Mu for Dynamic Languages

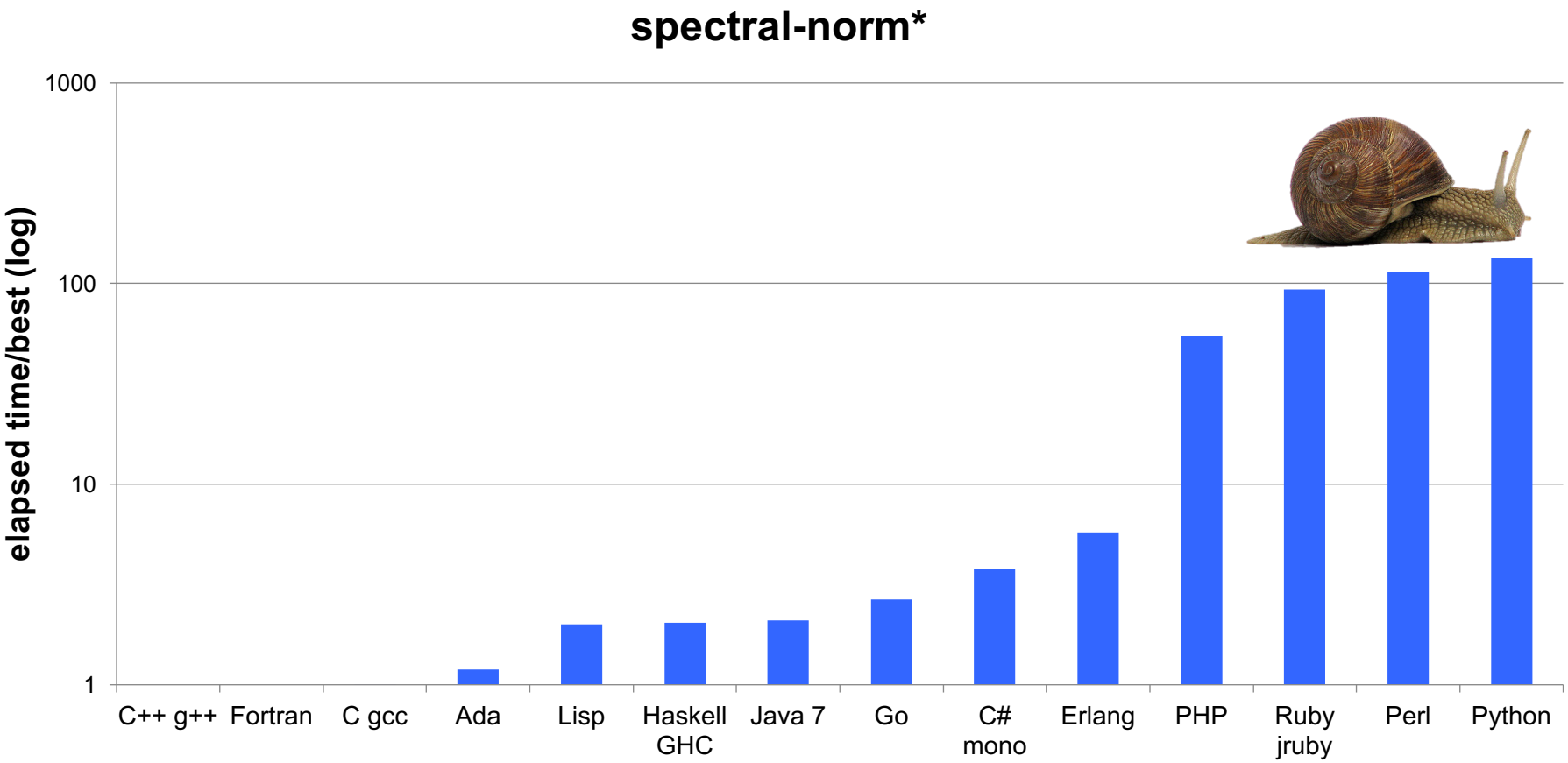
Zixian Cai

Supervisors: Steve Blackburn, Tony Hosking, Michael Norrish

```c
int NUM = 111181111;

int is_prime(int n) {
    int i;
    for(i = 2; i < n; i++) {
      if (n % i == 0) {
        return 0;
      }
    }
    return 1;
}
```

```python
NUM = 111181111

def is_prime(n):
    i = 2
    while i < n:
        if n % i == 0:
            return False
        i += 1

    return True
```

0.624s 25X difference! 15.609s

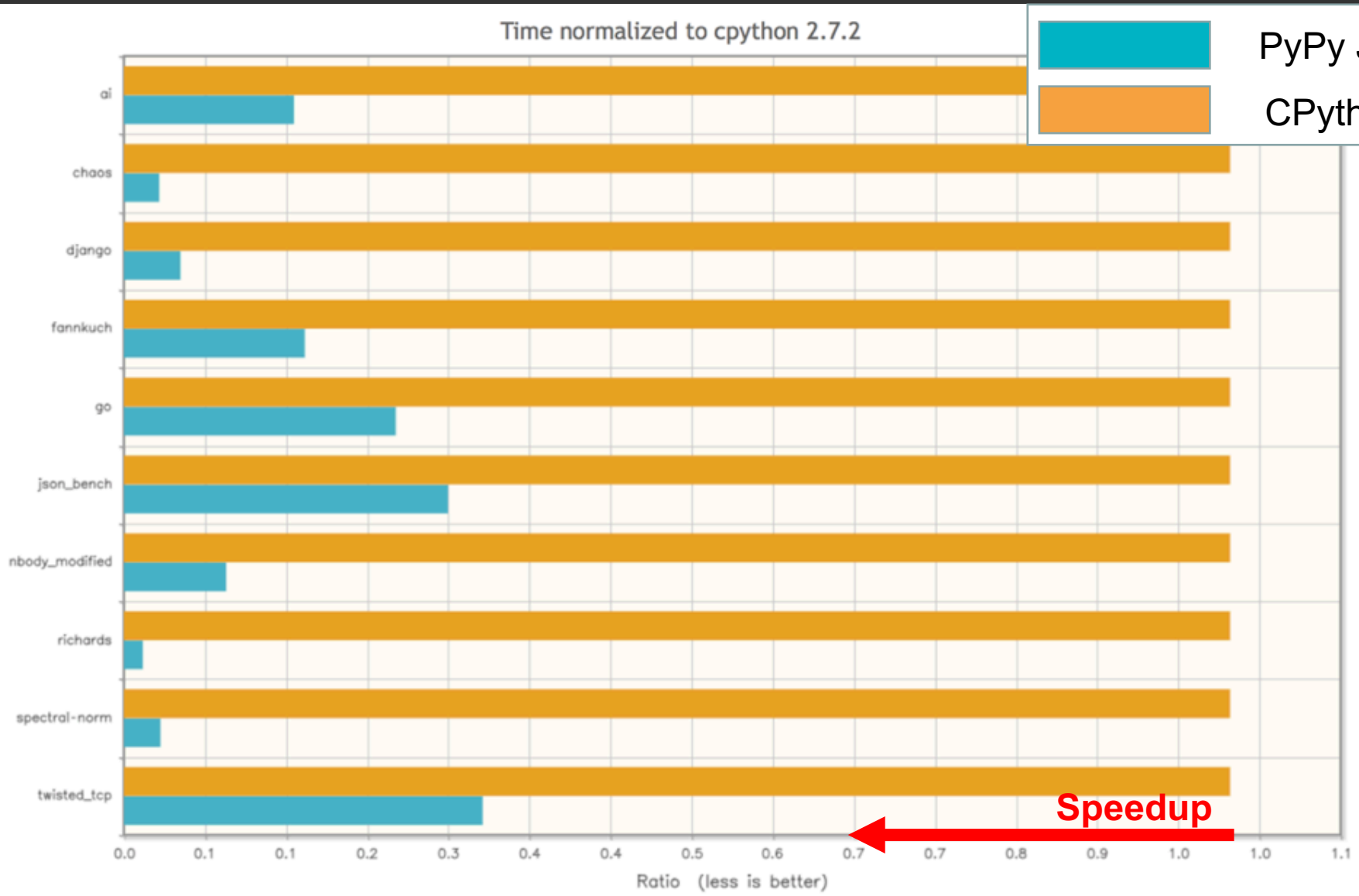


134X difference!

*these numbers are actually pretty meaningless, but the graph makes a point

Source: http://benchmarksgame.alioth.debian.org

Approach 1: PyPy



Source: http://speed.pypy.org/comparison/

Complexity

Assembler in PyPy

```
if isinstance(scaled_loc, ImmedLoc):
    # FIXME: What if base_loc is ebp or esp?
    self._location_code = 'm'
    self.loc_m = (base_loc.value, (scaled_loc.value << scale) + static_offset)
```

```
JM1_l = insn('\xE9', relative(1))
JM1_r = insn(rex_nw, '\xFF', orbyte(4<<3), register(1), '\xC0')
# FIXME: J_il8 and JMP_l8 assume the caller will do the appropriate
# calculation to find the displacement, but J_il does it for the caller.
# We need to be consistent.
JM1_l8 = insn('\xEB', immediate(1, 'b'))
J_il8 = insn(immediate(1, 'o'), '\x70', immediate(2, 'b'))
J_il = insn('\x0F', immediate(1,'o'), '\x80', relative(2))
```

```
mc.SUB_ri(esp.value, 16 - WORD)  # restore 16-byte alignment
# magically, the above is enough on X86_32 to reserve 3 stack places
if kind == 'fixed':
    mc.SUB_rr(edx.value, ecx.value) # compute the size we want
    if IS_X86_32:
        mc.MOV_sr(0, edx.value)     # store the length
        if hasattr(self.cpu.gc_ll_descr, 'passes_frame'):
            mc.MOV_sr(WORD, ebp.value)        # for tests only
    else:
        mc.MOV_rr(edi.value, edx.value)   # length argument
        if hasattr(self.cpu.gc_ll_descr, 'passes_frame'):
            mc.MOV_rr(esi.value, ebp.value)   # for tests only
elif kind == 'str' or kind == 'unicode':
    if IS_X86_32:
        # stack layout: [---][---][---][ret].. with 3 free stack places
        mc.MOV_sr(0, edx.value)     # store the length
    elif IS_X86_64:
        mc.MOV_rr(edi.value, edx.value)   # length argument
else:
    if IS_X86_32:
        # stack layout: [---][---][---][ret][gcmap][itemsize]...
        mc.MOV_sr(WORD * 2, edx.value)  # store the length
        mc.MOV_sr(WORD * 1, ecx.value)  # store the tid
        mc.MOV_rs(edx.value, WORD * 5)  # load the itemsize
        mc.MOV_sr(WORD * 0, edx.value)  # store the itemsize
    else:
        # stack layout: [---][ret][gcmap][itemsize]...
        # (already in edx)              # length
        mc.MOV_rr(esi.value, ecx.value) # tid
        mc.MOV_rs(edi.value, WORD * 3)  # load the itemsize
```

```
$ cloc x86
      64 text files.
      64 unique files.
       3 files ignored.


github.com/AlDanial/cloc v 1.74  T=0.48 s (133.5 files/s, 24068.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          63           1340           1412           8764
Bourne Shell                     1              4              4             12
-------------------------------------------------------------------------------
SUM:                            64           1344           1416           8776
-------------------------------------------------------------------------------
```

```
$ cloc arm
      61 text files.
      61 unique files.
       4 files ignored.


github.com/AlDanial/cloc v 1.74  T=0.42 s (146.3 files/s, 25571.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          60           1279           1177           8201
Bourne Shell                     1              0              0              2
-------------------------------------------------------------------------------
SUM:                            61           1279           1177           8203
-------------------------------------------------------------------------------
```

```
$ cloc ppc
      56 text files.
      55 unique files.
       3 files ignored.


github.com/AlDanial/cloc v 1.74  T=0.33 s (166.5 files/s, 29596.7 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          55           1366           1208           7203
-------------------------------------------------------------------------------
SUM:                            55           1366           1208           7203
-------------------------------------------------------------------------------
```

```
$ cloc zarch
      48 text files.
      48 unique files.
       5 files ignored.


github.com/AlDanial/cloc v 1.74  T=0.35 s (137.7 files/s, 27312.7 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          48           1257           1061           7203
-------------------------------------------------------------------------------
SUM:                            48           1257           1061           7203
-------------------------------------------------------------------------------
```

Approach 2: Repurpose Existing JIT

On the Benefits and Pitfalls of Extending a Statically Typed Language JIT Compiler for Dynamic Scripting Languages

Jose Castanos David Edelsohn Kazuaki Ishizaki Priya Nagpurkar Toshio Nakatani
Takeshi Ogasawara Peng Wu

IBM Thomas J. Watson Research Center
IBM Research - Tokyo
{castanos,edelsohn,pnagpurkar,pengwu}@us.ibm.com
{nakatani,takeshi}@jp.ibm.com

is an example of this problem. As a state-of-the-art, RJIT compiler for Python, the Fiorano JIT compiler outperforms two other RJIT compilers (Unladen Swallow and Jython), but still shows a noticeable performance gap compared to PyPy, today's best performing Python JIT compiler. In this

hoped-for performance boosts. The performance of JVM languages, for instance, often lags behind standard interpreter implementations. Even more customized solutions that extend the internals of a JIT compiler for the target language compete poorly with those designed specifically for dynamically typed languages. Our own Fiorano JIT compiler is an example of this problem. As a state-of-the-art, RJIT compiler for Python, the Fiorano JIT compiler outperforms two other RJIT compilers (Unladen Swallow and Jython), but still shows a noticeable performance gap compared to PyPy, today's best performing Python JIT compiler. In this paper, we discuss techniques that have proved effective in

1. Introduction

Dynamically typed languages are becoming increasingly popular due to productivity improvements enabled by rapid prototyping and incremental deployment cycles. While programmers rely on the flexibility of dynamic typing, higher-level data structures, and meta-programming to continuously improve their applications and unfold new features, the same features that appeal to programmers directly impact performance and make code optimization very challenging.

The initial implementation of a dynamically typed script-

Approach 3?

Mu

Abstraction

- Hardware
- Memory
- Concurrency

Mu

Abstraction

- Hardware (x86, AArch64, …)
- Memory
- Concurrency

Mu

Abstraction

- Hardware
- Memory (GC, stack/object layout)
- Concurrency

Specialization, Specialization, Specialization

```
1. Check a is int, or goto 10
2. Check b is int, or goto 10
3. %1 = get int from a
4. %2 = get int from b
5. %3 = ADD %1, %2
6. Create new int object
7. Store %3 in that object
8. Assign the object to c
9. Return
10. Check a is str, or goto 17
11. Check b is str, or goto 17
12. Create new str object
13. Copy string from a
14. Copy string from b
15. Assign the object to c
16. Return
17. Get the __dict__ from a
18. Get the __add__ method from the dict
19. Call __add__(a, b)
20. Assign the return value to c
21. Return
```

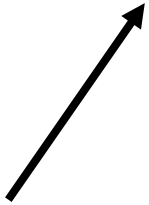
Runtime Information

```
1.  ADD r2, r0, r1
```

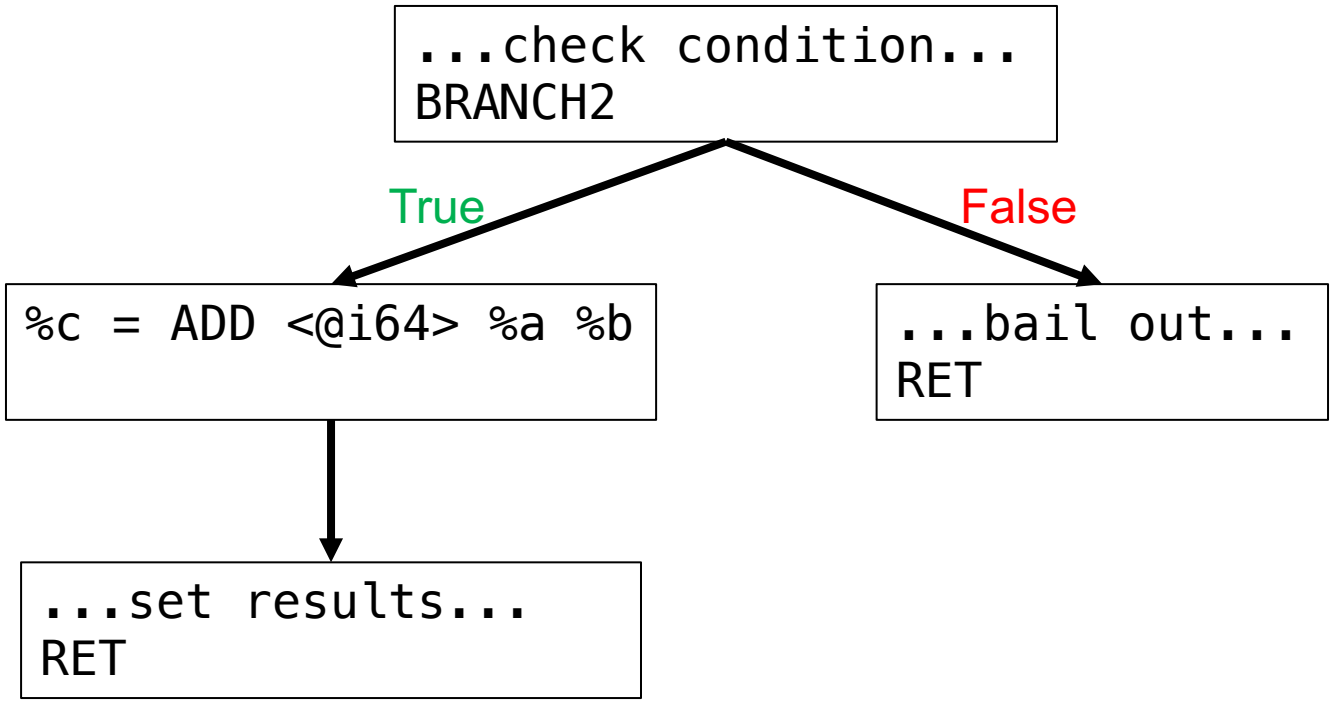
c = a + b

Runtime Conditions

```
1. guard_class(p1, ConstClass(W_IntObject))
2. guard_class(p2, ConstClass(W_IntObject))
3. i1 = getfield_gc_i(p1)
4. i2 = getfield_gc_i(p2)
5. i3 = int_add (i1, i2)
```



Specialized Code


...check condition...
BRANCH2
True
False
%c = ADD <@i64> %a %b
...bail out...
RET
...set results...
RET

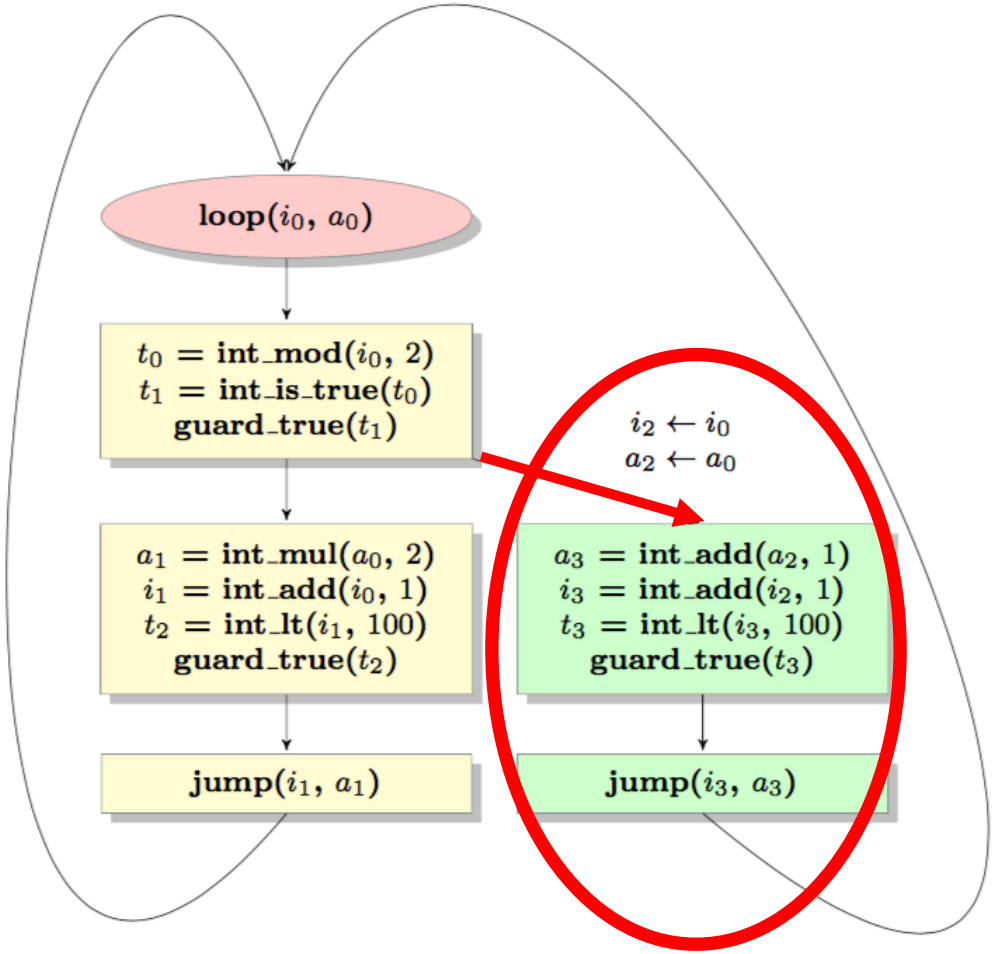
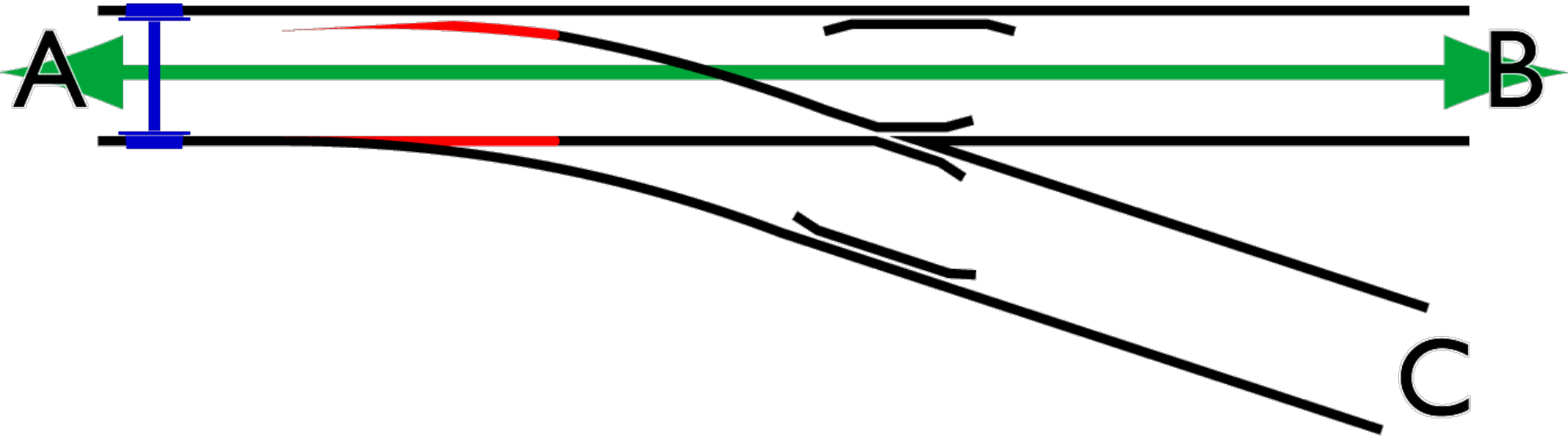
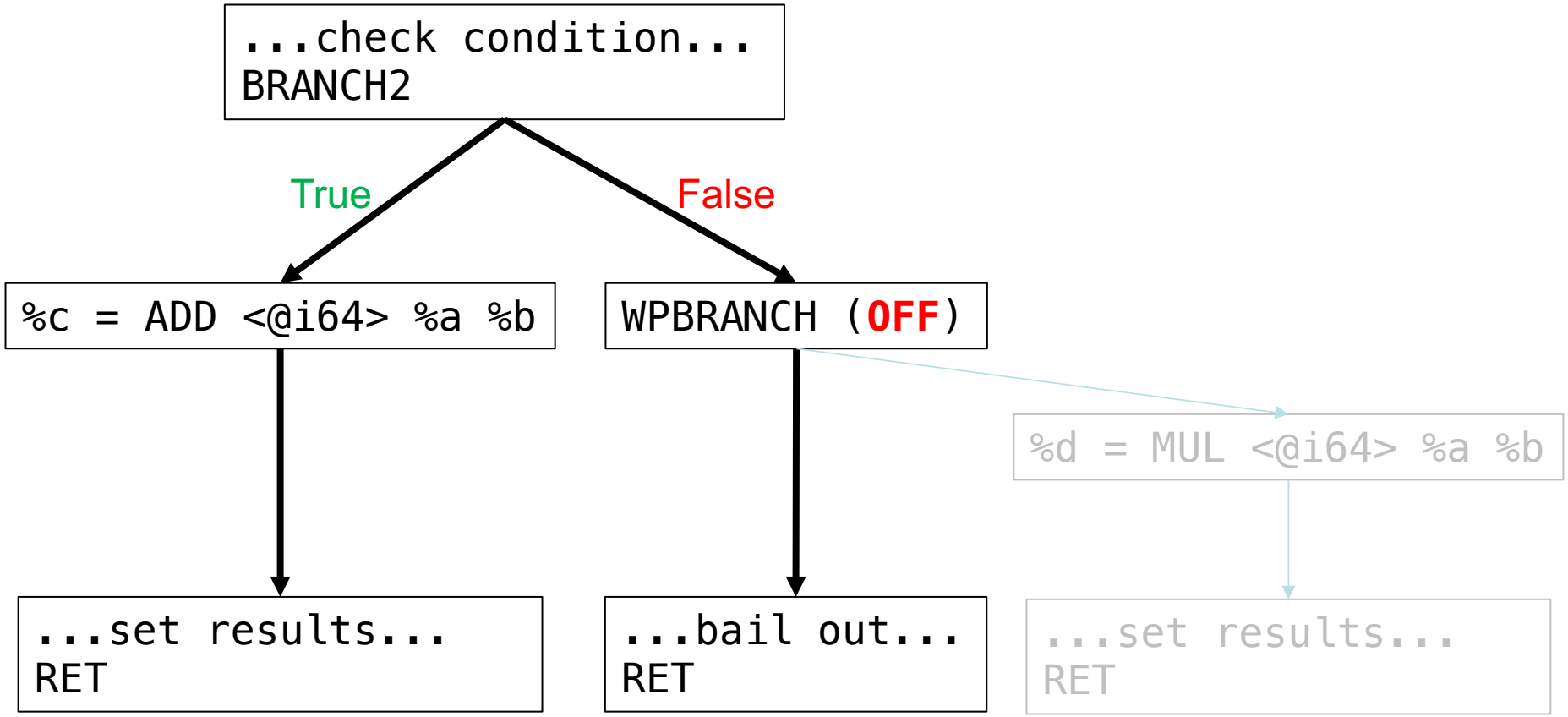
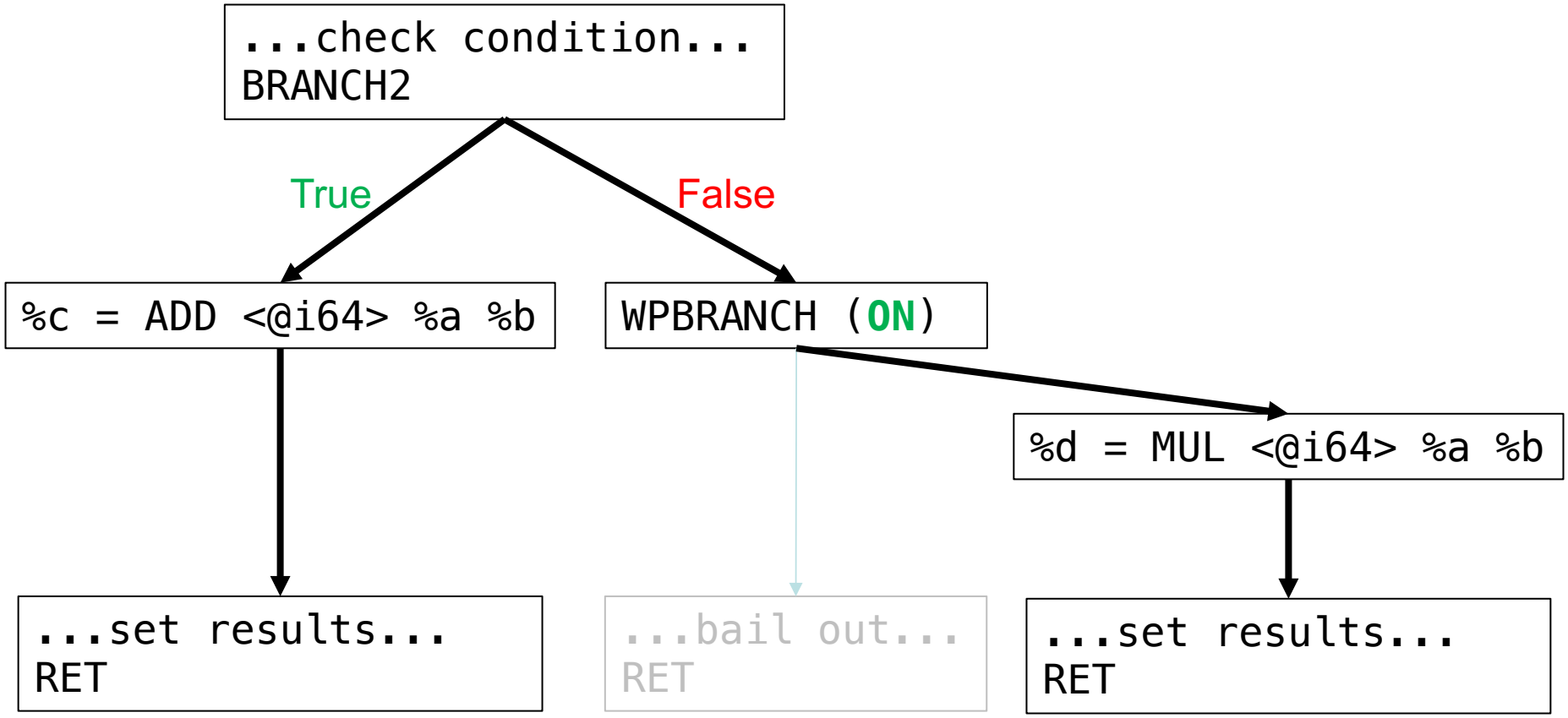


Figure 7.8: *The trace tree produced when the guard becomes hot*

Source: High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages.




...check condition...
BRANCH2
True
False
%c = ADD <@i64> %a %b
WPBRANCH (OFF)
%d = MUL <@i64> %a %b
...set results...
RET
...bail out...
RET
...set results...
RET


...check condition...
BRANCH2
True
False
%c = ADD <@i64> %a %b
WPBRANCH (ON)
%d = MUL <@i64> %a %b
...set results...
RET
...bail out...
RET
...set results...
RET

Challenges

Current Generation of JIT Bytecode

Allocate `Struct`

```
new(SizeDescr(size))
```

Allocate `Array`

```
new_array(length,
          ArrayDescr(basesize, itemsize))
```

Carrying Type Information

Allocate `Struct`

```
new(SizeDescr(size, TYPE))
```

Allocate `Array`

```
new_array(length,
          ArrayDescr(basesize, itemsize,
                     TYPE))
```

Summary

- Unique and abundant optimization opportunity: specialization
- Implementation of loops, bridges and guards
- Encode type information in descriptors