, x30, [sp], #48
    br      x4
```

Hoist the slow path

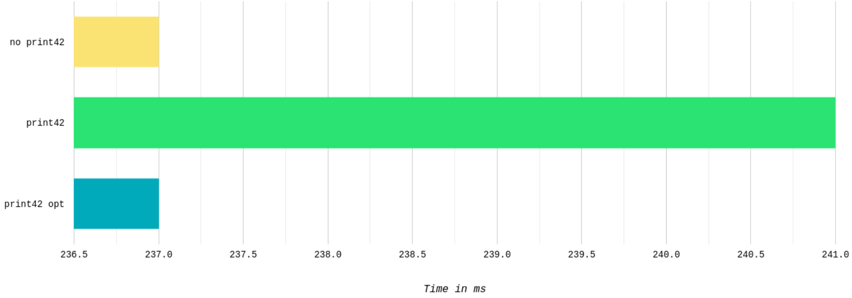
```
int do_execute_print42(bytecode_ip ip, int* vstack_ptr,
                       bytecode_ip* cstack_ptr, const bytecode& bc)
{
    if (vstack_ptr[0] == 42)
        [[clang::musttail]] return do_print_impl(…);

    ++ip;
    [[clang::musttail]] return execute_table[int(ip->op)](…);
}
[[gnu::noinline]] int do_print_impl(bytecode_ip ip, int* vstack_ptr,
                       bytecode_ip* cstack_ptr, const bytecode& bc)
{
    std::puts("42");
    ++ip;
    [[clang::musttail]] return execute_table[int(ip->op)](…);
}
```

Generated assembly

```
do_execute_print42:
    ldr     w8, [x1]
    cmp     w8, #42
    b.ne    .LBB1_2
    b       do_print_impl
.LBB1_2:
    ldrb    w8, [x0, #1]!
    adrp    x9, execute_table
    add     x9, x9, :lo12:execute_table
    ldr     x4, [x9, x8, lsl #3]
    br      x4
```

think-cell

Benchmark


no print42
print42
print42 opt
236.5
237.0
237.5
238.0
238.5
239.0
239.5
240.0
240.5
241.0
Time in ms

You can't actually return!

```
int do_execute_recurse(bytecode_ip ip, int* vstack_ptr,
                       bytecode_ip* cstack_ptr, const bytecode& bc)
{
    if (*cstack_ptr == END_OF_CALL_STACK)
        [[clang::musttail]] return grow_call_stack(…);

    *cstack_ptr++ = ip + 1;
    ip            = bc.data();
    [[clang::musttail]] return execute_table[int(ip->op)](…);
}
[[gnu::noinline]] int grow_call_stack(bytecode_ip ip, int* vstack_ptr,
                       bytecode_ip* cstack_ptr, const bytecode& bc)
{
    cstack_ptr = allocate_bigger_and_copy_old(cstack_ptr);
    [[clang::musttail]] return do_execute_recurse(…);
}
```

Conclusion?

`[[clang::musttail]]` enables threading via function calls:

- Detailed performance tracking in `perf record`
- Force the compiler to use a particular register assignment
- Remember to hoist slow paths; no regular function calls in the hot code

Conclusion?

`[[clang::musttail]]` enables threading via function calls:

- Detailed performance tracking in `perf record`
- Force the compiler to use a particular register assignment
- Remember to hoist slow paths; no regular function calls in the hot code

Trick the compiler into generating the exact assembly you want.

think-cell

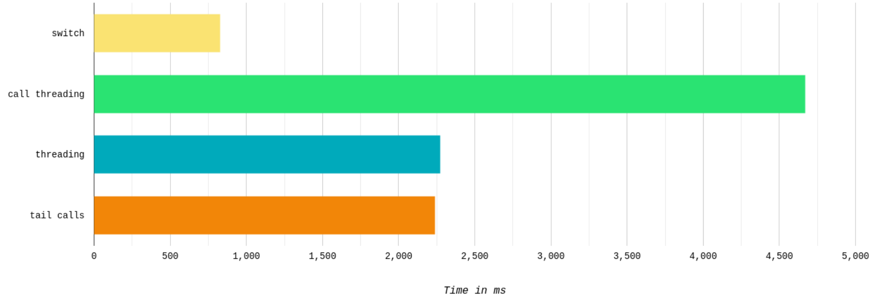
Let's benchmark on my old laptop

Benchmarks

New benchmarks: 2016 Thinkpad 13 running Arch Linux and clang 14.

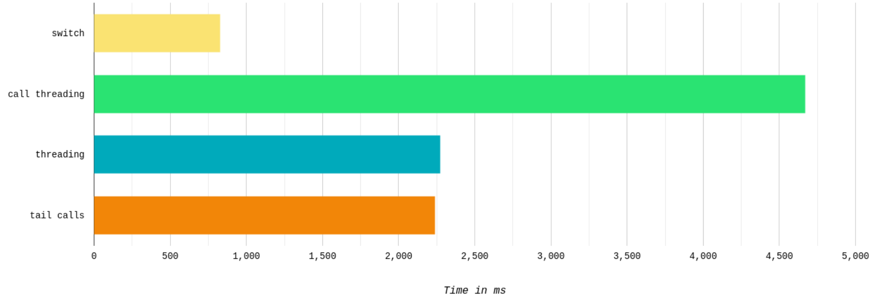
Benchmarks

New benchmarks: 2016 Thinkpad 13 running Arch Linux and clang 14.



Benchmarks

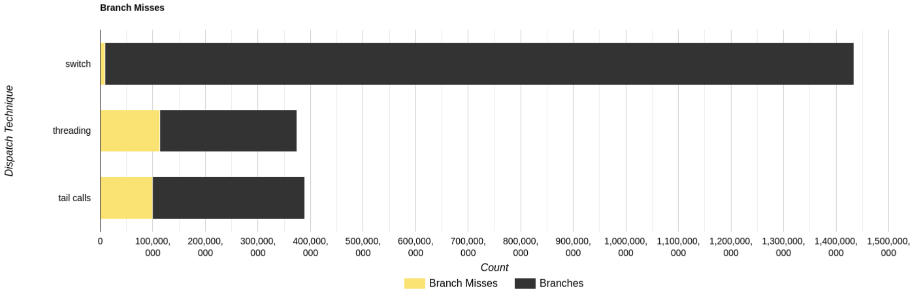
New benchmarks: 2016 Thinkpad 13 running Arch Linux and clang 14.



... Uhm?

think-cell

Branch misses!



Interlude: Branch *target* prediction

Simplified

Conditional branch, fixed target:

```
.loop:
    ldrb w8, [x0]
    cmp  w8, #0
    b.eq .push
    cmp  w8, #1
    b.eq .add
    …
    b .exit
```

Unconditional branch, variable target:

```
adrp    x9, execute_table
add     x9, x9, :lo12:execute_table
ldr     x4, [x9, x8, lsl #3]
br      x4
```

think-cell

Interlude: Branch *target* prediction

Simplified

Conditional branch, fixed target:

```
.loop:
    ldrb w8, [x0]
    cmp  w8, #0
    b.eq .push
    cmp  w8, #1
    b.eq .add
    …
    b .exit
```

Unconditional branch, variable target:

```
adrp    x9, execute_table
add     x9, x9, :lo12:execute_table
ldr     x4, [x9, x8, lsl #3]
br      x4
```

Branch target prediction: determine *where* a branch is going.

Workaround bad branch target prediction

```
[[clang::musttail]] return execute_table[int(ip->op)](…);
```

Workaround bad branch target prediction

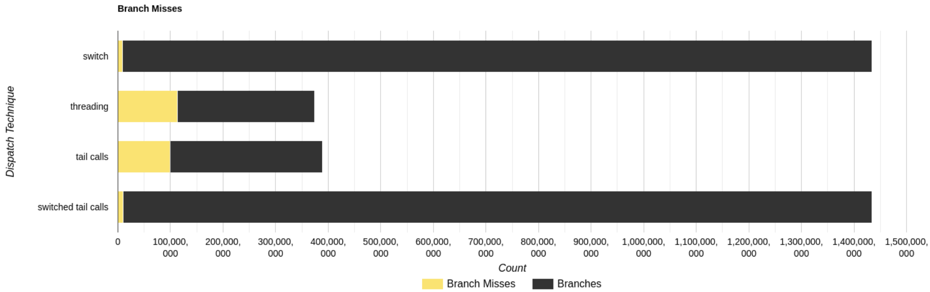
```
[[clang::musttail]] return execute_table[int(ip->op)](…);
```

```
switch (ip->op)
{
case bytecode_op::push:
    [[clang::musttail]] return do_execute_push(…);
case bytecode_op::add:
    [[clang::musttail]] return do_execute_add(…);
…
default:
    __builtin_unreachable();
}
```

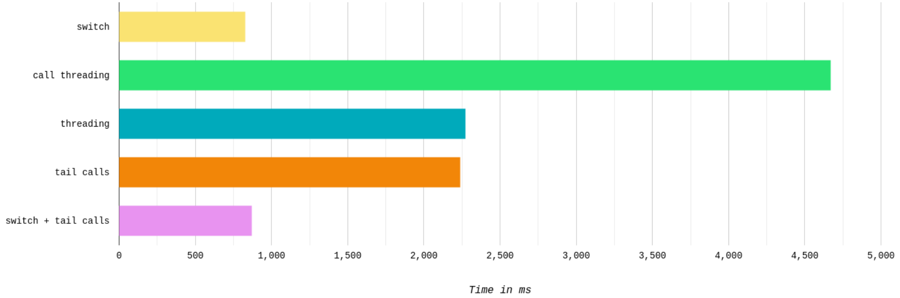
think-cell

Branch misses

2016 Thinkpad 13 running Arch Linux and clang 14.



2016 Thinkpad 13 running Arch Linux and clang 14.



Conclusion?

Trust the compiler to do dispatching, it knows best.

think-cell

Actual generated assembly for a switch

```
if (ip->op < 4) // 0-3
{
    if (ip->op <= 1)  // 0-1
    {
        if (ip->op == 0)
            goto push;
        else
            goto add;
    }
    else // 2-3
    {
        …
    }
}
else
{
```

Actual actual generated assembly for a switch

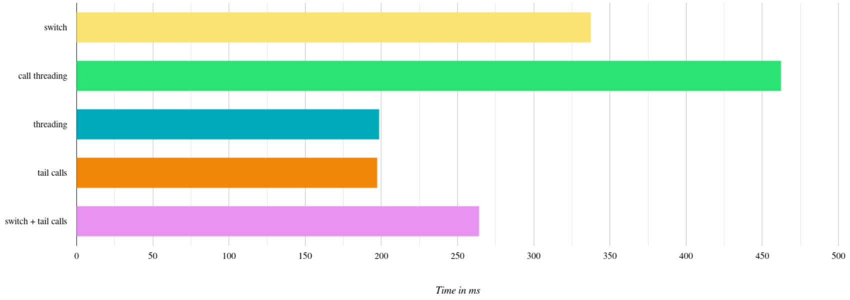
```
.loop:
    ldrb    w8, [x23]
    adr     x9, .push
    ldrb    w10, [x24, x8]
    add     x9, x9, x10, lsl #2
    br      x9

.push:
    …

.add:
    …
```

Actual actual generated assembly for a switch

```
.loop:
    ldrb    w8, [x23]
    adr     x9, .push
    ldrb    w10, [x24, x8]
    add     x9, x9, x10, lsl #2
    br      x9

.push:
    …

.add:
    …
```

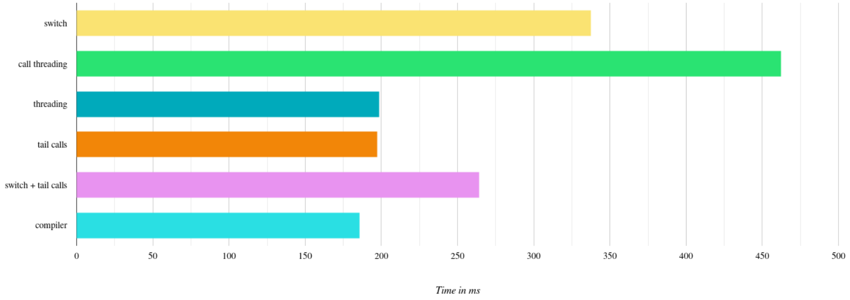
That's a jump table.

New new benchmarks: 2021 Thinkpad X1 Carbon (Intel® Core™ i5-1145G7) running Arch Linux and clang 15.



Benchmarks

New new benchmarks: 2021 Thinkpad X1 Carbon (Intel® Core™ i5-1145G7) running Arch Linux and clang 15.



think-cell

Generated assembly on x86_64

Manual jump table

```
movzx   eax, byte ptr [rbx]                 ; rax := ip->op
jmp     qword ptr [r13 + 8*rax]             ; goto *execute_table[rax]
```

Generated assembly on x86_64

Manual jump table

```
movzx   eax, byte ptr [rbx]              ; rax := ip->op
jmp     qword ptr [r13 + 8*rax]          ; goto *execute_table[rax]
```

Switch jump table

```
movzx   eax, byte ptr [rbx]              ; rax := ip->op
movsxd  rax, dword ptr [r13 + 4*rax]     ; rax := execute_table[rax]
add     rax, r13                         ; rax := rax + &execute_table
jmp     rax                              ; goto
```

Generated assembly on x86_64

Manual jump table

```
movzx    eax, byte ptr [rbx]             ; rax := ip->op
jmp      qword ptr [r13 + 8*rax]         ; goto *execute_table[rax]
```

Switch jump table

```
movzx    eax, byte ptr [rbx]             ; rax := ip->op
movsxd   rax, dword ptr [r13 + 4*rax]    ; rax := execute_table[rax]
add      rax, r13                        ; rax := rax + &execute_table
jmp      rax                             ; goto
```

Compiler generates jump table with 4 byte relative offsets, not 8 byte absolute offsets.

think-cell

Conclusion

???

Conclusion

???

Benchmark on the target hardware, then optimize.

Advanced dispatch techniques

Token-threaded dispatch

Computed goto

```
enum class bytecode_op
{
  push,
  …
};

std::array execute_table
  = {&&do_execute_push, …};

do_execute_push:
  …
  goto *execute_table[ip->op];
```

Tail calls

```
enum class bytecode_op
{
  push,
  …
};

std::array execute_table
  = {&do_execute_push, …};

int do_execute_push(…) {
  …
  return execute_table[ip->op](…);
}
```

Direct-threaded dispatch

Computed goto

```
namespace bytecode_op
{
  void* push;
  …
}

do_execute_push:
  …
  goto *ip;
```

Tail calls

```
namespace bytecode_op
{
  int push(…);
  …
}

int bytecode_op::push(…) {
  …
  return ip(…);
}
```

think-cell

Generated assembly

```
do_execute_push:
    ldrb    w8, [x0, #8]
    str     w8, [x1], #4
    ldr     x4, [x0, #16]!
    br      x4
```

think-cell

Direct-threaded dispatch

Pro best dispatch code so far

Direct-threaded dispatch

Pro best dispatch code so far

Con

- Opcode is 64-bit
- Requires branch target prediction
- Trivial remote code execution exploits possible

think-cell

Inline-threaded dispatch

```
// 1 + 2
push 1;


push 2;



add;
```

Inline-threaded dispatch

```
// 1 + 2
push 1;


push 2;



add;
```

```
ldrb    w8, [x0, #1]
str     w8, [x1], #4
add     x0, x0, #2
ldrb    w8, [x0, #1]
str     w8, [x1], #4
add     x0, x0, #2
ldr     w8, [x1, #-4]!
ldur    w9, [x1, #-4]
add     w8, w9, w8
stur    w8, [x1, #-4]
add     x0, x0, #1
```

think-cell

Inline-threaded dispatch

```
// 1 + 2
push 1;


push 2;



add;
```

```
ldrb    w8, [x0, #1]
str     w8, [x1], #4
add     x0, x0, #2
ldrb    w8, [x0, #1]
str     w8, [x1], #4
add     x0, x0, #2
ldr     w8, [x1, #-4]!
ldur    w9, [x1, #-4]
add     w8, w9, w8
stur    w8, [x1, #-4]
add     x0, x0, #1
```

Copy & paste assembly

think-cell

Inline-threaded dispatch

Pro the fastest dispatch is no dispatch
Con requires JIT compilation

Conclusion

Benchmark on the target hardware, then optimize.

jonathanmueller.dev/talk/deep-dive-dispatch

Mastodon: @foonathan@fosstodon.org

YouTube: youtube.com/@foonathan

think-cell