Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines

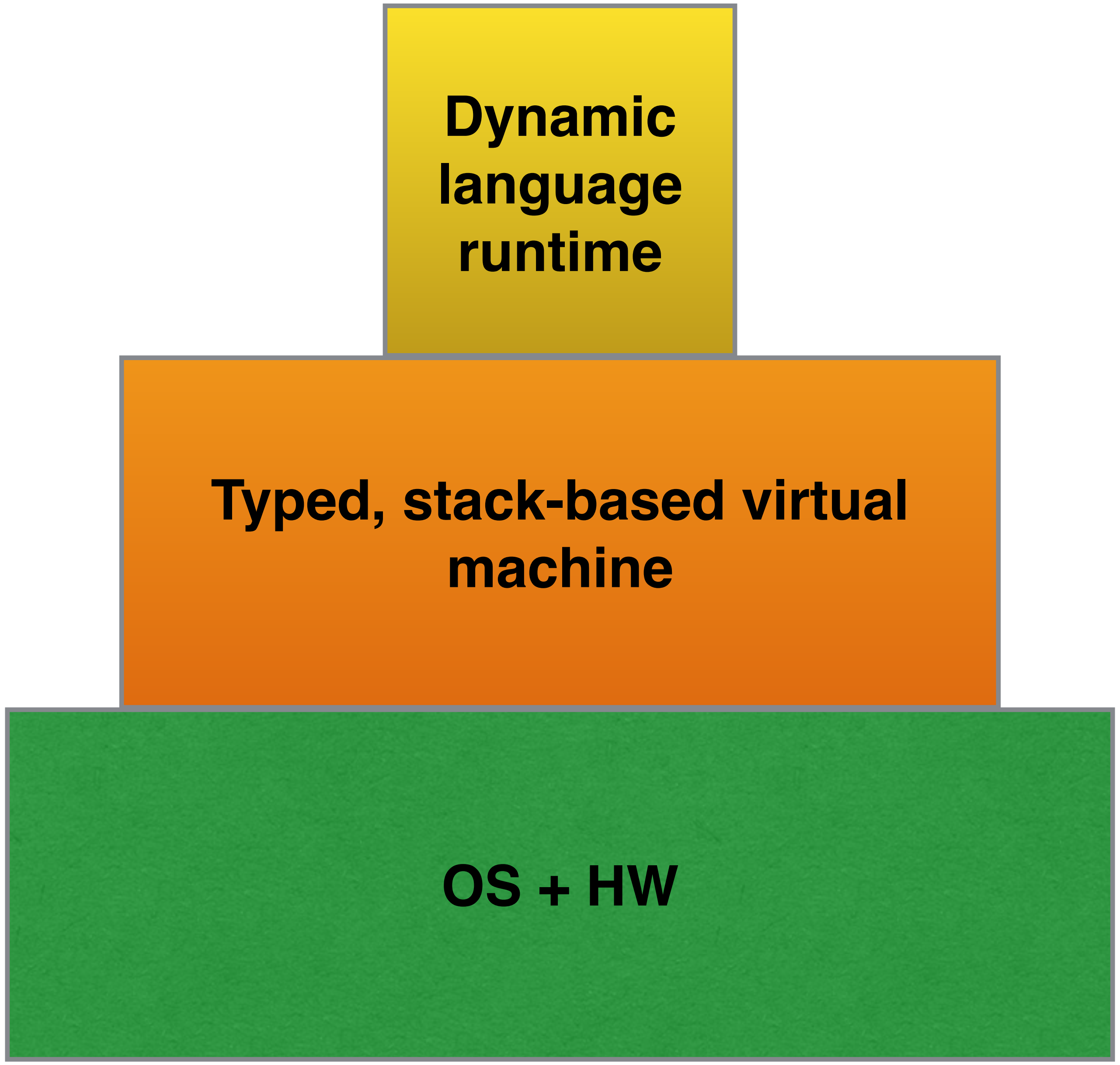
Madhukar N. Kedlaya[1], Behnam Robatmili[2], Calin Cascaval[2], Ben Hardekopf[1]

University of California, Santa Barbara[1]
Qualcomm Research Silicon Valley[2]

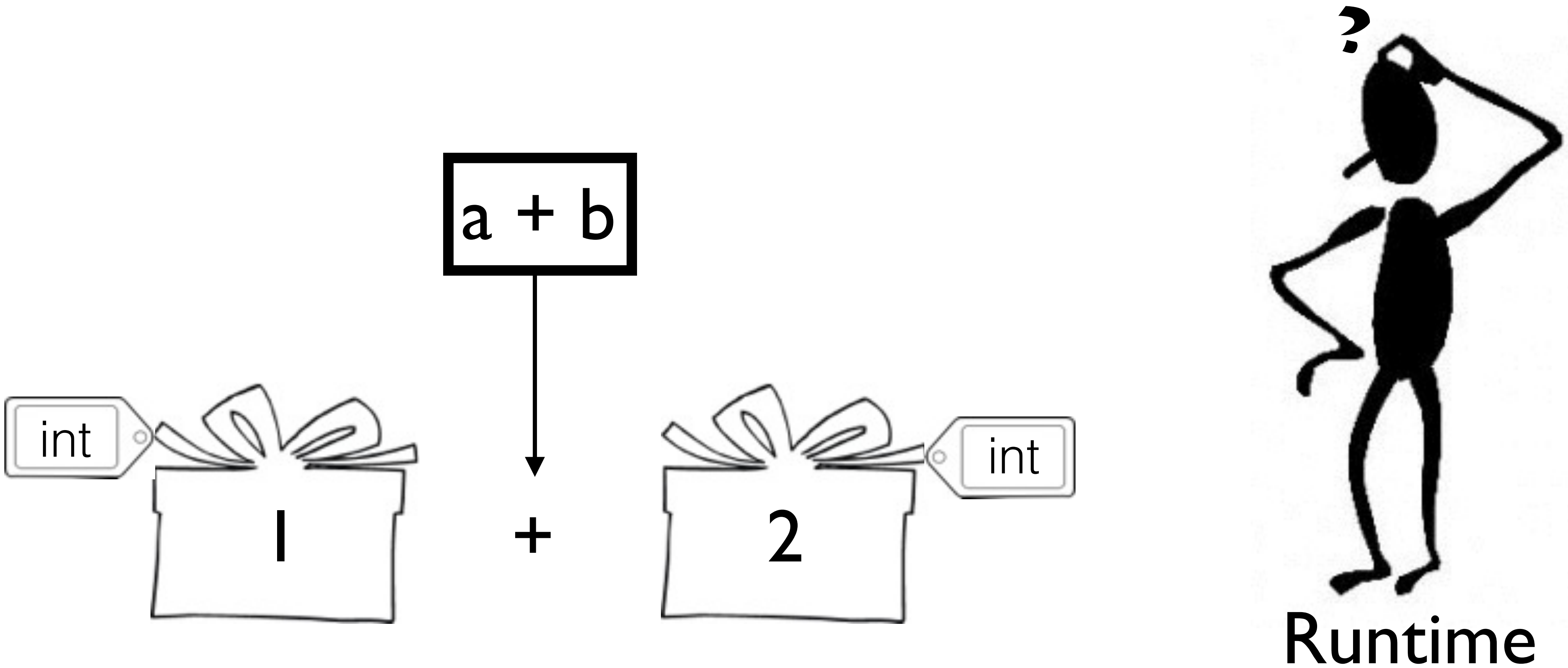
Motivation

- Interested in implementing efficient **dynamic language runtimes** on top of **virtual machines**.
- **Type specialization** is an important optimization for a dynamic language runtime.
- State-of-the-art technique of type specialization involves **deoptimization**.
- Current deoptimization techniques **do not work** on top of VMs.

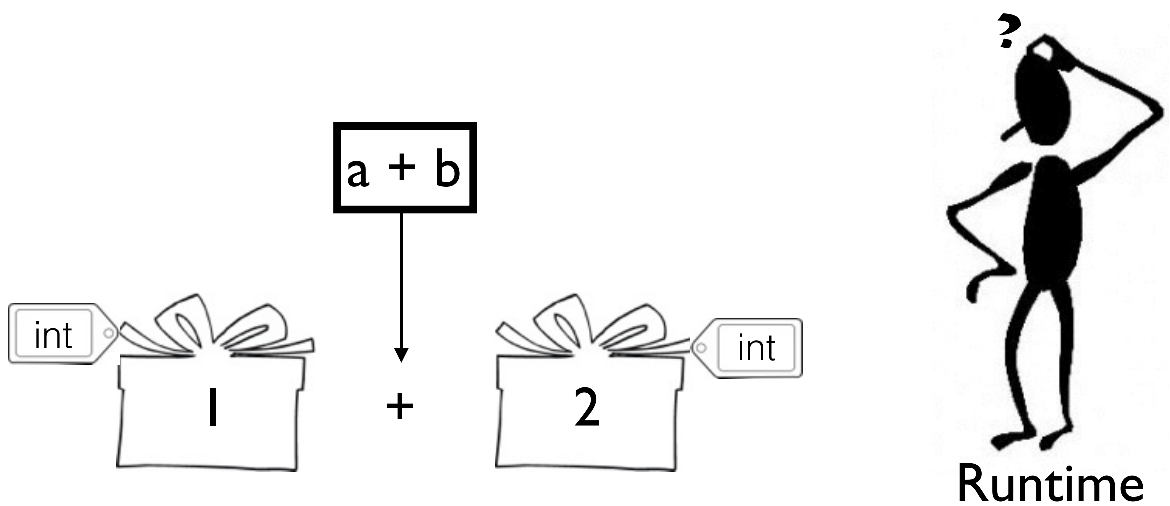
Layered Architecture



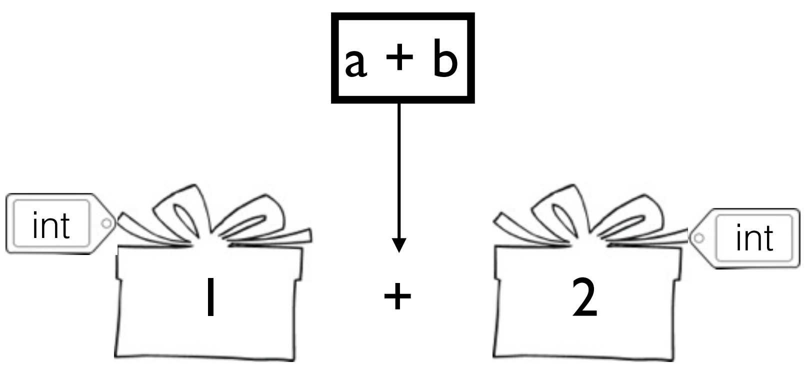
Type Specialization



Type Specialization



Type Specialization



Runtime

```
if (a.type == int && b.type == int)
  IntAdd(a.ToInt(), b.ToInt());
else if (a.type == double && b.type == double)
  DoubleAdd(a.ToDouble(), b.ToDouble());
else if (a.type == int && b.type == Double)
  DoubleAdd(a.ToInt(), b.ToDouble());
else if (a.type == double && b.type == int)
  DoubleAdd(a.ToDouble(), b.ToInt());
else if (a.type == string && b.type == string)
  StringConcat(a.ToString(), b.ToString());
…
…
```

Fast path + slow path

```
Example:
c = a + b
```

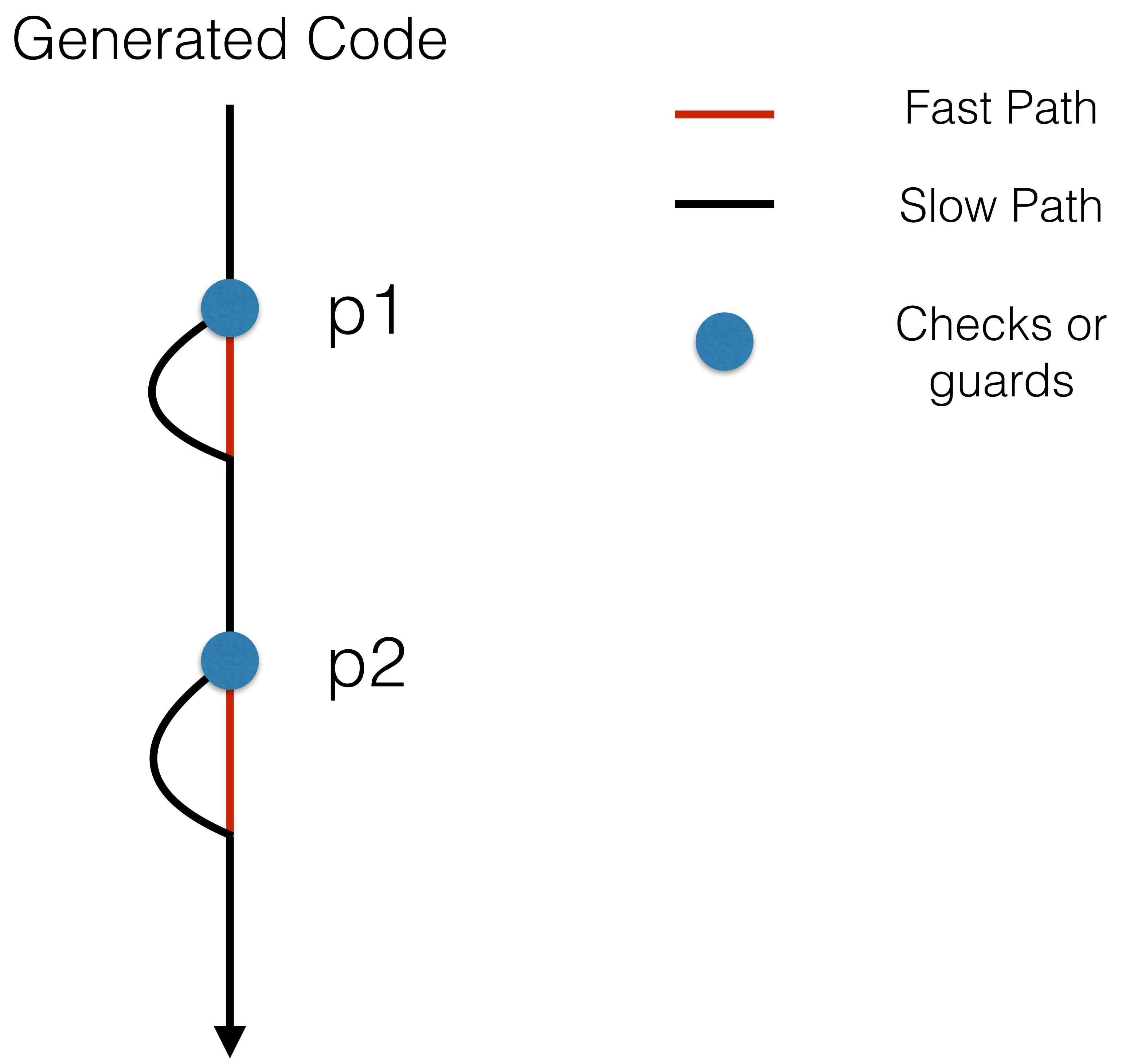
```
if (a.type == Int && b.type == Int) {
  /* Fast Path */
  c =     IntAdd(a.ToInt(), b.ToInt());
}
else {
  /* Slow Path */
  c = GenericAdd(a, b);
}
```

Fast path + slow path

```
Example:
c = a + b
```

```
if (a.type == Int && b.type == Int) {
  /* Fast Path */
  c = Box(IntAdd(a.ToInt(), b.ToInt()));
}
else {
  /* Slow Path */
  c = GenericAdd(a, b);
}
```

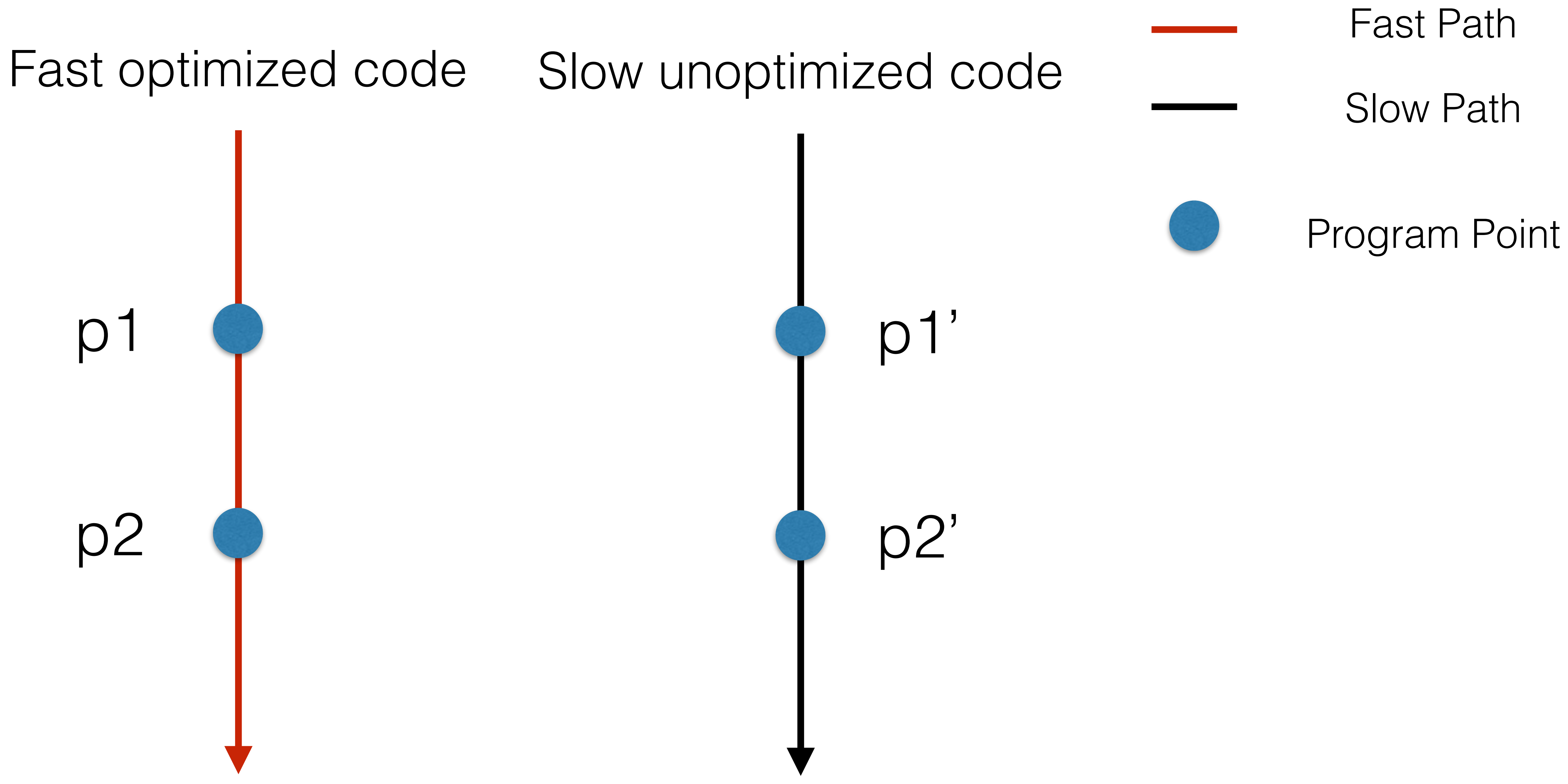
Fast path + slow path



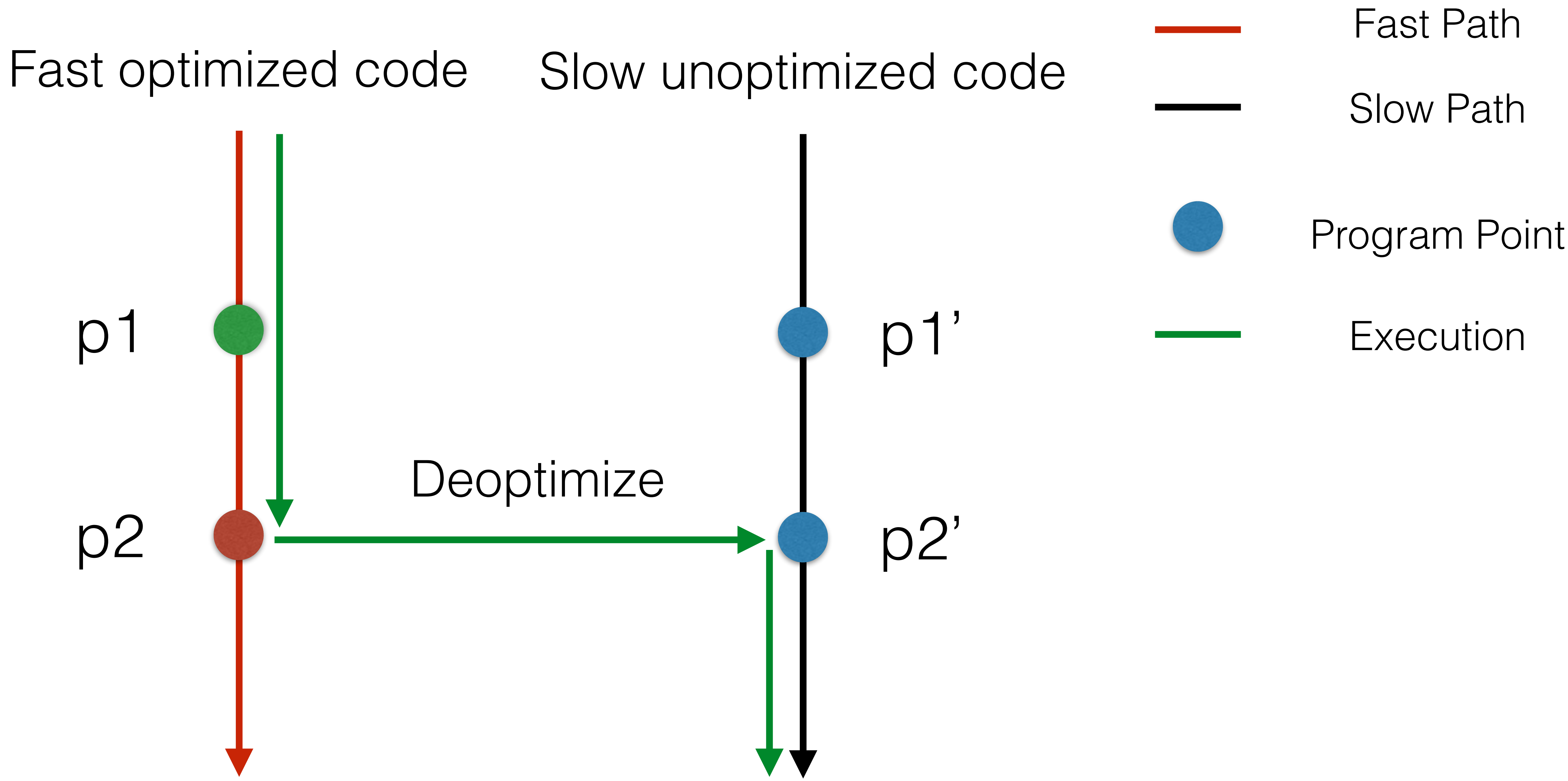
Deoptimization

- Generate only fast path of the code with type checks.
- When the assumptions do not hold, the compiled code is no longer valid.
- Deoptimization is a process of transferring the execution state from fast-optimized code to slow non-optimized code.

Deoptimization



Deoptimization



Problem

Common Deoptimization Techniques (that do not work on top of VMs)

- On-stack replacement/Code patching.
  - ✴ Cannot modify generated bytecode during execution.
- Long jumps to unoptimized code.
  - ✴ Violates bytecode verification rules.

Our Approach

- Novel deoptimization approach of code generation without modifying the underlying VM.
- **Control transfer** uses **Exception handling**
- **State transfer** uses **Bytecode verifier**
- Deoptimization target is a subroutine threaded interpreter.

Control Transfer

```
try {
  if (GetType(variable) != ProfiledType) {
    /* capture state here */
    throw new GuardFailureException(subroutineIndex);
  }
  /* fast Path */
}

catch (GuardFailureException e) {
  /* capture state here */
  SubroutineThreadedInterp(e.subroutineIndex, state);
}
```

State of Execution

- **state** data structure captures the current state of execution of the function.
- Two parts.
  - Values of **local variables**.
  - Values in **operand stack**.

State Transfer

```
try {
  if (GetType(variable) != ProfiledType) {
    for (value in operandStack) {
      state.stack.enqueue(value);
    }
    throw new GuardFailureException(subroutineIndex);
  }
  /* Fast Path */
  …
}
catch (GuardFailureException e) {
  for (variable in localVariables) {
    state.variables[variable] = GetValue(variable);
  }
  SubroutineThreadedInterp(e.subroutineIndex, state);
}
```

State Transfer

```
for (value in operandStack) {
  state.stack.enqueue(value);
}
```

Q: Which enqueue method does the runtime call?
enqueue(int)?
enqueue(string)?
enqueue(double)?
…

A: Depends on the types of values present in the operand stack.

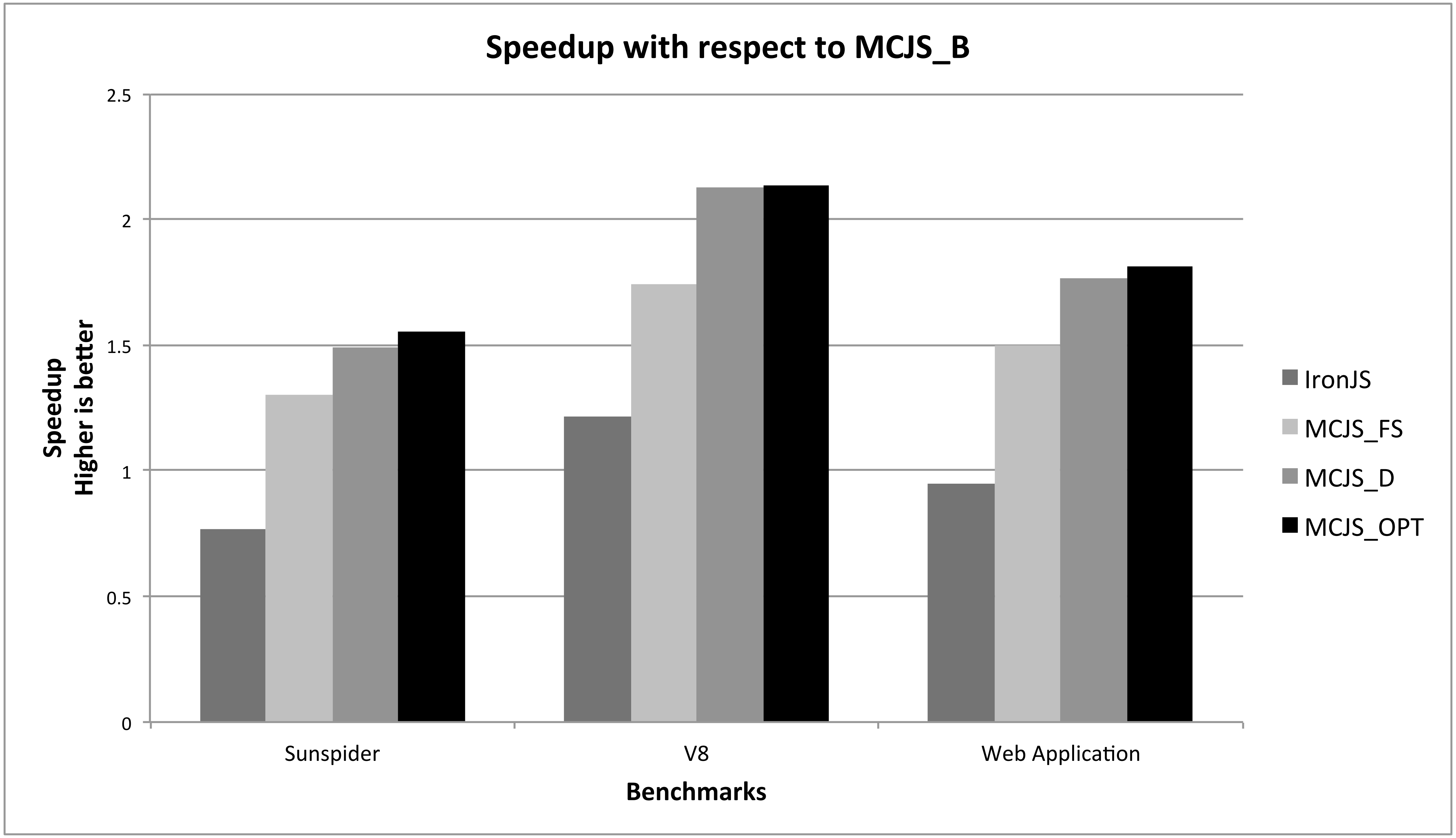
State Transfer

- Bytecode verifier checks type-safety of the Common Intermediate Language (CIL) code while generating it.
- Bytecode verifier uses a **type stack** to track the types of values in operand stack
- Code generator uses the type stack to generate the calls to proper enqueue methods at each of the deoptimization points.

Results

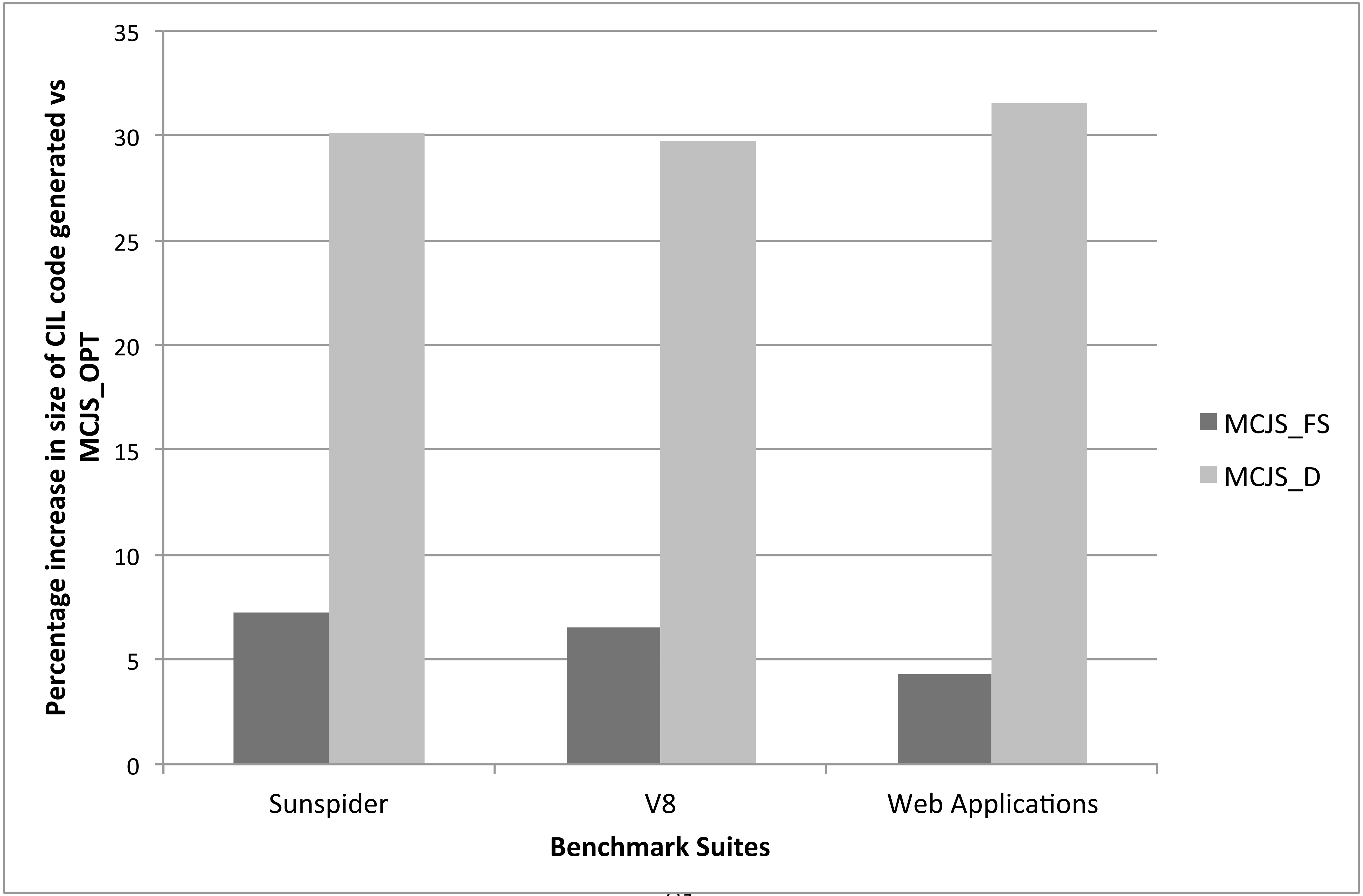
- Implemented in MCJS, a research JavaScript engine running on top of Mono 3.2.3.
- Benchmarks: Sunspider, V8, and JS1k web application benchmark suites.
- 5 different configurations
  - MCJS_B - MCJS baseline without any type specialization
  - MCJS_FS - MCJS with fast path + slow path
  - MCJS_D - MCJS with fast path + deoptimization
  - MCJS_OPT - MCJS with fast path without deoptimization (unsound)
  - IronJS - A DLR based implementation with fast path + slow path

Speedups



Speedup with respect to MCJS_B
Speedup
Higher is better
2.5
2
1.5
1
0.5
0
Sunspider
V8
Web Application
Benchmarks
IronJS
MCJS_FS
MCJS_D
MCJS_OPT

Code Bloat



35
30
25
20
15
10
5
0
Percentage increase in size of CIL code generated vs MCJS_OPT
Sunspider
V8
Web Applications
Benchmark Suites
MCJS_FS
MCJS_D

Conclusion and Future Work

- Novel **deoptimization** based **type-specialized** code generation for a **dynamic language runtime** implemented on top of a typed, stack-based **virtual machine**.
- Does not require modifications to the VM
- **1.16×** and **1.88×** speedup vs. fast path + slow path approach implemented in MCJS and IronJS respectively.
- Deoptimization technique is generic; and can be extended to implement function inlining with minor modifications.

Questions?