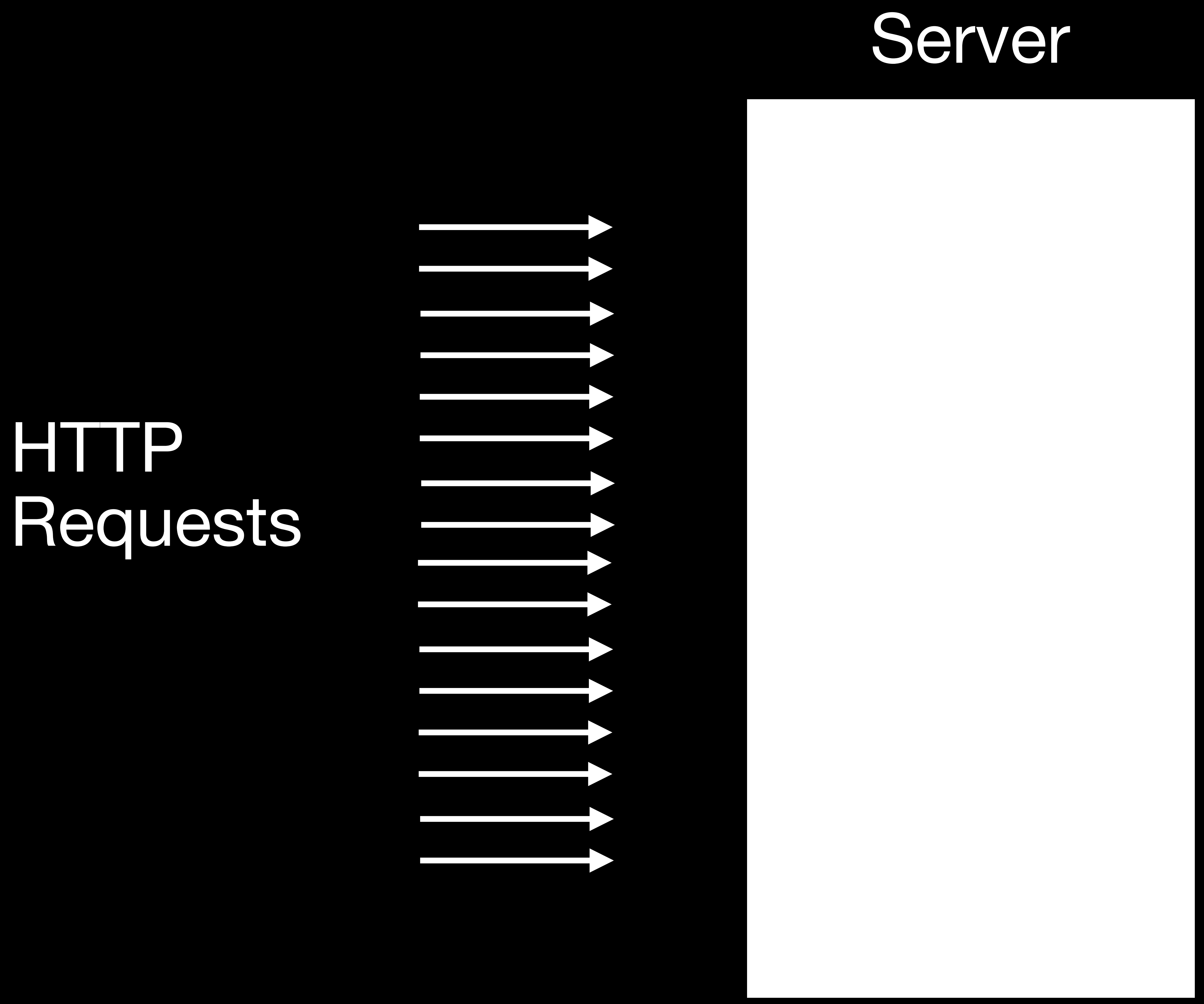
wevaling the wasms: AOT JS Compilation

Or: Stuffing a Dynamic Language onto a Very Static Platform

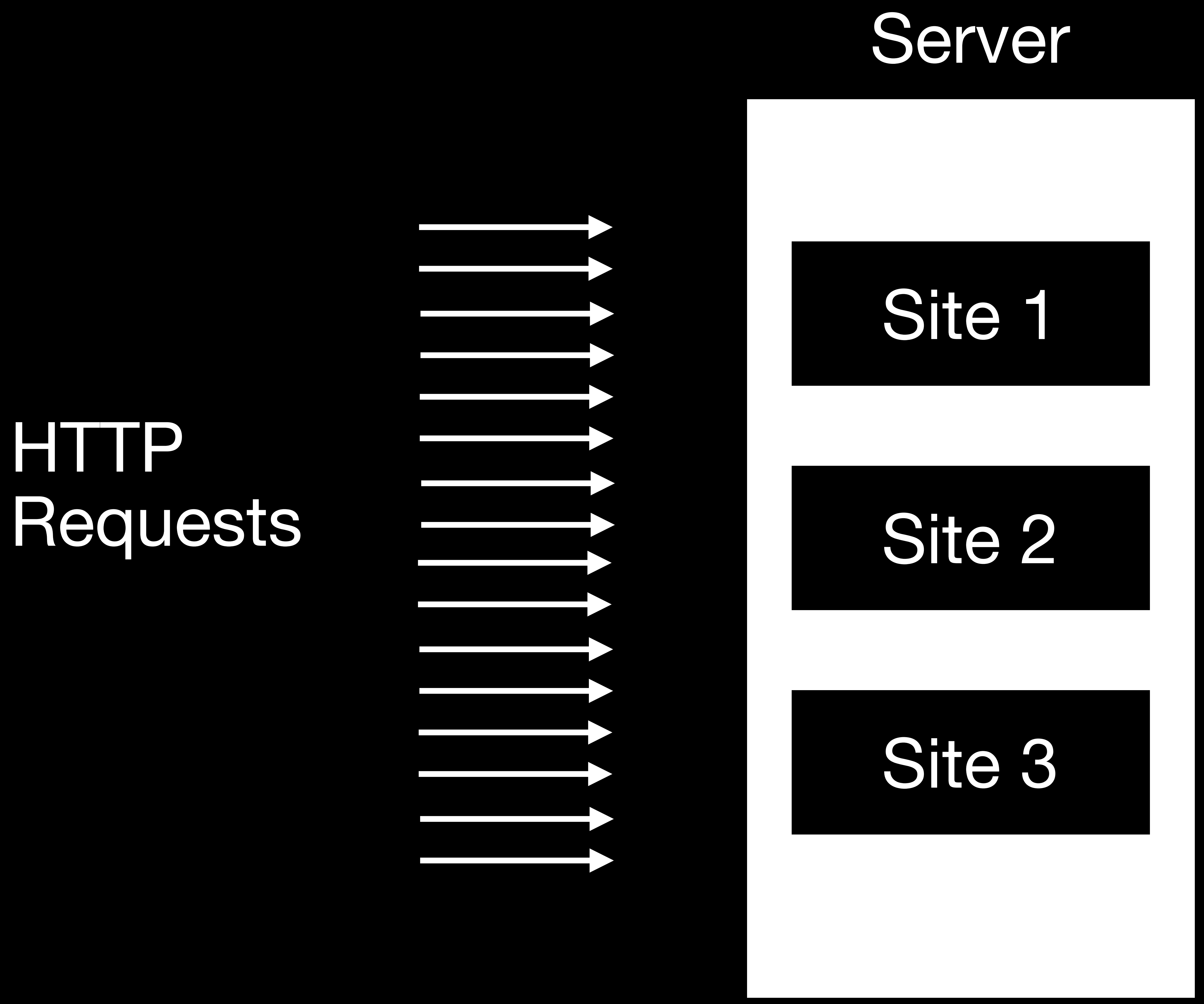
Chris Fallin *(Principal Software Engineer @ Fastly)*

The Problem

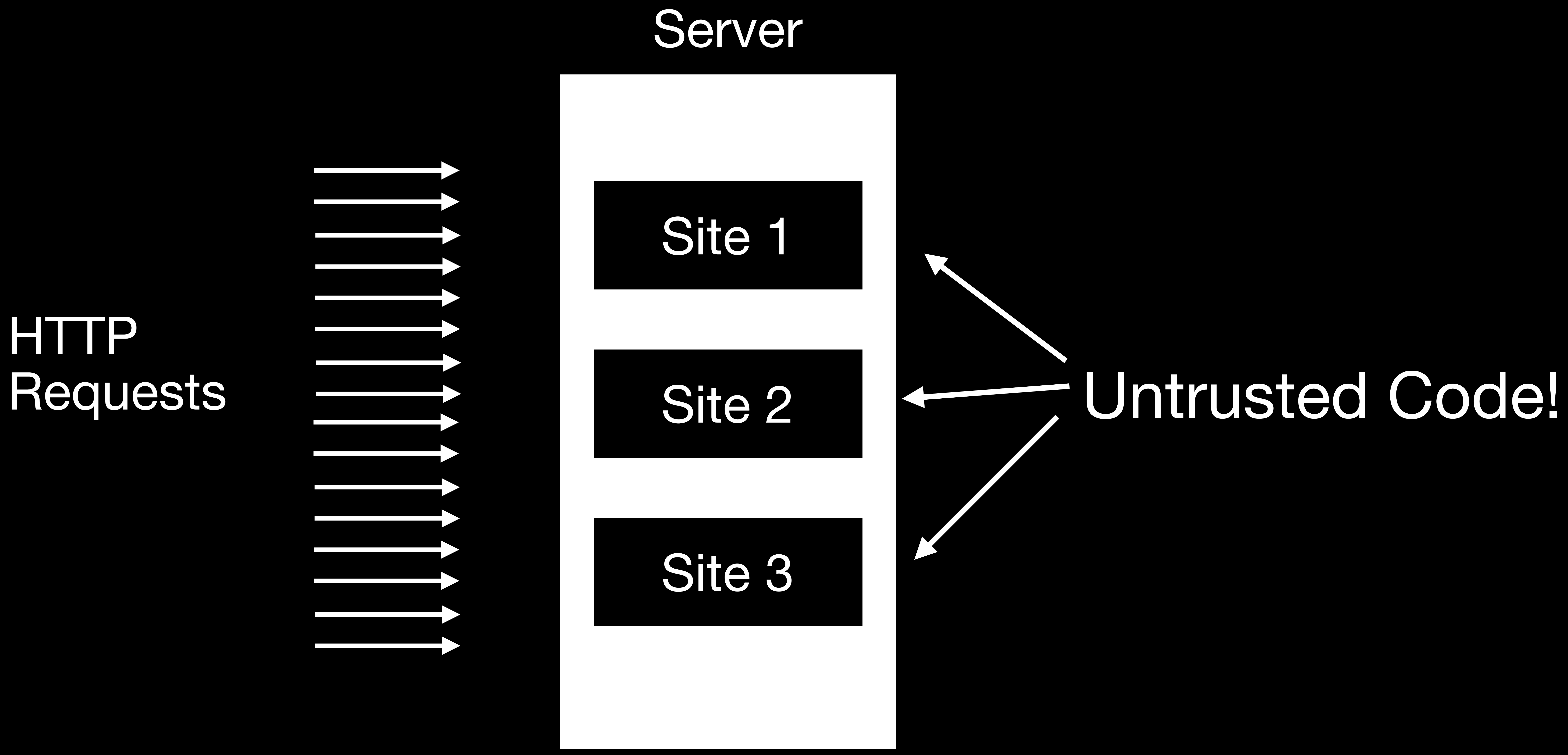
The Problem



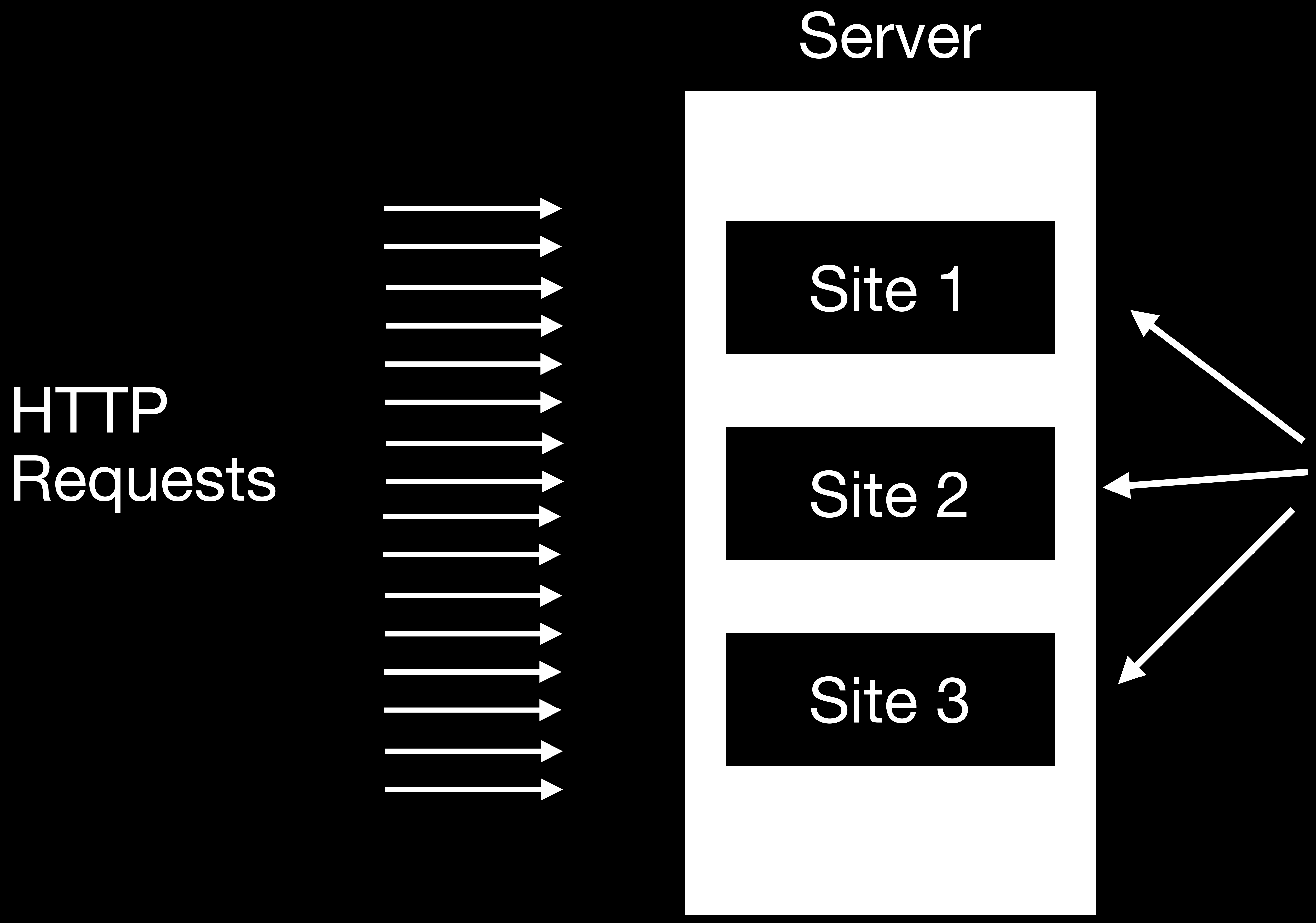
The Problem



The Problem



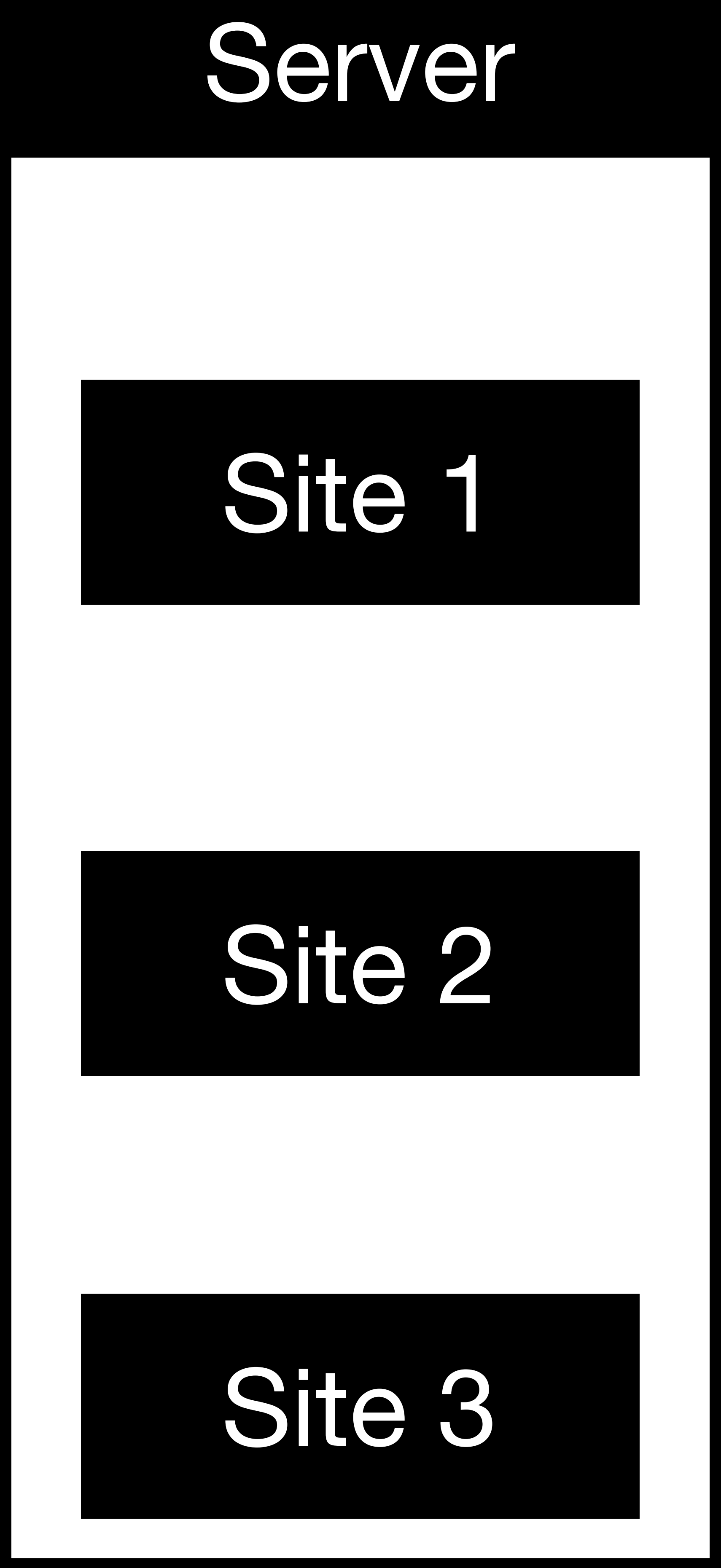
The Problem



Untrusted Code!

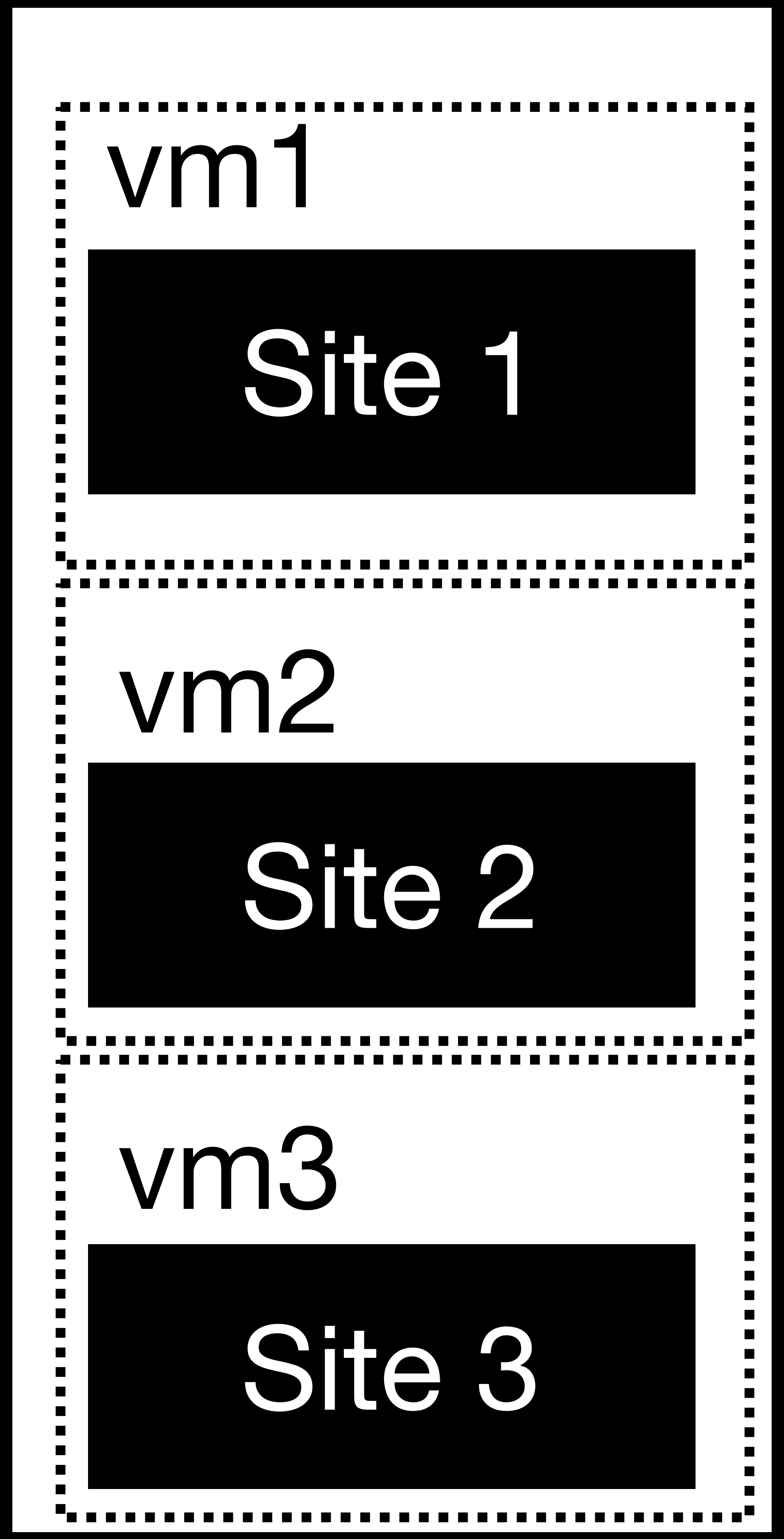
- Site 1 should not interfere with Site 2
- Site 1 should not interfere with host or infrastructure
- Request 1 on Site 1 should not interfere with request 2 on Site 1

The Problem



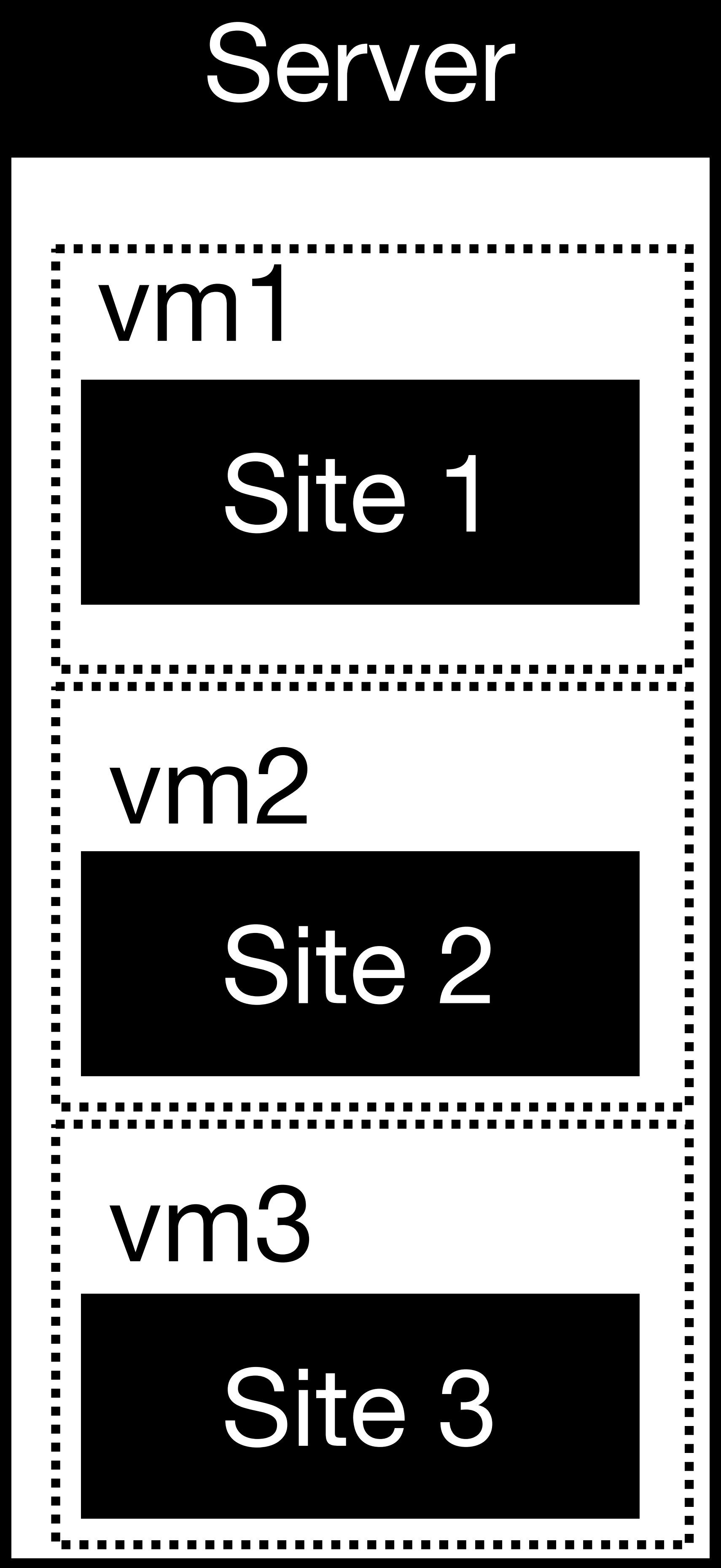
The Problem

Server



Virtual machines?

The Problem

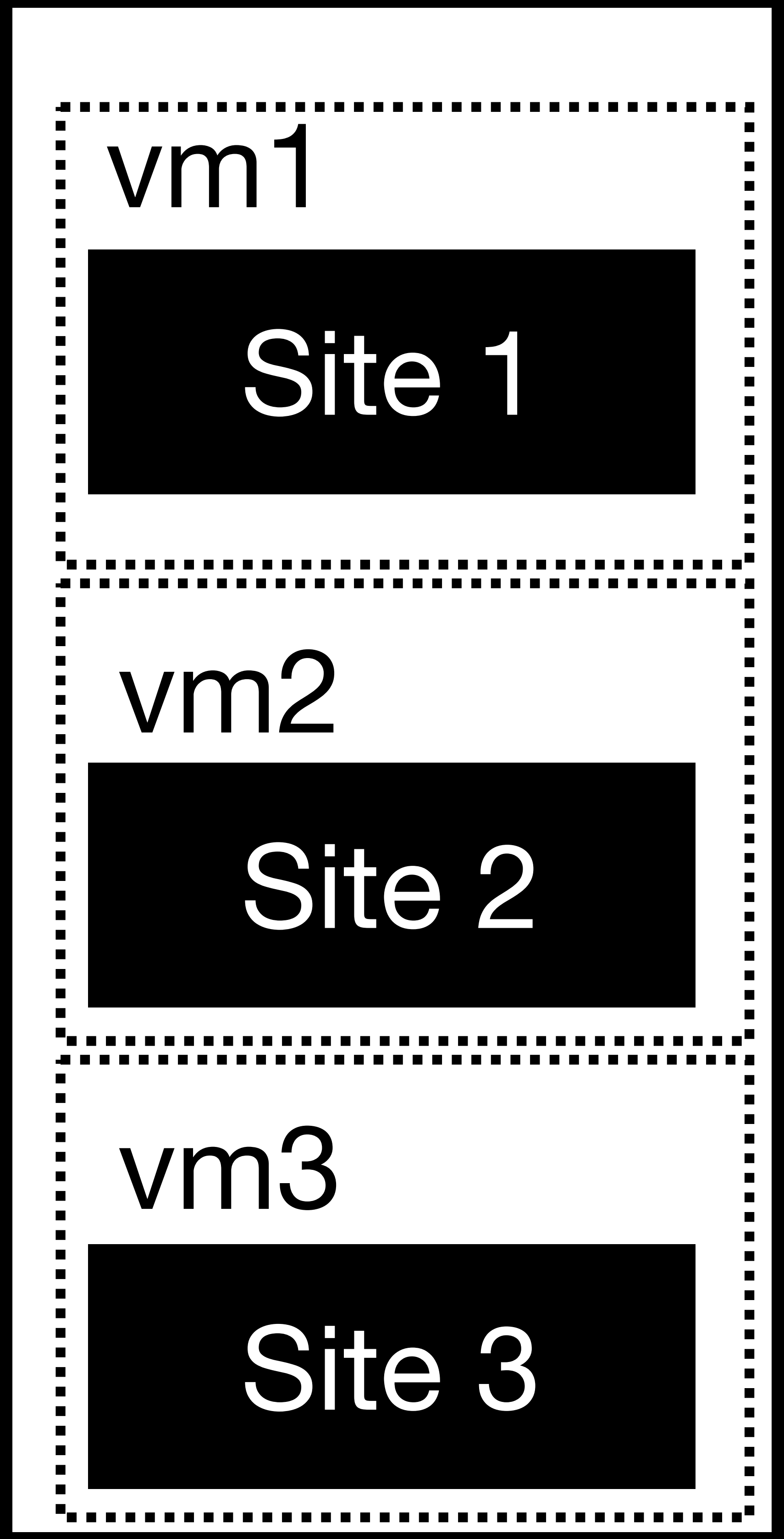


Virtual machines?

+ Extremely well-tested isolation boundary (trusted by cloud providers, …)
+ Conceptually simple: “single-tenant software stack” in each VM

The Problem

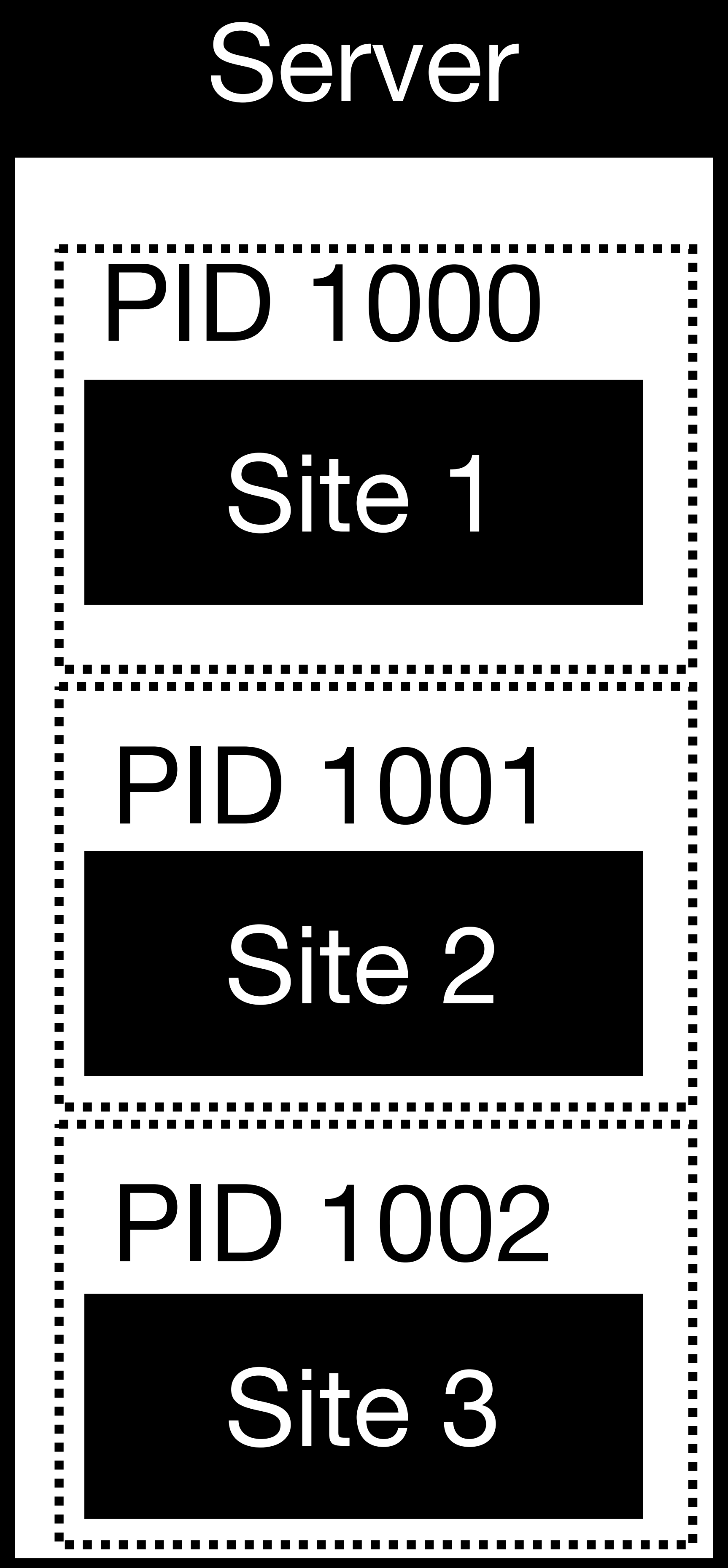
Server



Virtual machines?

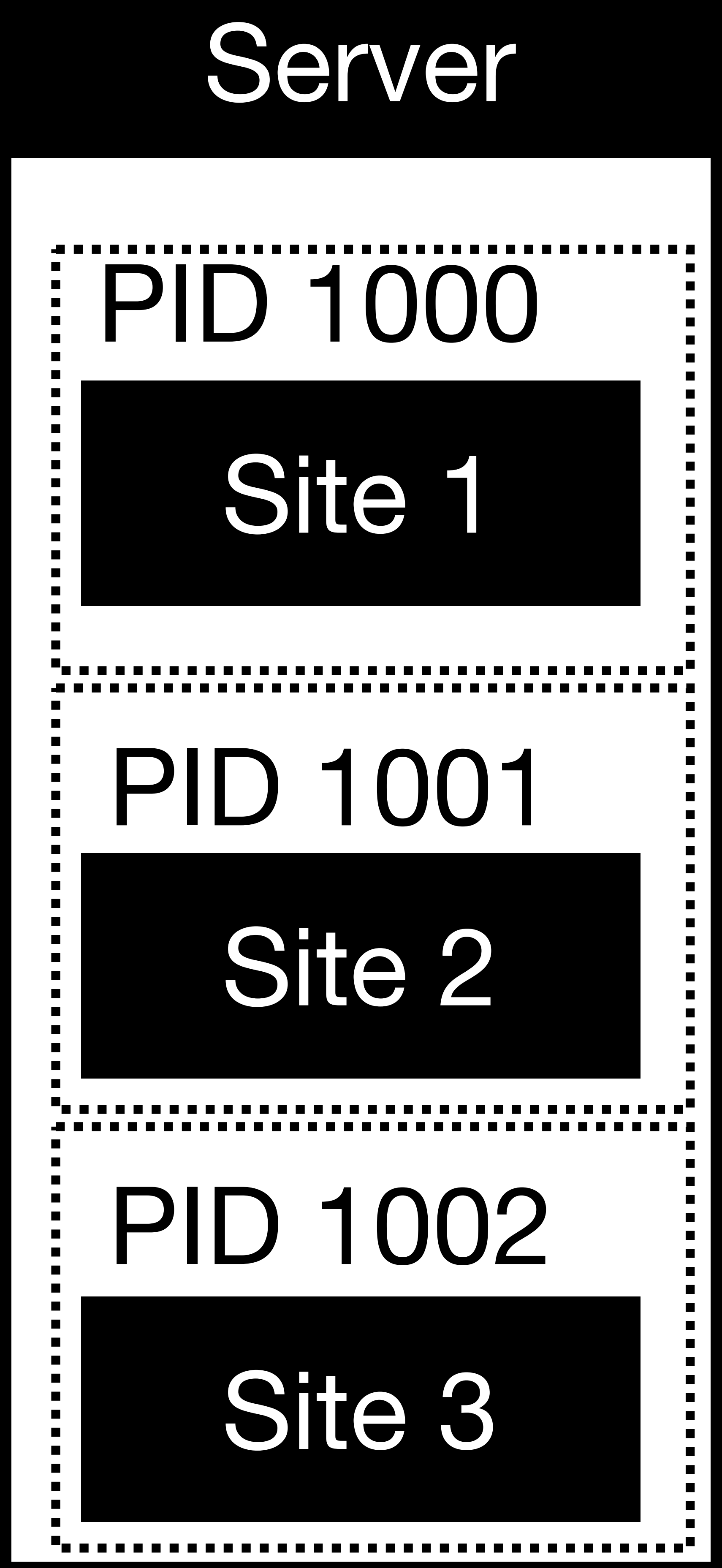
+ Extremely well-tested isolation boundary (trusted by cloud providers, …)
+ Conceptually simple: “single-tenant software stack” in each VM
- Horrible overhead: RAM + disk for full software stack + kernel in each VM!
- Fixed resource partitioning: cannot dynamically rebalance RAM if one site has spiky demand
- Doesn’t address “request isolation”

The Problem



Separate processes in containers?

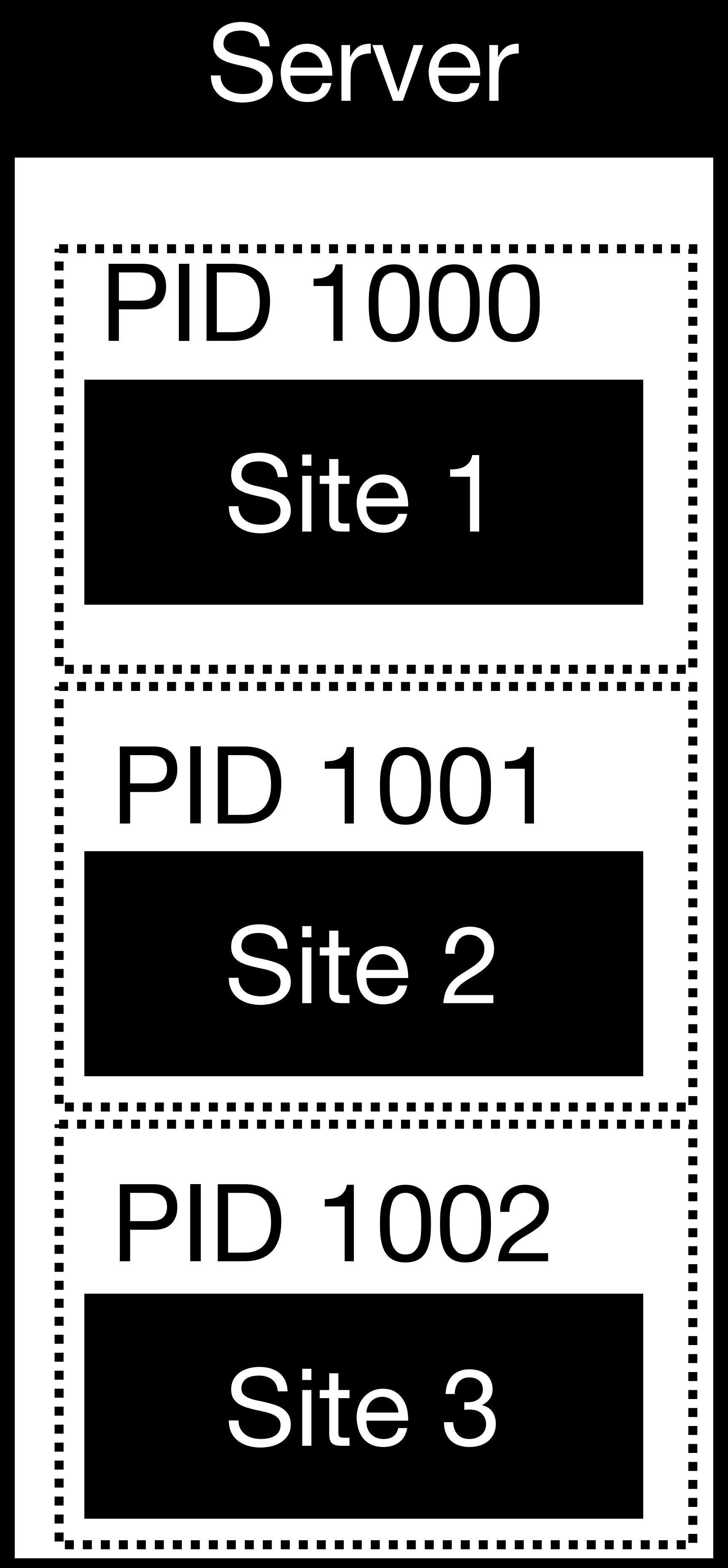
The Problem



Separate processes in containers?

+ Fairly well-tested isolation boundary (less than VMs, but emerging standard)
+ Software stack also looks similar to VM case: conceptually a “separate server” for every site

The Problem



Separate processes in containers?

+ Fairly well-tested isolation boundary (less than VMs, but emerging standard)
+ Software stack also looks similar to VM case: conceptually a “separate server” for every site
- Still too much overhead
- Processes must always be running for fast “cold start”
- Still no per-request isolation

The Problem

New process spawned for every request?

The Problem

New process spawned for every request?

+ This is a classic! Ask anyone from 90s webdev about “cgi-bin scripts” and Perl
+ Potentially good isolation/security, if properly sandboxed; good per-request isolation (fresh state for every request)

The Problem

New process spawned for every request?

+ This is a classic! Ask anyone from 90s webdev about "cgi-bin scripts" and Perl

+ Potentially good isolation/security, if properly sandboxed; good per-request isolation (fresh state for every request)

- Horrendous latency: OS process startup + binary load + script parse + connect to DB + parse configuration + initialize the universe + …

- Nonstarter for competitive modern web APIs

The Problem: Desired Properties

- We want good isolation for security:
  - Code for each site lives in some sort of sandbox with minimal attack surface
  - Code for each request starts fresh, with no “leftover state” that could leak private data from other user

The Problem: Desired Properties

- We want good isolation for security:
  - Code for each site lives in some sort of sandbox with minimal attack surface
  - Code for each request starts fresh, with no “leftover state” that could leak private data from other user
- We want extremely low latency:
  - Can’t afford to start a VM or a new OS-level process
  - Should strive to reuse high-cost setup (e.g., parsing the script)

The Problem: Desired Properties

- We want good isolation for security:
  - Code for each site lives in some sort of sandbox with minimal attack surface
  - Code for each request starts fresh, with no “leftover state” that could leak private data from other user
- We want extremely low latency:
  - Can’t afford to start a VM or a new OS-level process
  - Should strive to reuse high-cost setup (e.g., parsing the script)
- We want good *utilization* (pack many sites onto one server):
  - Can’t afford a few GB of RAM for a VM for every site
  - Probably can’t afford an OS process for every site

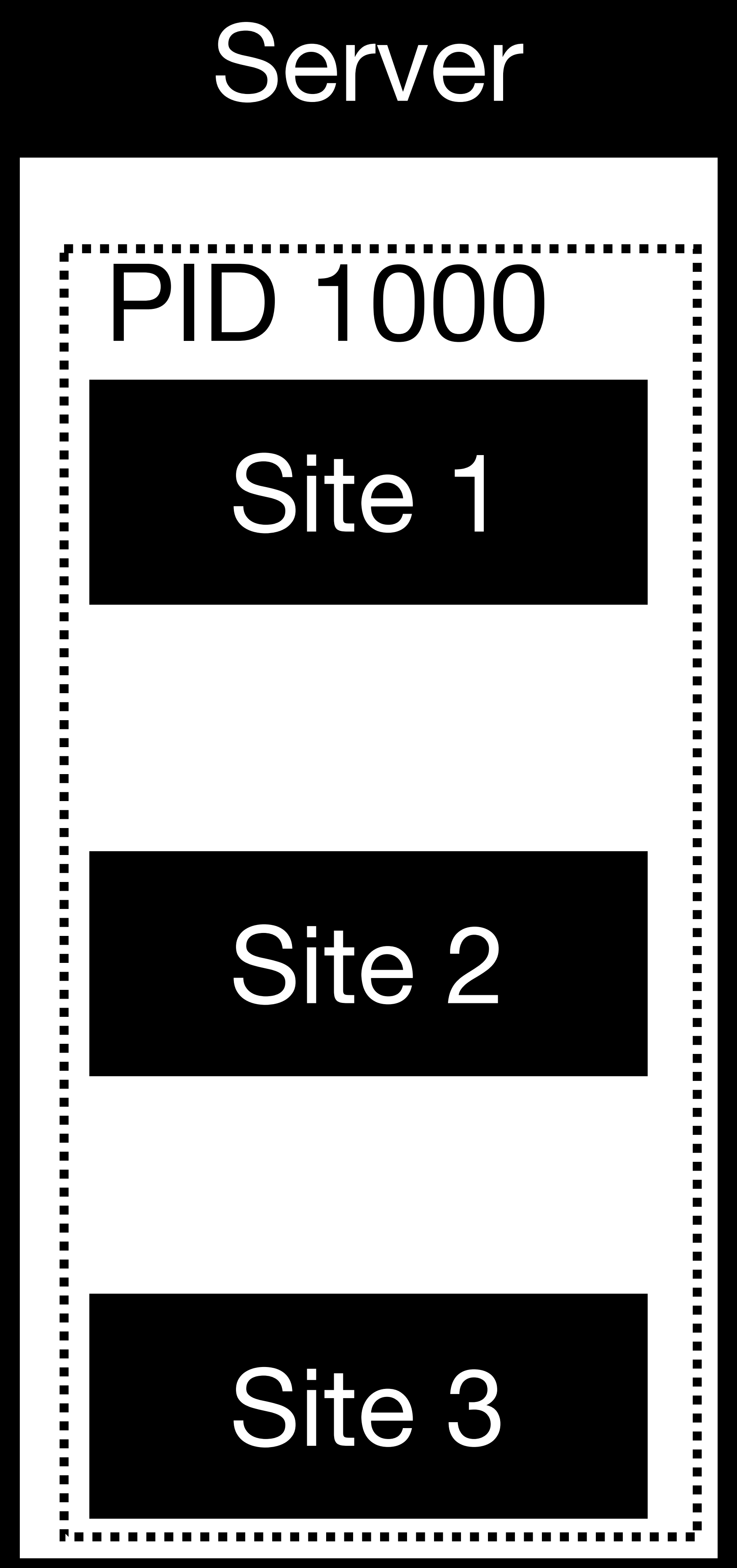
The Problem: Desired Properties

- We want good isolation for security:
  - Code for each site lives in some sort of sandbox with minimal attack surface
  - Code for each request starts fresh, with no “leftover state” that could leak private data from other user

(for now!)

- We want extremely low latency:
  - Can’t afford to start a VM or a new OS-level process
  - Should strive to reuse high-cost setup (e.g., parsing the script)
- We want good *utilization* (pack many sites onto one server):
  - Can’t afford a few GB of RAM for a VM for every site
  - Probably can’t afford an OS process for every site

The Problem: Desired Properties



Every site served from a single “global” process?

+ No startup latency: load code once, share setup + long-lived resources
  - Code is always present + a single function call away!

+ Conceptually, overhead of a site is just an object in some data structure

+ Conceptually, overhead of a request is just an object holding its state

+ Site software has a simple model: “just write a function”
  - This is *Function as a Service*

The Problem: Desired Properties

- We want good isolation for security:
  - Code for each site lives in some sort of sandbox with minimal attack surface
  - Code for each request starts fresh, with no “leftover state” that could leak private data from other user

The Problem: Desired Properties

- We want good isolation for security:
  - Code for each site lives in some sort of sandbox with minimal attack surface
  - Code for each request starts fresh, with no “leftover state” that could leak private data from other user

- Idea: what if there were a very simple “virtual CPU” to run the functions?
  - Give each function execution its own “memory” (array of a few KB or MB)
  - We could design it carefully to minimize attack surface —> good sandbox
  - Deterministic —> snapshots —> fast startup

The Problem: Desired Properties

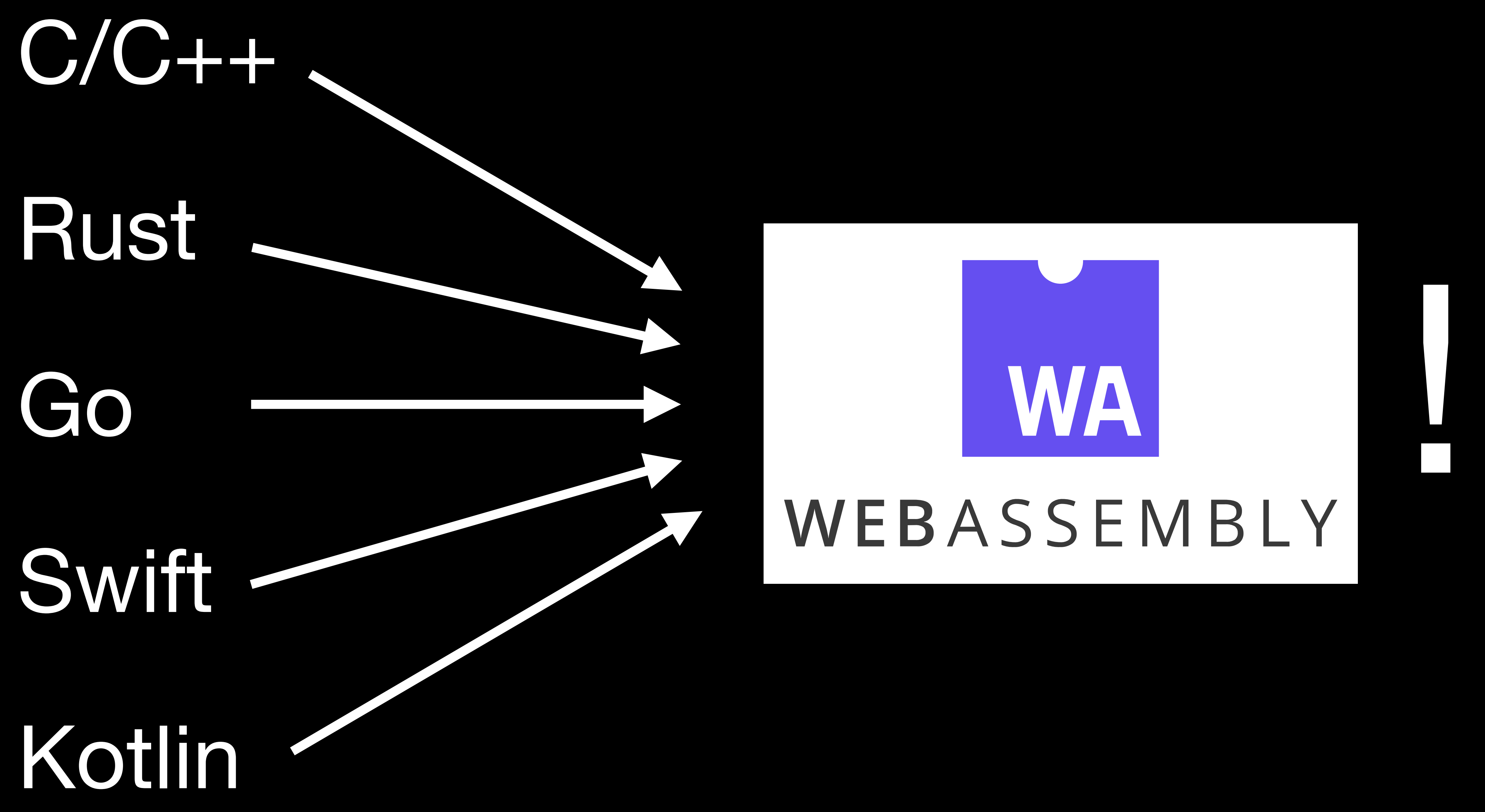
WA WEBASSEMBLY !

The Problem: Desired Properties

WEBASSEMBLY !

Portable bytecode with low-level (byte-addressable) memory and explicit hostcall imports (capability sandboxing)

The Problem: Desired Properties

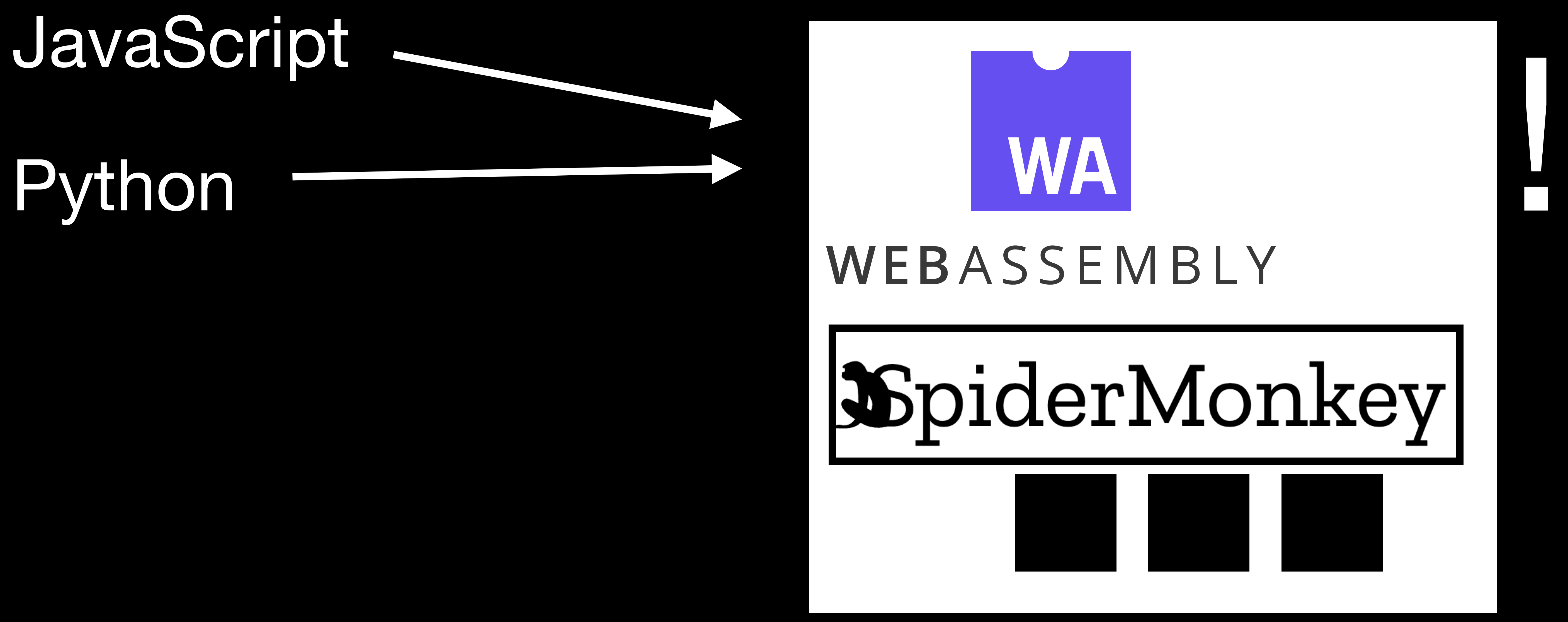


The Problem: Desired Properties

JavaScript → WebAssembly !

Python → WebAssembly !

The Problem: Desired Properties

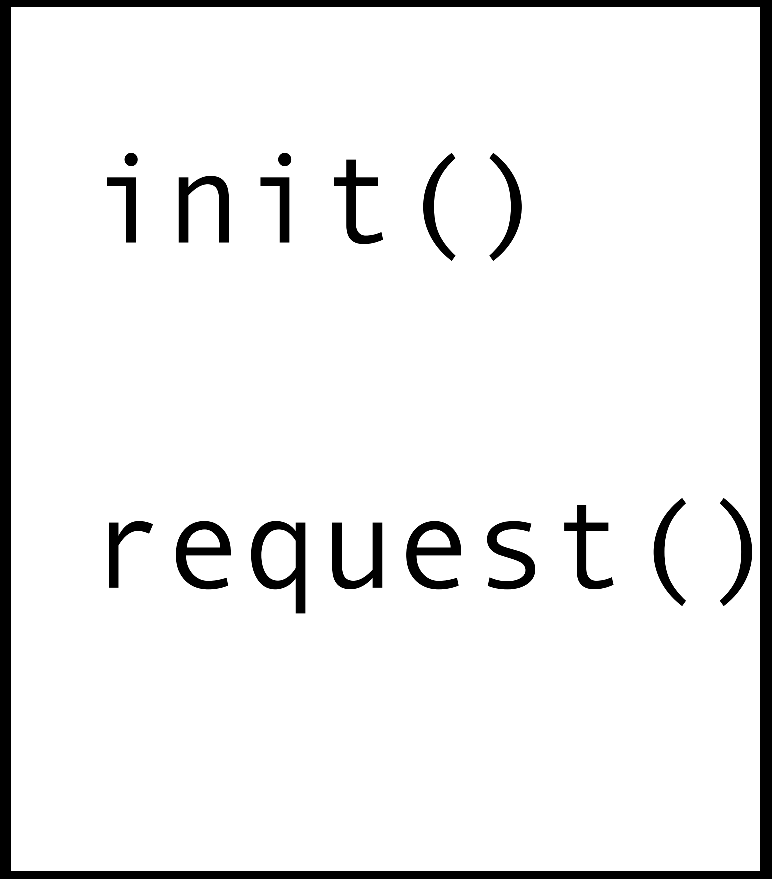


pre-load function bytecode into memory image

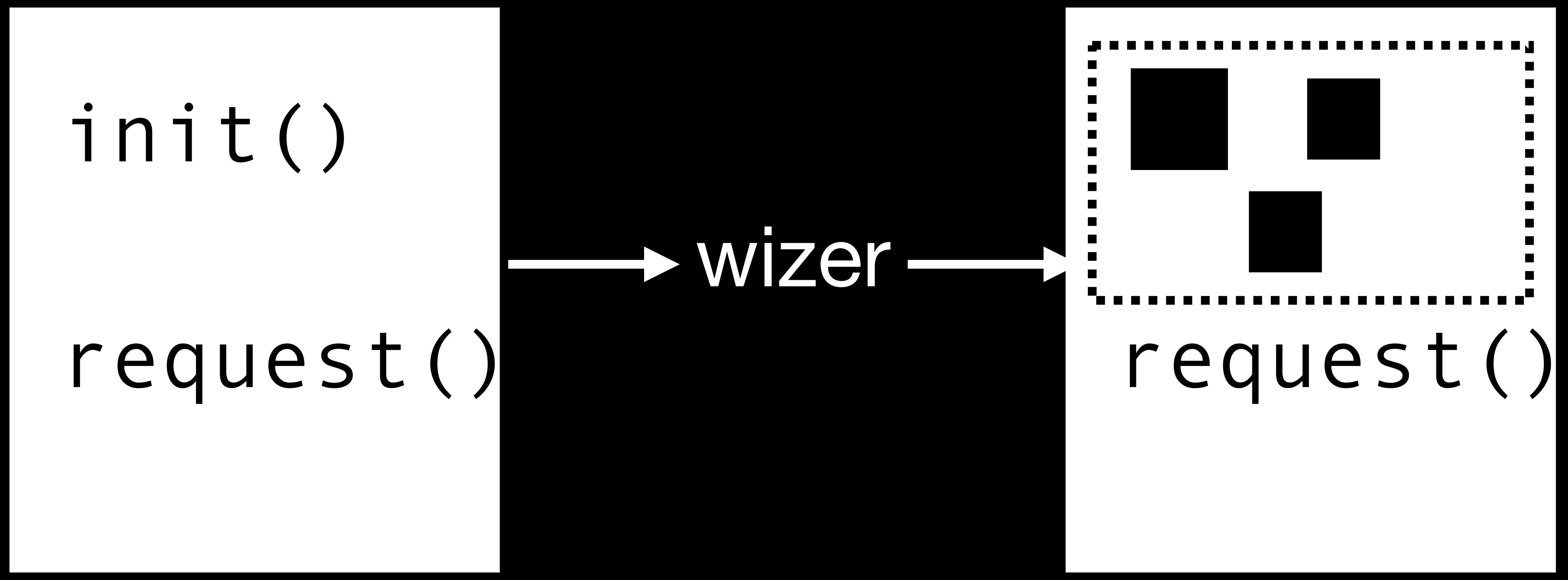
Wasm Snapshotting

```
init()
  main =
    parseScript()
request()
  execute(main)
```

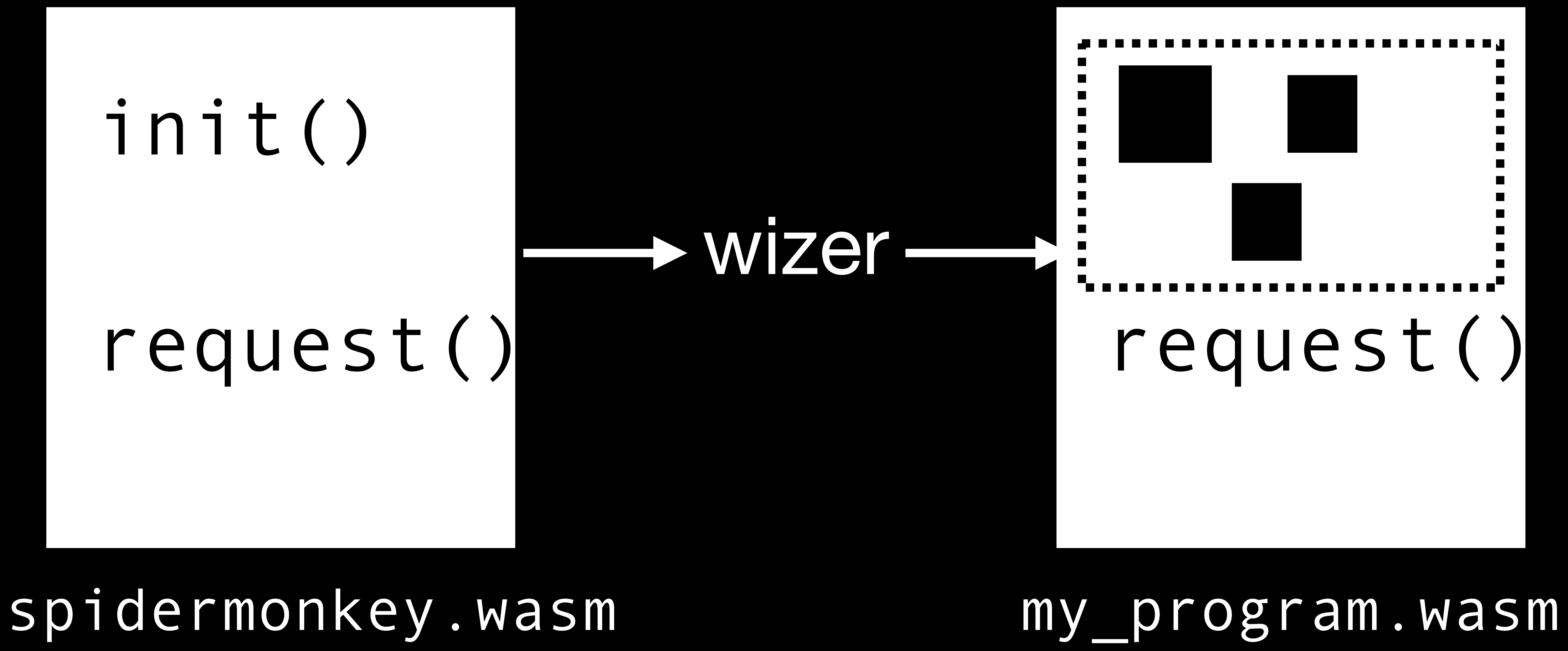
Wasm Snapshotting



Wasm Snapshotting



Wasm Snapshotting



Wasm-based Request Isolation

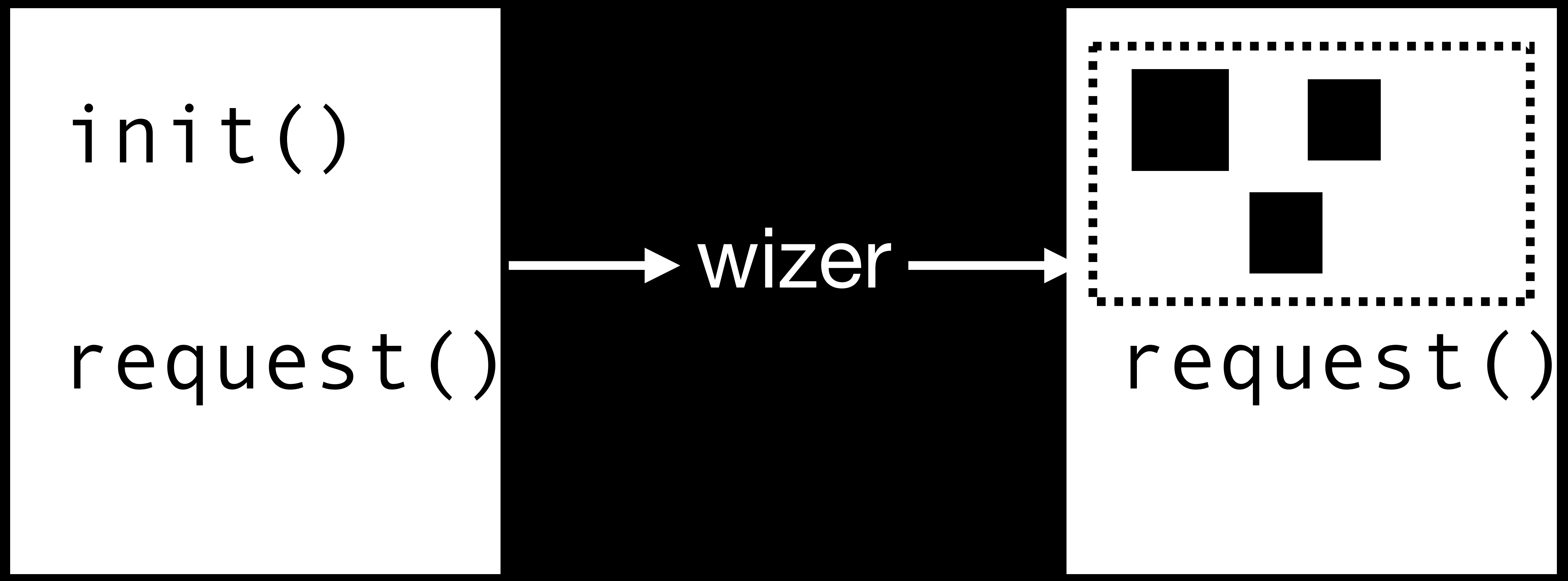
- Spawn a new Wasm *instance* with its own memory for every request

Wasm-based Request Isolation

- Spawn a new Wasm *instance* with its own memory for every request
- Virtual memory (copy-on-write) for 5-μs instantiation times

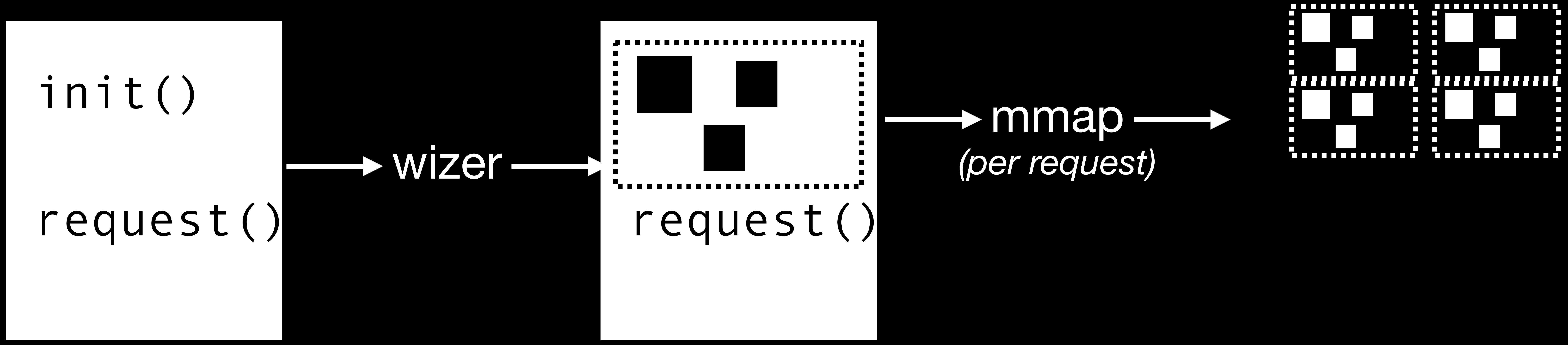
Wasm-based Request Isolation

- Spawn a new Wasm *instance* with its own memory for every request
- Virtual memory (copy-on-write) for 5-μs instantiation times



Wasm-based Request Isolation

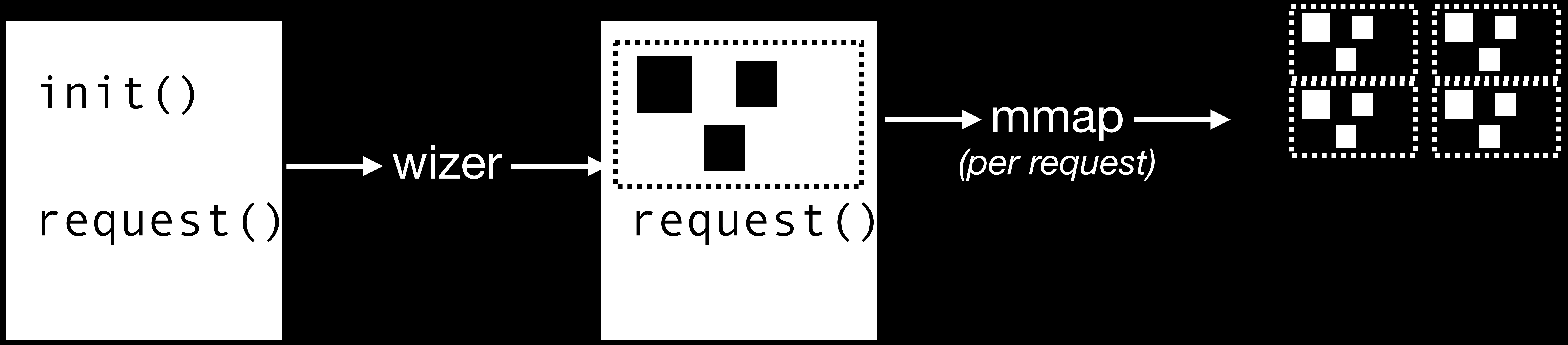
- Spawn a new Wasm *instance* with its own memory for every request
- Virtual memory (copy-on-write) for 5-μs instantiation times



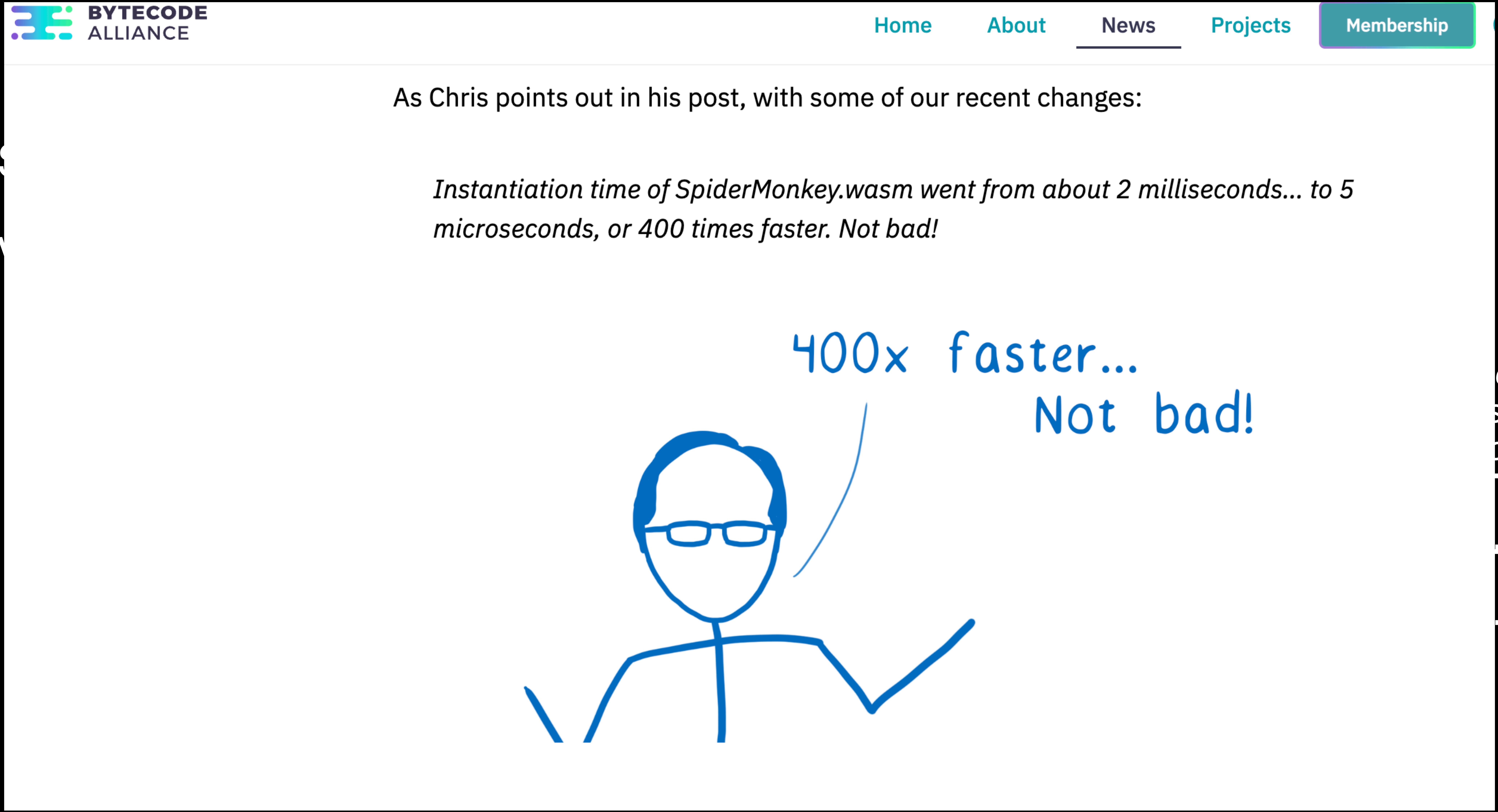
Wasm-based Request Isolation

- Spawn a new Wasm *instance* with its own memory for every request
- Virtual memory (copy-on-write) for 5-μs instantiation times

** actually madvise() if reusing a slot;*
avoid taking process address space lock;
lots of work on reducing IPIs/TLB shootdowns;
lazy init of VM structs too; happy to talk more



Wasm-based Request Isolation


BYTECODE ALLIANCE

Home | About | News | Projects | Membership

As Chris points out in his post, with some of our recent changes:

Instantiation time of SpiderMonkey.wasm went from about 2 milliseconds… to 5 microseconds, or 400 times faster. Not bad!

e lock;
shootdowns;
o talk more

Interpreter-inside-Wasm-with-Snapshotting

- Secure!
  - Wasm: Harvard architecture
    (separate code space, no codegen)

Interpreter-inside-Wasm-with-Snapshotting

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/unwind/…) and structured ctrl flow

Interpreter-inside-Wasm-with-Snapshotting

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/unwind/…) and structured ctrl flow
  - Instance-per-request: no possible state leakage between executions

Interpreter-inside-Wasm-with-Snapshotting

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/unwind/…) and structured ctrl flow
  - Instance-per-request: no possible state leakage between executions
- JIT engine’s favorite activities

Interpreter-inside-Wasm-with-Snapshotting

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/unwind/…) and structured ctrl flow
  - Instance-per-request: no possible state leakage between executions
- JIT engine’s favorite activities
  - generate code at runtime

Interpreter-inside-Wasm-with-Snapshotting

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/ unwind/…) and structured ctrl flow
  - Instance-per-request: no possible state leakage between executions
- JIT engine’s favorite activities
  - generate code at runtime
  - manage stack frames manually, implement O(1) unwind, on-stack replacement, multi entry + return points, jumps between IC stubs

Interpreter-inside-Wasm-with-Snapshotting

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/unwind/…) and structured ctrl flow
  - Instance-per-request: no possible state leakage between executions

- JIT engine's favorite activities
  - generate code at runtime
  - manage stack frames manually, implement O(1) unwind, on-stack replacement, multi entry + return points, jumps between IC stubs
  - Warm up and specialize code over time (many executions/requests)

Aside: “JIT Hooks for Wasm”?

- Straightforward adaptation:
  - Add a Wasm hostcall (or core feature) to add a new function at runtime (accept only Wasm bytecode — preserve the sandboxing still)
  - Work around other incongruities in machine model: O(n) unwind, tail calls for IC stubs, just don’t do OSR, …

Aside: “JIT Hooks for Wasm”?

- Straightforward adaptation:
  - Add a Wasm hostcall (or core feature) to add a new function at runtime (accept only Wasm bytecode — preserve the sandboxing still)
  - Work around other incongruities in machine model: O(n) unwind, tail calls for IC stubs, just don’t do OSR, …

WASM I/O 2024 · 14 -15 MAR · BARCELONA

RUNNING JS VIA WASM FASTER WITH JIT

DMITRY BEZHETSKOV - IGALIA

Aside: “JIT Hooks for Wasm”?

RUNNING JS VIA WASM FASTER WITH JIT
DMITRY BEZHETSKOV - IGALIA

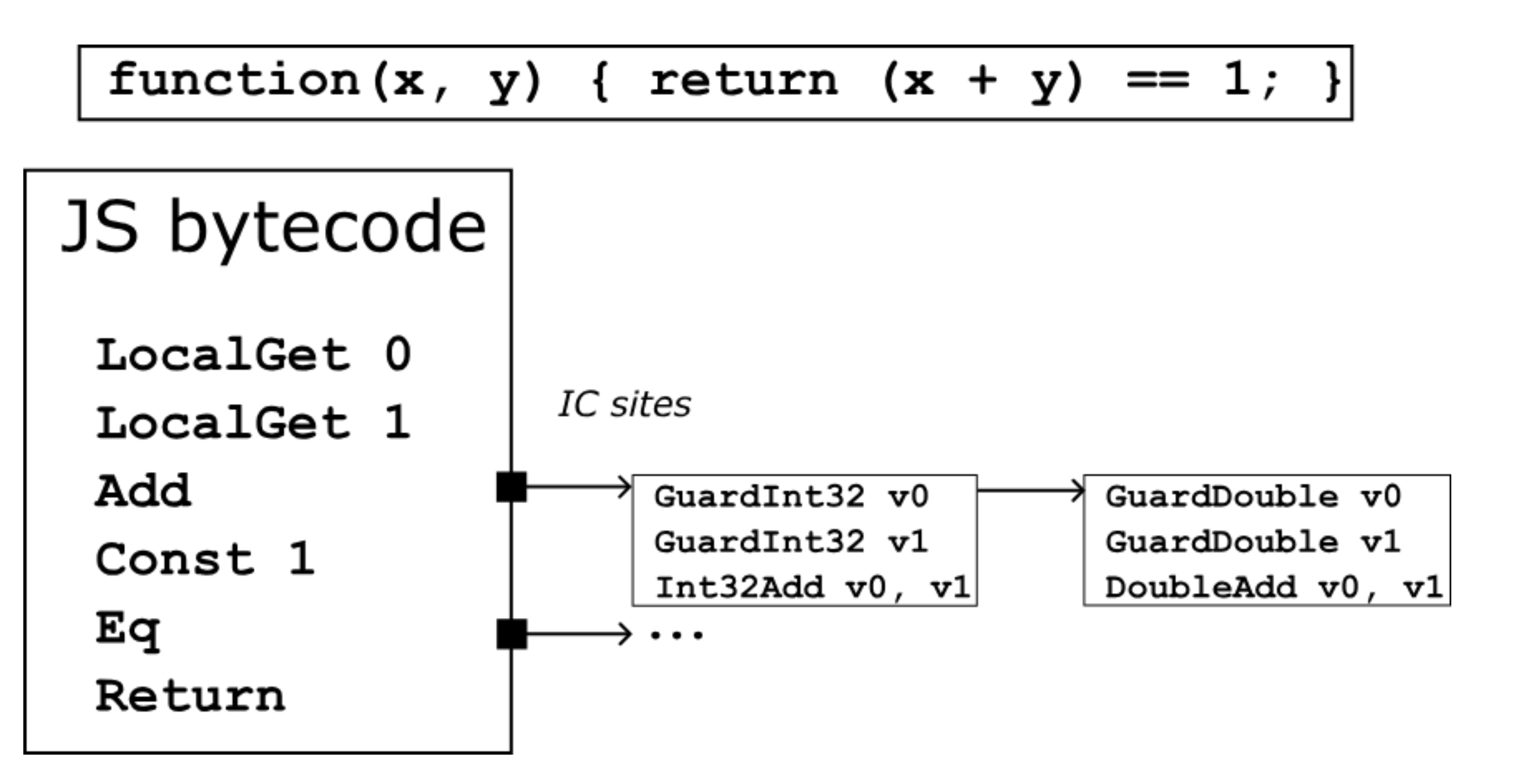
- Straightforward adaptation:
  - Add a Wasm hostcall (or core feature) to add a new function at runtime (accept only Wasm bytecode — preserve the sandboxing still)
  - Work around other incongruities in machine model: O(n) unwind, tail calls for IC stubs, just don’t do OSR, …
- Really impressive results: 2x-11x (similar to native ISA baseline-compiler)

Aside: "JIT Hooks for Wasm"?

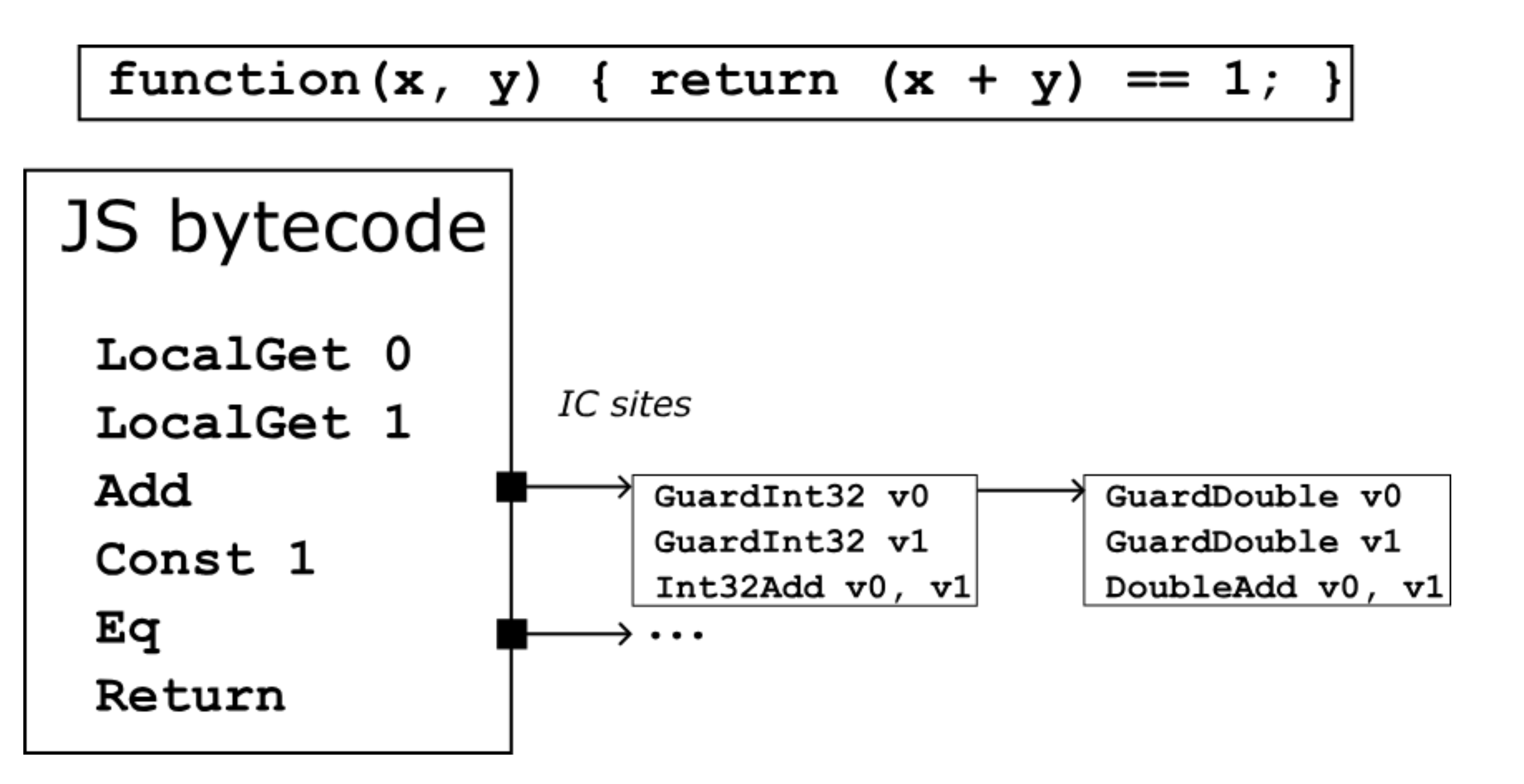
RUNNING JS VIA WASM FASTER WITH JIT
DMITRY BEZHETSKOV - IGALIA

- Straightforward adaptation:
  - Add a Wasm hostcall (or core feature) to add a new function at runtime (accept only Wasm bytecode — preserve the sandboxing still)
  - Work around other incongruities in machine model: O(n) unwind, tail calls for IC stubs, just don't do OSR, …
- Really impressive results: 2x-11x (similar to native ISA baseline-compiler)
- Downside: requires *test data* and *profiling run* (nonstandard user experience) or compiler-in-the-loop and saving state across requests (JIT data structures)

Fast Dynamic Languages: ICs for Late Binding



Fast Dynamic Languages: ICs for Late Binding



Key idea: *late binding* for execution semantics (dynamic types) becomes *late binding* in compilation strategy (indirect call via IC head)

CacheIR: Systematic Fast-Paths

- SpiderMonkey has a *straight-line IR* with specific "guard" (predicate) and "action" opcodes

```
GuardInt32 v0
GuardInt32 v1
Int32Add v0, v1
```

- Engine is well-populated with *many* fast paths developed over the years
    - Property accesses, including JS oddities (chain of prototype-chain guards)
    - Special cases for calls to well-known functions (String.length(), etc)
    - Hundreds of opcodes, ~hundreds-thousands of IC cases
    - Let's reuse this if we can!

See also: de Mooij et al. *CacheIR: The Benefits of Structured Representation for Inline Caches.* SPLASH 2023.

Compilation Levels

Level	Data Required	JS opcode dispatch	ICs	Optimization Scope	CacheIR dispatch	Codegen at Runtime?
Generic Interpreter	JS bytecode	interpreter	none	none	—	no
Baseline Interpreter	JS bytecode + IC stub cases	interpreter	dynamic dispatch	within one op (via IC)	compiled	yes
Baseline Compiler	JS bytecode + IC stub cases	compiled	dynamic dispatch	within one op (via IC)	compiled	yes
Optimizing Compiler	JS bytecode + warmed-up ICs	compiled	inlined	entire function	compiled	yes

Step 1: Portable Baseline Interpreter (PBL)

- New interpreter tier in SpiderMonkey: no codegen, but run ICs via *interpreter*

Step 1: Portable Baseline Interpreter (PBL)

- New interpreter tier in SpiderMonkey: no codegen, but run ICs via *interpreter*

Level	Data Required	JS opcode dispatch	ICs	Optimization Scope	CacheIR dispatch	Codegen at Runtime?
Generic Interpreter	JS bytecode	interpreter	none	none	—	no
Baseline Interpreter	JS bytecode + IC stub cases	interpreter	dynamic dispatch	within one op (via IC)	compiled	yes

Step 1: Portable Baseline Interpreter (PBL)

- New interpreter tier in SpiderMonkey: no codegen, but run ICs via *interpreter*

Level	Data Required	JS opcode dispatch	ICs	Optimization Scope	CacheIR dispatch	Codegen at Runtime?
Generic Interpreter	JS bytecode	interpreter	none	none	—	no
Portable Baseline Interpreter	JS bytecode	interpreter	dynamic dispatch	none	interpreter	no
Baseline Interpreter	JS bytecode + IC stub cases	interpreter	dynamic dispatch	within one op (via IC)	compiled	yes

Step 1: Portable Baseline Interpreter (PBL)

- New interpreter tier in SpiderMonkey: no codegen, but run ICs via *interpreter*
  - Key insight: this shifts the tradeoff; not all ICs will be profitable anymore —> "hybrid ICs": (optionally) use ICs only for property accesses, calls
  - This allows faster execution even for "code we have never met before" (`eval()` in production…)
- Implemented and upstreamed; used in production; 33% geomean speedup

Step 1: Portable Baseline Interpreter (PBL)

```
Bench               Base     PBL
----
Richards             164     280 (1.71x)
DeltaBlue            167     321 (1.92x)
Crypto               453     566 (1.25x)
RayTrace             498     786 (1.58x)
EarleyBoyer          712    1070 (1.50x)
RegExp               273     337 (1.23x)
Splay               1293    2147 (1.66x)
NavierStokes         684     763 (1.32x)
PdfJS               2220    2512 (1.31x)
Mandreel             189     233 (1.23x)
Gameboy             1479    1774 (1.20x)
CodeLoad           19765   18994 (0.96x)
Box2D                943    1328 (1.41x)
----
Overall              848    1127 (1.33x)
```

Step 1: Portable Baseline Interpreter (PBL)

- How does this help us *compile* JavaScript?!

JS Compilation

Level	Data Required	JS opcode dispatch	ICs	Optimization Scope	CacheIR dispatch	Codegen at Runtime?
Generic Interpreter	JS bytecode	interpreter	none	none	—	no
Portable Baseline Interpreter	JS bytecode	interpreter	dynamic dispatch	none	interpreter	no
Baseline Interpreter	JS bytecode + IC stub cases	interpreter	dynamic dispatch	within one op (via IC)	compiled	yes
Baseline Compiler	JS bytecode + IC stub cases	compiled	dynamic dispatch	within one op (via IC)	compiled	yes
Optimizing Compiler	JS bytecode + warmed-up ICs	compiled	inlined	entire function	compiled	yes

JS Compilation

Level	Data Required	JS opcode dispatch	ICs	Optimization Scope	CacheIR dispatch	Codegen at Runtime?
Generic Interpreter	JS bytecode	interpreter	none	none	—	no
Portable Baseline Interpreter	JS bytecode	interpreter	dynamic dispatch	none	interpreter	no
Baseline Interpreter	JS bytecode + IC stub cases	interpreter	dynamic dispatch	within one op (via IC)	compiled	yes
Baseline Compiler	JS bytecode + IC stub cases	compiled	dynamic dispatch	within one op (via IC)	compiled	yes
Optimizing Compiler	JS bytecode + warmed-up ICs	compiled	inlined	entire function	compiled	yes

JS Compilation

Level	Data Required	JS opcode dispatch	ICs	Optimization Scope	CacheIR dispatch	Codegen at Runtime?
Generic Interpreter	JS bytecode	interpreter	none	none	—	no
Portable Baseline Interpreter	JS bytecode	interpreter	dynamic dispatch	none	interpreter	no
Baseline Interpreter	JS bytecode + IC stub cases	interpreter	dynamic dispatch	within one op (via IC)	compiled	yes
Baseline Compiler	JS bytecode + IC stub cases	compiled	dynamic dispatch	within one op (via IC)	compiled	yes
Optimizing Compiler	JS bytecode + warmed-up ICs	compiled	inlined	entire function	compiled	yes

Compilation Phasing (“can we AOT?”)

Level	Data Required	JS opcode dispatch	ICs	Optimization Scope	CacheIR dispatch	Codegen at Runtime?
Generic Interpreter	JS bytecode	interpreter	none	none	—	no
Portable Baseline Interpreter	JS bytecode	interpreter	dynamic dispatch	none	interpreter	no
Baseline Interpreter	JS bytecode + IC stub cases	interpreter	dynamic dispatch	within one op (via IC)	compiled	yes
Baseline Compiler	JS bytecode + IC stub cases	compiled	dynamic dispatch	within one op (via IC)	compiled	yes
Optimizing Compiler	JS bytecode + warmed-up ICs	compiled	inlined	entire function	compiled	yes

AOT: ✅

Compilation Phasing (“can we AOT?”)

Level	Data Required	JS opcode dispatch	ICs	Optimization Scope	CacheIR dispatch	Codegen at Runtime?
Generic Interpreter	JS bytecode	interpreter	none	none	—	no
Portable Baseline Interpreter	[illegible]	[illegible]	dynamic [illegible]	[illegible]	[illegible]	no
Baseline Interpreter	JS bytecode + IC stub cases	interpreter	dynamic dispatch	within one op (via IC)	compiled	yes
Baseline Compiler	JS bytecode + IC stub cases	compiled	dynamic dispatch	within one op (via IC)	[illegible]	yes
Optimizing Compiler	JS bytecode + warmed-up ICs	[illegible]	inlined	entire function	compiled	yes

Key insight: collect a corpus of “common ICs” (once)

AOT: ✅

AOT: ✅

Compilation Phasing ("can we AOT?")

Level	Data Required	JS opcode dispatch	ICs	Optimization Scope	CacheIR dispatch	Codegen at Runtime?
Generic Interpreter	JS bytecode	interpreter	none	none	—	no
Portable Baseline Interpreter	[illegible]	[illegible]	dynamic [illegible]	[illegible]	[illegible]	no
Baseline Interpreter	[illegible] IC stub cases	[illegible]	[illegible] dispatch	[illegible] (via IC)	[illegible]	yes
Baseline Compiler	JS bytecode + IC stub cases	compiled	dynamic dispatch	within one op (via IC)	[illegible]	yes
Optimizing Compiler	JS bytecode + warmed-up ICs	[illegible]	inlined	entire function	compiled	yes

Key insight: collect a corpus of "common ICs" (once) —> pushes PGO to engine developer, not engine user

AOT: ✅

AOT: ✅

Compiler Backend

- Great! Let’s write a compiler backend!

Compiler Backend

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/unwind/…) and structured ctrl flow
  - Instance-per-request: no possible state leakage between executions

- JIT engine's favorite activities
  - generate code at runtime
  - manage stack frames manually, implement O(1) unwind, on-stack replacement, multi entry + return points, jumps between IC stubs
  - Warm up and specialize code over time (many executions/requests)

Compiler Backend

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/unwind/…) and structured ctrl flow
  - Instance-per-request: no possible state leakage between executions

- JIT engine's favorite activities
  - ~~generate code at runtime~~
  - manage stack frames manually, implement O(1) unwind, on-stack replacement, multi entry + return points, jumps between IC stubs
  - ~~Warm up and specialize code over time (many executions/requests)~~

Compiler Backend

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/unwind/…) and structured ctrl flow
  - Instance-per-request: no possible state leakage between executions

- JIT engine's favorite activities
  - ~~generate code at runtime~~
  - manage stack frames manually, implement O(1) unwind, on-stack replacement, multi entry + return points, jumps between IC stubs
  - ~~Warm up and specialize code over time (many executions/requests)~~

Wasm is a *weird* architecture —> maintenance burden concerns

Compiler Backend

- Secure!
  - Wasm: Harvard architecture (separate code space, no codegen)
  - Wasm: first-class call stack (stack frames are VM-managed, no jumps/unwind/…) and structured ctrl flow
  - Instance-per-request: no possible state leakage between executions

- JIT engine's favorite activities
  - ~~generate code at runtime~~
  - manage stack frames manually, implement O(1) unwind, on-stack replacement, multi entry + return points, jumps between IC stubs
  - ~~Warm up and specialize code over time (many executions/requests)~~

Wasm is a *weird* architecture —> maintenance burden concerns
… also, we already have an interpreter (PBL) with exactly the logic we want

Compiler Backend?

```
switch(*pc++) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    break;
  case RET:
    return pop();
}
```

→

func:

ADD
RET

Compiler Backend?

```
switch(*pc++) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    break;
  case RET:
    return pop();
}
```

→

func:

ADD
RET

```
func() {
  auto a = pop();
  auto b = pop();
  push(a + b);
  return pop();
}
```

Compiler Backend?

```
switch(*pc++) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    break;
  case RET:
    return pop();
}
```

→

func:

ADD
RET

```
func() {
  auto a = pop();
  auto b = pop();
  push(a + b);
  return pop();
}
```

Key insight: Wasm is a small, introspectable, well-behaved IR;
partial evaluation should be tractable (moreso than on native code)

Futamura Projections

- Given Program(Input) —> Output:
  - Split Input into static and dynamic parts: Program(Static, Dynamic) = Output
  - Curry Program with Static: PEval(Program, Static) -> Program*
  - Then Program*(Dynamic) = Program(Static, Dynamic)

Futamura Projections

- Given Program(Input) —> Output:
  - Split Input into static and dynamic parts: Program(Static, Dynamic) = Output
  - Curry Program with Static: PEval(Program, Static) -> Program*
  - Then Program*(Dynamic) = Program(Static, Dynamic)
- Interesting cases:
  - First Futamura Projection: PEval(Interp, ProgramText) -> CompiledProgram

Futamura Projections

- Given Program(Input) —> Output:
  - Split Input into static and dynamic parts: Program(Static, Dynamic) = Output
  - Curry Program with Static: PEval(Program, Static) -> Program*
  - Then Program*(Dynamic) = Program(Static, Dynamic)
- Interesting cases:
  - First Futamura Projection: PEval(Interp, ProgramText) -> CompiledProgram
  - Second Futamura Projection: PEval(PEval, Interp) -> Compiler

Futamura Projections

- Given Program(Input) —> Output:
  - Split Input into static and dynamic parts: Program(Static, Dynamic) = Output
  - Curry Program with Static: PEval(Program, Static) -> Program*
  - Then Program*(Dynamic) = Program(Static, Dynamic)
- Interesting cases:
  - First Futamura Projection: PEval(Interp, ProgramText) -> CompiledProgram
  - Second Futamura Projection: PEval(PEval, Interp) -> Compiler
  - Third Futamura Projection: PEval(PEval, PEval) -> InterpreterToCompilerCompiler

Futamura Projections

- Given Program(Input) —> Output:
  - Split Input into static and dynamic parts: Program(Static, Dynamic) = Output
  - Curry Program with Static: PEval(Program, Static) -> Program*
  - Then Program*(Dynamic) = Program(Static, Dynamic)
- Interesting cases:
  - First Futamura Projection: PEval(Interp, ProgramText) -> CompiledProgram
  - Second Futamura Projection: PEval(PEval, Interp) -> Compiler
  - Third Futamura Projection: PEval(PEval, PEval) -> InterpreterToCompilerCompiler

weval: Partial Evaluation of Wasm

- Key idea: produce *specializations* of functions in a Wasm module with respect to some constant inputs (namely, interpreted bytecode)

weval: Partial Evaluation of Wasm

- Key idea: produce *specializations* of functions in a Wasm module with respect to some constant inputs (namely, interpreted bytecode)
    - Very very very important guiding principle: *no magic*, only semantics-preserving transforms; specialized function behaves identically to original

weval: Partial Evaluation of Wasm

- Key idea: produce *specializations* of functions in a Wasm module with respect to some constant inputs (namely, interpreted bytecode)
  - Very very very important guiding principle: *no magic*, only semantics-preserving transforms; specialized function behaves identically to original
  - Gives us a compiler “for free” once we have an interpreter

weval: Partial Evaluation of Wasm

- Key idea: produce *specializations* of functions in a Wasm module with respect to some constant inputs (namely, interpreted bytecode)
  - Very very very important guiding principle: *no magic*, only semantics-preserving transforms; specialized function behaves identically to original
  - Gives us a compiler “for free” once we have an interpreter
- Related work: GraalVM for JVM (TruffleRuby, …)
  - Main distinction in abstraction level: AST interpreter using Graal classes vs. general pre-existing interpreter in Wasm

Specialization Intrinsics

```
void call_function(function* f, int arg1, int arg2) {
  interp(f->bytecode, arg1, arg2);
}
```

```
void prepare_function(function* f) {
  weval::weval(
          &f->funcptr, &interp,
          ConstantMemory(f->bytecode),
          Runtime<int>(), Runtime<int>());
}
```

```
void call_function(function* f, …) {
  if (f->funcptr) f->funcptr(…);
  else interp(f->bytecode, …);
}
```

Specialization Intrinsics

```
void call_function(function* f, int arg1, int arg2) {
  interp(f->bytecode, arg1, arg2);
}
```

1. *asynchronous* request (“fill in this function pointer later”); integrates with wizening
2. specialized function behaves exactly the same as original interp()
3. for each argument, we provide constant value or “runtime”

Specialization Intrinsics

```
void interp(bytecode* pc) {

  switch (*pc++) {
    case OP1:
      …

      break;
    case OP2:
      …

      break;
  }
}
```

Specialization Intrinsics

```
void interp(bytecode* pc) {

  switch (*pc++) {
    case OP1:
      …

      break;
    case OP2:
      …

      break;
  }
}
```

```
void interp(bytecode* pc) {
  weval::push_context(pc);
  switch (*pc++) {
    case OP1:
      …
      weval::update_context(pc);
      break;
    case OP2:
      …
      weval::update_context(pc);
      break;
  }
}
```

Specialization Intrinsics

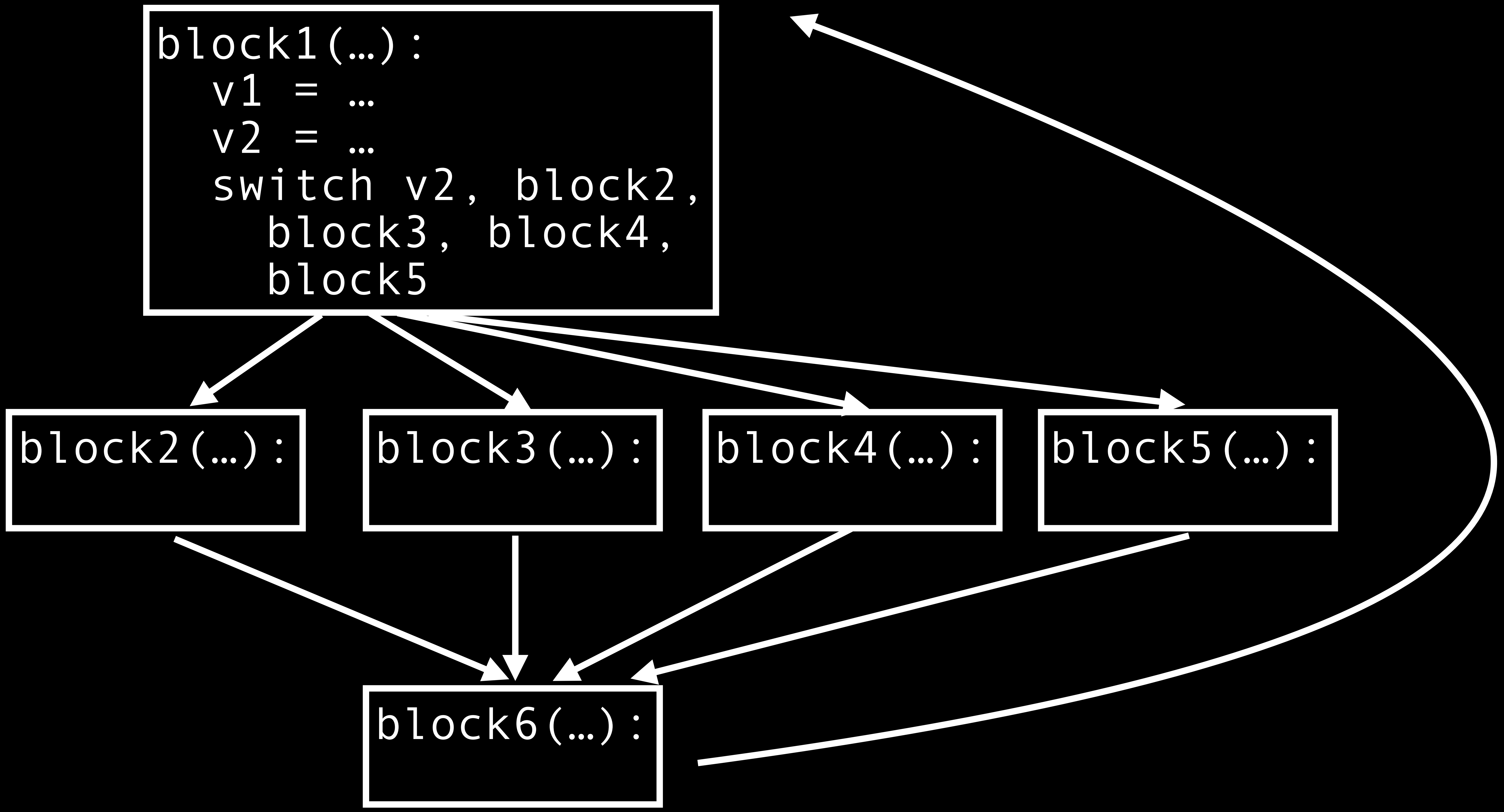
1. "No magic": only expand code where interpreter specifies via *context* mechanism

2. Partially evaluate iterations of the interpreter loop in a *context-sensitive* way, where the context is the bytecode PC

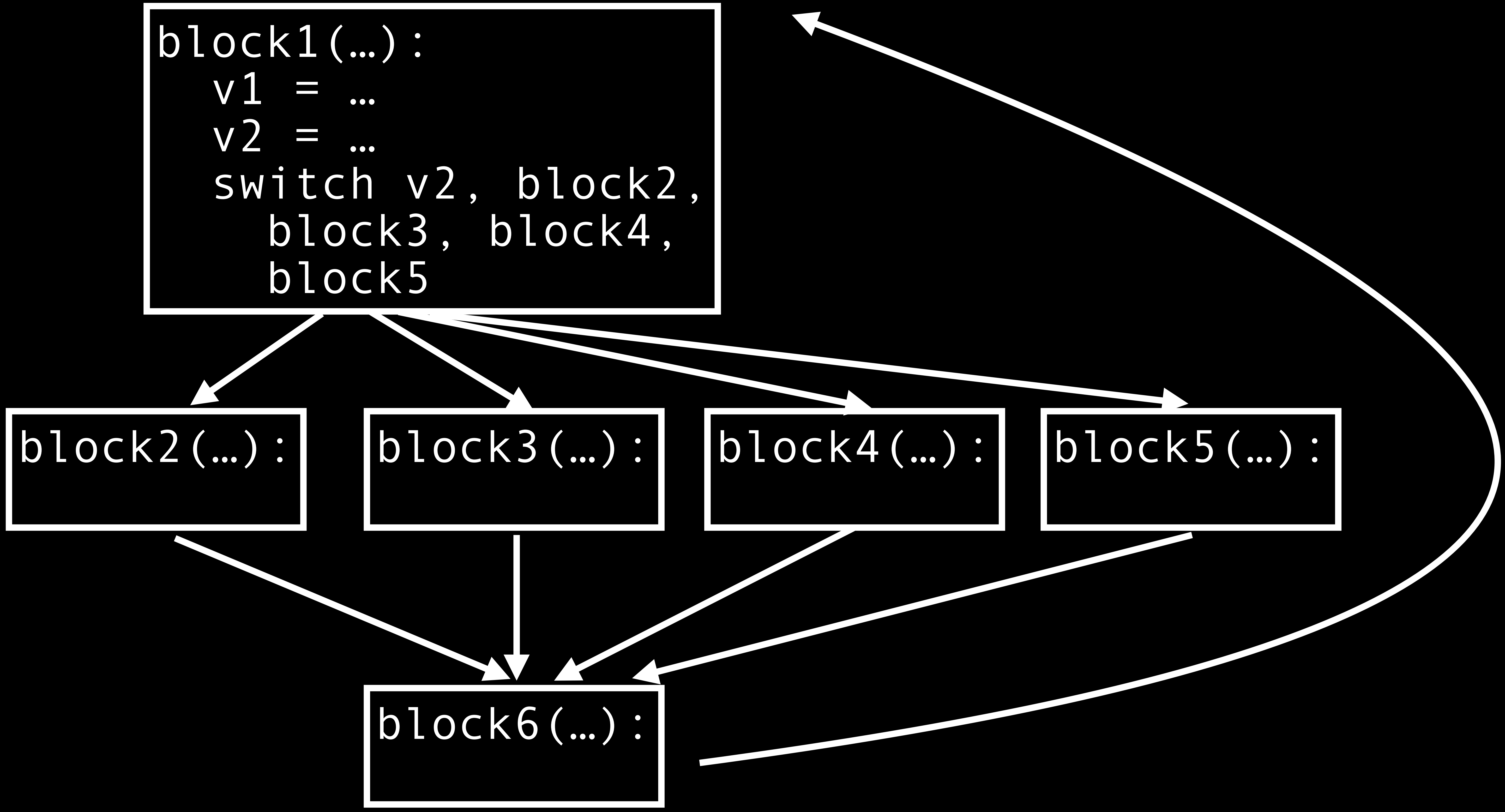
3. … and that's it.

```
void interp(bytecode* pc) {
  weval::push_context(pc);
  switch (*pc++) {
    case OP1:
      …
      weval::update_context(pc);
      break;
    case OP2:
      …
      weval::update_context(pc);
      break;
  }
}
```

The weval Transform



The weval Transform

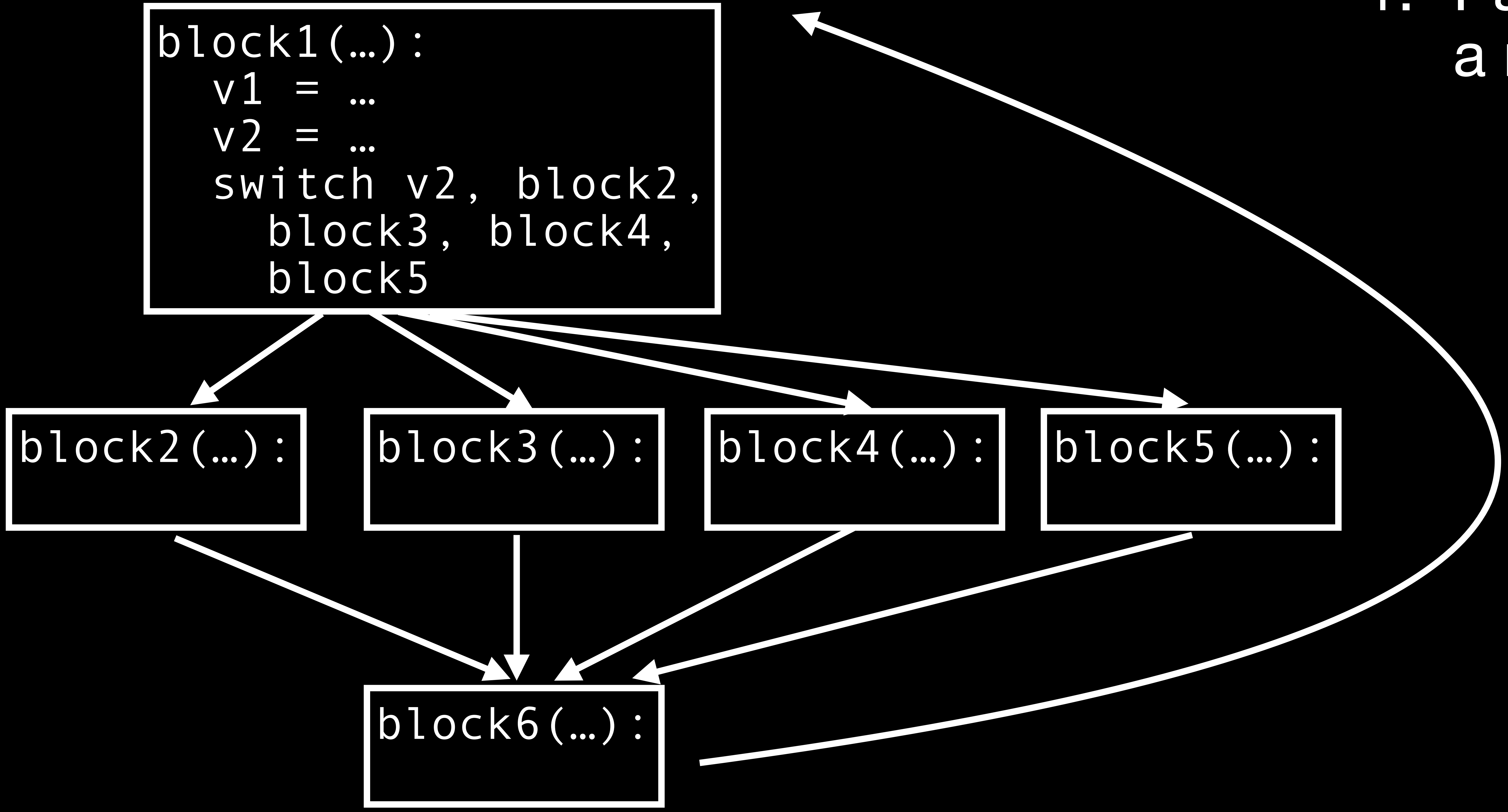


Generic

blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

Specialized

The weval Transform



1. Partially evaluate a block using a runtime/constant lattice

Generic

blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

Specialized

The weval Transform

1. Partially evaluate a block using a runtime/constant lattice

```
block1(…):
  v1 = …
  v2 = …
  switch v2, block2,
    block3, block4
    b
```

```
block2(…):
```

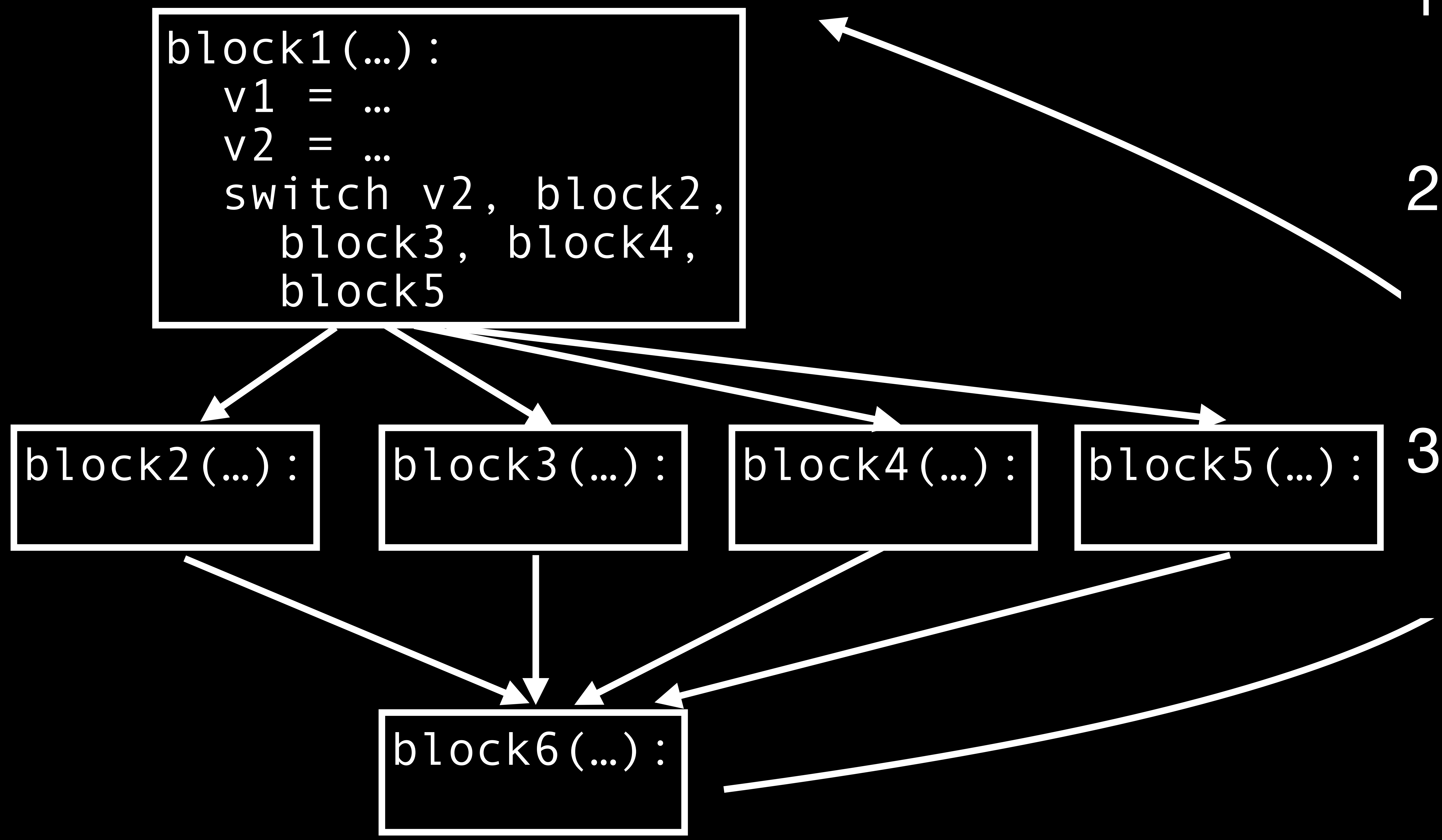
```
pub enum AbstractValue {
    /// "top" default value; undefined.
    Top,
    /// A value known at specialization time.
    Concrete(WasmVal),
    /// A value that points to memory known at specialization time,
    /// with the given offset.
    ConcreteMemory(MemoryBufferIndex, u32),
    /// A value only computed at runtime. The instruction that
    /// computed it is specified, if known.
    Runtime(Option<waffle::Value>),
}
```

blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

Generic

Specialized

The weval Transform



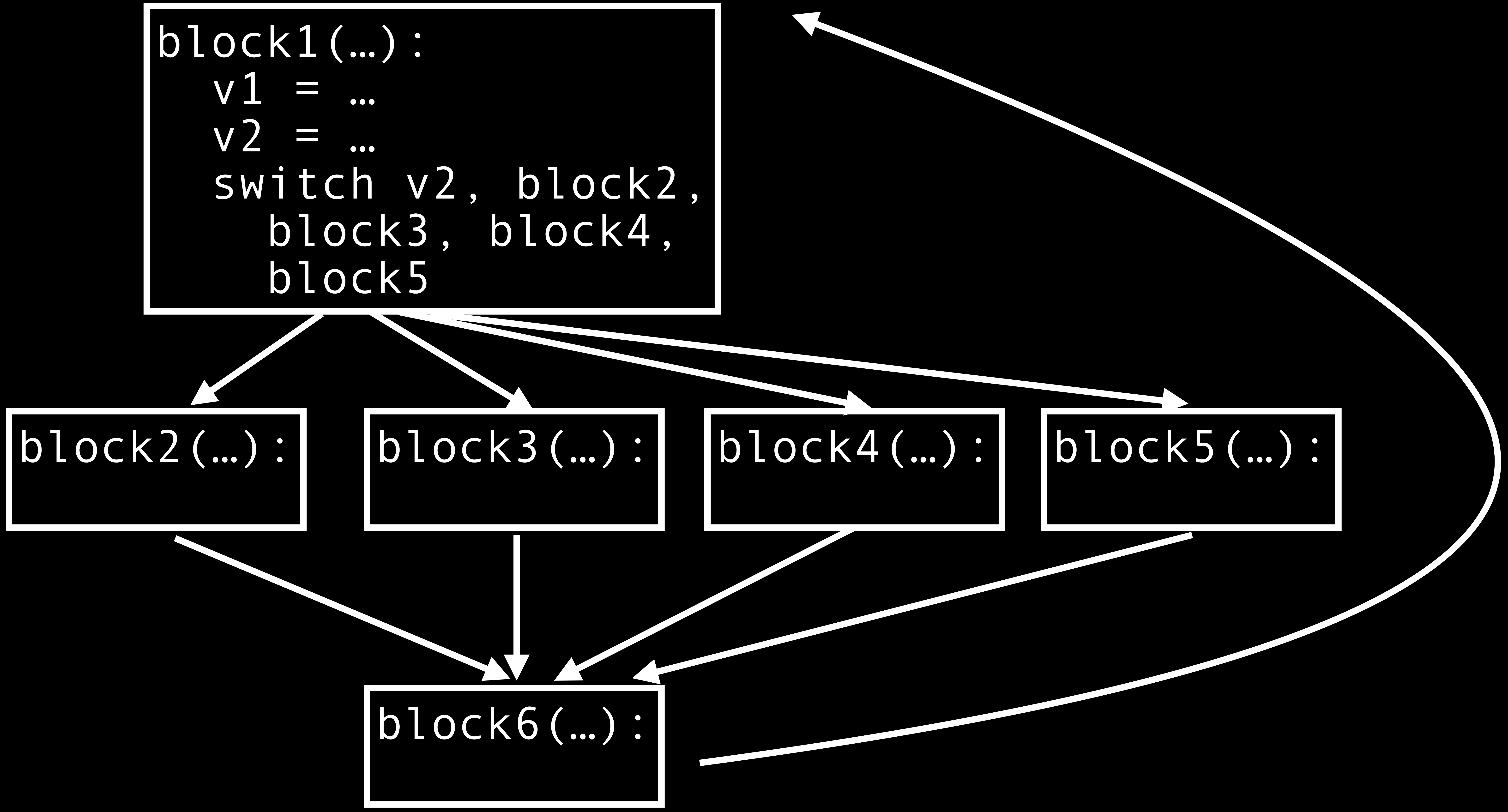
1. Partially evaluate a block using a runtime/constant lattice
2. Track *context* as part of flow-sensitive state; update at intrinsics
3. At branches, enqueue targets

blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

Generic

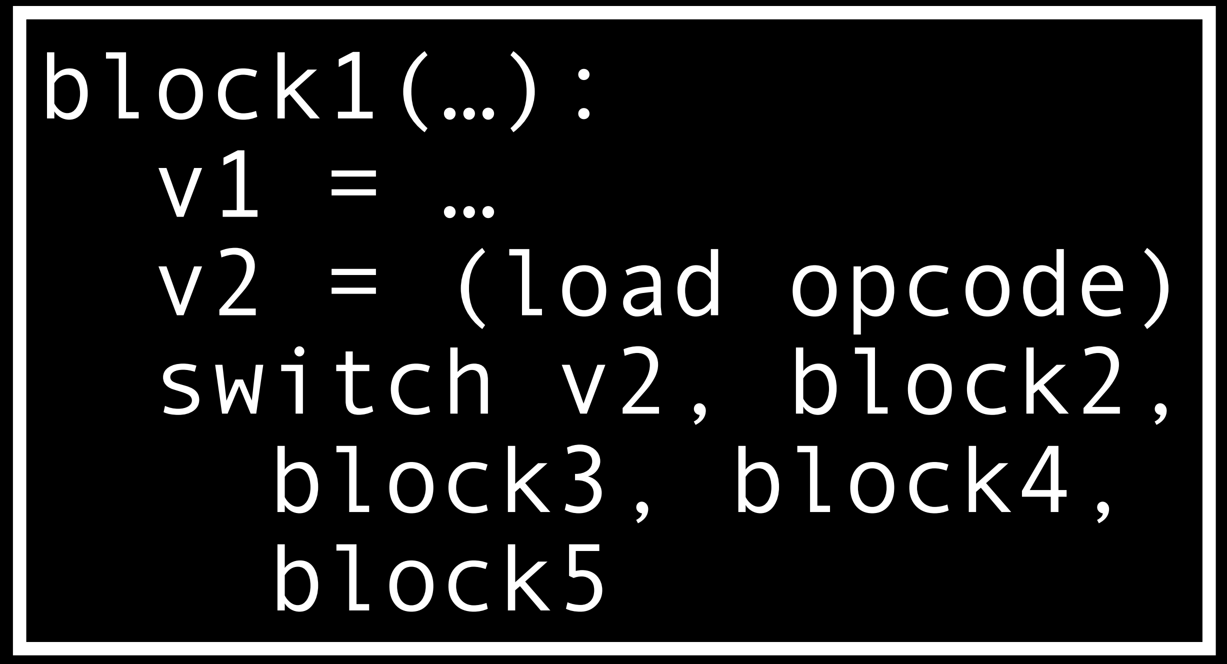
Specialized

The weval Transform



Generic

Context: PC 0

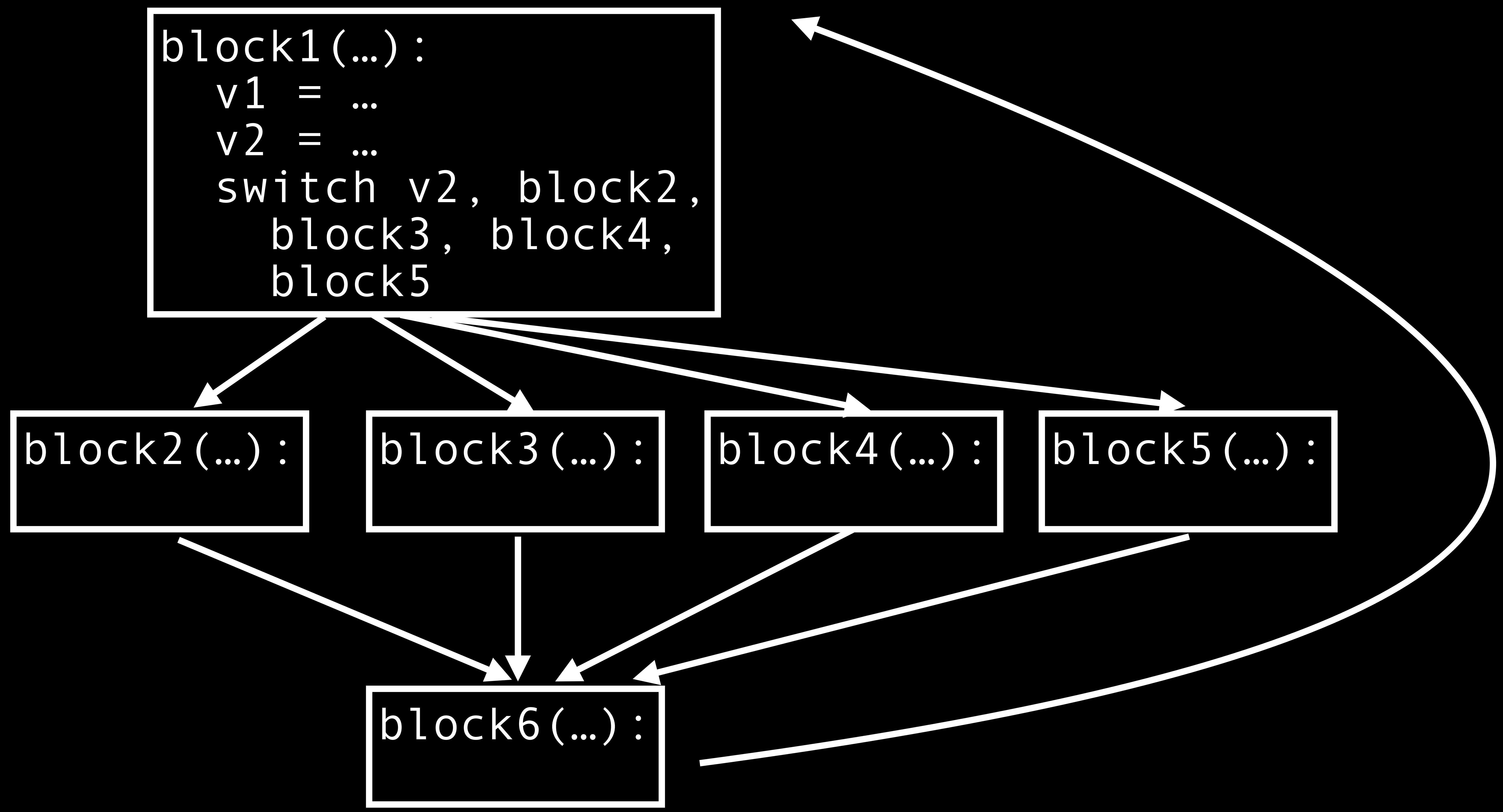


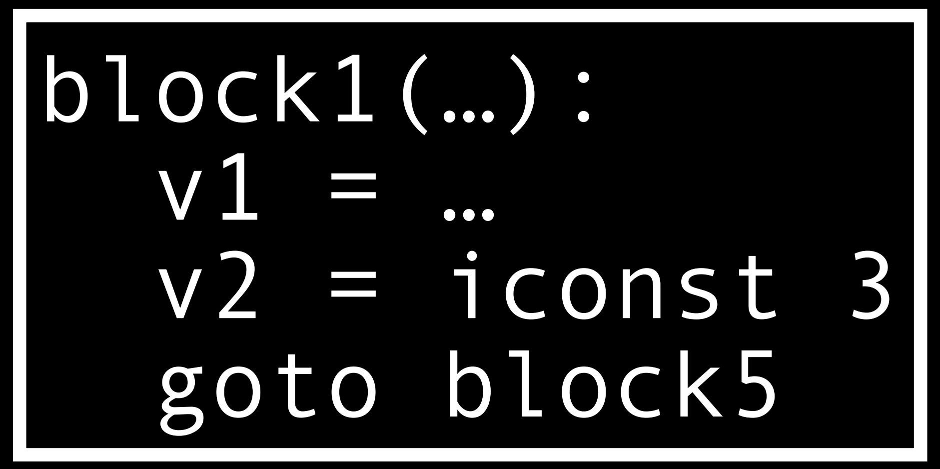
Specialized

blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

The weval Transform

Context: PC 0





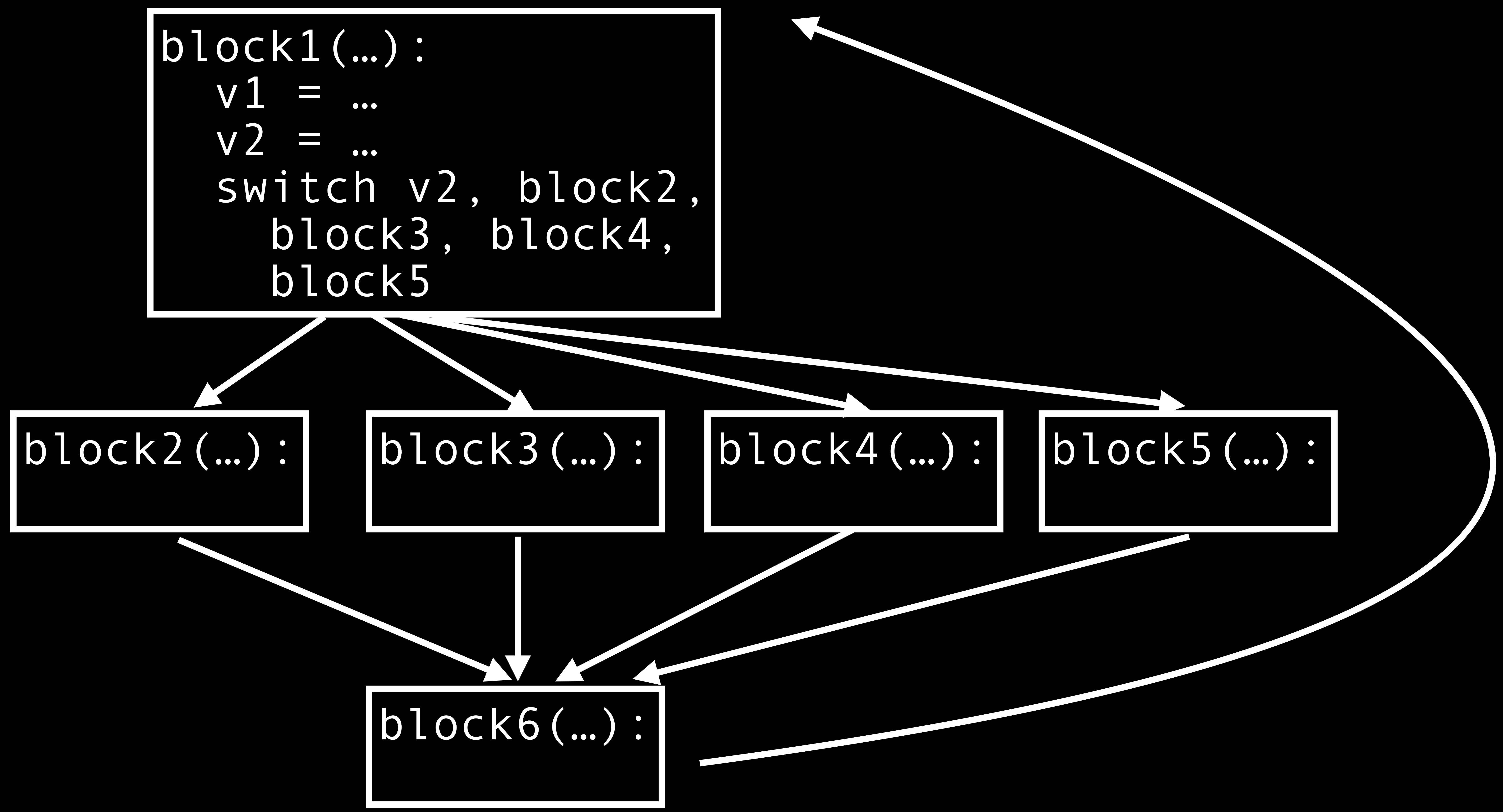
blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

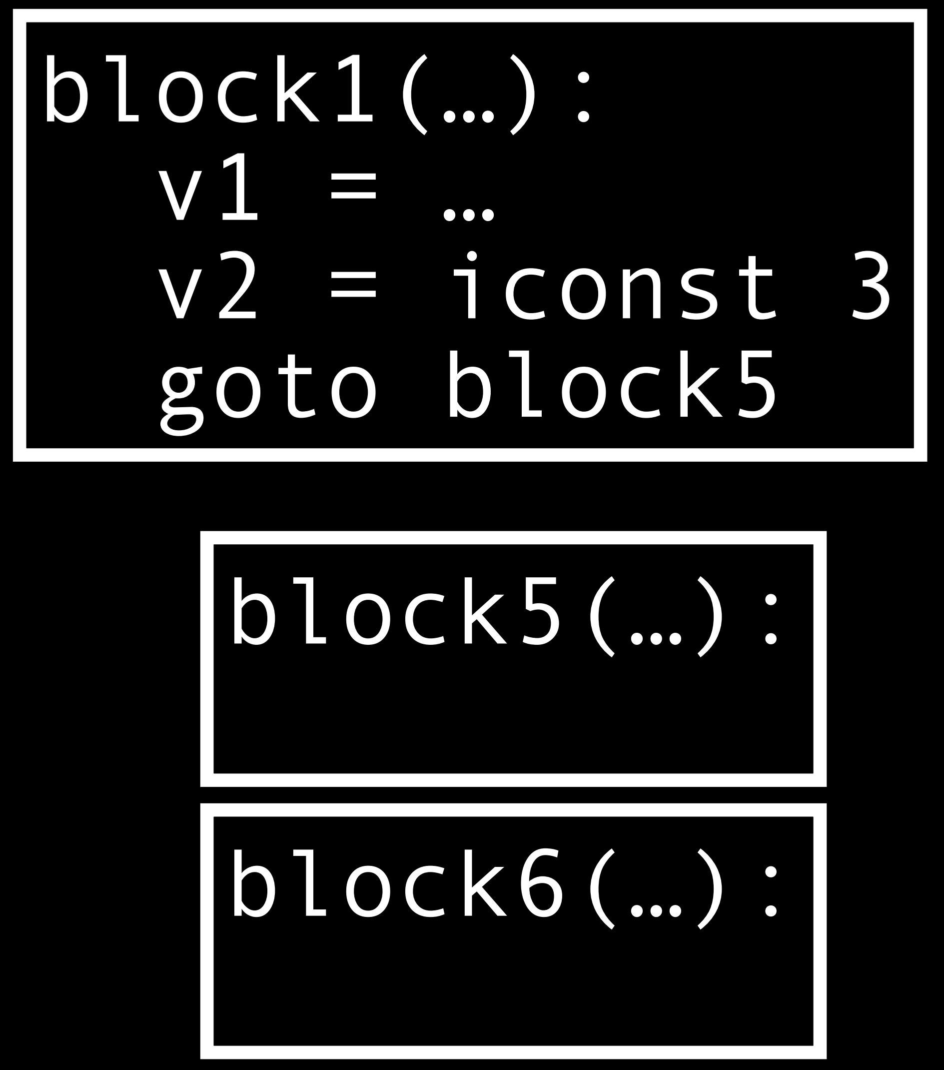
Generic

Specialized

The weval Transform

Context: PC 0



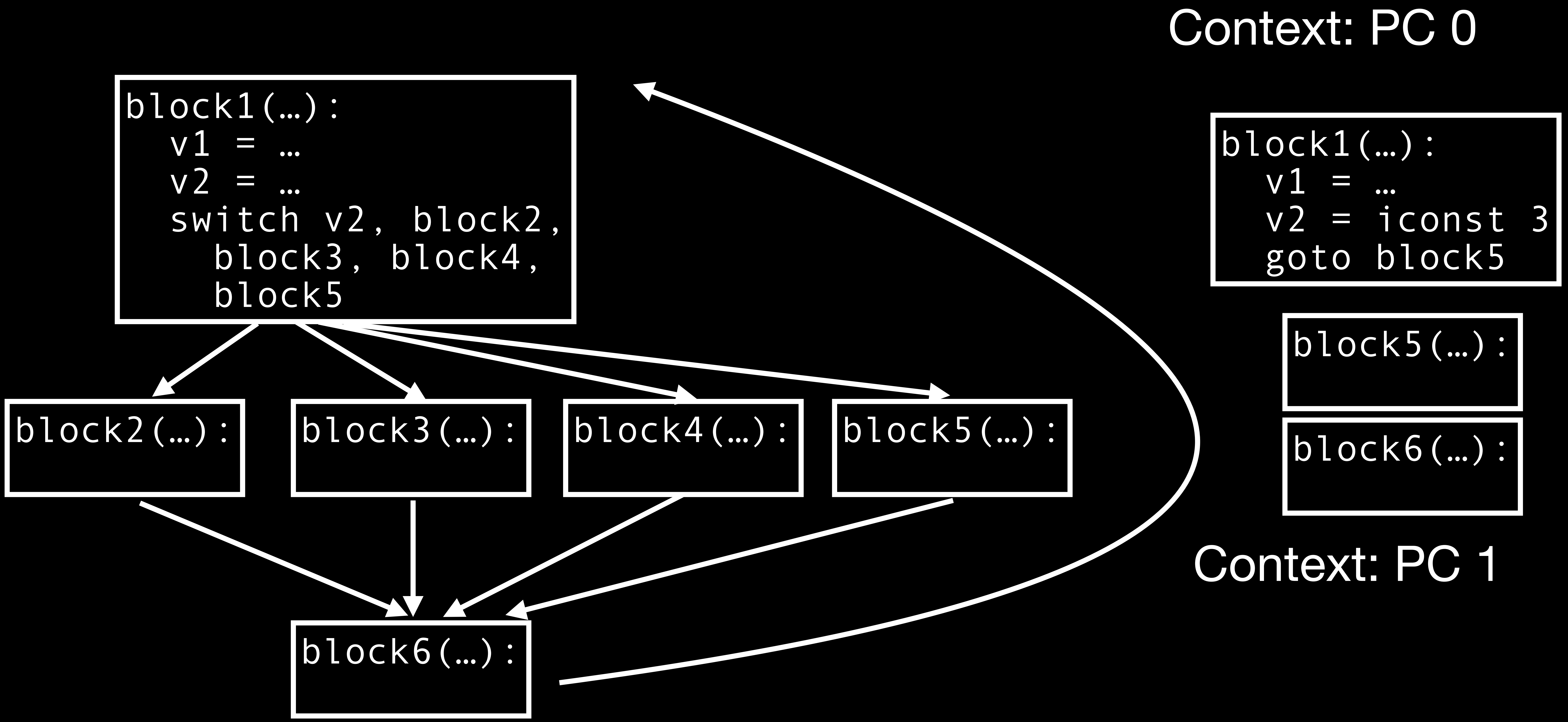


blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

Generic

Specialized

The weval Transform

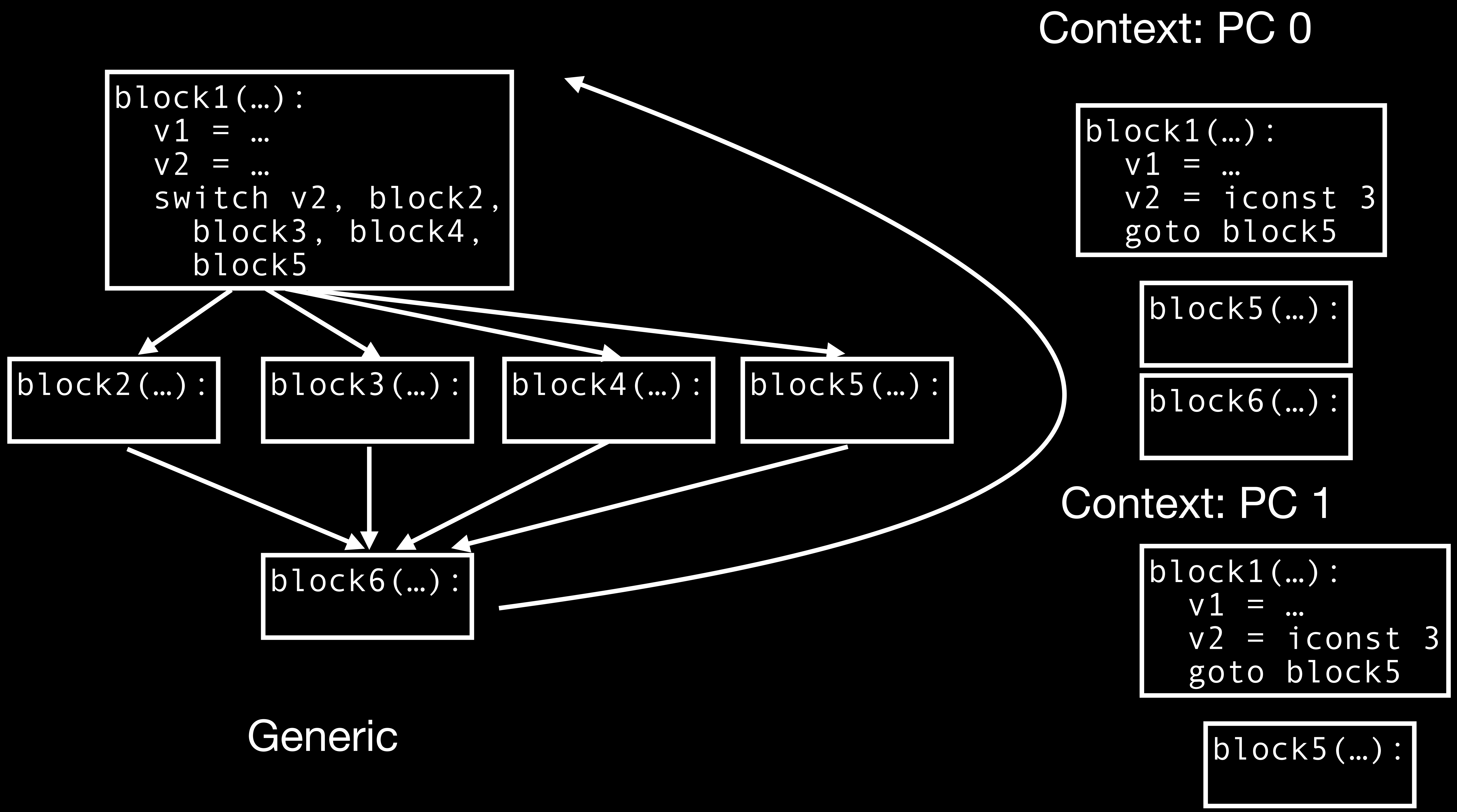


blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

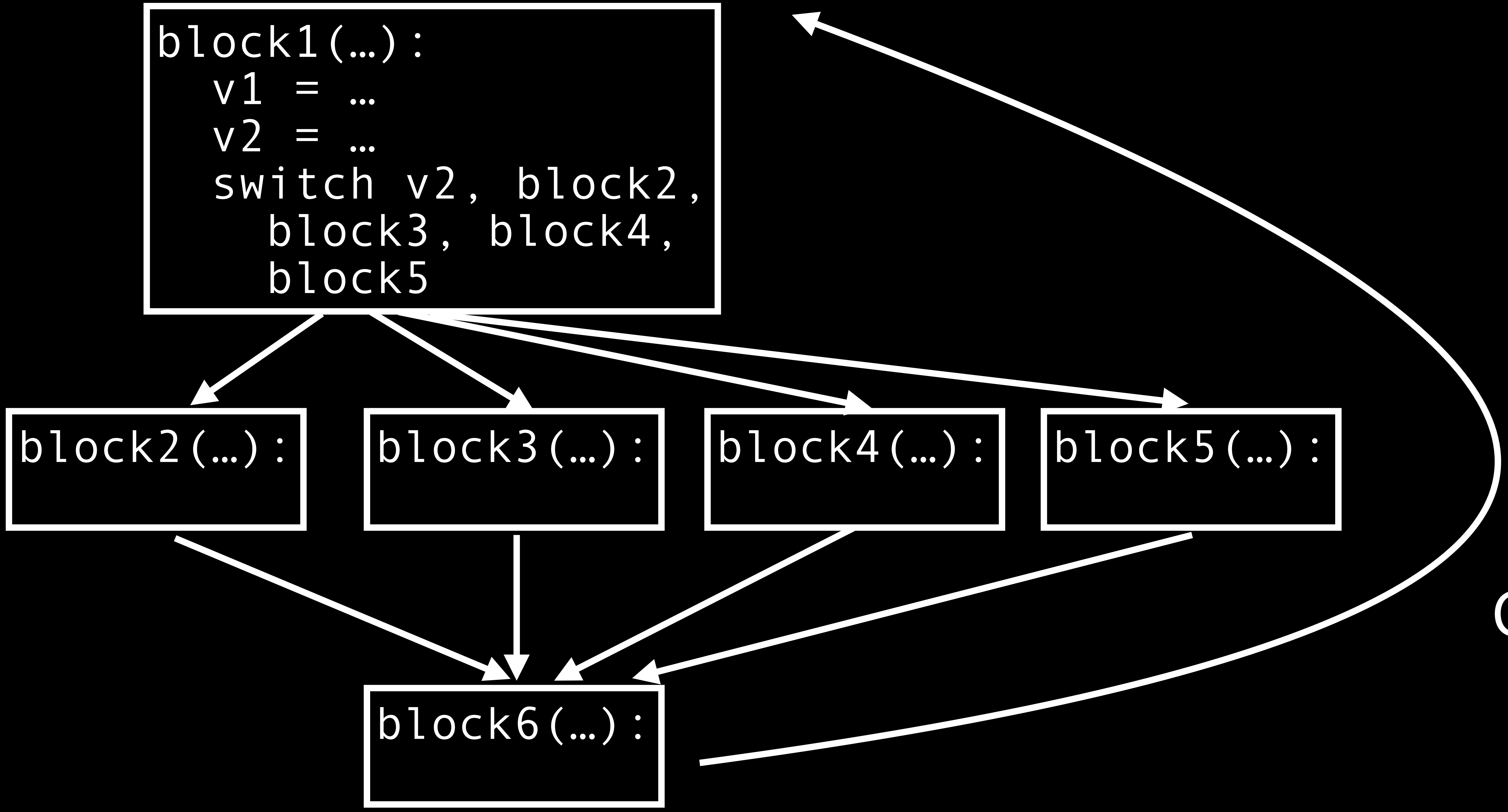
Generic

Specialized

The weval Transform



The weval Transform



Generic

PC 0: ADD
PC 1: GOTO 0

Context: PC 0

```
block1(…):
  v1 = …
  v2 = iconst 3
  goto block5
```

```
block5(…):
```

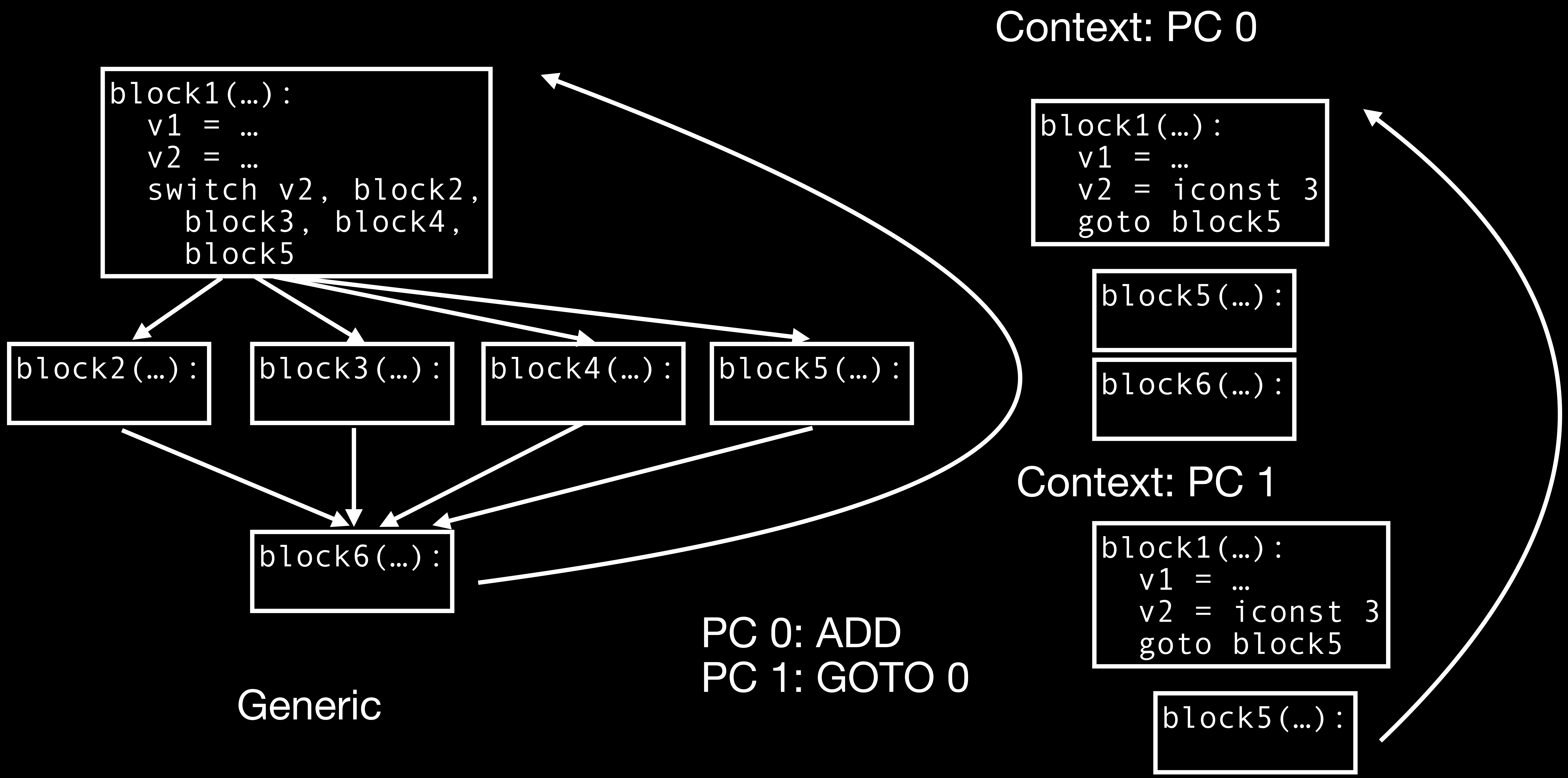
```
block6(…):
```

Context: PC 1

```
block1(…):
  v1 = …
  v2 = iconst 3
  goto block5
```

```
block5(…):
```

The weval Transform



The weval Transform

Context: PC 0

Resulting CFG is a *convolution* of interpreter's CFG and bytecode's CFG

bloc

PC 1: GOTO 0

Generic

block5(…):

The weval Transform

Context: PC 0

Resulting CFG is a *convolution* of interpreter’s CFG and bytecode’s CFG

🤯

bloc

PC 1: GOTO 0

Generic

block5(…):

A Note on SSA

- SSA validity is defined in terms of the dominator tree (def dominates uses)
- weval transform breaks dominance
- naive approach (worked at first!): convert to "maximal SSA" before transform
  - all live values passed via phis/blockparams at every block edge; then use only local values ("all other SSA is just an optimization of maximal SSA")
- Much better: find "cut blocks" based on "highest ancestor with same context" (property depends on position of ctx-change intrinsics)
  - Reduced value-number count in output code by 5x!

Other Intrinsics for Performance

- An interpreter will keep state in memory (“IC registers”) because of dynamic indexing; compiled code should be able to lift into SSA / dataflow in Wasm
- Intrinsics:

```
weval_read_reg(index)
weval_write_reg(index, value)
```

- New intrinsics are OK when they have well-defined semantics and *could* be polyfilled without weval transform
- Initially tried to implement “memory renaming” —> very fragile (pointer escapes, semantics on calls?, …)

Value Specialization

- Ideally, implementation of a control-flow op looks like

```
auto value = pop();
if (value) {
  pc = A;
  weval::update_context(pc);
  goto dispatch;
} else {
  pc = B;
  weval::update_context(pc);
  goto dispatch;
}
```

- Key property: edge to a different *static* block in bytecode should be a different *static* program point (otherwise edge resolution / block linkup doesn't work)

- “Switch” opcodes are problematic — load PC from a table

- offer a “specialize context N ways with i=0..N-1” intrinsic

Portable Specialization Requests and Time-Travel

- weval specialization request is (funcptr, args) tuple — plain old data, independent of Wasm heap state
  - Very important: bundle the *content* of constant memory, not just const ptr
- Collecting IC bodies: collect a bunch of weval requests, do them eagerly on subsequent wevalings, and inject a “look up by arg string” hashtable
- Also: makes deterministic weval caching very nice (processing speed!)

Requirements on Interpreter

- *Function-level* control flow in *interpreter* must match source language
  - Because weval specialization is function-to-function and Wasm functions are first-class
  - Often this means disabling an “inline call frame” optimization
  - We can’t support source-language tail calls until Wasm does
  - We can’t support O(1) exception unwind until Wasm does
    (… do O(n) unwind for now)
- Bytecode must remain constant, and PC must be statically context-sensitively resolvable (no “indirect branch to arbitrary offset” opcode)

Other Optimizations

- By itself, weval removes opcode-dispatch overhead, and puts opcode cases next to each other statically (—> opt opportunities), but still copy+pastes inefficiencies and bloat into target code
- Good idea for faster code *and* faster weval processing: out-of-line special cases
  - (Ab)use C++ template parameters to build several versions of interp(); specialized version tailcalls into generic version (non-wevaled) for error paths

Results

Bench	Base	PBL		wevaled PBL	

Richards	164	280	(1.71x)	444	(2.71x)
DeltaBlue	167	321	(1.92x)	435	(2.60x)
Crypto	453	566	(1.25x)	1231	(2.72x)
RayTrace	498	786	(1.58x)	827	(1.66x)
EarleyBoyer	712	1070	(1.50x)	1178	(1.65x)
RegExp	273	337	(1.23x)	421	(1.54x)
Splay	1293	2147	(1.66x)	2809	(2.17x)
NavierStokes	684	763	(1.32x)	1336	(1.95x)
PdfJS	2220	2512	(1.31x)	4150	(1.87x)
Mandreel	189	233	(1.23x)	399	(2.11x)
Gameboy	1479	1774	(1.20x)	3122	(2.11x)
CodeLoad	19765	18994	(0.96x)	17735	(0.90x)
Box2D	943	1328	(1.41x)	2134	(2.26x)

Overall	848	1127	(1.33x)	1654	(1.95x)

Results

Bench	Base	PBL		wevaled PBL	
Richards	164	280	(1.71x)	444	(2.71x)
DeltaBlue	167	321	(1.92x)	435	(2.60x)
Crypto	453	566	(1.25x)	1231	(2.72x)
RayTrace	498	786	(1.58x)	827	(1.66x)
Mandreel	189	233	(1.23x)	399	(2.11x)
Gameboy	1479	1774	(1.20x)	3122	(2.11x)
CodeLoad	19765	18994	(0.96x)	17735	(0.90x)
Box2D	943	1328	(1.41x)	2134	(2.26x)
Overall	848	1127	(1.33x)	1654	(1.95x)

Loop microbenchmark, latest optimizations, optimistic removal of some Wasm overheads (typed funcref table bounds checks, stacklimit checks):

~4x speedup
(native baseline compiler: 5.33x)

What’s Left?

- “Fast dispatch” intrinsics — indirect calls in Wasm are very slow
  - (sidenote: Igalia online-JIT emits direct calls to top IC when known)
- Intrinsics for operand stack — abstract-interpret push/pop
  - Limited due to GC-safepoint constraints but still some opportunity
- Optimize Wasmtime/Cranelift with these workloads in mind
  - Silly ABI hacks (pack i64s into i64x2 SIMD to get more arg registers…)
- … and build an optimizing JIT compiler with “cloud PGO”

Thanks! Questions?