The Efficient Handling of Guards in the Design of RPython's Tracing JIT

David Schneider   Carl Friedrich Bolz

Heinrich-Heine-Universität Düsseldorf, STUPS Group, Germany

2012 VMIL, 21st of October, 2012

RPython and PyPy

- Context: RPython
- a language for writing interpreters for dynamic languages
- a generic tracing JIT, applicable to many languages
- used to implement PyPy, an efficient Python interpreter

How fast is PyPy?

Tracing JITs Compile by Observing an Interpreter

- VM contains both an interpreter and the tracing JIT compiler
- JIT works by observing and logging what the interpreter does
- for interesting, commonly executed code paths
- produces a linear list of operations (trace)

Code

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

Frames

a = a_1
j = j_1

Trace

[j_1, a_1]

Code

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

Frames

a = a_1
j = j_2

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)

Guards

- Points of control flow divergence are marked with guards
- Operations that check whether conditions are still true
- When a guard fails, execution of the trace stops and continues in the interpreter

Guards

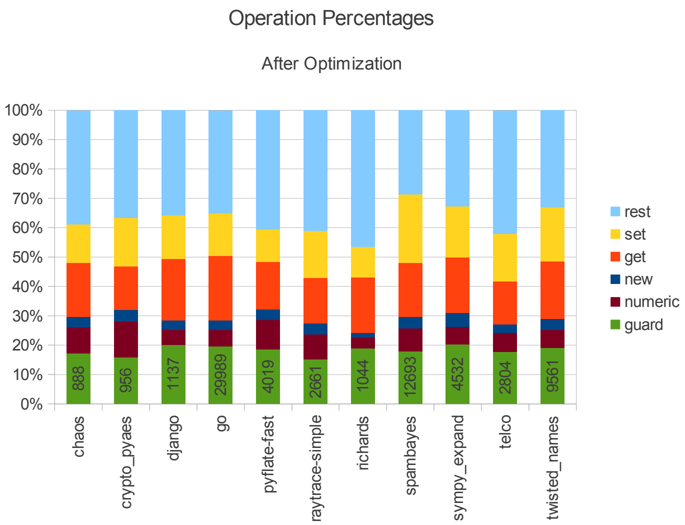
- Points of control flow divergence are marked with guards
- Operations that check whether conditions are still true
- When a guard fails, execution of the trace stops and continues in the interpreter
- similar to deoptimization points, but more common, and patchable
- This talk: technology and design decisions of guards

Guards

- Points of control flow divergence are marked with guards
- Operations that check whether conditions are still true
- When a guard fails, execution of the trace stops and continues in the interpreter
- similar to deoptimization points, but more common, and patchable
- This talk: technology and design decisions of guards

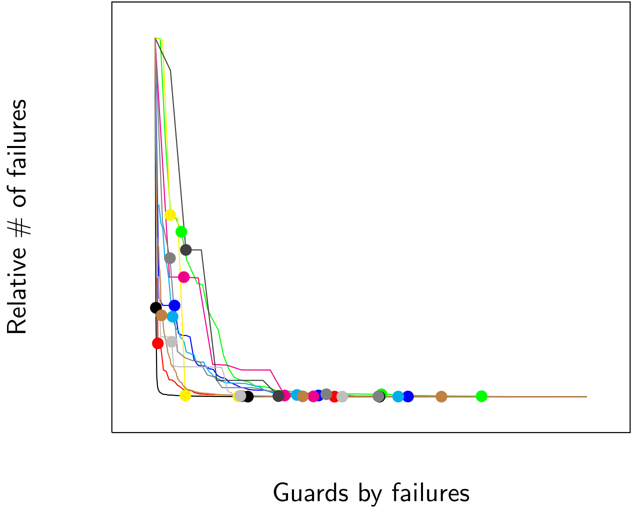
Guard Characteristics

- lots of them, up to 20% guards
- most never fail
- need big information attached to them


Operation Percentages
After Optimization
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
888
956
1137
29989
4019
2661
1044
12693
4532
2804
9561
chaos
crypto_pyaes
django
go
pyflate-fast
raytrace-simple
richards
spambayes
sympy_expand
telco
twisted_names
rest
set
get
new
numeric
guard

Guard Failure Rates / Go Benchmark (29989 Guards)

Guard Failure Rates



Code | Frames | Trace

```
while j < 100:
  j += 1
  if a is None:
    break
  a = a.f()
```

a = a_1
j = j_2

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)

Inlining

Tracing automatically does (potentially deep) inlining

Code	Frames	Trace

Code

```
while j < 100:
  j += 1
  if a is None:
    break
  a = a.f()
```

Frames

a = a_1
j = j_2

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)

Code

Frames

Trace

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

a = a_1
j = j_2

n =
self = a_1

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)

Code	Frames	Trace

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

a = a_1
j = j_2

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

n = i_2
self = a_1

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)

Code	Frames	Trace

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

a = a_1
j = j_2

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

n = i_2
self = a_1

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)

Code	Frames	Trace

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

a = a_1
j = j_2

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

n = i_2
self = a_1

```
if n & 1 == 0:
    return Even(n)
else:
    return Odd(n)
```

n = i_2

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)
i_3 = int_and(i_2, 1)
i_4 = int_is_zero(i_3)
guard_true(i_4)

Code

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

```
if n & 1 == 0:
    return Even(n)
else:
    return Odd(n)
```

Frames

a = a_1
j = j_2

n = i_2
self = a_1

n = i_2

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)
i_3 = int_and(i_2, 1)
i_4 = int_is_zero(i_3)
guard_true(i_4)
a_2 = new(Even)

Code

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

```
if n & 1 == 0:
    return Even(n)
else:
    return Odd(n)
```

```
self.value = n
```

Frames

a = a_1
j = j_2

n = i_2
self = a_1

n = i_2

self = a_2

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)
i_3 = int_and(i_2, 1)
i_4 = int_is_zero(i_3)
guard_true(i_4)
a_2 = new(Even)
setfield_gc(a_2, descr='value')

Code

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

```
if n & 1 == 0:
    return Even(n)
else:
    return Odd(n)
```

```
self.value = n
```

Frames

a = a_2
j = j_2

n = i_2
self = a_1

n = i_2

self = a_2

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)
i_3 = int_and(i_2, 1)
i_4 = int_is_zero(i_3)
guard_true(i_4)
a_2 = new(Even)
setfield_gc(a_2, descr='value')
b_2 = int_lt(j_2, 100)
guard_true(b_2)

Code

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

```
if n & 1 == 0:
    return Even(n)
else:
    return Odd(n)
```

```
self.value = n
```

Frames

a = a_2
j = j_2

n = i_2
self = a_1

n = i_2

self = a_2

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)
i_3 = int_and(i_2, 1)
i_4 = int_is_zero(i_3)
guard_true(i_4)
a_2 = new(Even)
setfield_gc(a_2, descr='value')
b_2 = int_lt(j_2, 100)
guard_true(b_2)
jump(j_2, a_2)

Symbolic Frame Capturing

- Guard can fail deep inside inlined function
- when going back to the interpreter, call stack needs to be re-created
- done with the help of symbolic frame stacks
- these show how trace variables fill the to-be-built interpreter stack frames

Code

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

Frames

a = a_1
j = j_2

n = i_2
self = a_1

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)

Code

```
while j < 100:
  j += 1
  if a is None:
    break
  a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

Frames

a = a_1
j = j_2

n = i_2
self = a_1

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)

Symbolic Frame Compression

- There are a lot of guards
- Naively storing symbolic frames would be costly in terms of memory
- need to store them compactly
- observation: from one guard to the next, the non-top stack frames don't change
- share these between subsequent guards

Code	Frames	Trace

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

a = a_1
j = j_2

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

n = i_2
self = a_1

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)

Code

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

Frames

a = a_1
j = j_2

n = i_2
self = a_1

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)

Code

```
while j < 100:
    j += 1
    if a is None:
        break
    a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

```
if n & 1 == 0:
    return Even(n)
else:
    return Odd(n)
```

Frames

a = a_1
j = j_2

n = i_2
self = a_1

n = i_2

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)
i_3 = int_and(i_2, 1)
i_4 = int_is_zero(i_3)
guard_true(i_4)

Code

```
while j < 100:
  j += 1
  if a is None:
      break
  a = a.f()
```

```
n = self.value >> 2
if n == 1:
    return None
return self.build(n)
```

```
if n & 1 == 0:
    return Even(n)
else:
    return Odd(n)
```

Frames

a = a_1
j = j_2

n = i_2
self = a_1

n = i_2

Trace

[j_1, a_1]
j_2 = int_add(j_1, 1)
guard_nonnull(a_1)
guard_class(a_1, Even)
i_1 = getfield_gc(a_1, descr='value')
i_2 = int_rshift(i_1, 2)
b_1 = int_eq(i_2, 1)
guard_false(b_1)
i_3 = int_and(i_2, 1)
i_4 = int_is_zero(i_3)
guard_true(i_4)

Compact Representation

also need a byte-saving binary representation, but that's just boring work

Interaction with Optimization

- Some optimizations make it necessary to store extra information in symbolic frames

Interaction with Optimization

- Some optimizations make it necessary to store extra information in symbolic frames
- examples:
  - allocation removal (need to allocate objects before resuming)
  - delayed heap stores (need to do stores before resuming interpreter)
- can be compressed using similar techniques

Emitting Guards

Guards are compiled as

- quick check if the condition holds
- and a mapping of machine locations to JIT-variables

Emitting Guards

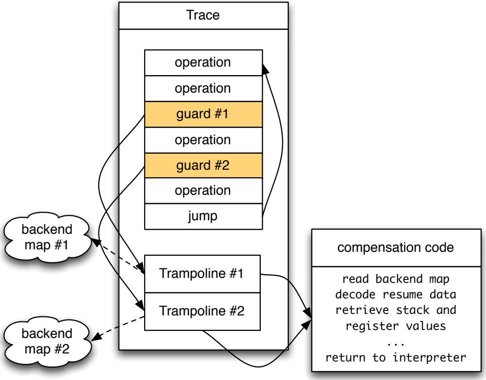
Guards are compiled as

- quick check if the condition holds
- and a mapping of machine locations to JIT-variables

In case of failure

- execution jumps to shared compensation code, decodes and stores mapping
- returns to interpreter that rebuilds state

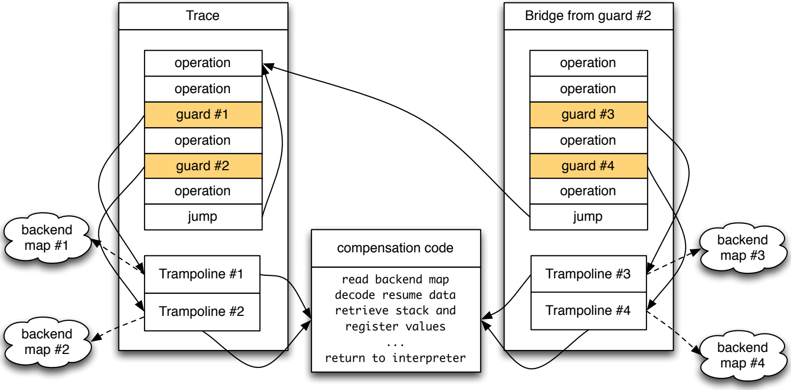
Compiling a Trace



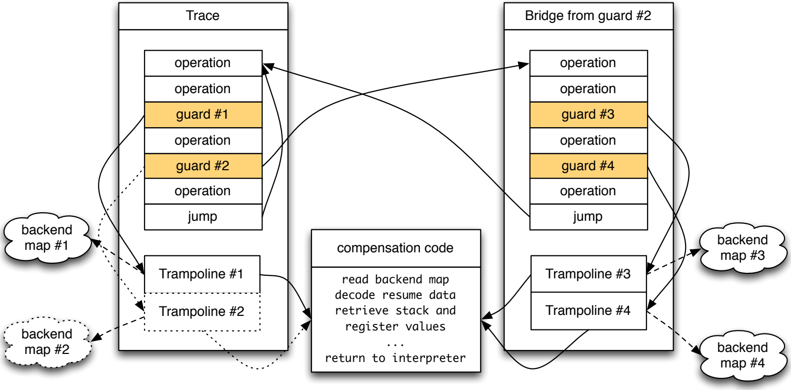
Bridges

- When a trace fails often, it becomes worth to attach a new trace to it
- This is called a bridge
- The bridge is attached by patching the guard machine code
- when this guard fails in the future, the new trace is executed instead

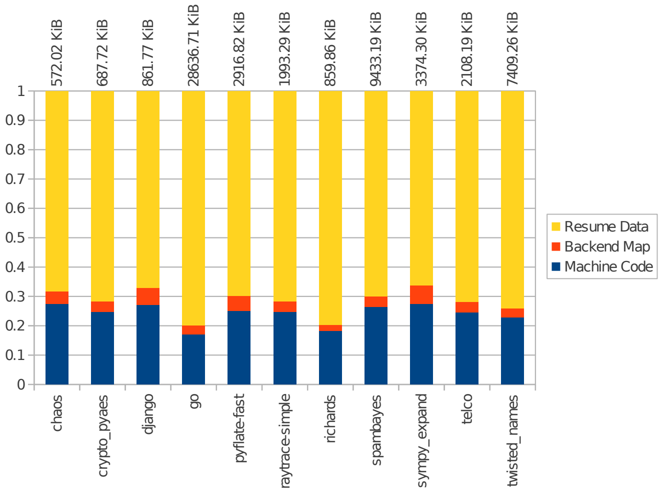
Compiling a Bridge

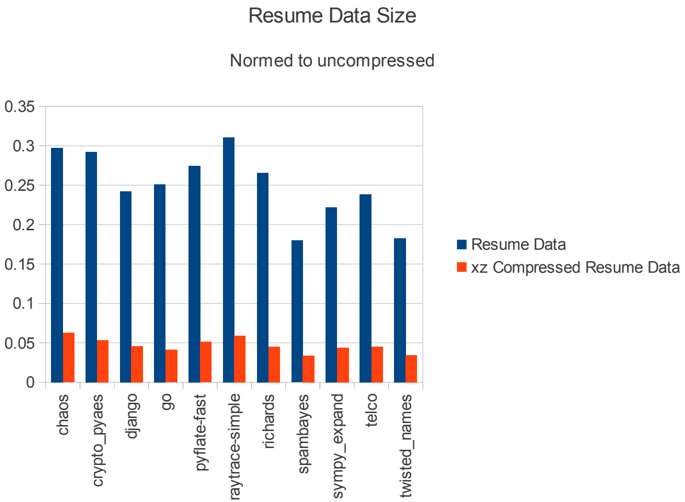


Patching Guards for Bridges



JIT memory overhead




Resume Data Size
Normed to uncompressed
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
chaos
crypto_pyaes
django
go
pyflate-fast
raytrace-simple
richards
spambayes
sympy_expand
telco
twisted_names
Resume Data
xz Compressed Resume Data

Conclusion

- Things that sound simple still often need careful engineering

Conclusion

- Things that sound simple still often need careful engineering
- guards are fundamental part of tracing JITs, need to be implemented well
- not even any direct performance gains
- keep memory usage sane
- allows good bridges

Thank you! Questions?

- Things that sound simple still often need careful engineering
- guards are fundamental part of tracing JITs, need to be implemented well
- not even any direct performance gains
- keep memory usage sane
- allows good bridges