Feb 28, 2025

# A Quick Ramp-Up On Ramping Up Quickly

Iain Ireland

Mozilla

# So you wake up one morning, and you’re a JavaScript engine…

# A website hands you some code

**JavaScript**

```
function add(a, b) {
    return a + b;
}
```

# A website hands you some code

**JavaScript**

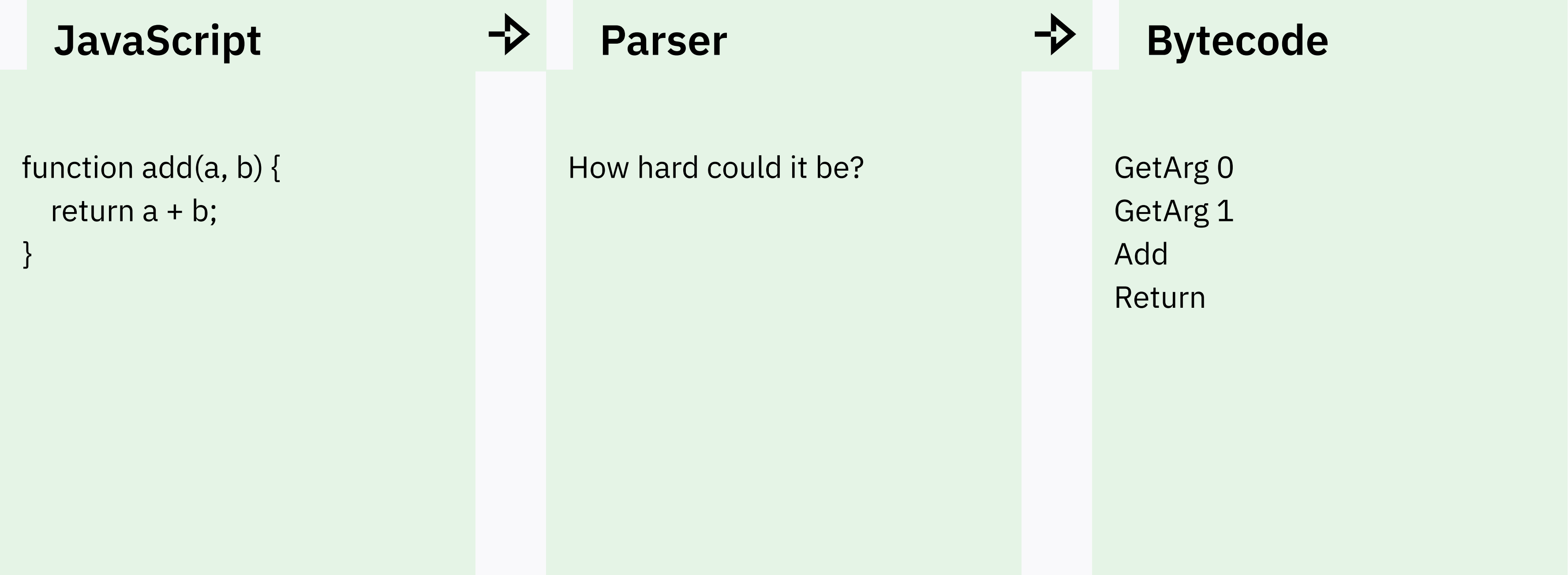
```
function add(a, b) {
   return a + b;
}
```

**Parser**

How hard could it be?

https://github.com/mozilla-spidermonkey/jsparagus/blob/master/js-quirks.md

# A website hands you some code



**JavaScript**

```
function add(a, b) {
    return a + b;
}
```

**Parser**

How hard could it be?

**Bytecode**

```
GetArg 0
GetArg 1
Add
Return
```

# Interpreter

# Interpreter

```
JSOp op = readOp();
switch (op) {
```

# Interpreter

```
JSOp op = readOp();
switch (op) {
    ...
  case JSOp::GetArg:
      uint16_t argno = readUint16();
      push(getArg(argno));
      break;
    ...
```

# Interpreter

```
JSOp op = readOp();
switch (op) {
    ...
  case JSOp::GetArg:
      uint16_t argno = readUint16();
      push(getArg(argno));
      break;
    ...
  case JSOp::Add:
      Value lhs = pop();
      Value rhs = pop();
      push(add(lhs, rhs));
      break;
```

# Interpreter

```
JSOp op = readOp();
switch (op) {
     ...
    case JSOp::GetArg:
        uint16_t argno = readUint16();
        push(getArg(argno));
        break;
      ...
    case JSOp::Add:
        Value lhs = pop();
        Value rhs = pop();
        push(add(lhs, rhs));
        break;
    case JSOp::Sub:
       ...
```

# Interpreter

```
while (true) {

  JSOp op = readOp();
  switch (op) {
     ...
    case JSOp::GetArg:
      uint16_t argno = readUint16();
      push(getArg(argno));
      break;
      ...
    case JSOp::Add:
      Value lhs = pop();
      Value rhs = pop();
      push(add(lhs, rhs));
      break;
    case JSOp::Sub:
      ...
  }
}
```

# Interpreter

```
while (true) {

  JSOp op = readOp();
  switch (op) {
     ...
    case JSOp::GetArg:
      uint16_t argno = readUint16();
      push(getArg(argno));
      break;

     ...
    case JSOp::Add:
      Value lhs = pop();
      Value rhs = pop();
      push(add(lhs, rhs));
      break;
    case JSOp::Sub:

      ...
  }
}
```

# The website comes back...

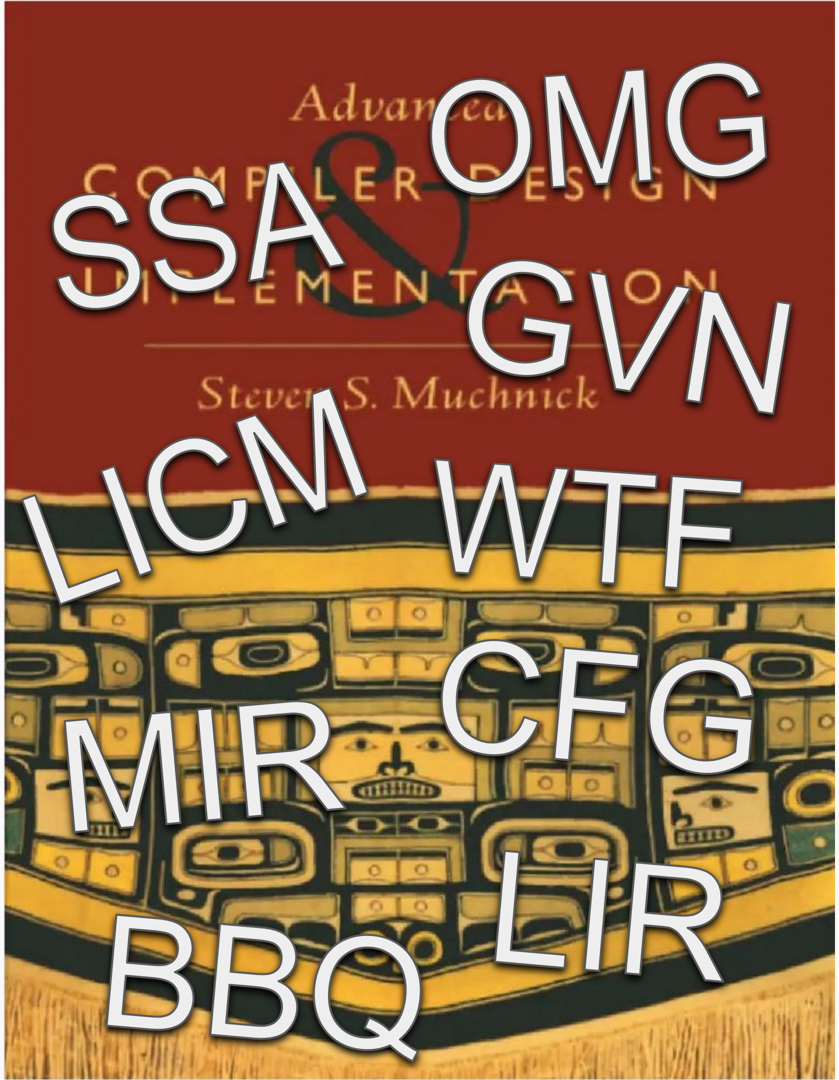
```
while (true) {

    JSOp op = readOp();
    switch (op) {
        ...
        case JSOp::GetArg:
            uint16_t argno = readUint16();
            push(getArg(argno));
            break;
        ...
        case JSOp::Add:
            Value lhs = pop();
            Value rhs = pop();
            push(add(lhs, rhs));
            break;
        case JSOp::Sub:
            ...
    }
}
```

# The website comes back...

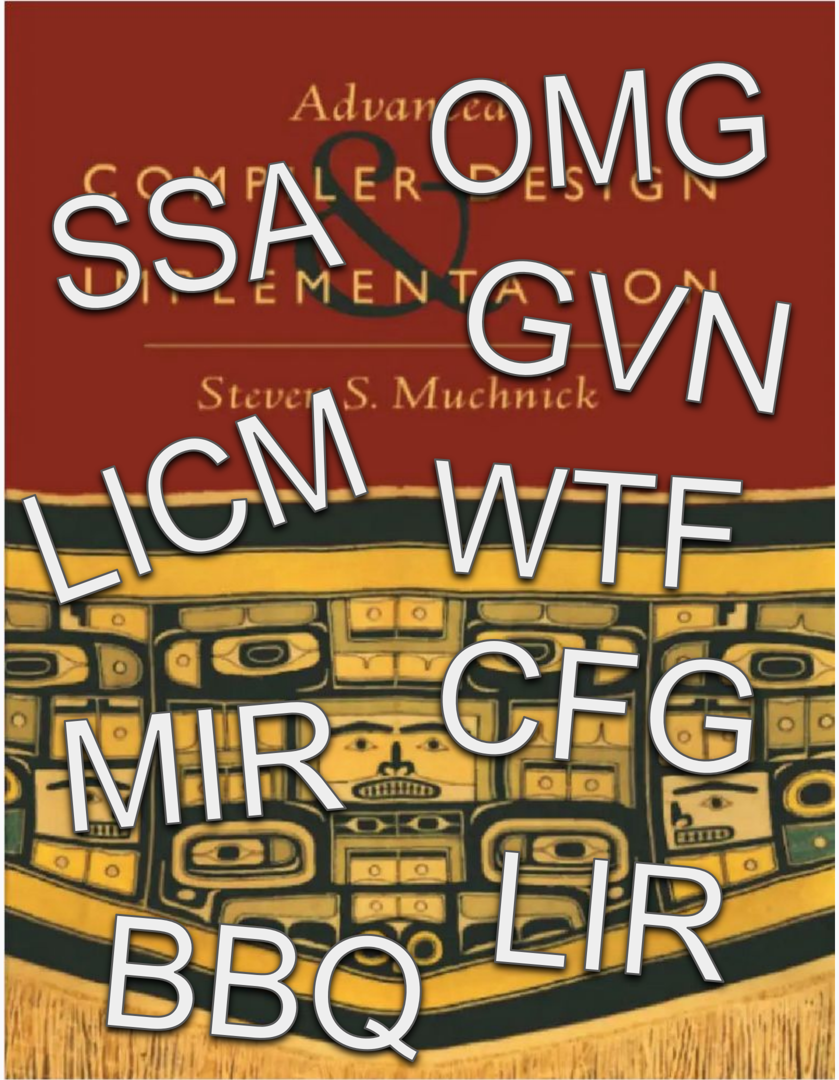
```
while (true) {

  JSOp op = readOp();
  switch (op) {
     ...
    case JSOp::GetArg:
      uint16_t argno = readUint16();
      push(getArg(argno));
      break;

    case JSOp::Add:
      Value lhs = pop();
      Value rhs = pop();
      push(add(lhs, rhs));
      break;
    case JSOp::Sub:

       ...
  }
}
```

TOO SLOW

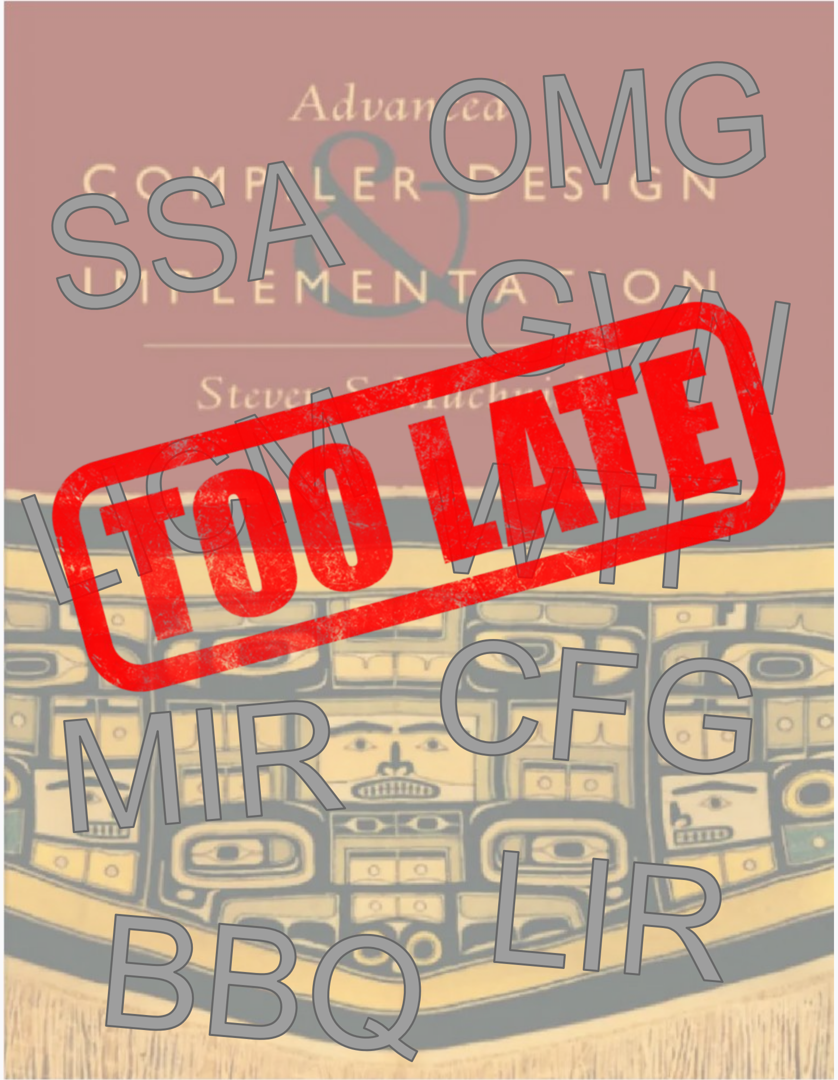
# It’s time to get serious



# The website comes back...



# The website comes back...



# AOT vs JIT

# AOT vs JIT

**1** AOT compilation only costs developer time.

# AOT vs JIT

1. AOT compilation only costs developer time.
2. Time spent JIT compiling competes with running code.

# AOT vs JIT

1. AOT compilation only costs developer time.

2. Time spent JIT compiling competes with running code.

3. AOT-compiled code is available immediately.

# AOT vs JIT

1. AOT compilation only costs developer time.
2. Time spent JIT compiling competes with running code.
3. AOT-compiled code is available immediately.
4. JIT-compiled code is available when the compile finishes.

# AOT vs JIT

1. AOT compilation only costs developer time.
2. Time spent JIT compiling competes with running code.
3. AOT-compiled code is available immediately.
4. JIT-compiled code is available when the compile finishes.
5. AOT compilation can be amortized across more uses.

# Tiering

**1** Code varies wildly in importance.

# Tiering

1. Code varies wildly in importance.
2. Some code is very hot.

# Tiering

1. Code varies wildly in importance.
2. Some code is very hot.
3. Some code runs exactly once, or never runs at all.

# Tiering

1. Code varies wildly in importance.
2. Some code is very hot.
3. Some code runs exactly once, or never runs at all.
4. No one-size-fits-all solution. Multiple tiers are necessary.

# Tiering

1. Code varies wildly in importance.
2. Some code is very hot.
3. Some code runs exactly once, or never runs at all.
4. No one-size-fits-all solution. Multiple tiers are necessary.
5. Lower tiers can collect profiling data for higher tiers.

# Baseline Compiler

# Baseline Compiler

**Bytecode**

```
GetArg 0
GetArg 1
Add
Return
```

# Baseline Compiler

## Bytecode

**GetArg 0**
GetArg 1
Add
Return

# Baseline Compiler

## Bytecode

**GetArg 0**
GetArg 1
Add
Return

## Handler: GetArg

1. Load argument from caller’s stack frame.
2. Push it on the stack.

# Baseline Compiler

## Bytecode

**GetArg 0**
GetArg 1
Add
Return

## Handler: GetArg

1. Load argument from caller’s stack frame.
2. Push it on the stack.

## Machine Code

**push 0x28(rbp)**

# Baseline Compiler

## Bytecode

GetArg 0
**GetArg 1**
Add
Return

## Handler

## Machine Code

push 0x28(rbp)

# Baseline Compiler

## Bytecode

GetArg 0
**GetArg 1**
Add
Return

## Handler: GetArg

1. Load argument from caller’s stack frame.
2. Push it on the stack.

## Machine Code

push 0x28(rbp)

# Baseline Compiler

## Bytecode

GetArg 0
**GetArg 1**
Add
Return

## Handler: GetArg

1. Load argument from caller’s stack frame.
2. Push it on the stack.

## Machine Code

push 0x28(rbp)
**push 0x30(rbp)**

# Baseline Compiler

### Bytecode

```
GetArg 0
GetArg 1
Add
Return
```

### Handler: Add

**???**

### Machine Code

```
push 0x28(rbp)
push 0x30(rbp)
```

# What is an Add?

# What is an Add?

**1**

**Math**

Numbers, BigInts

```
add(1, 2);

add(1.5, Math.PI);

add(1n, 10000000000000000000n);
```

# What is an Add?

1 **Math**
Numbers, BigInts

```
add(1, 2);

add(1.5, Math.PI);

add(1n, 10000000000000000000n);
```

2 **Concatenation**
Strings

```
add(“hello “, “world”);
```

# What is an Add?

**1** **Math**
Numbers, BigInts

```
add(1, 2);

add(1.5, Math.PI);

add(1n, 10000000000000000000n);
```

**2** **Concatenation**
Strings

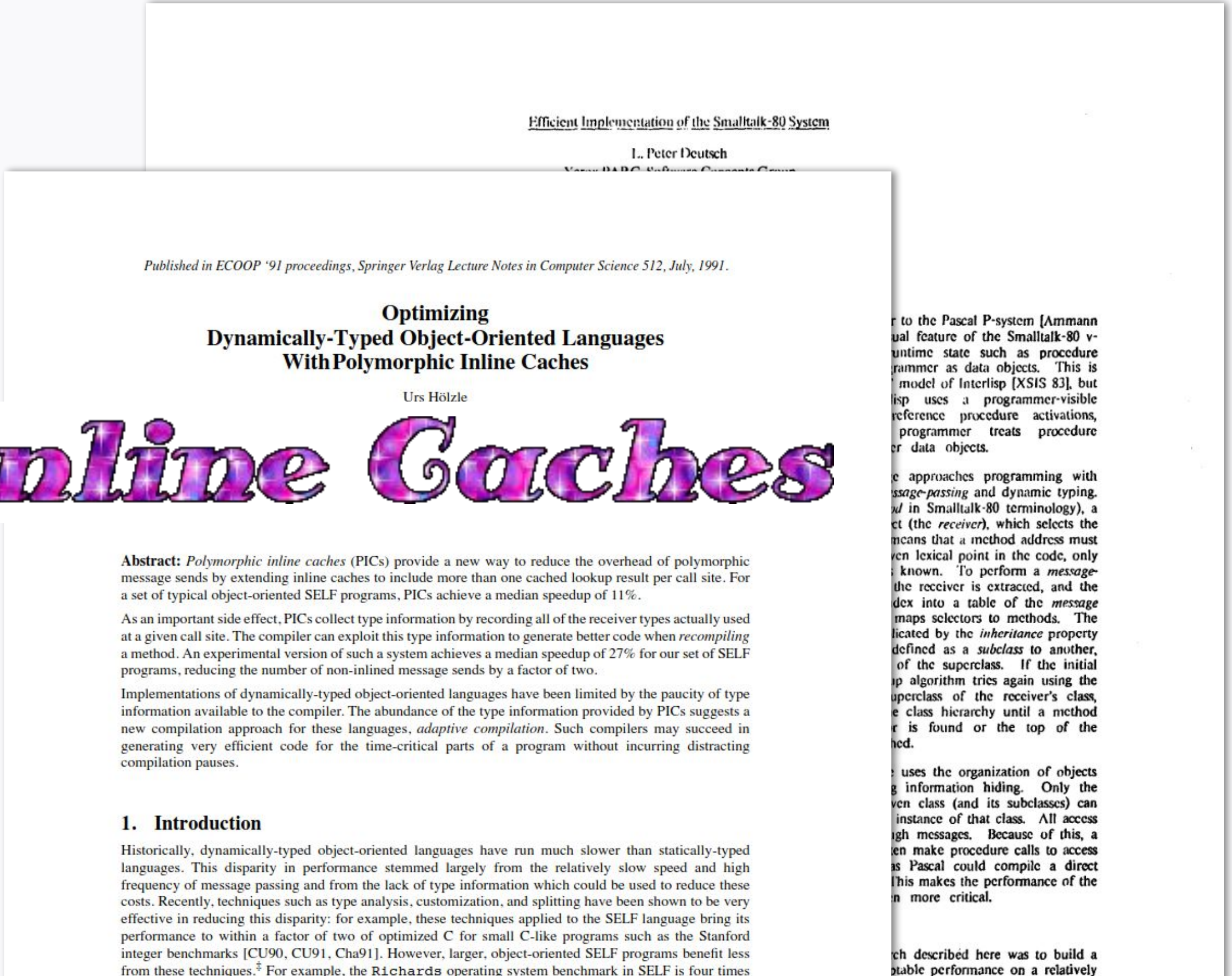
```
add(“hello “, “world”);
```

**3** **Arbitrary Nonsense**
Objects

```
add(“hello ”, { toString: ( ) => “world” });
```

# The most sincere form of flattery

Inline Caches



*Published in ECOOP '91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, July, 1991.*

## Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches

Urs Hölzle

**Abstract:** *Polymorphic inline caches* (PICs) provide a new way to reduce the overhead of polymorphic message sends by extending inline caches to include more than one cached lookup result per call site. For a set of typical object-oriented SELF programs, PICs achieve a median speedup of 11%.

As an important side effect, PICs collect type information by recording all of the receiver types actually used at a given call site. The compiler can exploit this type information to generate better code when *recompiling* a method. An experimental version of such a system achieves a median speedup of 27% for our set of SELF programs, reducing the number of non-inlined message sends by a factor of two.

Implementations of dynamically-typed object-oriented languages have been limited by the paucity of type information available to the compiler. The abundance of the type information provided by PICs suggests a new compilation approach for these languages, *adaptive compilation*. Such compilers may succeed in generating very efficient code for the time-critical parts of a program without incurring distracting compilation pauses.

### 1. Introduction

Historically, dynamically-typed object-oriented languages have run much slower than statically-typed languages. This disparity in performance stemmed largely from the relatively slow speed and high frequency of message passing and from the lack of type information which could be used to reduce these costs. Recently, techniques such as type analysis, customization, and splitting have been shown to be very effective in reducing this disparity: for example, these techniques applied to the SELF language bring its performance to within a factor of two of optimized C for small C-like programs such as the Stanford integer benchmarks [CU90, CU91, Cha91]. However, larger, object-oriented SELF programs benefit less from these techniques.[‡] For example, the `Richards` operating system benchmark in SELF is four times

---

Efficient Implementation of the Smalltalk-80 System

L. Peter Deutsch

r to the Pascal P-system [Ammann
ual feature of the Smalltalk-80 v-
untime state such as procedure
rammer as data objects. This is
model of Interlisp [XSIS 83], but
isp uses a programmer-visible
reference procedure activations,
programmer treats procedure
r data objects.

e approaches programming with
ssage-passing and dynamic typing.
d in Smalltalk-80 terminology), a
ct (the *receiver*), which selects the
means that a method address must
ven lexical point in the code, only
known. To perform a *message-*
the receiver is extracted, and the
dex into a table of the *message*
maps selectors to methods. The
licated by the *inheritance* property
defined as a *subclass* to another,
of the superclass. If the initial
p algorithm tries again using the
uperclass of the receiver's class,
e class hierarchy until a method
r is found or the top of the
hed.

uses the organization of objects
information hiding. Only the
ven class (and its subclasses) can
instance of that class. All access
gh messages. Because of this, a
en make procedure calls to access
s Pascal could compile a direct
This makes the performance of the
n more critical.

ch described here was to build a
table performance on a relatively

# Inline Caches

**Dynamic Languages**
Mostly static!

Past behaviour is highly correlated with future behaviour.

**Deciding what to do is hard**
Doing it is easy

There's a common pattern:

- Many things *could* happen
- Picking the right path is slow
- Validating a single path is fast

**Cache a fast path**
Next time will be better

You only need a few cheap guards, once you know what inputs to expect.

# Baseline Compiler

## Bytecode

GetArg 0
GetArg 1
**Add**
Return

## Handler: Add

## Machine Code

push 0x28(rbp)
push 0x30(rbp)

# Baseline Compiler

## Bytecode

GetArg 0
GetArg 1
**Add**
Return

## Handler: Add

1. **Load the array of IC entries.**

## Machine Code

push 0x28(rbp)
push 0x30(rbp)
**movq -0x28(rbp), rdi**

# Baseline Compiler

## Bytecode

GetArg 0
GetArg 1
**Add**
Return

## Handler: Add

1. Load the array of IC entries.
2. **Load the entry for this op.**

## Machine Code

```
push 0x28(rbp)
push 0x30(rbp)
movq -0x28(rbp), rdi
movq 0x10(rdi), rdi
```

# Baseline Compiler

## Bytecode

```
GetArg 0
GetArg 1
Add
Return
```

## Handler: Add

1. Load the array of IC entries.
2. Load the entry for this op.
3. **Call it!**

## Machine Code

```
push 0x28(rbp)
push 0x30(rbp)
movq -0x28(rbp), rdi
movq 0x10(rdi), rdi
callq (rdi)
```

# Baseline Compiler

## Bytecode

GetArg 0
GetArg 1
Add
**Return**

## Handler

## Machine Code

```
push 0x28(rbp)
push 0x30(rbp)
movq -0x28(rbp), rdi
movq 0x10(rdi), rdi
callq (rdi)
```

# Baseline Compiler

## Bytecode

GetArg 0  
GetArg 1  
Add  
**Return**

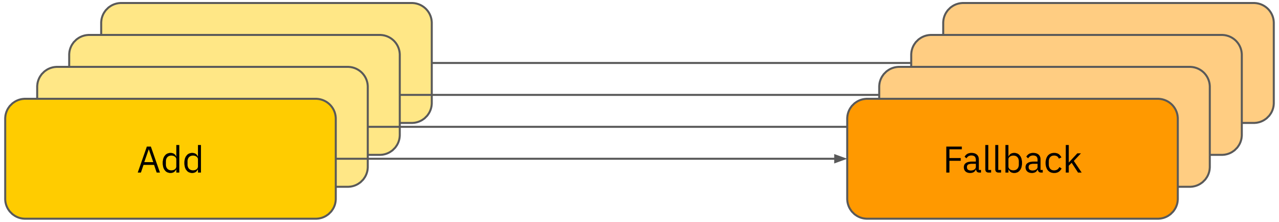
## Handler: Return

1. Load the return value into the return register.
2. Return!

## Machine Code

push 0x28(rbp)  
push 0x30(rbp)  
movq -0x28(rbp), rdi  
movq 0x10(rdi), rdi  
callq (rdi)  
**pop rax**  
**mov rbp, rsp**  
**pop rbp**  
**ret**

# Keeping your promises



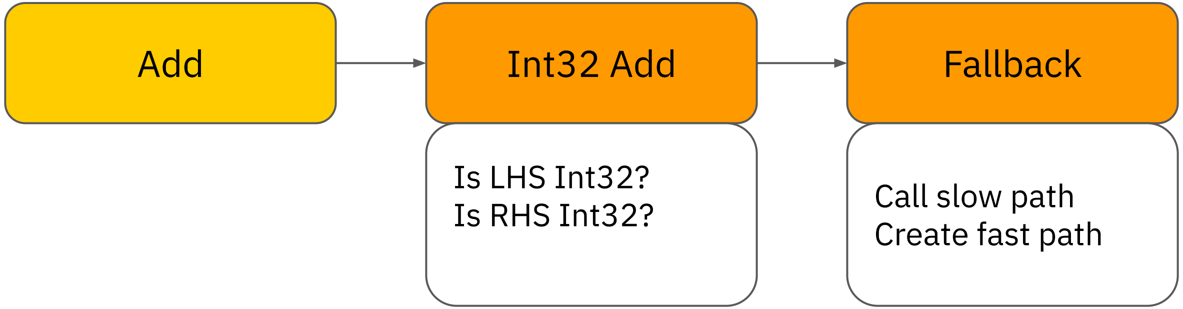
# Keeping your promises

Add

Fallback

Call slow path
Create fast path

# Keeping your promises



Add

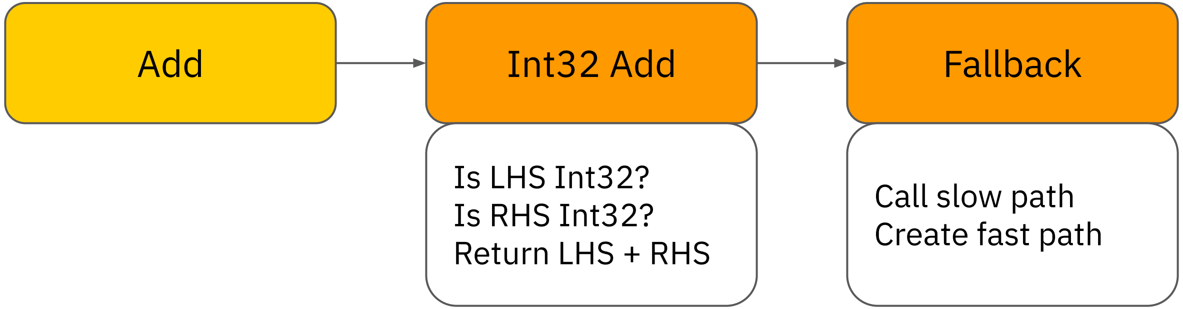
Int32 Add

Is LHS Int32?
Is RHS Int32?

Fallback

Call slow path
Create fast path

# Keeping your promises



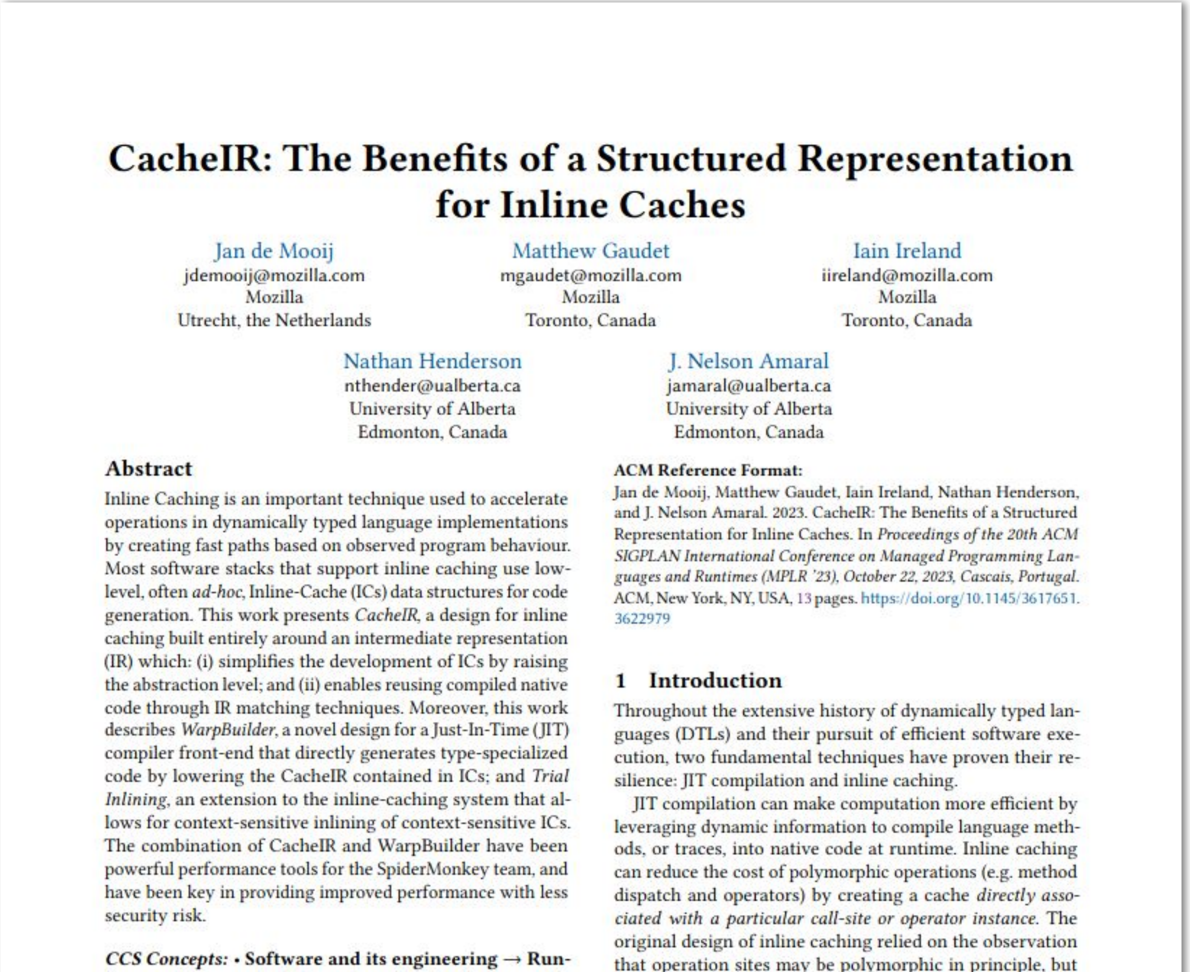
Add

Int32 Add

Is LHS Int32?
Is RHS Int32?
Return LHS + RHS

Fallback

Call slow path
Create fast path



# CacheIR: The Benefits of a Structured Representation for Inline Caches

Jan de Mooij
jdemooij@mozilla.com
Mozilla
Utrecht, the Netherlands

Matthew Gaudet
mgaudet@mozilla.com
Mozilla
Toronto, Canada

Iain Ireland
iireland@mozilla.com
Mozilla
Toronto, Canada

Nathan Henderson
nthender@ualberta.ca
University of Alberta
Edmonton, Canada

J. Nelson Amaral
jamaral@ualberta.ca
University of Alberta
Edmonton, Canada

## Abstract

Inline Caching is an important technique used to accelerate operations in dynamically typed language implementations by creating fast paths based on observed program behaviour. Most software stacks that support inline caching use low-level, often *ad-hoc*, Inline-Cache (ICs) data structures for code generation. This work presents *CacheIR*, a design for inline caching built entirely around an intermediate representation (IR) which: (i) simplifies the development of ICs by raising the abstraction level; and (ii) enables reusing compiled native code through IR matching techniques. Moreover, this work describes *WarpBuilder*, a novel design for a Just-In-Time (JIT) compiler front-end that directly generates type-specialized code by lowering the CacheIR contained in ICs; and *Trial Inlining*, an extension to the inline-caching system that allows for context-sensitive inlining of context-sensitive ICs. The combination of CacheIR and WarpBuilder have been powerful performance tools for the SpiderMonkey team, and have been key in providing improved performance with less security risk.

*CCS Concepts:* • **Software and its engineering → Run-**

**ACM Reference Format:**
Jan de Mooij, Matthew Gaudet, Iain Ireland, Nathan Henderson, and J. Nelson Amaral. 2023. CacheIR: The Benefits of a Structured Representation for Inline Caches. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3617651.3622979

## 1 Introduction

Throughout the extensive history of dynamically typed languages (DTLs) and their pursuit of efficient software execution, two fundamental techniques have proven their resilience: JIT compilation and inline caching.

JIT compilation can make computation more efficient by leveraging dynamic information to compile language methods, or traces, into native code at runtime. Inline caching can reduce the cost of polymorphic operations (e.g. method dispatch and operators) by creating a cache *directly associated with a particular call-site or operator instance*. The original design of inline caching relied on the observation that operation sites may be polymorphic in principle, but

# Baseline Compiler

# Baseline Compiler

**1** Per-opcode handlers directly generating machine code

# Baseline Compiler

1. Per-opcode handlers directly generating machine code
2. Dynamic behaviour via inline caches

# Baseline Compiler

1. Per-opcode handlers directly generating machine code
2. Dynamic behaviour via inline caches
3. Minimal optimization/overhead

# Baseline Compiler

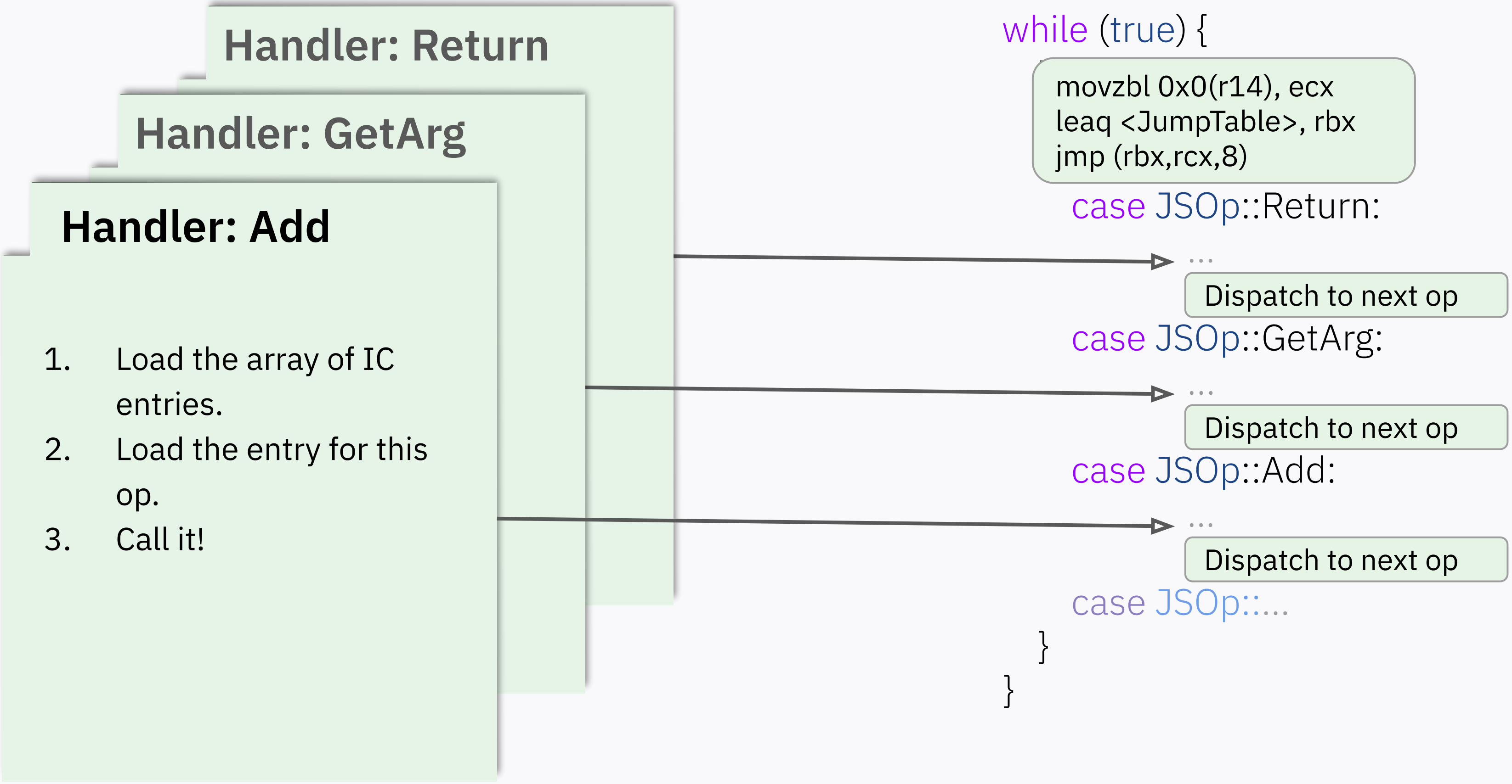
1. Per-opcode handlers directly generating machine code
2. Dynamic behaviour via inline caches
3. Minimal optimization/overhead
4. Faster to run than interpreter. Faster to compile than optimizing compiler.

# The website comes back...

NOT TOO BAD BUT...

1. Per-opcode handlers directly generating machine code
2. Dynamic behaviour via inline caches
3. Minimal optimization/overhead
4. Faster to run than interpreter. Faster to compile than optimizing compiler.

# Baseline Interpreter!



**Handler: Return**

**Handler: GetArg**

**Handler: Add**

1. Load the array of IC entries.
2. Load the entry for this op.
3. Call it!

```
while (true) {
  movzbl 0x0(r14), ecx
  leaq <JumpTable>, rbx
  jmp (rbx,rcx,8)
    case JSOp::Return:
      ...
      Dispatch to next op
    case JSOp::GetArg:
      ...
      Dispatch to next op
    case JSOp::Add:
      ...
      Dispatch to next op
    case JSOp::...
  }
}
```

# Baseline Interpreter

# Baseline Interpreter

**1** Lightning-fast startup

# Baseline Interpreter

1. Lightning-fast startup
2. Fast performance

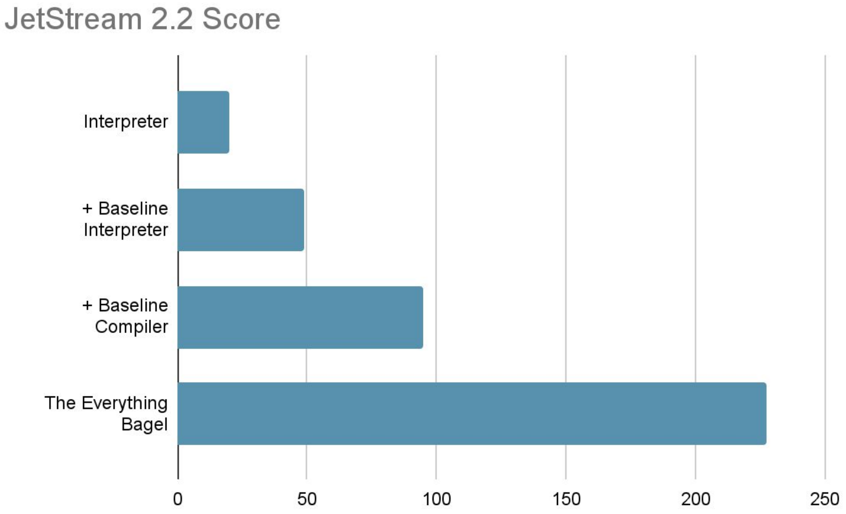
# Baseline Interpreter

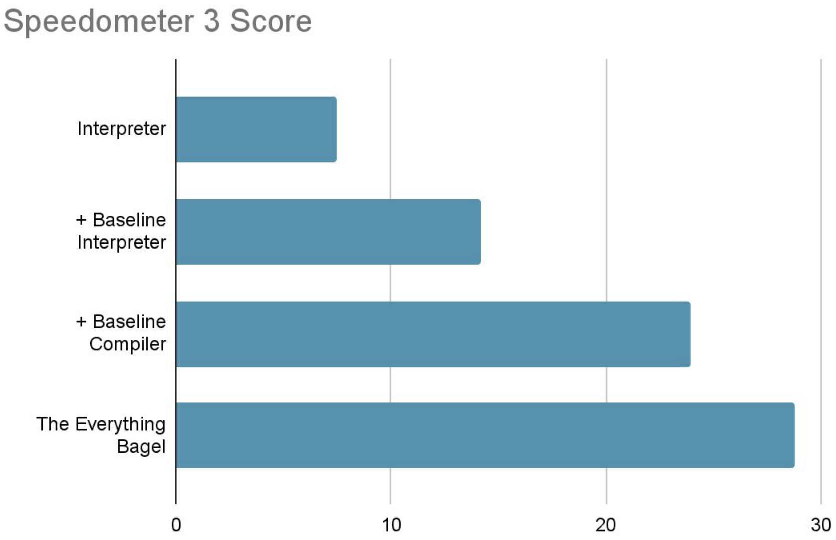
1. Lightning-fast startup
2. Fast performance
3. Code sharing

# Baseline Interpreter

1. Lightning-fast startup
2. Fast performance
3. Code sharing
4. Easy transition between tiers

# Is it any good?





Up to 8% page load improvements

# Is it any good?

# Thank you