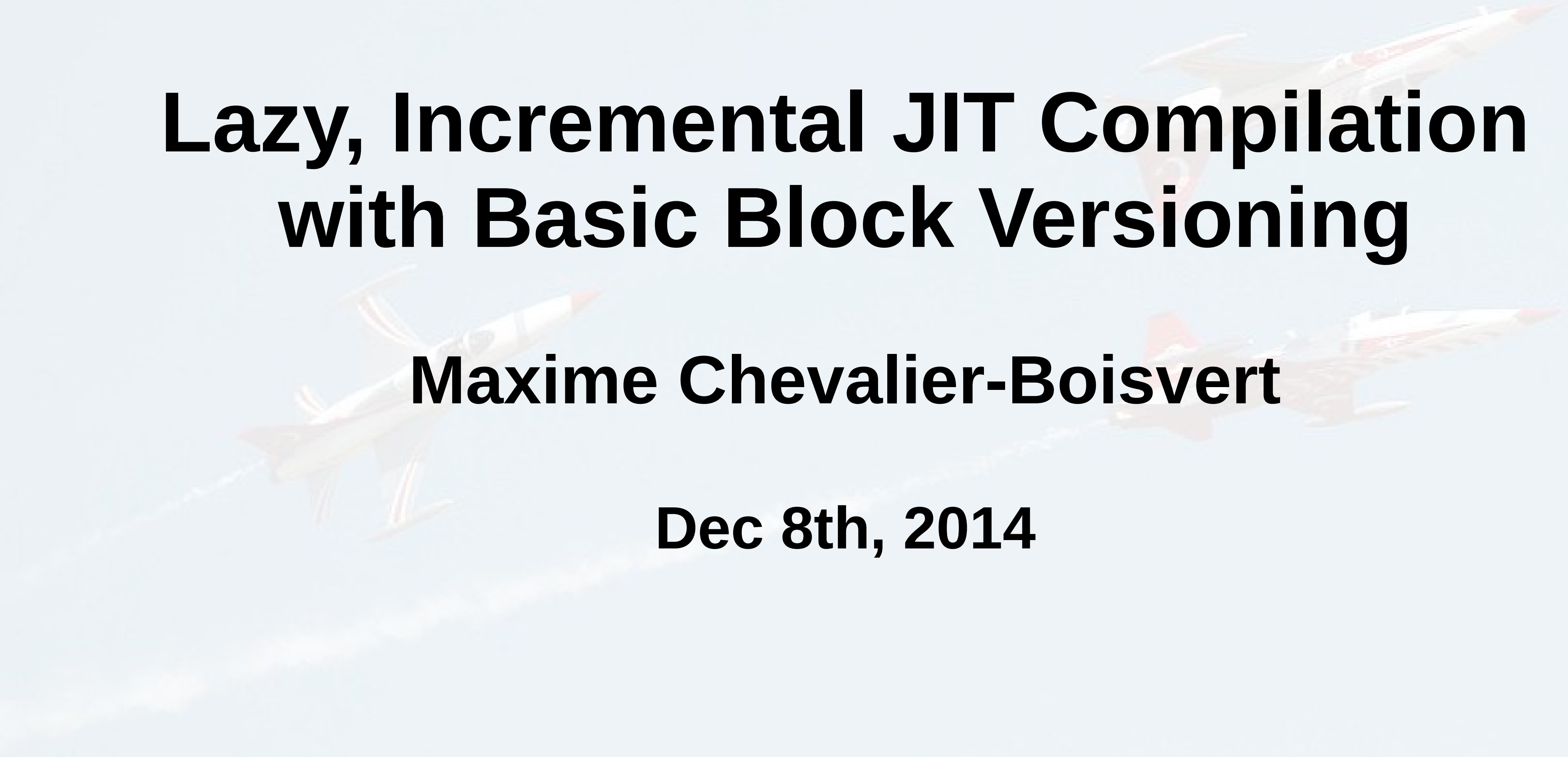


Lazy, Incremental JIT Compilation with Basic Block Versioning

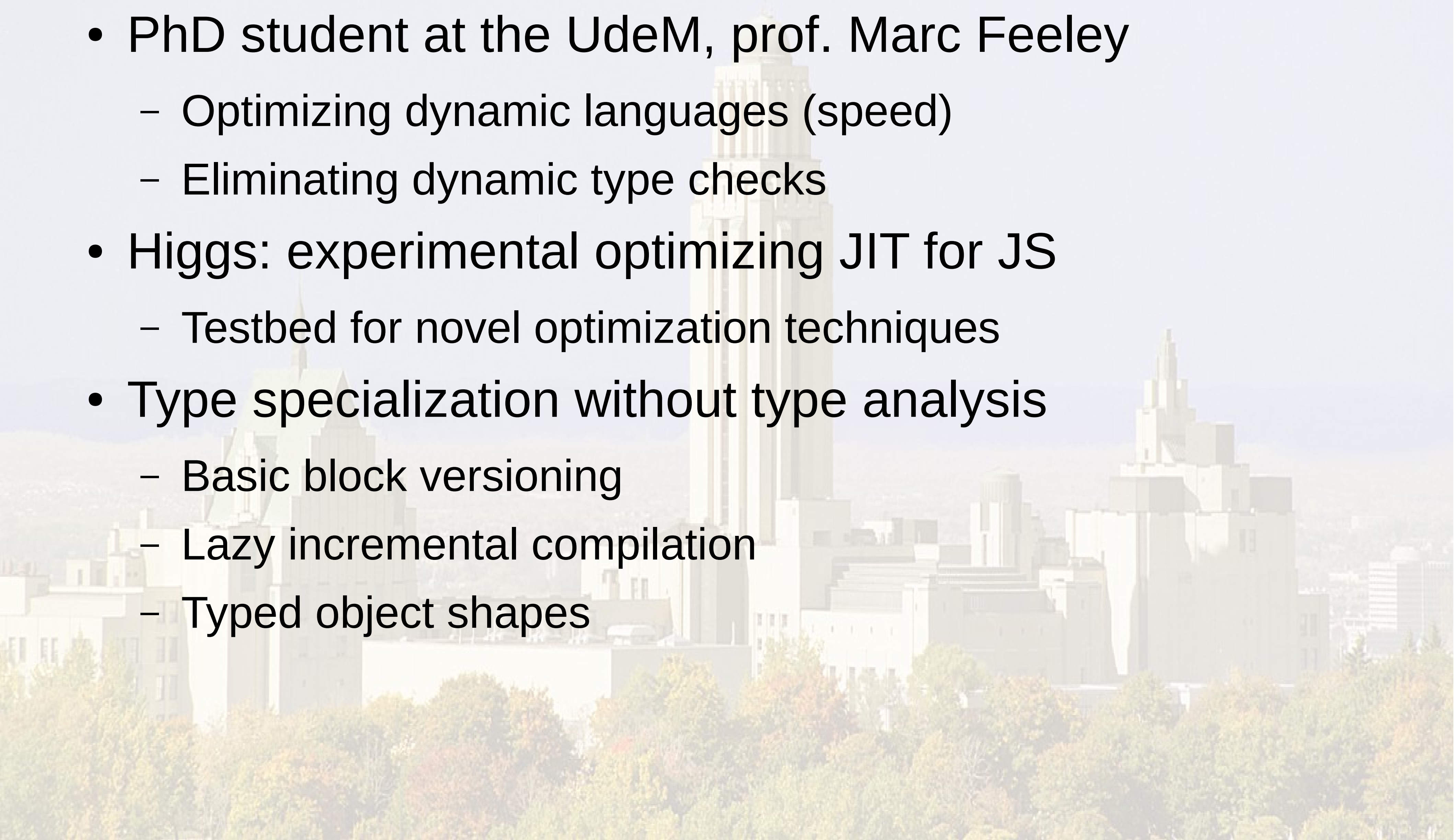
Maxime Chevalier-Boisvert

Dec 8th, 2014

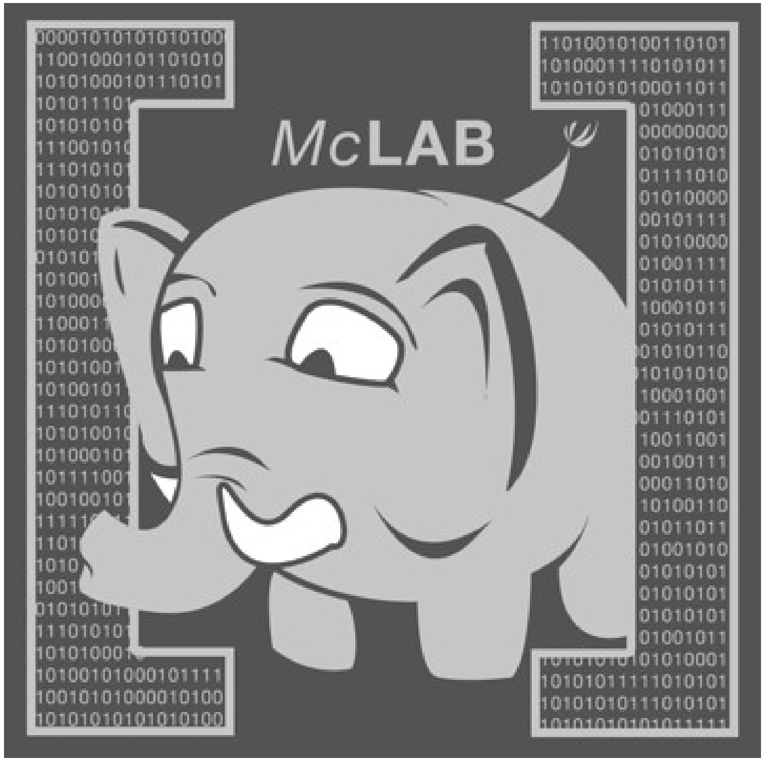
Université de Montréal

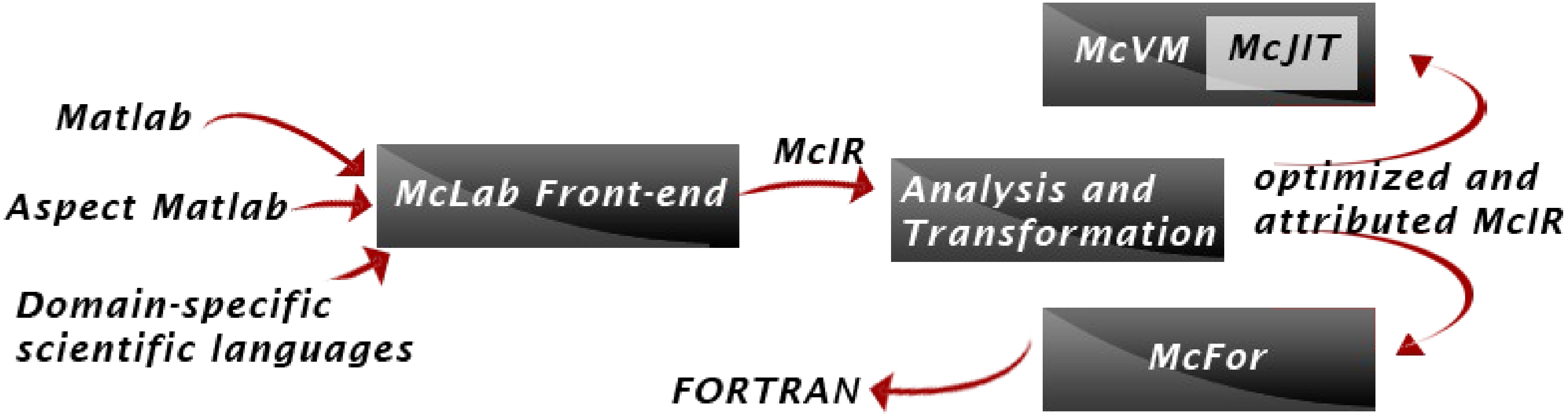
Introduction

- PhD student at the UdeM, prof. Marc Feeley
  - Optimizing dynamic languages (speed)
  - Eliminating dynamic type checks
- Higgs: experimental optimizing JIT for JS
  - Testbed for novel optimization techniques
- Type specialization without type analysis
  - Basic block versioning
  - Lazy incremental compilation
  - Typed object shapes

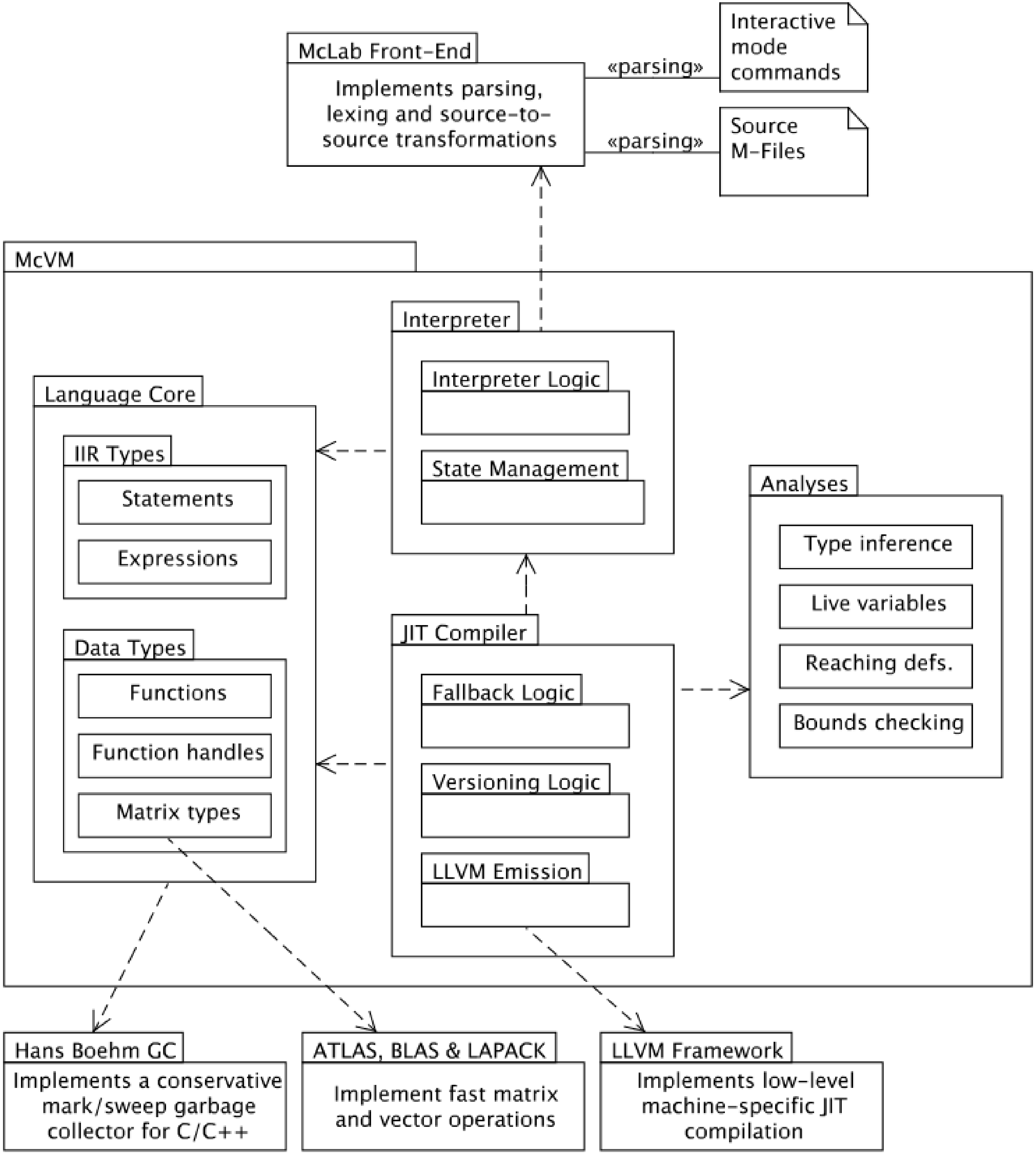


My M.Sc. Days


McLAB


Matlab
Aspect Matlab
Domain-specific scientific languages
McLab Front-end
McIR
Analysis and Transformation
McVM
McJIT
optimized and attributed McIR
McFor
FORTRAN

$$\begin{bmatrix}
N & \Sigma w & \Sigma x & \Sigma y & \Sigma x^2 & \Sigma wy & \Sigma xy & \Sigma wx & \Sigma w^2 & \Sigma y^2 & \Sigma x^3 & \Sigma w^3 & \Sigma y^3 \\
\Sigma w & \Sigma w^2 & \Sigma wx & \Sigma wy & \Sigma wx^2 & \Sigma w^2y & \Sigma wxy & \Sigma w^2x & \Sigma w^3 & \Sigma wy^2 & \Sigma wx^3 & \Sigma w^4 & \Sigma wy^3 \\
\Sigma x & \Sigma wx & \Sigma x^2 & \Sigma xy & \Sigma x^3 & \Sigma wxy & \Sigma x^2y & \Sigma wx^2 & \Sigma w^2x & \Sigma xy^2 & \Sigma x^4 & \Sigma xw^3 & \Sigma xy^3 \\
\Sigma y & \Sigma wy & \Sigma xy & \Sigma y^2 & \Sigma x^2y & \Sigma wy^2 & \Sigma xy^2 & \Sigma wxy & \Sigma w^2y & \Sigma y^3 & \Sigma x^3y & \Sigma w^3y & \Sigma y^4 \\
\Sigma x^2 & \Sigma wx^2 & \Sigma x^3 & \Sigma x^2y & \Sigma x^4 & \Sigma wx^2y & \Sigma x^3y & \Sigma wx^3 & \Sigma w^2x^2 & \Sigma x^2y^2 & \Sigma x^5 & \Sigma w^3x^2 & \Sigma x^2y^3 \\
\Sigma wy & \Sigma w^2y & \Sigma wxy & \Sigma wy^2 & \Sigma wx^2y & \Sigma w^2y^2 & \Sigma wxy^2 & \Sigma w^2xy & \Sigma w^3y & \Sigma wy^3 & \Sigma wyx^3 & \Sigma w^4y & \Sigma wy^4 \\
\Sigma xy & \Sigma wxy & \Sigma x^2y & \Sigma xy^2 & \Sigma x^3y & \Sigma wxy^2 & \Sigma x^2y^2 & \Sigma wx^2y & \Sigma w^2xy & \Sigma xy^3 & \Sigma x^4y & \Sigma w^3xy & \Sigma xy^4 \\
\Sigma wx & \Sigma w^2x & \Sigma wx^2 & \Sigma wxy & \Sigma wx^3 & \Sigma w^2xy & \Sigma wx^2y & \Sigma w^2x^2 & \Sigma w^3x & \Sigma wxy^2 & \Sigma wx^4 & \Sigma w^4x & \Sigma wxy^3 \\
\Sigma w^2 & \Sigma w^3 & \Sigma w^2x & \Sigma w^2y & \Sigma w^2x^2 & \Sigma w^3y & \Sigma w^2xy & \Sigma w^3x & \Sigma w^4 & \Sigma w^2y^2 & \Sigma w^2x^3 & \Sigma w^5 & \Sigma w^2y^3 \\
\Sigma y^2 & \Sigma wy^2 & \Sigma xy^2 & \Sigma y^3 & \Sigma x^2y^2 & \Sigma wy^3 & \Sigma xy^3 & \Sigma wxy^2 & \Sigma w^2y^2 & \Sigma y^4 & \Sigma x^3y^2 & \Sigma w^3y^2 & \Sigma y^5 \\
\Sigma x^3 & \Sigma wx^3 & \Sigma x^4 & \Sigma x^3y & \Sigma x^5 & \Sigma wx^3y & \Sigma x^4y & \Sigma wx^4 & \Sigma w^2x^3 & \Sigma x^3y^2 & \Sigma x^6 & \Sigma w^3x^3 & \Sigma x^3y^3 \\
\Sigma w^3 & \Sigma w^4 & \Sigma w^3x & \Sigma w^3y & \Sigma w^3x^2 & \Sigma w^4y & \Sigma w^3xy & \Sigma w^4x & \Sigma w^5 & \Sigma w^3y^2 & \Sigma w^3x^3 & \Sigma w^6 & \Sigma w^3y^3 \\
\Sigma y^3 & \Sigma wy^3 & \Sigma xy^3 & \Sigma y^4 & \Sigma x^2y^3 & \Sigma wy^4 & \Sigma xy^4 & \Sigma wxy^3 & \Sigma w^2y^3 & \Sigma y^5 & \Sigma x^3y^3 & \Sigma w^3y^3 & \Sigma y^6
\end{bmatrix}
\cdot
\begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \\ J \\ K \\ L \\ M \end{bmatrix}
=
\begin{bmatrix} \Sigma z \\ \Sigma wz \\ \Sigma xz \\ \Sigma yz \\ \Sigma x^2z \\ \Sigma wyz \\ \Sigma xyz \\ \Sigma wxz \\ \Sigma w^2z \\ \Sigma y^2z \\ \Sigma x^3z \\ \Sigma w^3z \\ \Sigma y^3z \end{bmatrix}$$


McLab Front-End
Implements parsing, lexing and source-to-source transformations
«parsing»
Interactive mode commands
«parsing»
Source M-Files
McVM
Interpreter
Interpreter Logic
State Management
Language Core
IIR Types
Statements
Expressions
Data Types
Functions
Function handles
Matrix types
Analyses
Type inference
Live variables
Reaching defs.
Bounds checking
JIT Compiler
Fallback Logic
Versioning Logic
LLVM Emission
Hans Boehm GC
Implements a conservative mark/sweep garbage collector for C/C++
ATLAS, BLAS & LAPACK
Implement fast matrix and vector operations
LLVM Framework
Implements low-level machine-specific JIT compilation

Optimizing Matlab through Just-In-Time Specialization *

Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge

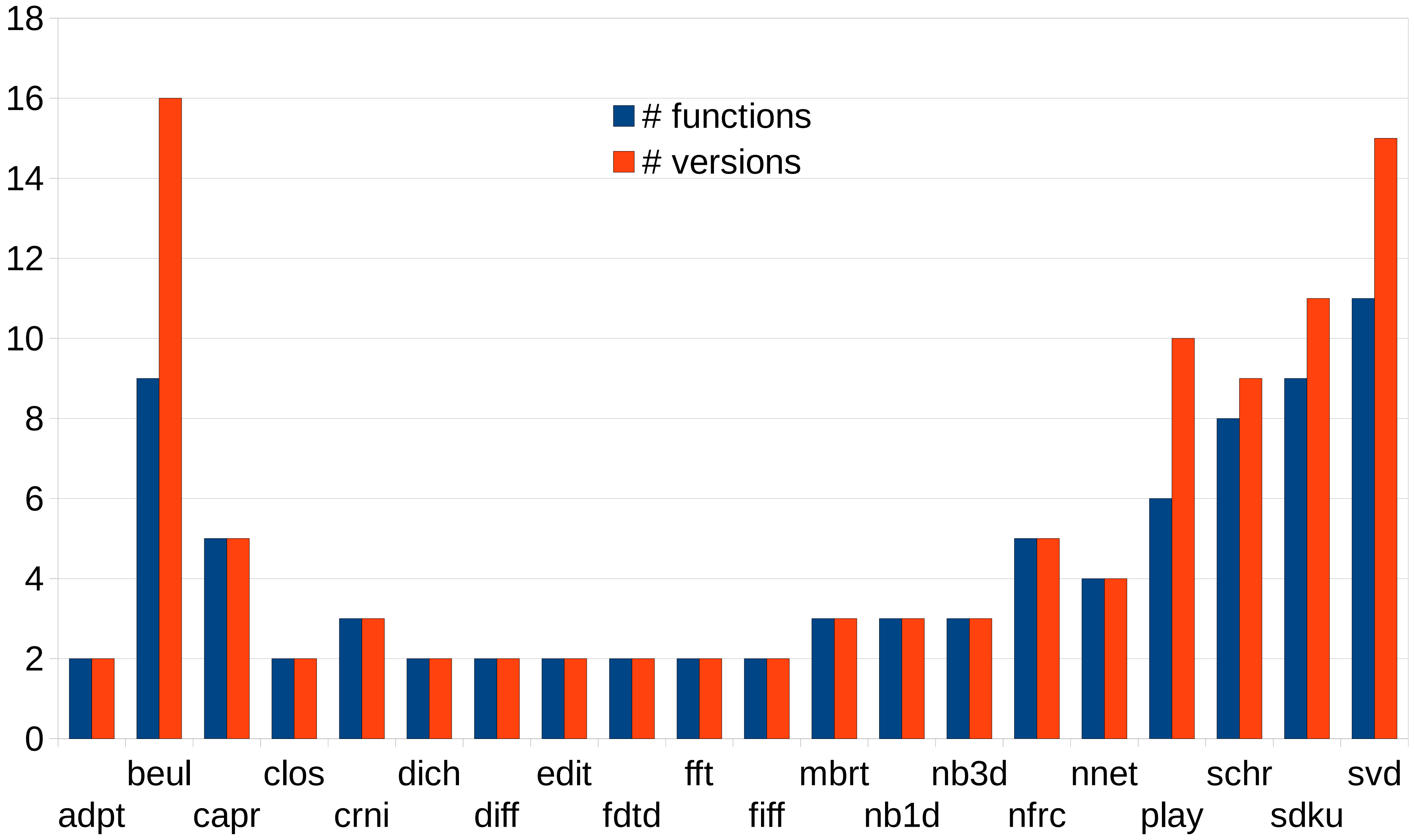
School of Computer Science, McGill University, Montreal, QC, Canada
{mcheva,hendren,clump}@cs.mcgill.ca

Abstract. Scientists are increasingly using dynamic programming languages like Matlab for prototyping and implementation. Effectively compiling Matlab raises many challenges due to the dynamic and complex nature of Matlab types. This paper presents a new JIT-based approach which specializes and optimizes functions on-the-fly based on the current types of function arguments.

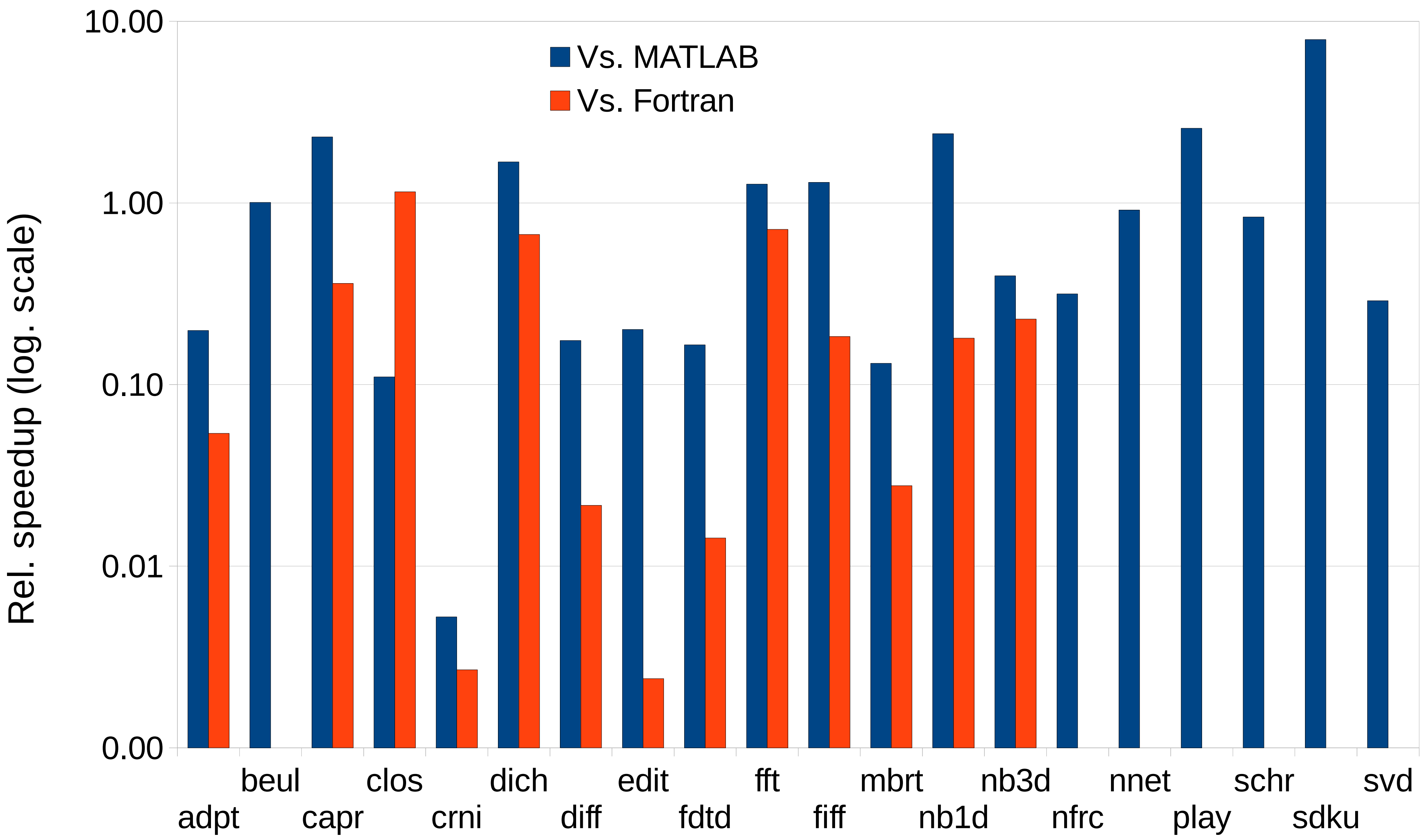
A key component of our approach is a new type inference algorithm which uses the run-time argument types to infer further type and shape information, which in turn provides new optimization opportunities. These techniques are implemented in McVM, our open implementation of a Matlab virtual machine. As this is the first paper reporting on McVM, a brief introduction to McVM is also given.

We have experimented with our implementation and compared it to several other Matlab implementations, including the Mathworks proprietary system, McVM without specialization, the Octave open-source interpreter and the McFor static compiler. The results are quite encouraging and indicate that specialization is an effective optimization—McVM with specialization outperforms Octave by a large margin and also sometimes outperforms the Mathworks implementation.

Functions to Versions



McVM vs MATLAB & Fortran



JavaScript

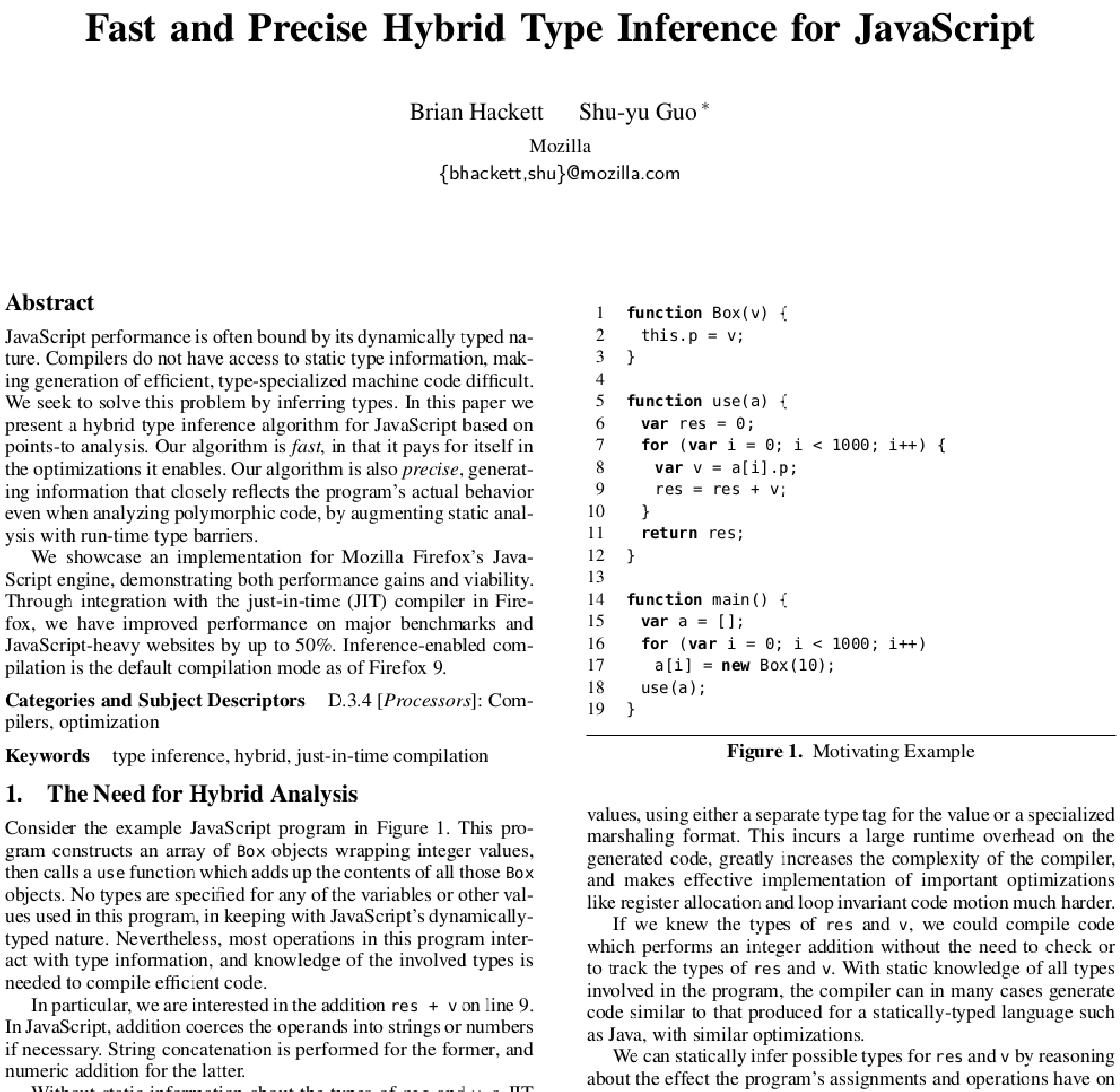
```
function add1(n)
{
    return n + 1;
}
```

```
add1(2)         ⟾ 3

add1('hello')   ⟾ 'hello1'

add1(true)      ⟾ 2
add1(null)      ⟾ 1
add1(undefined) ⟾ NaN

add1({ toString: function() { return '3'; } }) ⟾ '31'

add1({ toString: function() { return 3; } })   ⟾ 4
```

```
// JS less-than comparison operator (x < y)
function $rt_lt(x, y)
{
    // If x is integer
    if ($ir_is_int32(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_i32(x, y);

        if ($ir_is_float(y))
            return $ir_lt_f64($ir_i32_to_f64(x), y);
    }

    // If x is float
    if ($ir_is_float(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_f64(x, $ir_i32_to_f64(y));

        if ($ir_is_float(y))
            return $ir_lt_f64(x, y);
    }


    …
}
```

```
// JS less-than comparison operator (x < y)
function $rt_lt(x, y)
{
    // If x is integer
    if ($ir_is_int32(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_i32(x, y);

        if ($ir_is_float(y))
            return $ir_lt_f64($ir_i32_to_f64(x), y);
    }

    // If x is float
    if ($ir_is_float(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_f64(x, $ir_i32_to_f64(y));

        if ($ir_is_float(y))
            return $ir_lt_f64(x, y);
    }

    …
}
```

```
// JS less-than comparison operator (x < y)
function $rt_lt(x, y)
{
    // If x is integer
    if ($ir_is_int32(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_i32(x, y);

        if ($ir_is_float(y))
            return $ir_lt_f64($ir_i32_to_f64(x), y);
    }

    // If x is float
    if ($ir_is_float(x))
    {
        if ($ir_is_int32(y))
            return $ir_lt_f64(x, $ir_i32_to_f64(y));

        if ($ir_is_float(y))
            return $ir_lt_f64(x, y);
    }

    …
}
```



Fast and Precise Hybrid Type Inference for JavaScript

Brian Hackett Shu-yu Guo *
Mozilla
{bhackett,shu}@mozilla.com

Abstract

JavaScript performance is often bound by its dynamically typed nature. Compilers do not have access to static type information, making generation of efficient, type-specialized machine code difficult. We seek to solve this problem by inferring types. In this paper we present a hybrid type inference algorithm for JavaScript based on points-to analysis. Our algorithm is *fast*, in that it pays for itself in the optimizations it enables. Our algorithm is also *precise*, generating information that closely reflects the program's actual behavior even when analyzing polymorphic code, by augmenting static analysis with run-time type barriers.

We showcase an implementation for Mozilla Firefox's JavaScript engine, demonstrating both performance gains and viability. Through integration with the just-in-time (JIT) compiler in Firefox, we have improved performance on major benchmarks and JavaScript-heavy websites by up to 50%. Inference-enabled compilation is the default compilation mode as of Firefox 9.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, optimization

Keywords type inference, hybrid, just-in-time compilation

1. The Need for Hybrid Analysis

Consider the example JavaScript program in Figure 1. This program constructs an array of Box objects wrapping integer values, then calls a use function which adds up the contents of all those Box objects. No types are specified for any of the variables or other values used in this program, in keeping with JavaScript's dynamically-typed nature. Nevertheless, most operations in this program interact with type information, and knowledge of the involved types is needed to compile efficient code.

In particular, we are interested in the addition res + v on line 9. In JavaScript, addition coerces the operands into strings or numbers if necessary. String concatenation is performed for the former, and numeric addition for the latter.

Without static information about the types of res and v, a JIT

```
1  function Box(v) {
2    this.p = v;
3  }
4
5  function use(a) {
6    var res = 0;
7    for (var i = 0; i < 1000; i++) {
8      var v = a[i].p;
9      res = res + v;
10   }
11   return res;
12 }
13
14 function main() {
15   var a = [];
16   for (var i = 0; i < 1000; i++)
17     a[i] = new Box(10);
18   use(a);
19 }
```

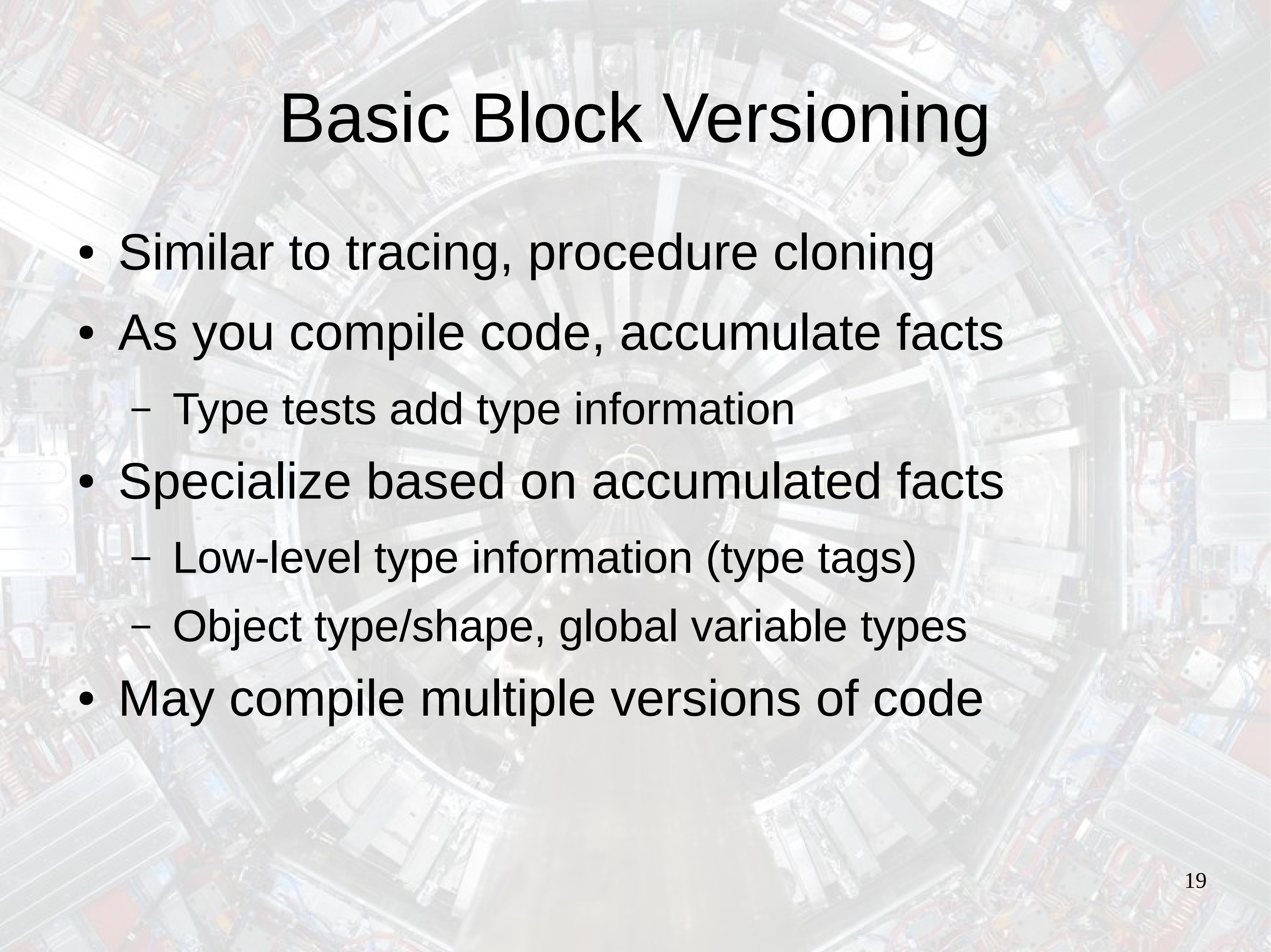
Figure 1. Motivating Example

values, using either a separate type tag for the value or a specialized marshaling format. This incurs a large runtime overhead on the generated code, greatly increases the complexity of the compiler, and makes effective implementation of important optimizations like register allocation and loop invariant code motion much harder.

If we knew the types of res and v, we could compile code which performs an integer addition without the need to check or to track the types of res and v. With static knowledge of all types involved in the program, the compiler can in many cases generate code similar to that produced for a statically-typed language such as Java, with similar optimizations.

We can statically infer possible types for res and v by reasoning about the effect the program's assignments and operations have on

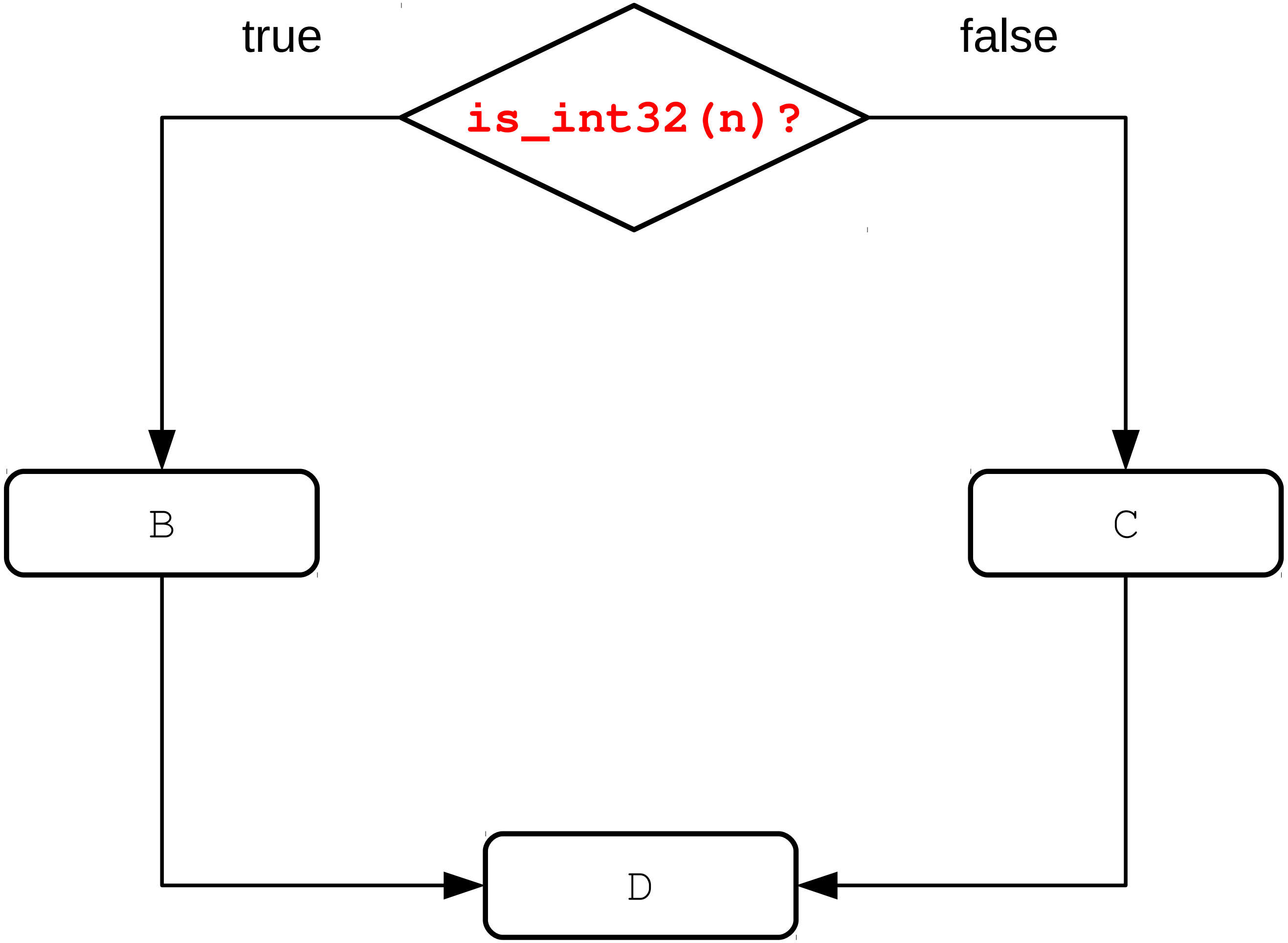
Basic Block Versioning

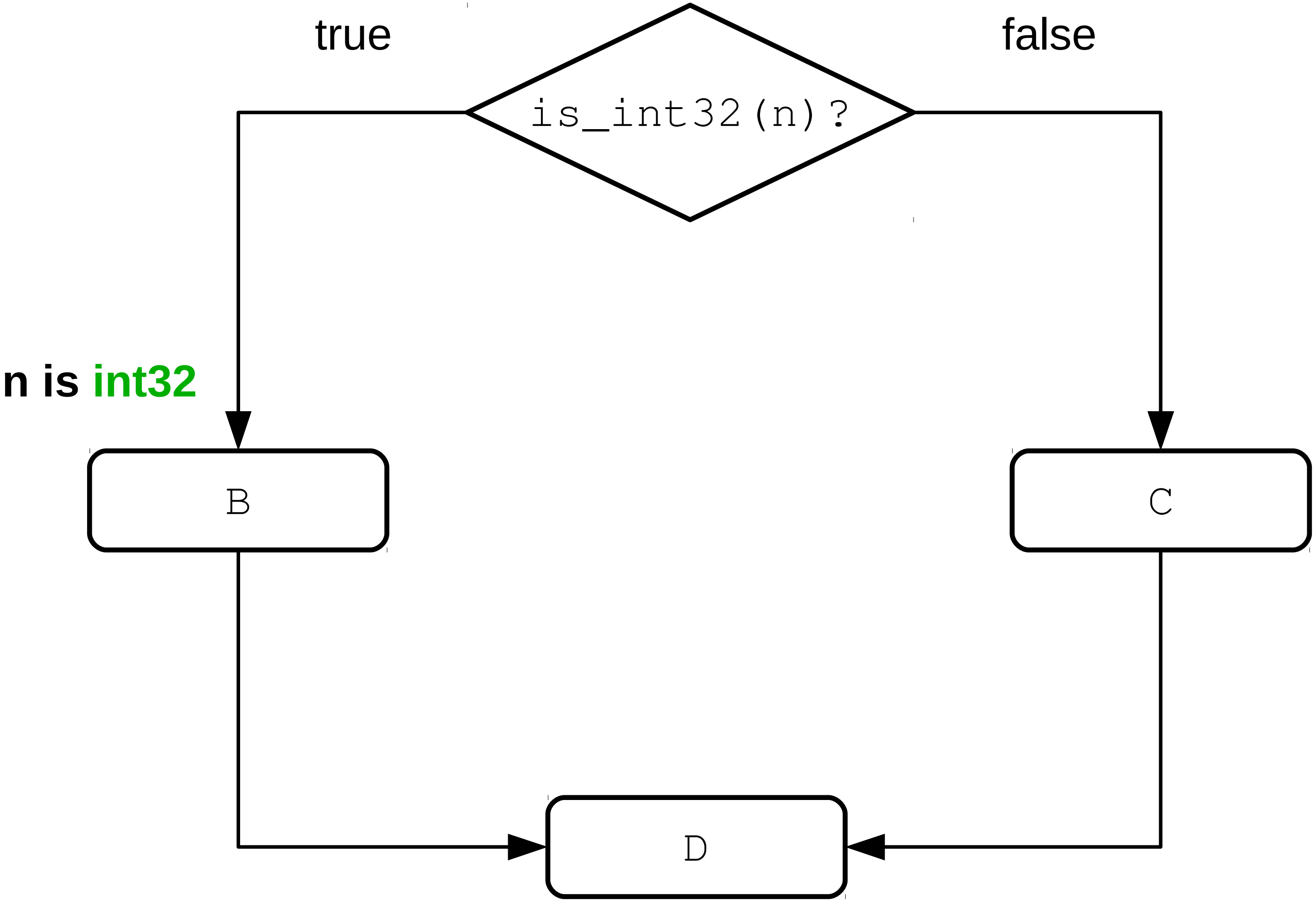


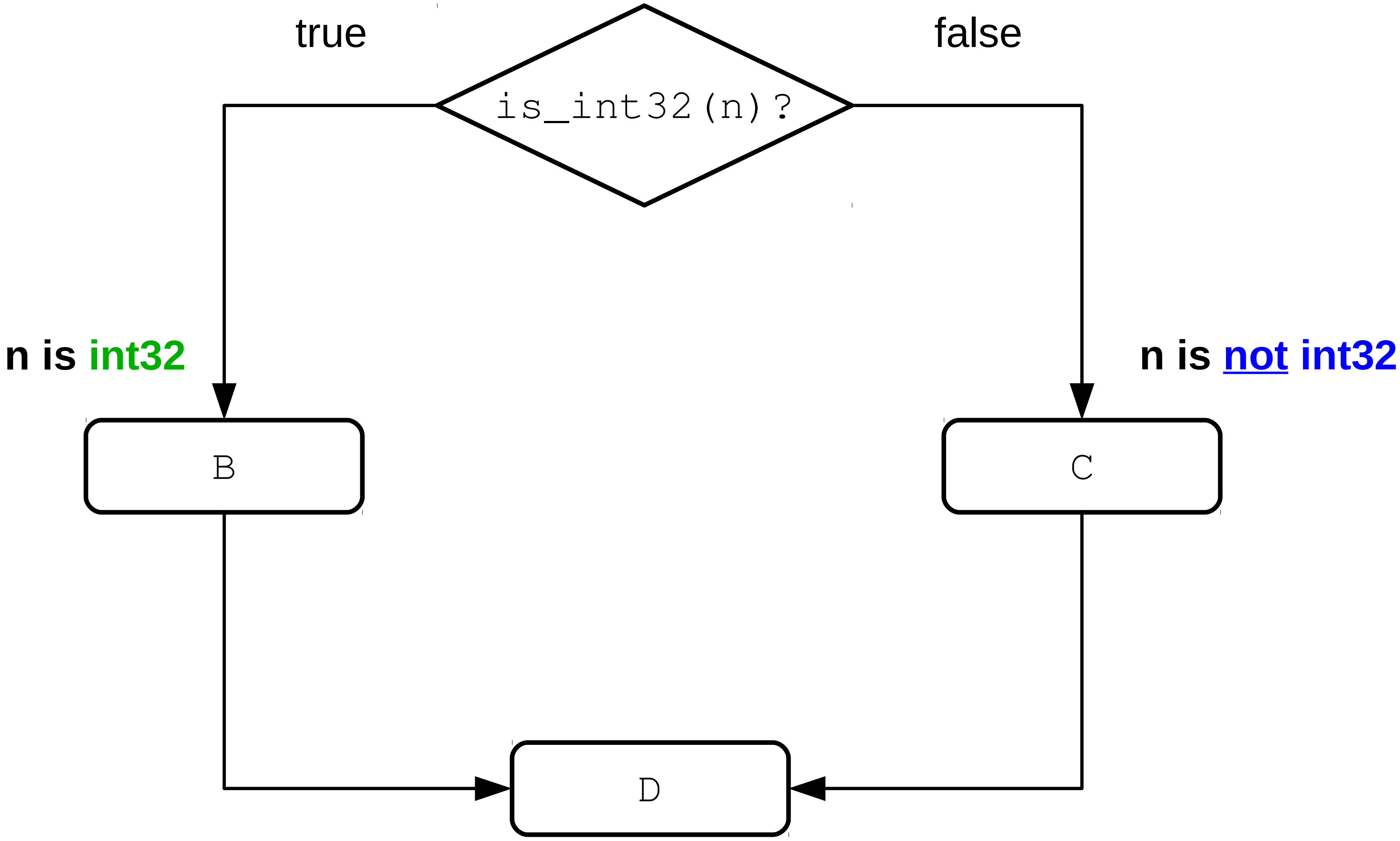
Basic Block Versioning

- Similar to tracing, procedure cloning
- As you compile code, accumulate facts
  - Type tests add type information
- Specialize based on accumulated facts
  - Low-level type information (type tags)
  - Object type/shape, global variable types
- May compile multiple versions of code

```
if (is_int32(n)) // A
{
    // B
    ...
}
else
{
    // C
    ...
}

// D
...
```


true
is_int32(n)?
false
B
C
D


true
is_int32(n)?
false
n is int32
B
C
D



true

false

`is_int32(n)?`

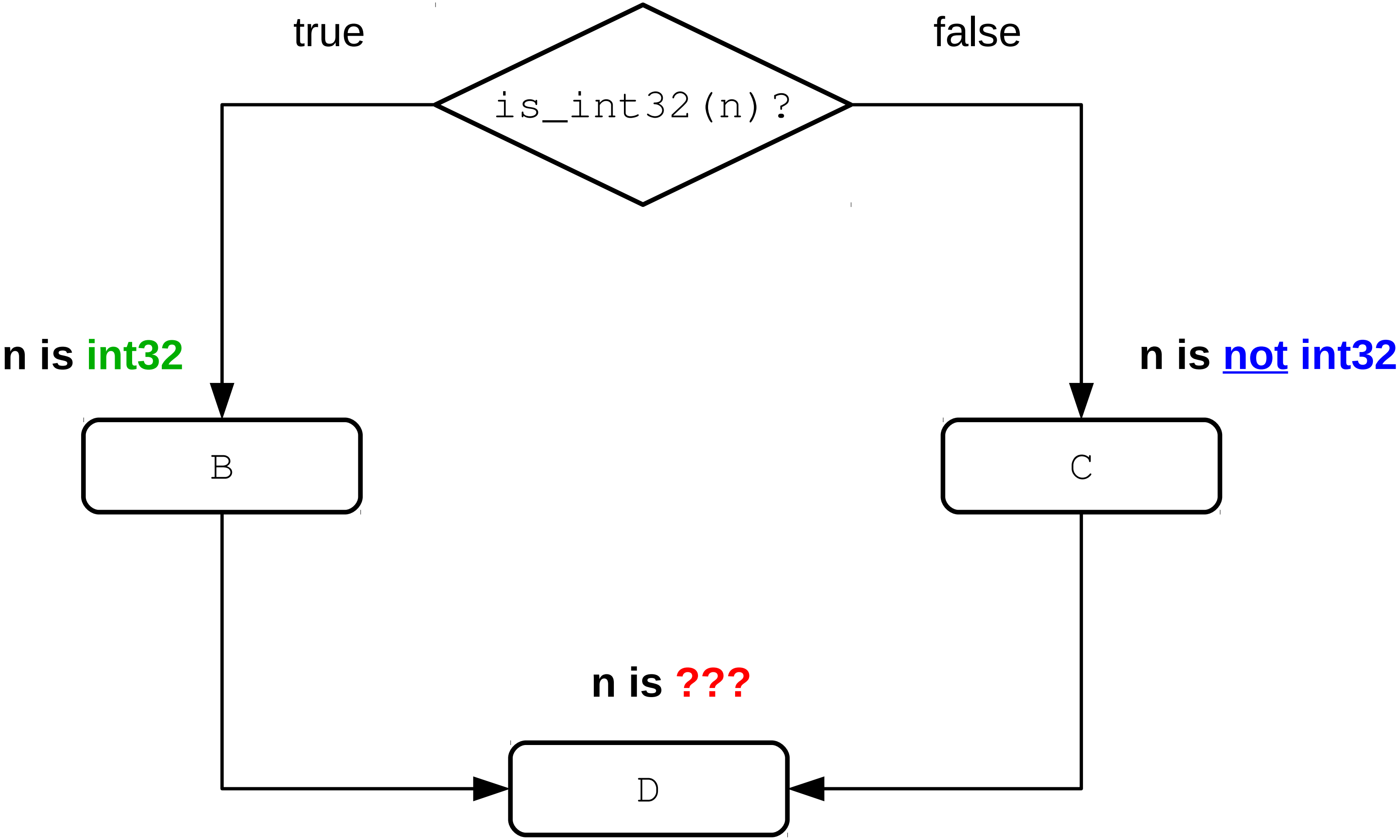
n is int32

n is not int32

B

C

D



true

`is_int32(n)?`

false

n is int32

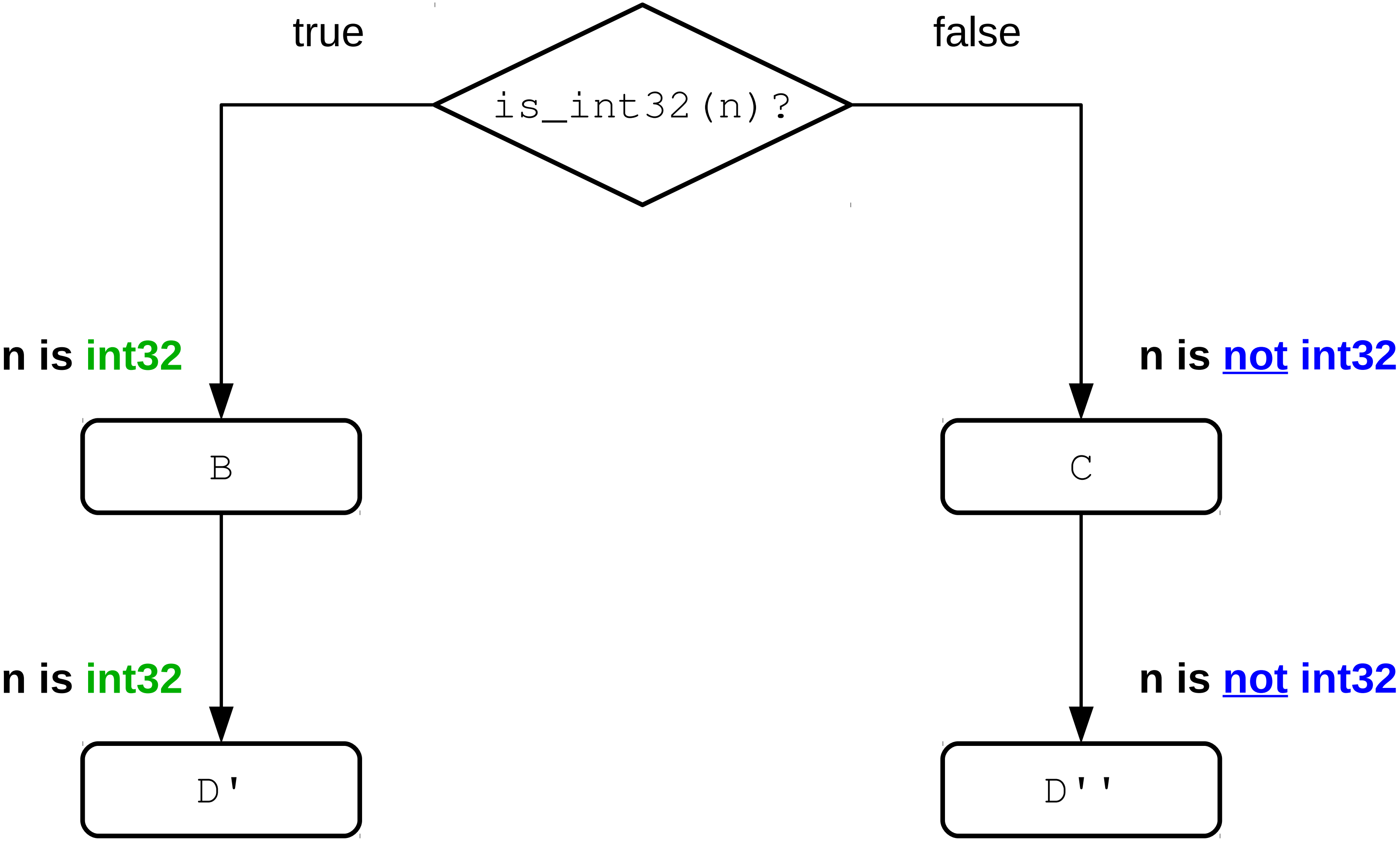
B

n is <u>not</u> int32

C

n is ???

D



true

is_int32(n)?

false

n is int32

n is not int32

B

C

n is int32

n is not int32

D'

D''

```
var v = 4294967296;
for (var i = 0; i < 600000; i++)
    v = v & i;

// From the SunSpider bitwise-and benchmark
```

```
var v = 4294967296;
for (var i = 0; less_than(i,600000); i = add(i,1))
    v = bitwise_and(v,i);
```

```
var v = 4294967296;
for (var i = 0; less_than(i,600000); i = add(i,1))
    v = bitwise_and(v,i);

function bitwise_and(x,y) {
   if (is_int32(x) && is_int32(y))
      return bitwise_and_int32(x,y); // Fast path

   return bitwise_and_int32(toInt32(x), toInt32(y));
}
```

```
var v = 4294967296;
for (var i = 0; less_than(i,600000); i = add(i,1))
    v = bitwise_and(v,i);

function bitwise_and(x,y) {
   if (is_int32(x) && is_int32(y))
      return bitwise_and_int32(x,y); // Fast path

   return bitwise_and_int32(toInt32(x), toInt32(y));
}

function add(x,y) {
   if (is_int32(x) && is_int32(y))
   {
      var r = add_int32(x,y); // Fast path
      if (cpu_overflow_flag)
         r = add_double(toDouble(x), toDouble(y));
      return r;
   }

   return add_general(x,y);
}
```

```
var v = 4294967296;
for (var i = 0; less_than(i,600000); i = add(i,1))
    v = bitwise_and(v,i);

function bitwise_and(x,y) {
   if (is_int32(x) && is_int32(y))
      return bitwise_and_int32(x,y); // Fast path

   return bitwise_and_int32(toInt32(x), toInt32(y));
}

function add(x,y) {
   if (is_int32(x) && is_int32(y))
   {
      var r = add_int32(x,y); // Fast path
      if (cpu_overflow_flag)
         r = add_double(toDouble(x), toDouble(y));
      return r;
   }

   return add_general(x,y);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (is_int32(i) && is_int32(600000))
      if (greater_eq_int32(i, 600000)) break;
   else
      if (greater_eq_general(i, 600000) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and_int32(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i) && is_int32(1)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (is_int32(i) && is_int32(600000))
      if (greater_eq_int32(i, 600000)) break;
   else
      if (greater_eq_general(i, 600000) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and_int32(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i) && is_int32(1)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
    // if (i >= 600000) break;
    if (is_int32(i))
        if (greater_eq_int32(i, 600000)) break;
    else
        if (greater_eq_general(i, 600000) break;

    // v = v & i
    if (is_int32(v) && is_int32(i))
        v = bitwise_and_int32(v,i);
    else
        v = bitwise_and_int32(toInt32(v), toInt32(i));

    // i = i + 1
    if (is_int32(i)) {
        i = add_int32(i,1);
        if (cpu_overflow_flag)
            i = add_double(toDouble(i), toDouble(1));
    }
    else
        i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0; // when we enter the loop, i is int32
for (;;) {
   // if (i >= 600000) break;
   if (is_int32(i))
      if (greater_eq_int32(i, 600000)) break;
   else
      if (greater_eq_general(i, 600000) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and_int32(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) { // assume i is int32 when entering loop
   // if (i >= 600000) break;
   if (is_int32(i))
      if (greater_eq_int32(i, 600000)) break;
   else
      if (greater_eq_general(i, 600000) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and_int32(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) { // assume i is int32 when entering loop
   // if (i >= 600000) break;
   if (is_int32(i))
      if (greater_eq_int32(i, 600000)) break;
   else
      if (greater_eq_general(i, 600000) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and_int32(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and_int32(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and_int32(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) { // assume both i and v are int32
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and_int32(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   if (is_int32(i)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   if (is_int32(i)) {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   v = bitwise_and_int32(v,i); // v remains int32 after this

   // i = i + 1
   i = add_int32(i,1);
   if (cpu_overflow_flag)
      i = add_double(toDouble(i), toDouble(1));
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   i = add_int32(i,1); // the add could overflow!
   if (cpu_overflow_flag)
      i = add_double(toDouble(i), toDouble(1));
      // i becomes a double
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   i = add_int32(i,1);
   if (cpu_overflow_flag) {
      i = add_double(toDouble(i), toDouble(1));

      // Jump to a loop version where i is a double
      NEW_LOOP_VERSION = gen_new_version({'i':'double'});
      goto NEW_LOOP_VERSION;
   }

   // If we make it here, i is still int32
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   i = add_int32(i,1);
   if (cpu_overflow_flag) {
      i = add_double(toDouble(i), toDouble(1));

      // Jump to a loop version where i is a double
      NEW_LOOP_VERSION = gen_new_version({'i':'double'});
      goto NEW_LOOP_VERSION;
   }

   // If we make it here, i is still int32
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break; // i < INT32_MAX

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   i = add_int32(i,1);
   if (cpu_overflow_flag) {
      i = add_double(toDouble(i), toDouble(1));

      // Jump to a loop version where i is a double
      NEW_LOOP_VERSION = gen_new_version({'i':'double'});
      goto NEW_LOOP_VERSION;
   }

   // If we make it here, i is still int32
}
```

```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break; // i < INT32_MAX

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   i = add_int32(i,1); // i + 1 <= INT32_MAX
   if (cpu_overflow_flag) {
      i = add_double(toDouble(i), toDouble(1));

      // Jump to a loop version where i is a double
      NEW_LOOP_VERSION = gen_new_version({'i':'double'});
      goto NEW_LOOP_VERSION;
   }

   // If we make it here, i is still int32
}
```

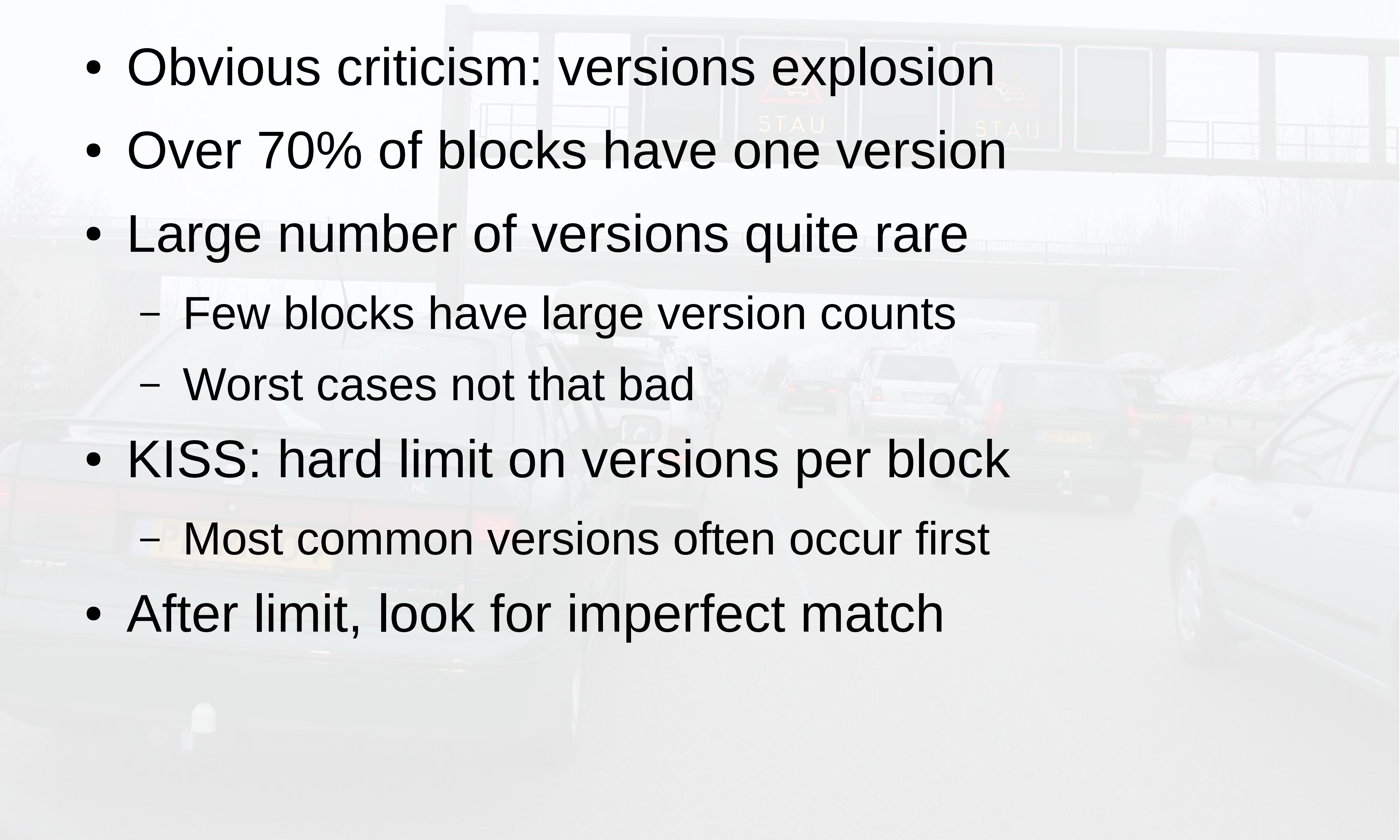
```
var v = 4294967296;
var i = 0;
for (;;) {
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break; // i < INT32_MAX

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   i = add_int32(i,1); // i + 1 <= INT32_MAX

   // If we make it here, i is still int32
}
```

A “multi-world” Approach

- Traditional type analysis
  - Fixed-point on types
  - Types found must agree with all inputs
  - Pessimistic, conservative answer
- Basic block versioning
  - Multiple solutions possible for each block
  - Don't necessarily have to sacrifice
  - Fixed-point on versioning of blocks

There is no Explosion

- Obvious criticism: versions explosion
- Over 70% of blocks have one version
- Large number of versions quite rare
  - Few blocks have large version counts
  - Worst cases not that bad
- KISS: hard limit on versions per block
  - Most common versions often occur first
- After limit, look for imperfect match

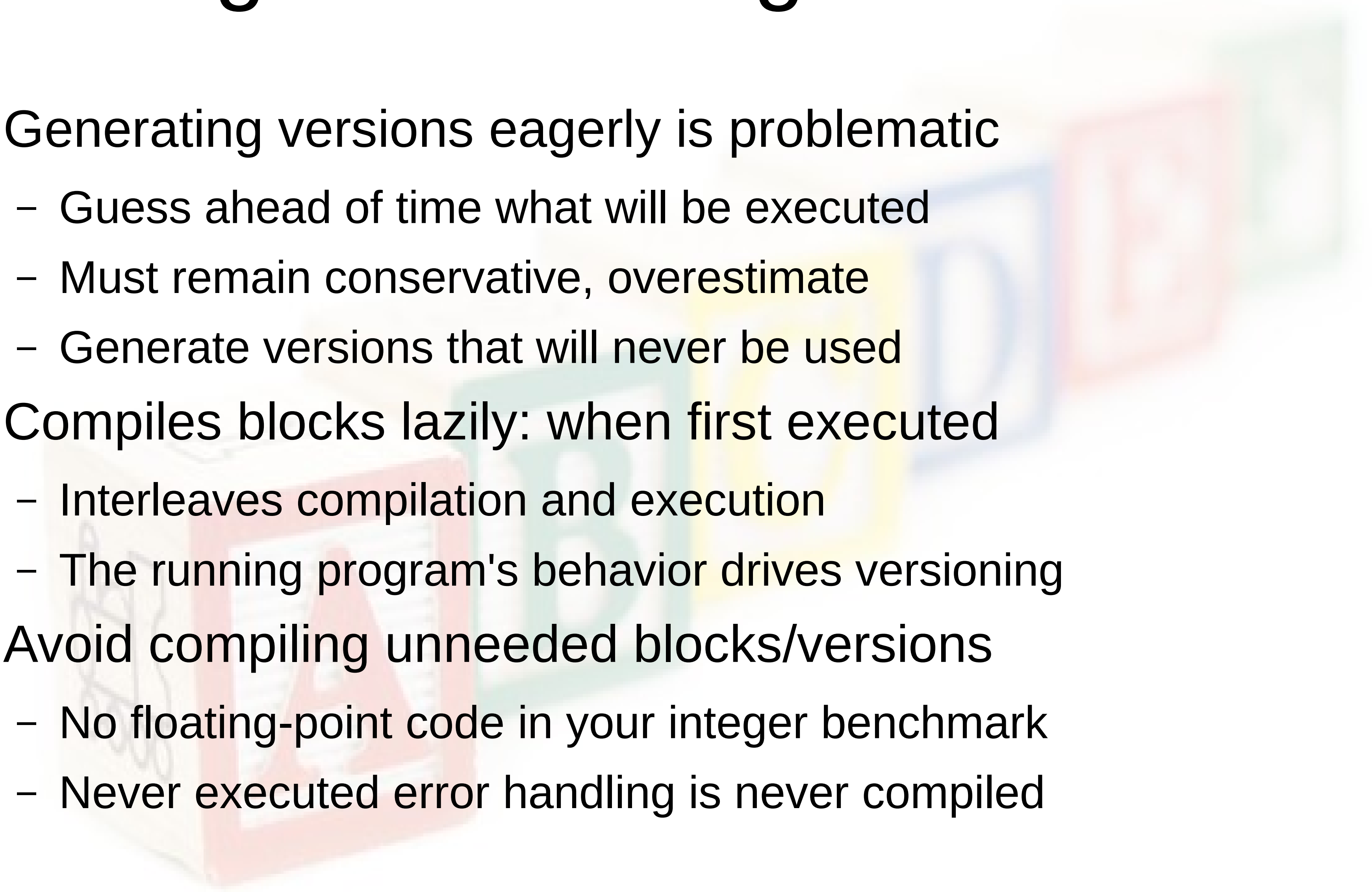


Unseen Benefits

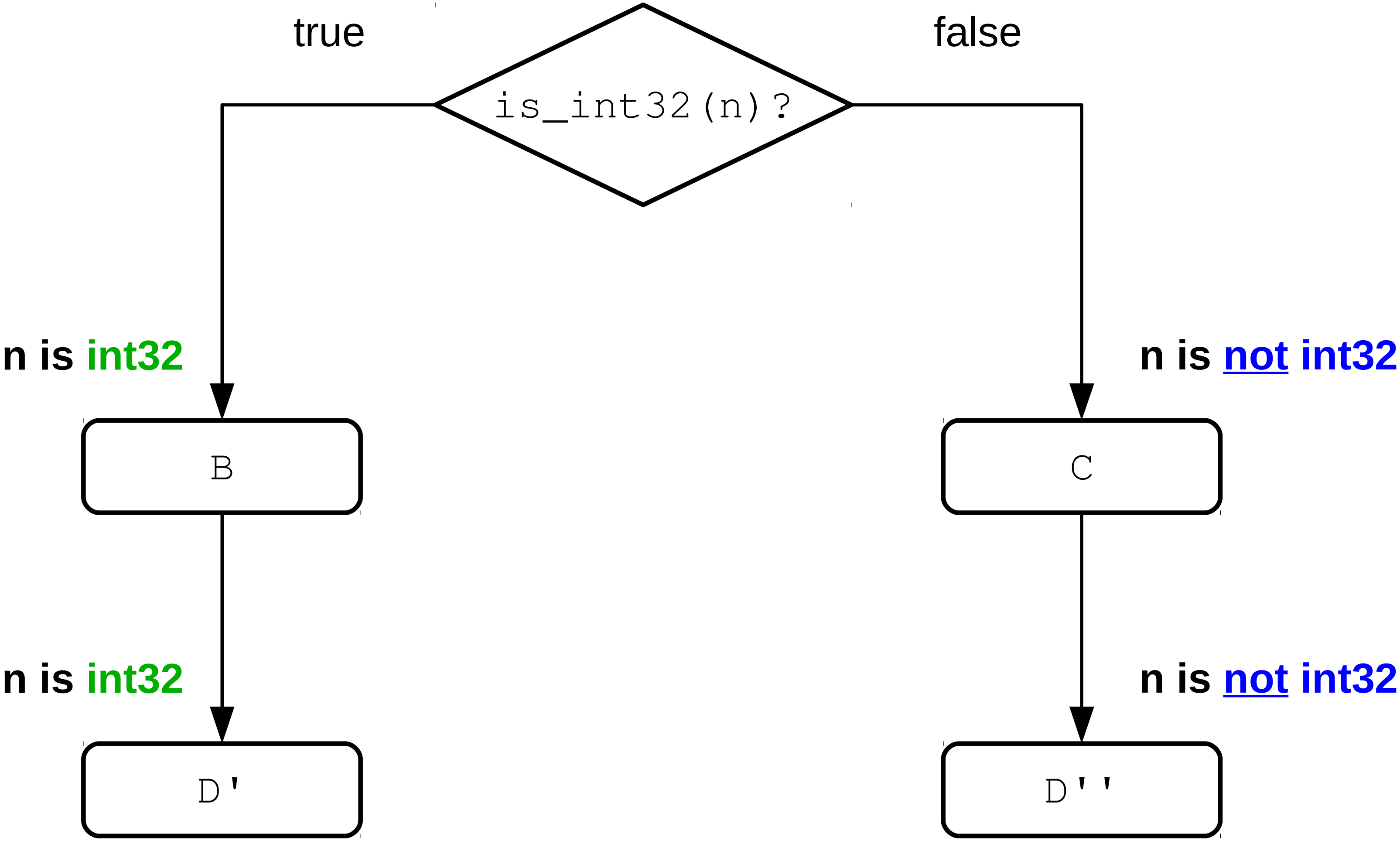
- Hoists redundant type tests out of loop bodies
- More powerful than traditional type analysis
  - Multiple separate optimized code paths
  - Works on code poorly amenable to analysis
  - Gives answers where a type analysis can't
- No iterative fixed-point, very fast
- Many interesting extensions possible

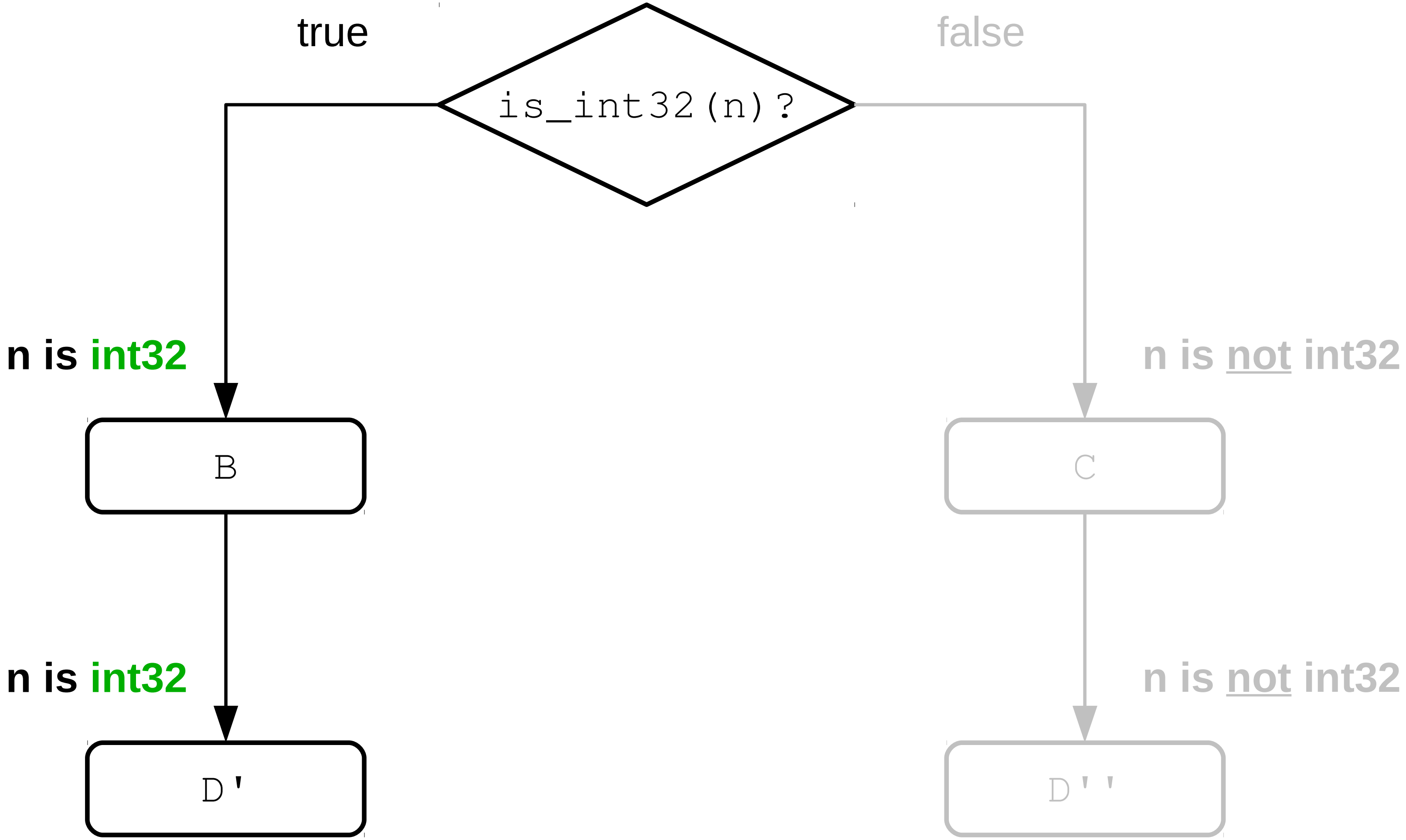
Lazy Incremental Compilation

Eager Versioning is No Good

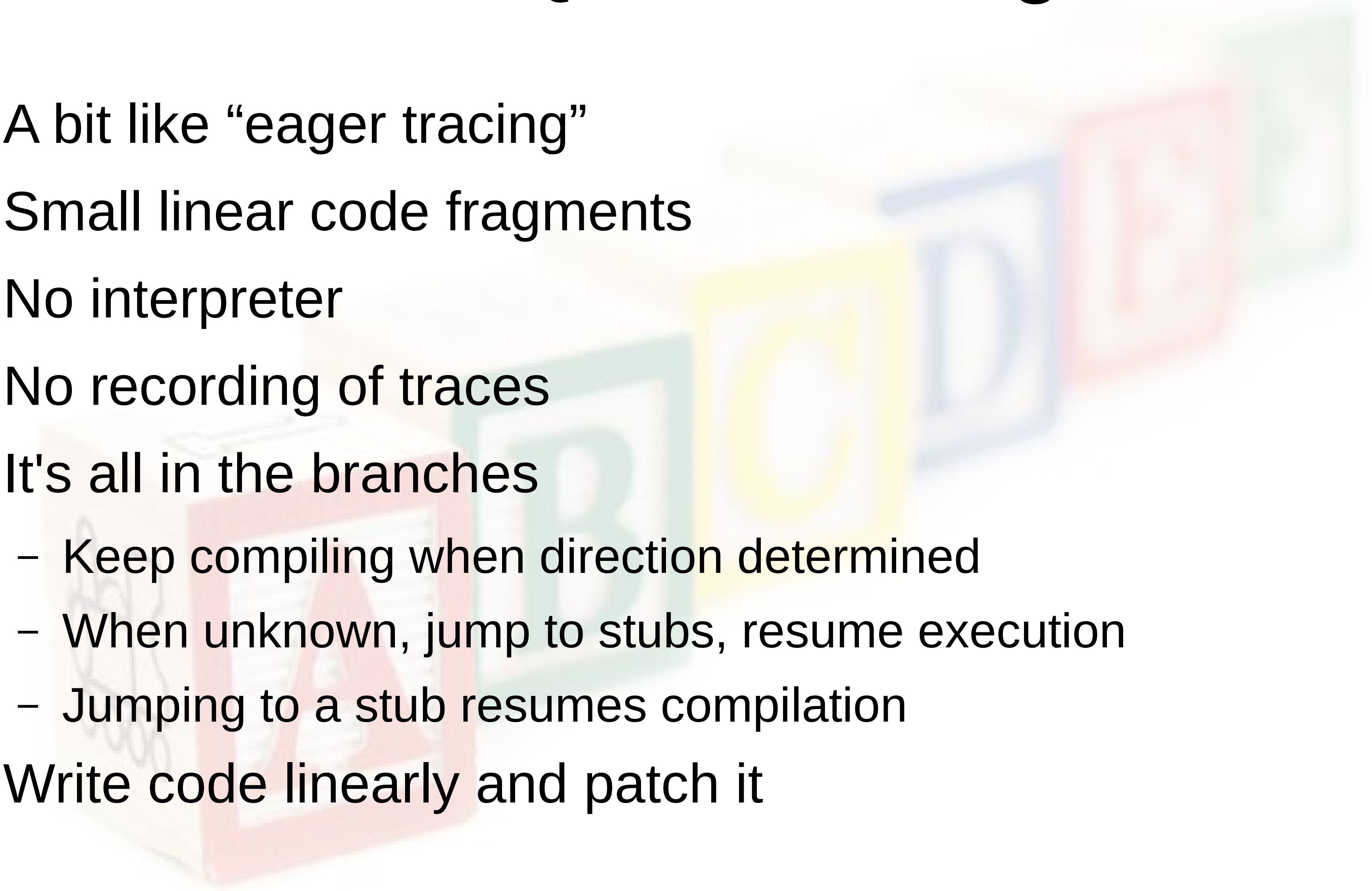


- Generating versions eagerly is problematic
  - Guess ahead of time what will be executed
  - Must remain conservative, overestimate
  - Generate versions that will never be used
- Compiles blocks lazily: when first executed
  - Interleaves compilation and execution
  - The running program's behavior drives versioning
- Avoid compiling unneeded blocks/versions
  - No floating-point code in your integer benchmark
  - Never executed error handling is never compiled


true
is_int32(n)?
false
n is int32
n is not int32
B
C
n is int32
n is not int32
D'
D''


true
is_int32(n)?
false
n is int32
n is not int32
B
C
n is int32
n is not int32
D'
D''

Not Quite Tracing



- A bit like “eager tracing”
- Small linear code fragments
- No interpreter
- No recording of traces
- It's all in the branches
  - Keep compiling when direction determined
  - When unknown, jump to stubs, resume execution
  - Jumping to a stub resumes compilation
- Write code linearly and patch it

Incremental Codegen Example

```
function sumInts(n)
{
    var sum = 0;
    for (var i = 0; i < n; i++)
        sum += i;

    return sum;
}

sumInts(600);
```

```
function sumInts(n)
{
    var sum = 0;
    for (var i = 0; i < n; i++)
        sum += i;

    return sum;
}

sumInts(600);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne not_int_stub
je is_int_stub
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne not_int_stub
je is_int_stub
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne not_int_stub
je is_int_stub
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jl branch_for_body(22427);
jmp branch_for_exit(22429);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jl branch_for_body(22427);
jmp branch_for_exit(22429);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jge branch_for_exit(22429);

for_body(22427):
; $0 = is_int32 $2
; $0 = is_int32 $3
; $8 = add_i32_ovf $2, $3
add ecx, edx;
jo branch_if_false(2249D);
jmp branch_call_merge(22485);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jge branch_for_exit(22429);

for_body(22427):
; $0 = is_int32 $2
; $0 = is_int32 $3
; $8 = add_i32_ovf $2, $3
add ecx, edx;
jo branch_if_false(2249D);
jmp branch_call_merge(22485);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jge branch_for_exit(22429);

for_body(22427):
; $0 = is_int32 $2
; $0 = is_int32 $3
; $8 = add_i32_ovf $2, $3
add ecx, edx;
jo branch_if_false(2249D);

call_merge(22485):
; $0 = is_int32 $3
; $15 = add_i32_ovf $3, 1
add edx, 1;
```

```
for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
jmp if_true(22453);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jge branch_for_exit(22429);

for_body(22427):
; $0 = is_int32 $2
; $0 = is_int32 $3
; $8 = add_i32_ovf $2, $3
add ecx, edx;
jo branch_if_false(2249D);

call_merge(22485):
; $0 = is_int32 $3
; $15 = add_i32_ovf $3, 1
add edx, 1;
```

```
for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
jmp if_true(22453);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jge branch_for_exit(22429);

for_body(22427):
; $0 = is_int32 $2
; $0 = is_int32 $3
; $8 = add_i32_ovf $2, $3
add ecx, edx;
jo branch_if_false(2249D);

call_merge(22485):
; $0 = is_int32 $3
; $15 = add_i32_ovf $3, 1
add edx, 1;
```

```
for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
jmp if_true(22453);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jge branch_for_exit(22429);

for_body(22427):
; $0 = is_int32 $2
; $0 = is_int32 $3
; $8 = add_i32_ovf $2, $3
add ecx, edx;
jo branch_if_false(2249D);

call_merge(22485):
; $0 = is_int32 $3
; $15 = add_i32_ovf $3, 1
add edx, 1;
```

```
for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
jmp if_true(22453);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jge branch_for_exit(22429);

for_body(22427):
; $0 = is_int32 $2
; $0 = is_int32 $3
; $8 = add_i32_ovf $2, $3
add ecx, edx;
jo branch_if_false(2249D);

call_merge(22485):
; $0 = is_int32 $3
; $15 = add_i32_ovf $3, 1
add edx, 1;
```

```
for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
jmp if_true(22453);
```

```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
cmp [byte r13 + 26], 1;
jne branch_if_join(22456);

if_true(22453):
; $0 = lt_i32 $3, $26
mov r12, [qword r14 + 208];
cmp edx, r12d;
jge branch_for_exit(22429);

for_body(22427):
; $0 = is_int32 $2
; $0 = is_int32 $3
; $8 = add_i32_ovf $2, $3
add ecx, edx;
jo branch_if_false(2249D);

call_merge(22485):
; $0 = is_int32 $3
; $15 = add_i32_ovf $3, 1
add edx, 1;
```

```
for_test(22426):
; $0 = is_int32 $3
; $0 = is_int32 $26
jmp if_true(22453);

for_exit(22429):
; ret $2
mov dl, 1;
mov eax, [dword r14 + 200];
sub eax, 1;
xor ebx, ebx;
cmp eax, 0;
cmovl eax, ebx;
add eax, 27;
mov rbx, [qword r14 + 176];
add r13, rax;
shl rax, 3;
add r14, rax;
jmp rbx;
```

But Wait, There's More!

Typed Object Shapes

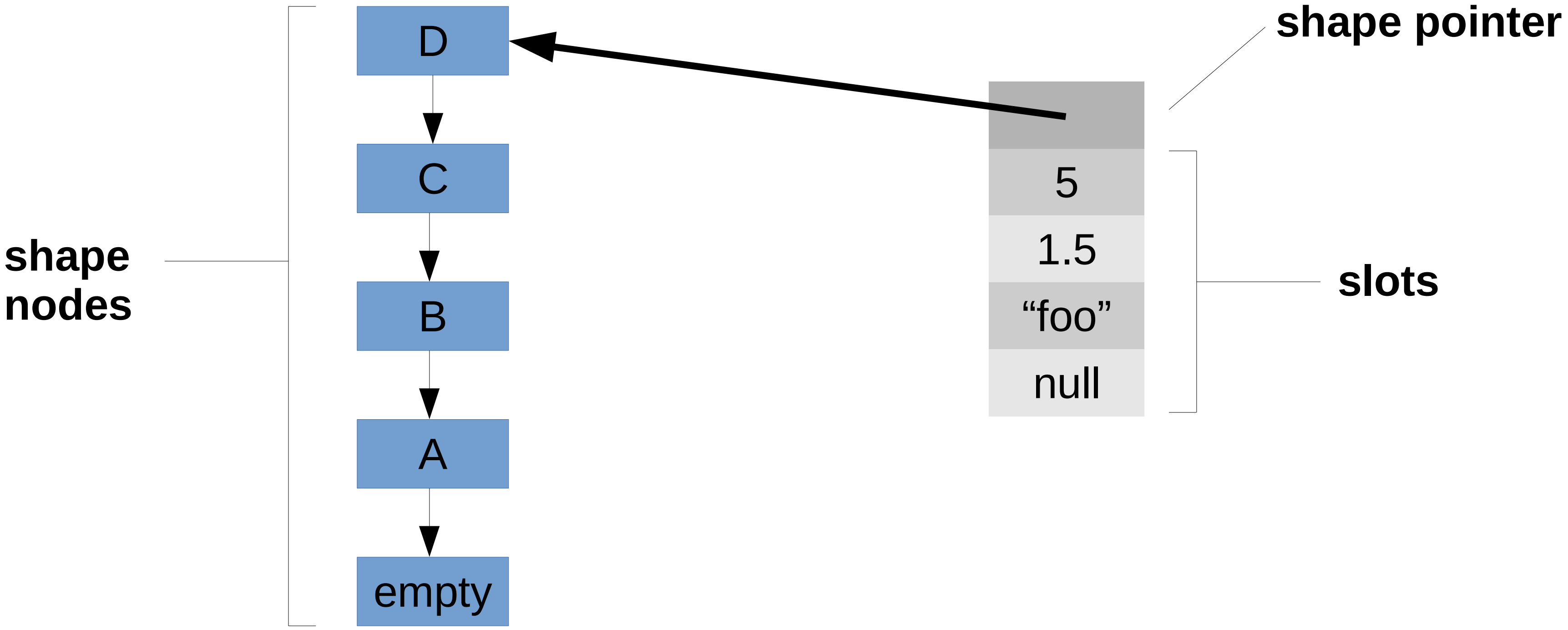
JavaScript Objects

- Are dynamic with complex semantics
  - Dynamic addition & deletion of properties
  - Dynamic typing of properties
  - Modifiable property attributes
  - Dynamically installable getters & setters
  - Indexable, behave like *dictionaries*
- Need to be efficient
  - Must perform better than dictionaries

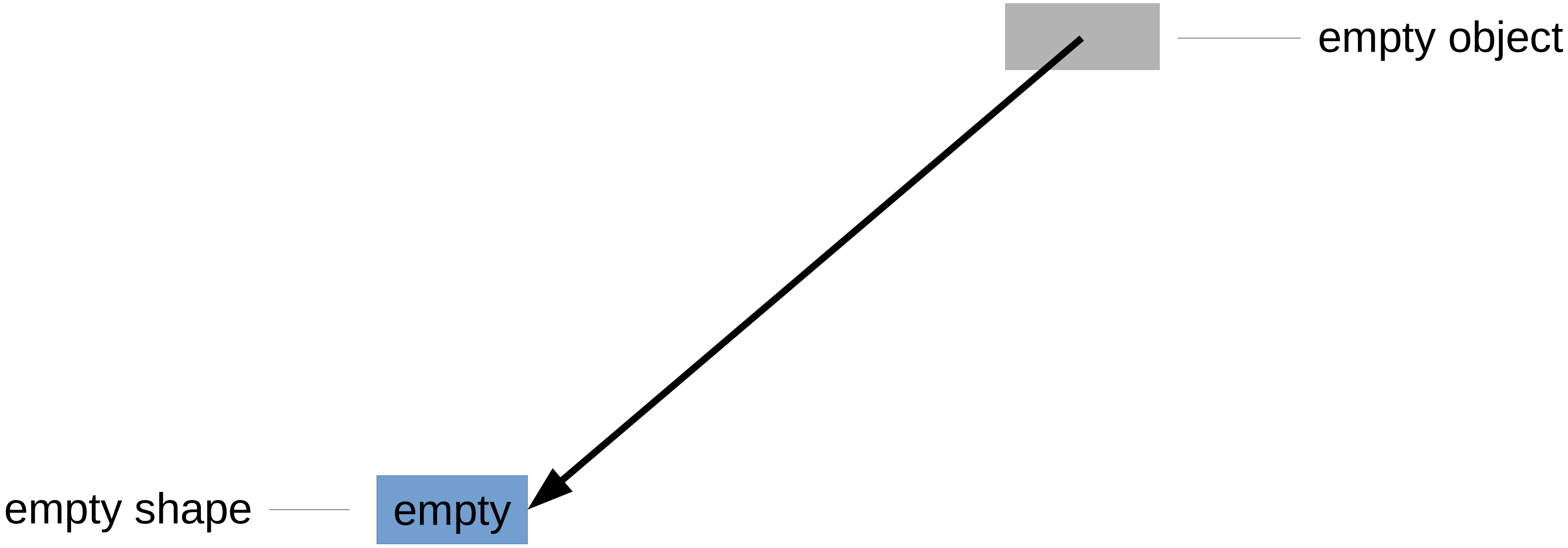
Objects as Dictionaries

Attribute flags	Keys	Values
WEC	A	5
WEC	B	1.5
--	--	--
E	C	“foo”
WE	D	null
--	--	--

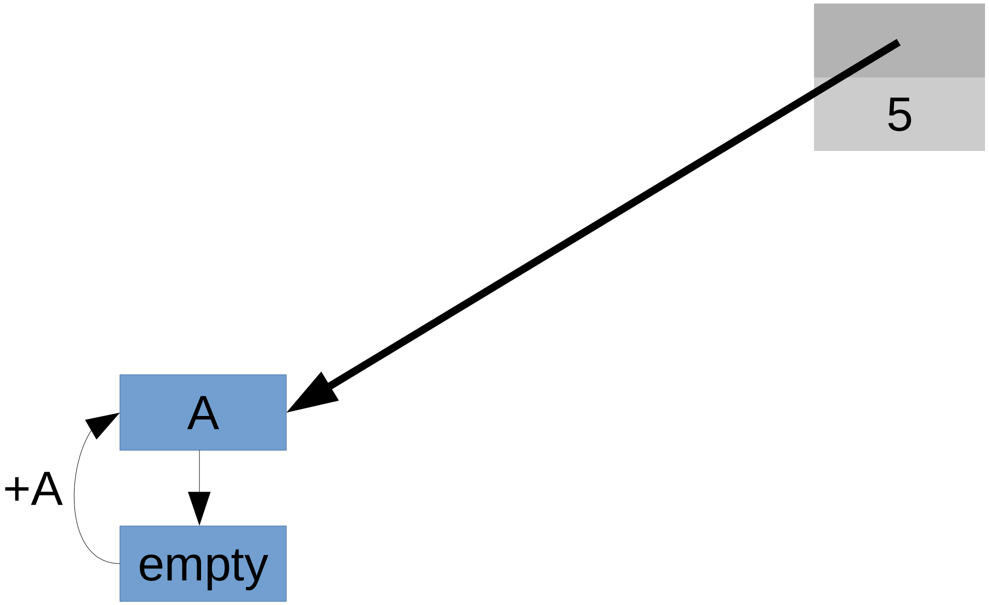
Shapes



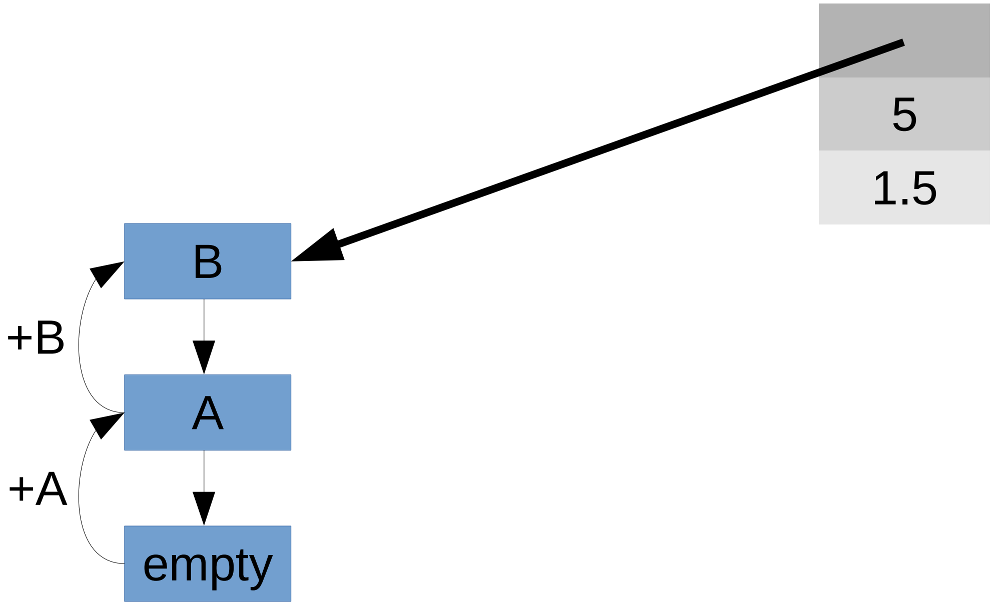
The Empty Object



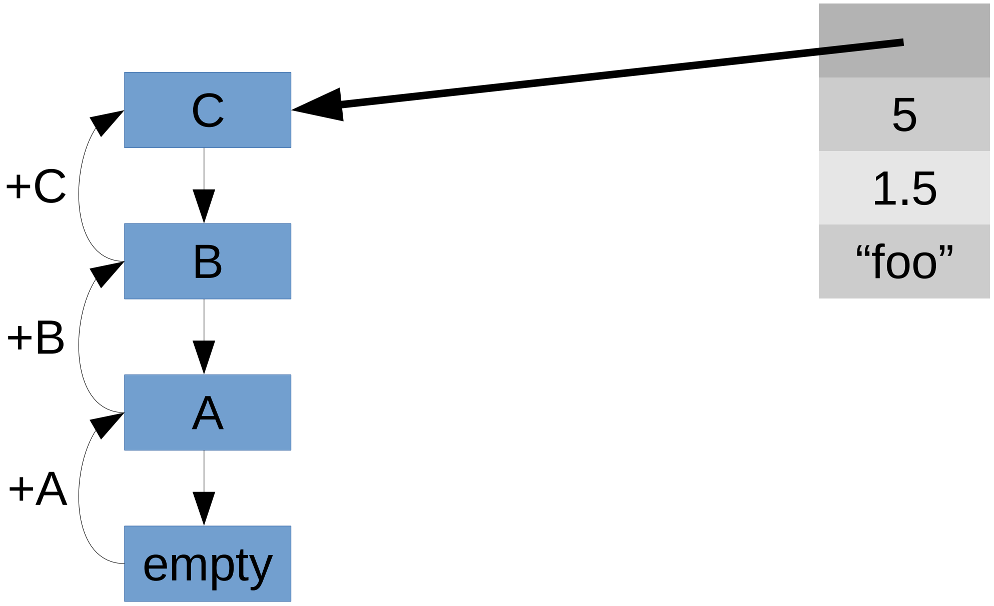
Growing Objects



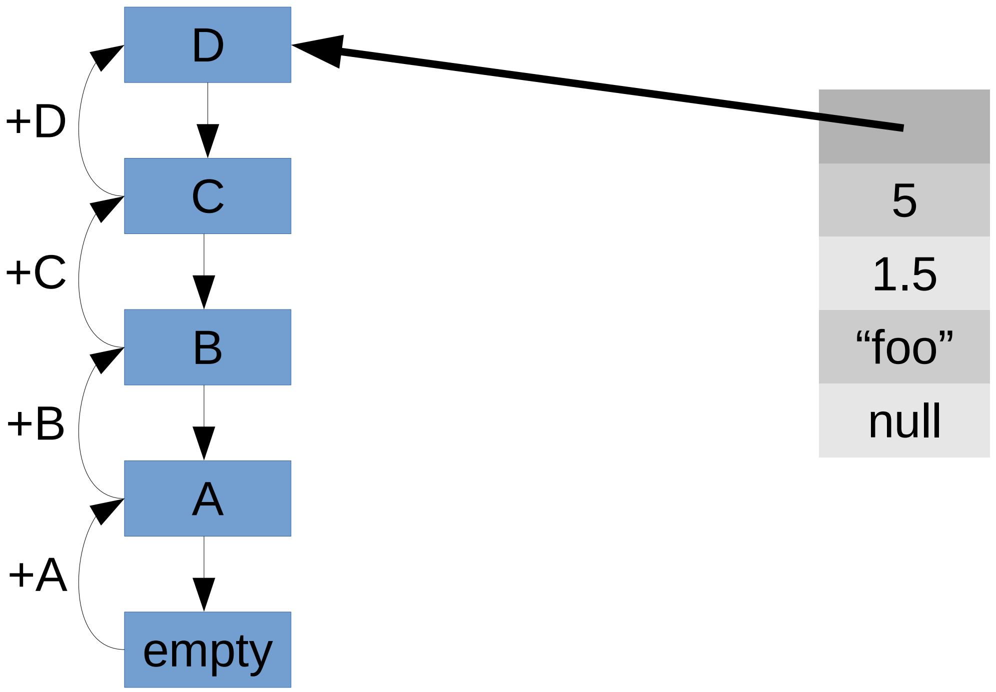
Growing Objects



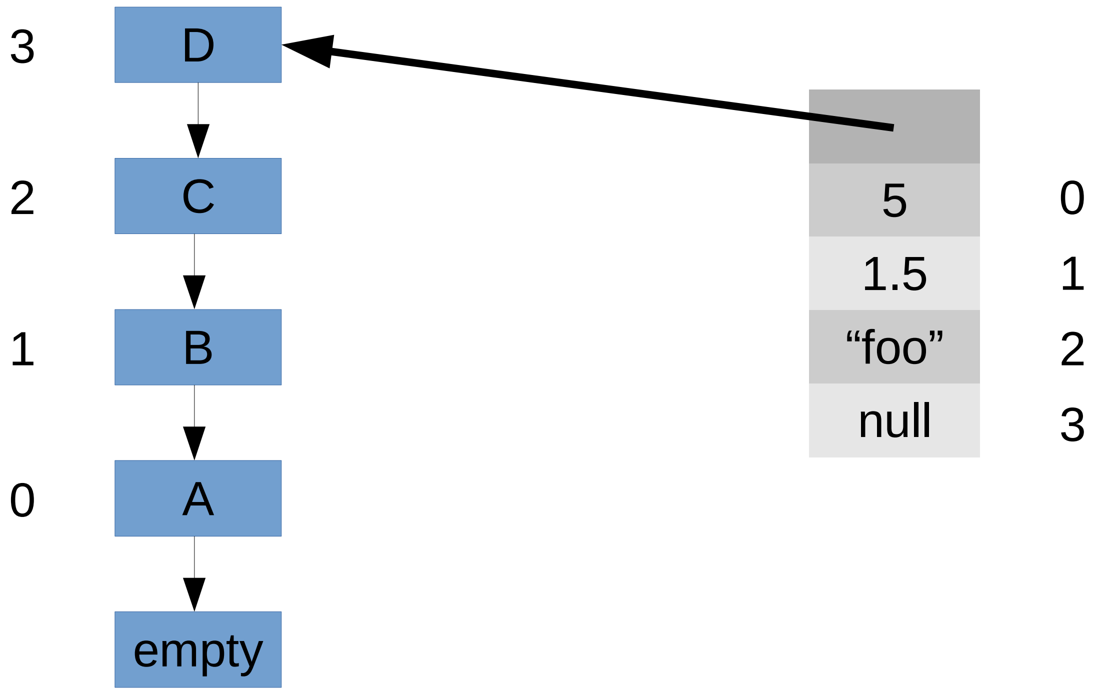
Growing Objects



Growing Objects



Inline Caching



3 D

2 C

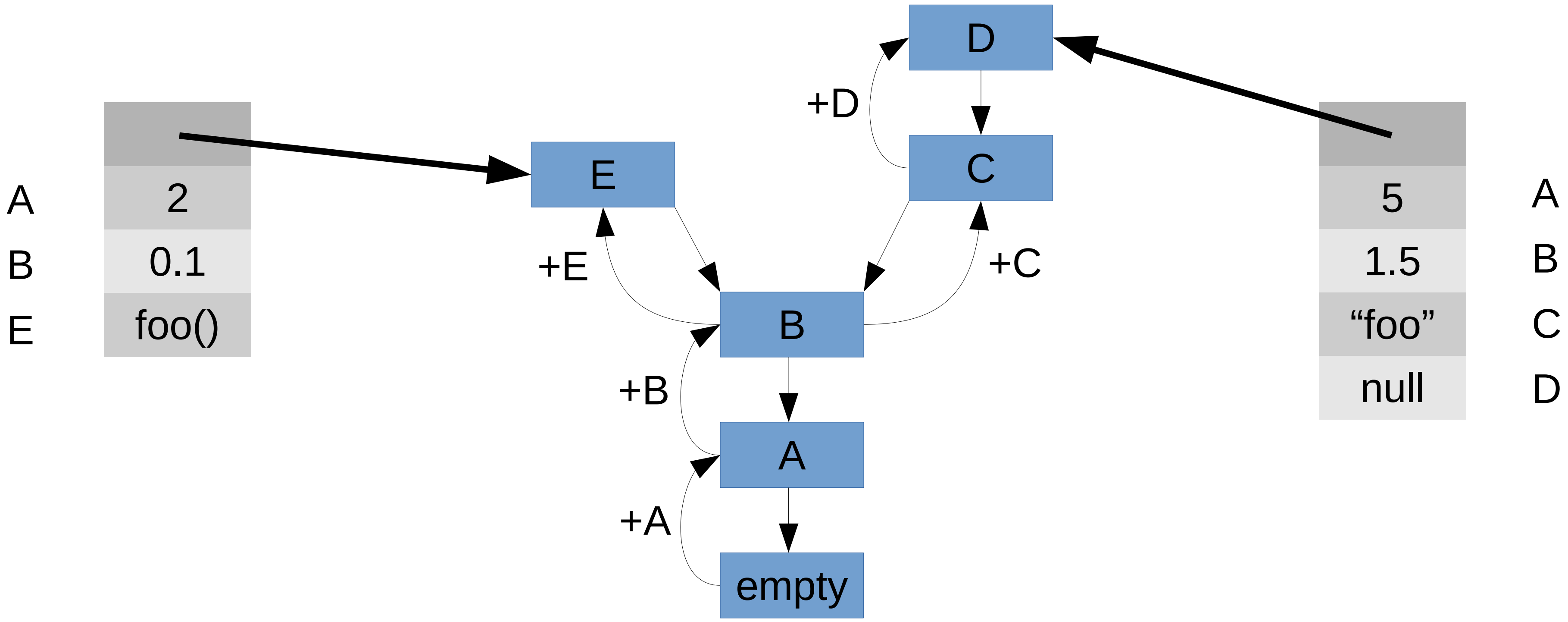
1 B

0 A

empty

5	0
1.5	1
“foo”	2
null	3

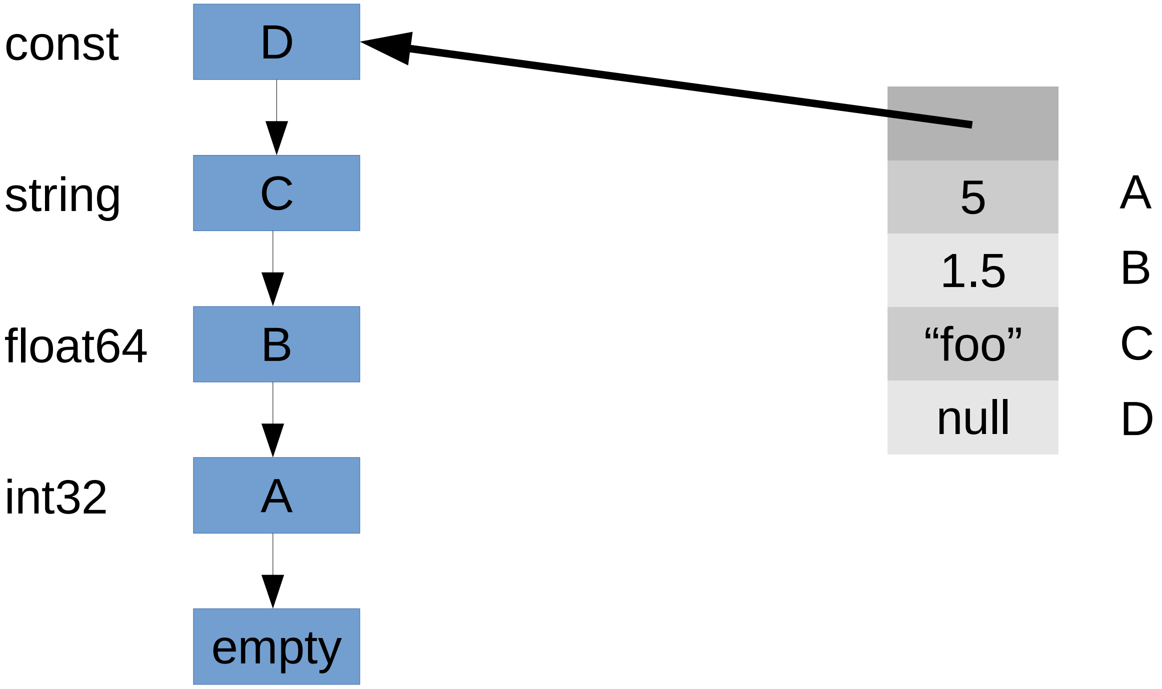
Shape Trees and Sharing



A	2
B	0.1
E	foo()

5	A
1.5	B
“foo”	C
null	D

Typed Shapes



const D

string C

float64 B

int32 A

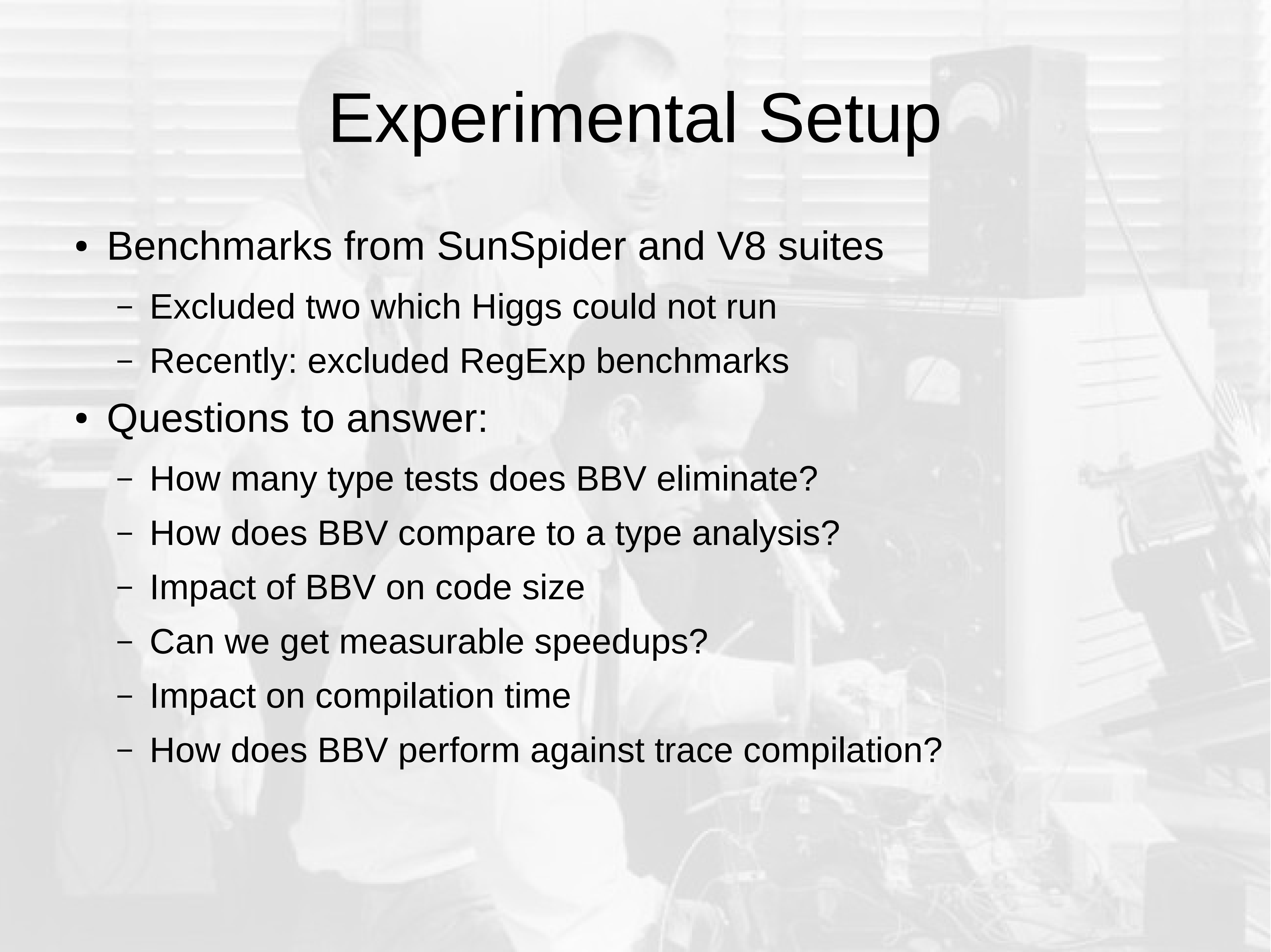
empty

5	A
1.5	B
“foo”	C
null	D

Typed Shapes

- Extend shapes to store property types
  - Type tags
  - Function pointers
- Allows versioning based on shapes
- Permits the elimination of:
  - Missing property checks
  - Getter/setter checks
  - Property type checks
  - Boxing/unboxing overhead
  - Dynamic dispatch on function calls

Experimental Results



Experimental Setup

- Benchmarks from SunSpider and V8 suites
  - Excluded two which Higgs could not run
  - Recently: excluded RegExp benchmarks
- Questions to answer:
  - How many type tests does BBV eliminate?
  - How does BBV compare to a type analysis?
  - Impact of BBV on code size
  - Can we get measurable speedups?
  - Impact on compilation time
  - How does BBV perform against trace compilation?

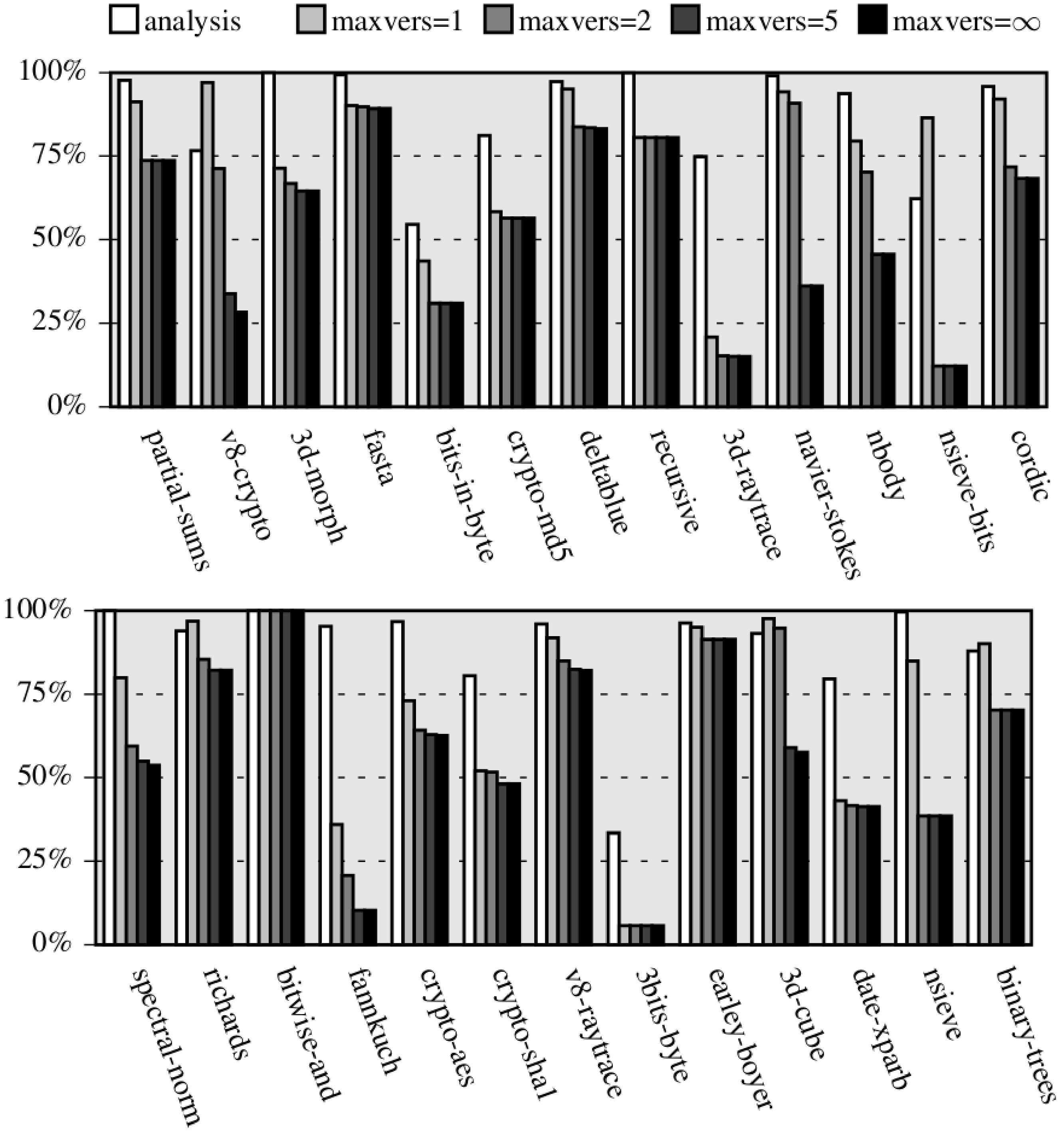


Figure 5. Counts of dynamic type tests (relative to baseline)

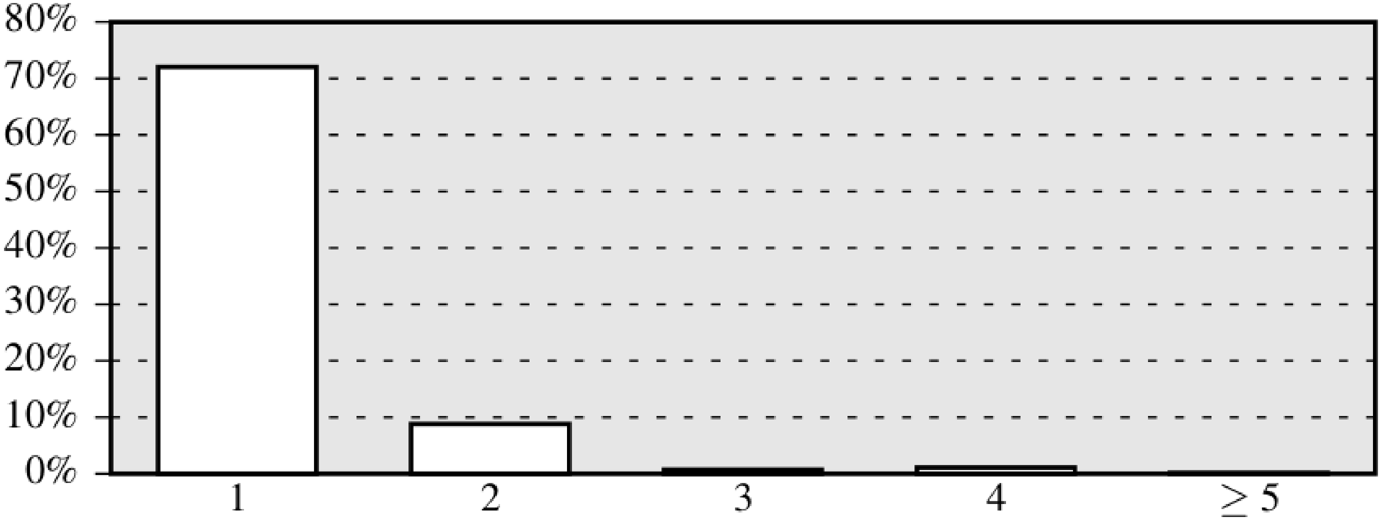


Figure 6. Relative occurrence of block version counts

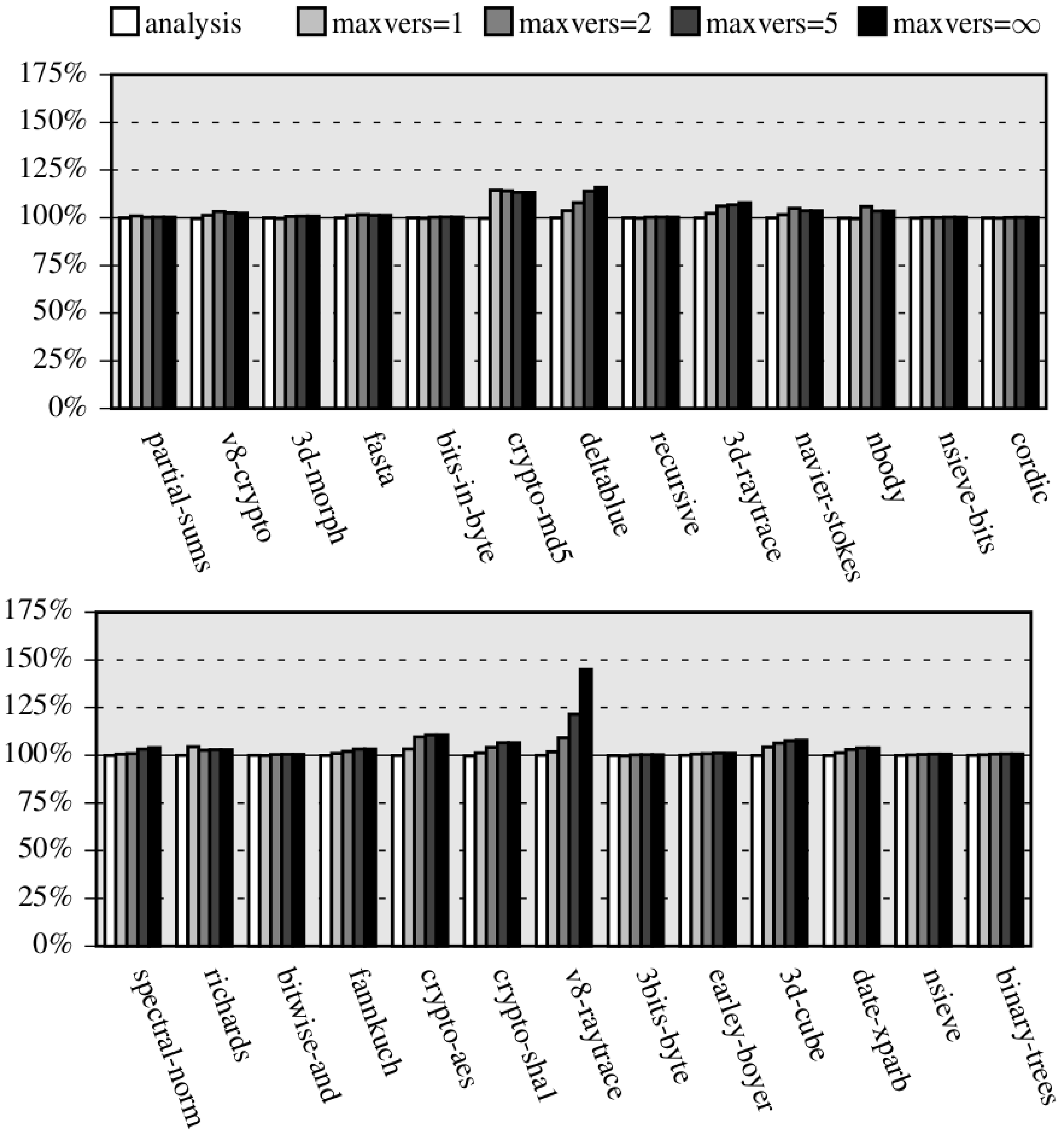


Figure 7. Code size growth for different block version limits

2013

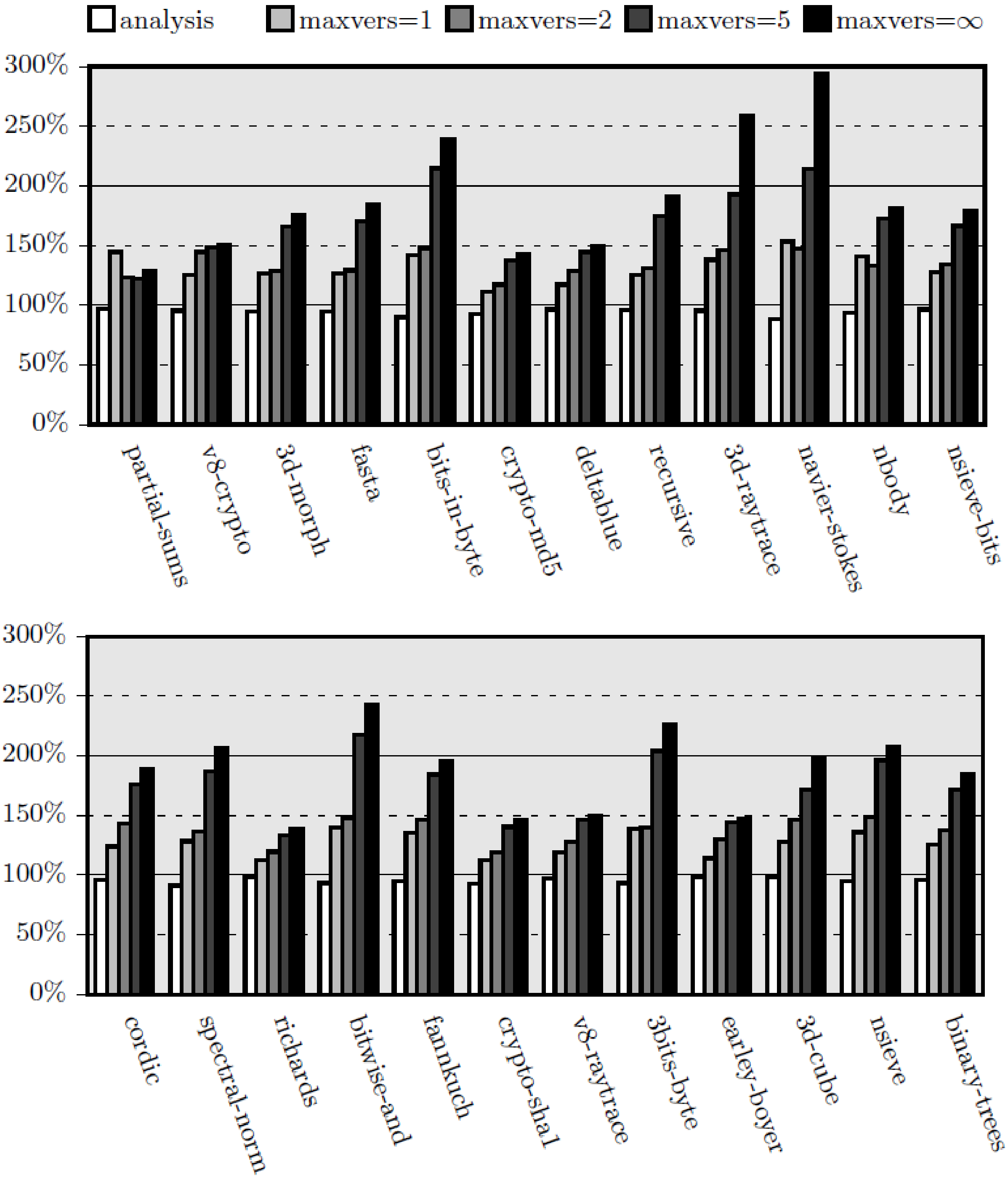


Fig. 10. Code size growth for different block version limits

2013

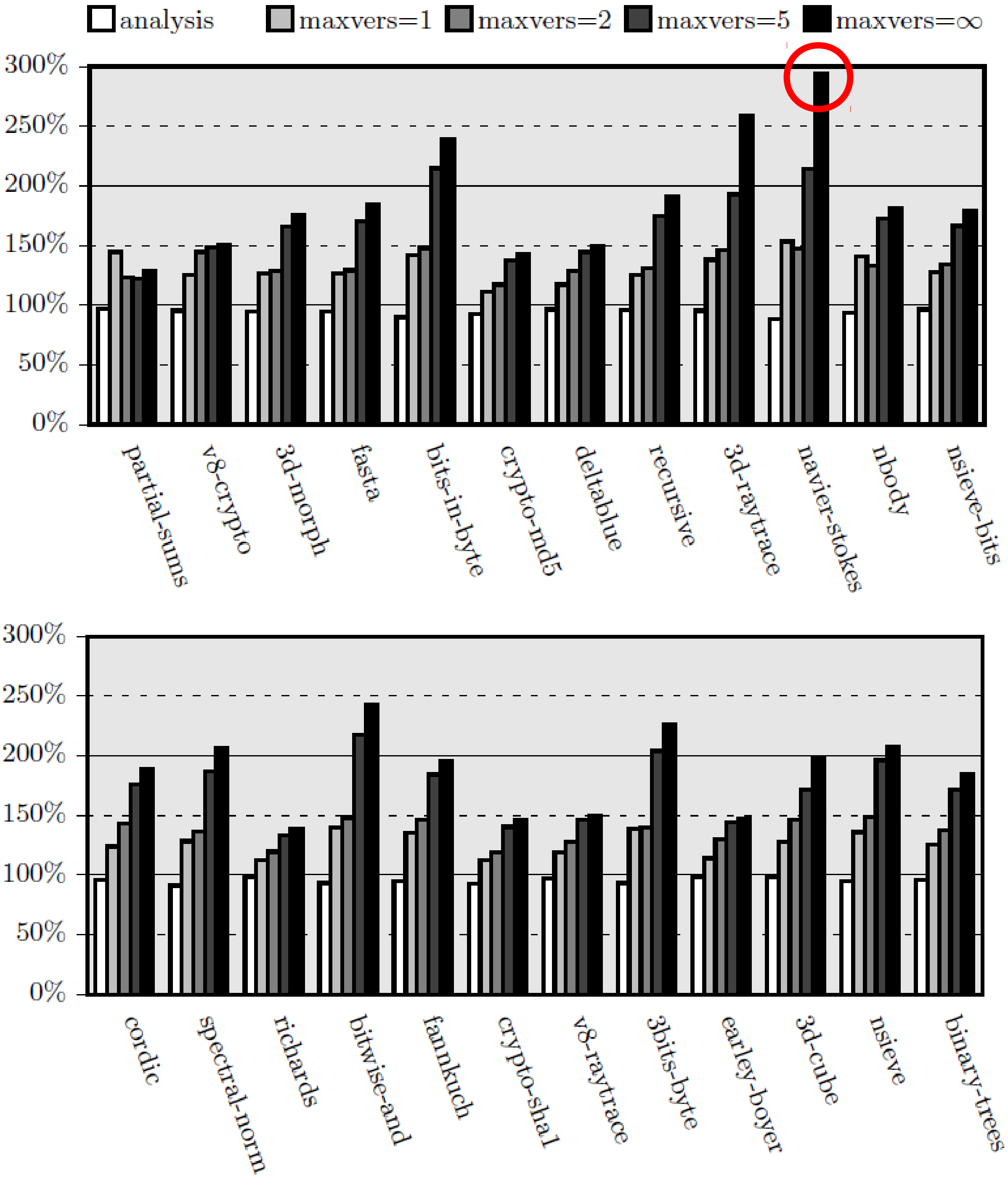


Fig. 10. Code size growth for different block version limits

worst case: almost 300%

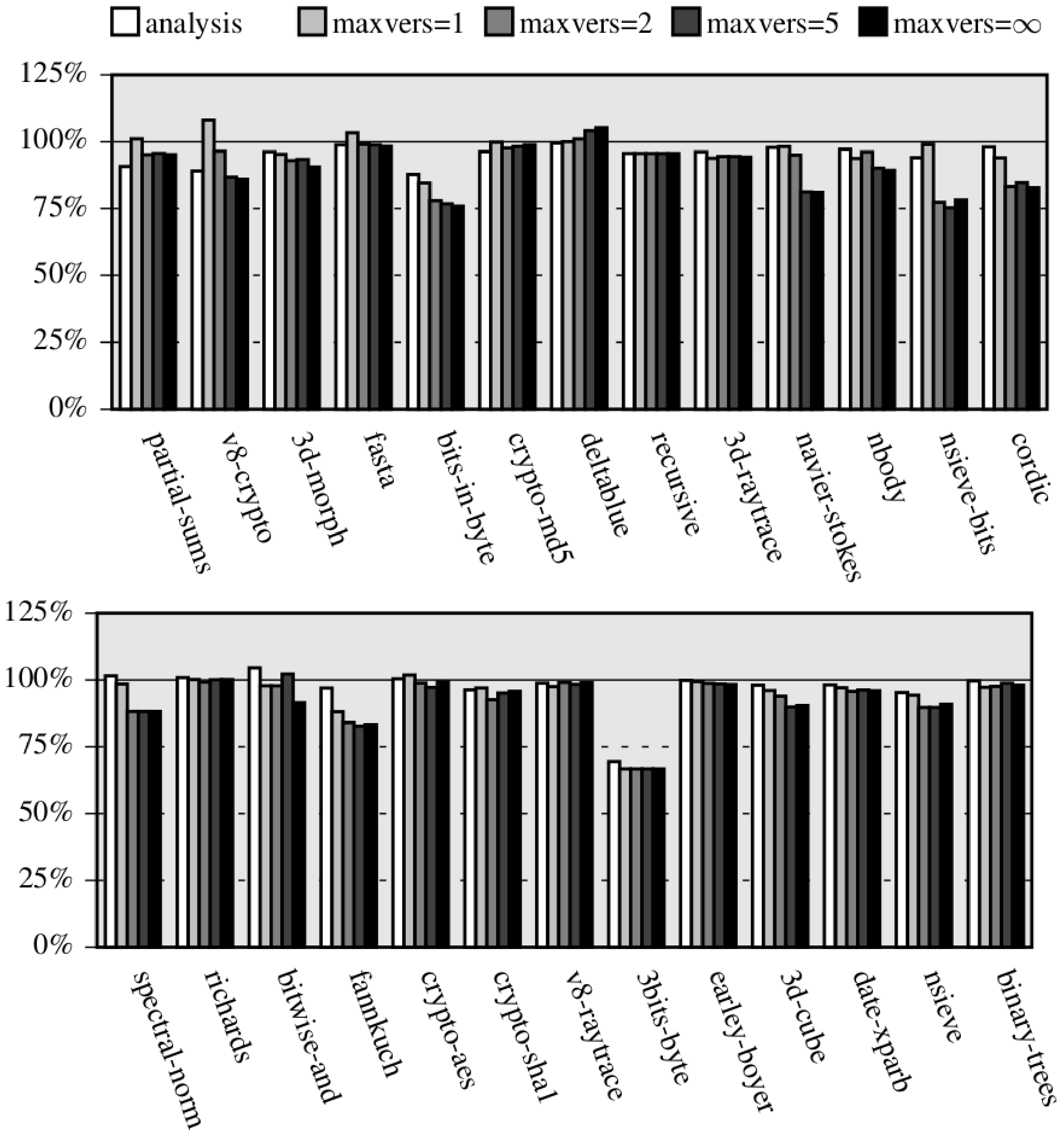


Figure 8. Execution time change for different block version limits

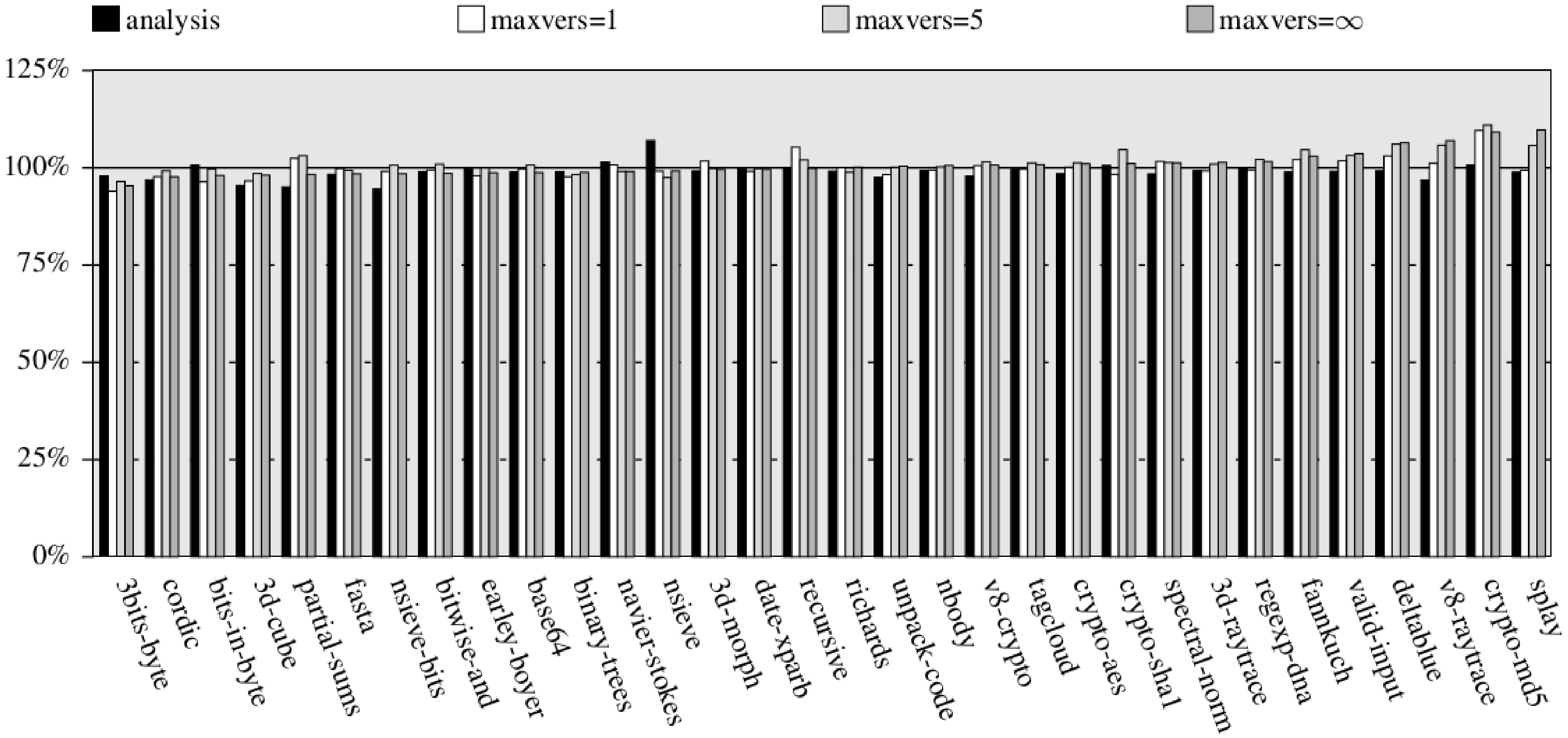


Figure 12. Compilation time for various block version limits (relative to baseline)

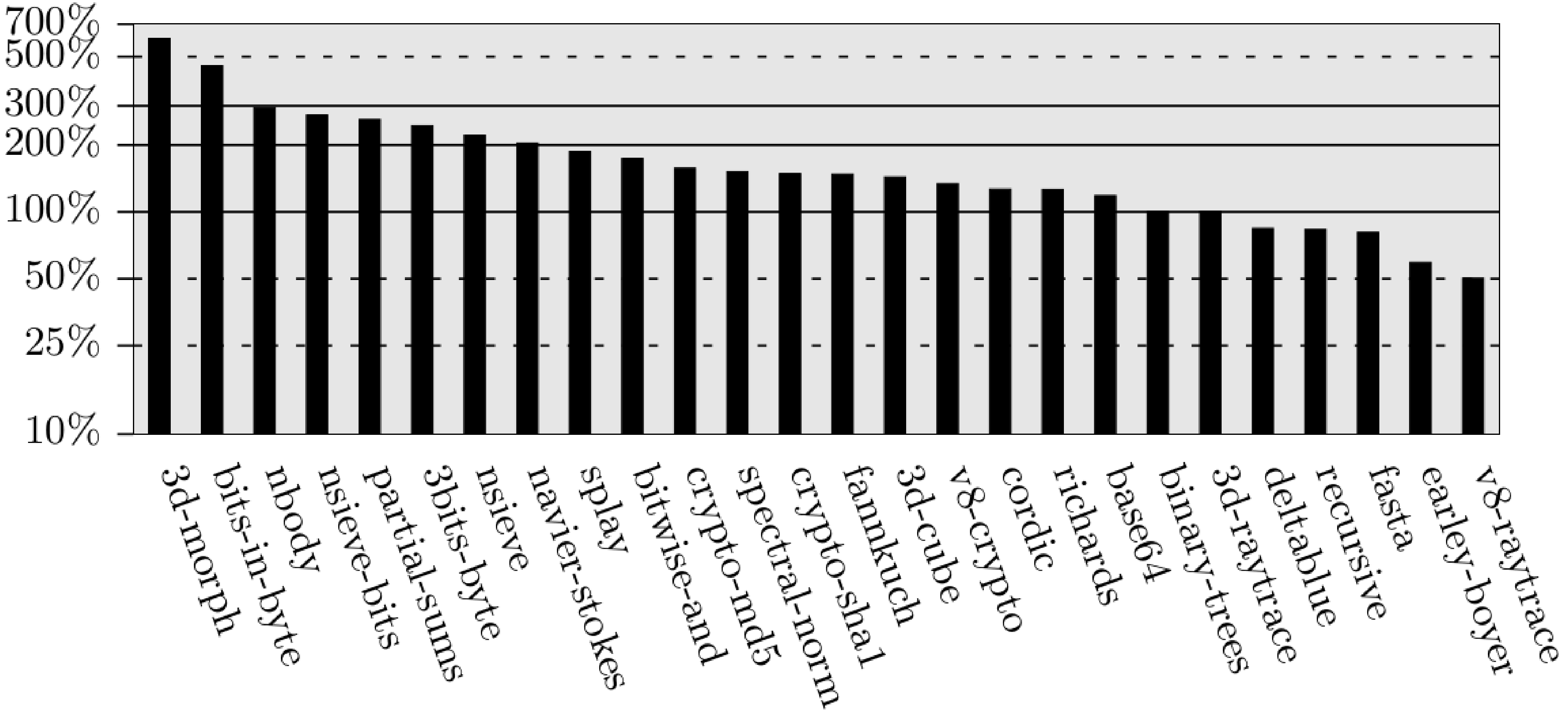


Fig. 13. Speedup relative to V8 baseline (log scale, higher is better)

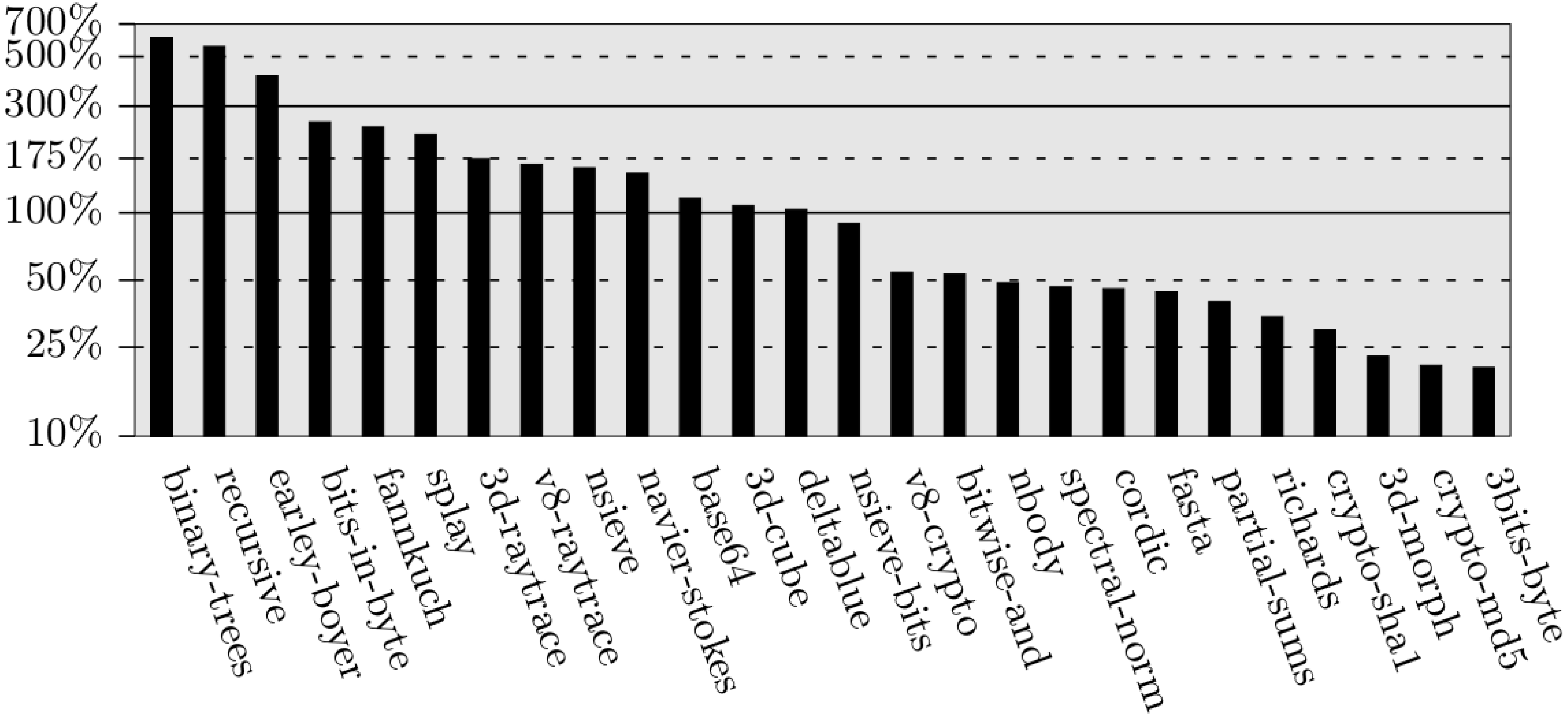
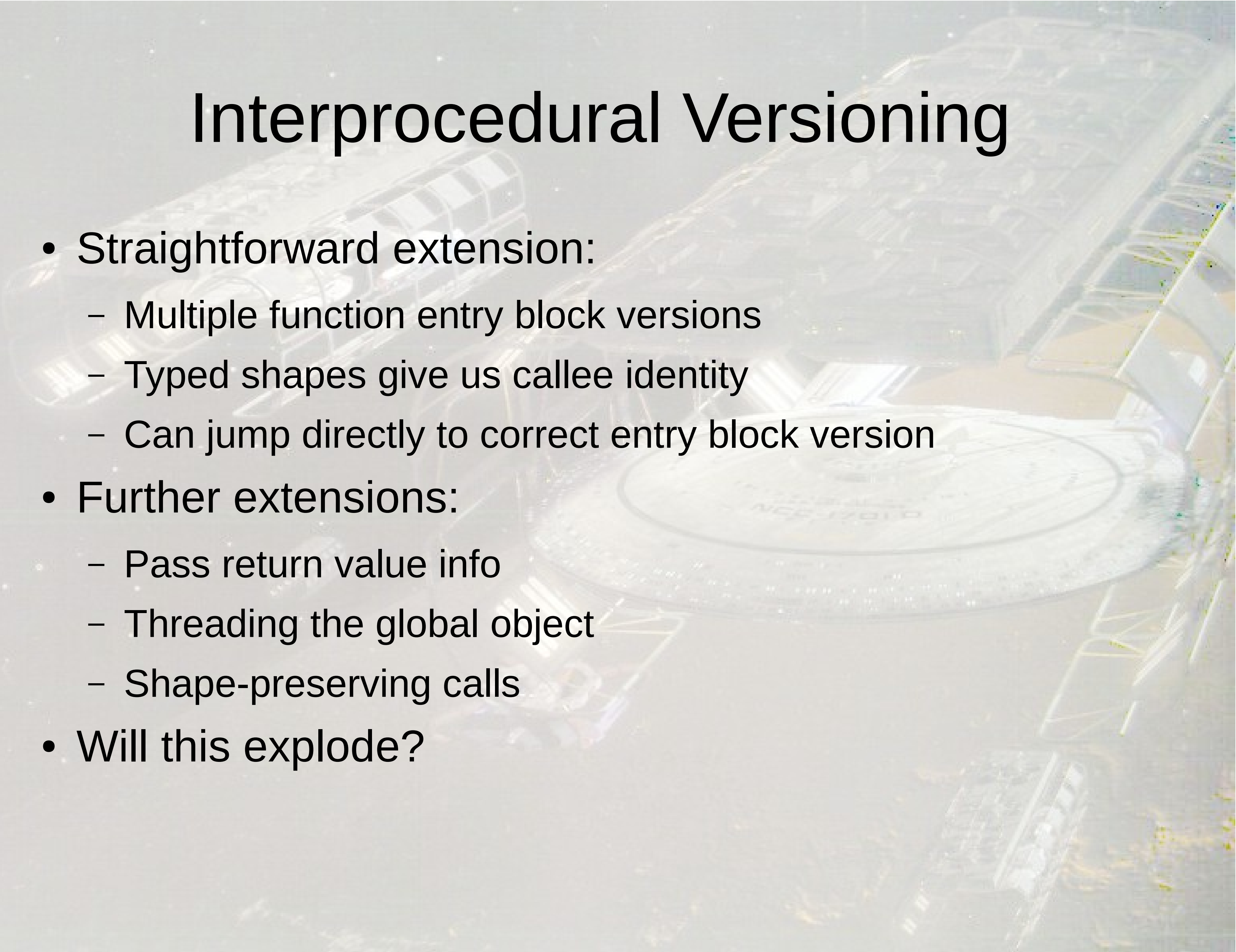


Fig. 14. Speedup relative to TraceMonkey (log scale, higher is better)

Future Work



Interprocedural Versioning

- Straightforward extension:
  - Multiple function entry block versions
  - Typed shapes give us callee identity
  - Can jump directly to correct entry block version
- Further extensions:
  - Pass return value info
  - Threading the global object
  - Shape-preserving calls
- Will this explode?

Incremental Inlining

- Incremental compilation + inlining
- Inline functions one block at a time
  - Only what gets executed
  - Particularly useful for self-hosted runtimes
- Avoid expensive CFG transformations
- Avoid needing to recompile whole functions
  - No need for separate inlining pass
  - No need for on-stack replacement

IR-level Versioning

- In Higgs, versioning happens in the backend
  - Backend maintains basic block “instances”
  - Each instance has a context object, type info
- Backend's view of code is rather myopic
  - Too late for complex code transformations
- BBV should be done by transforming the IR
  - Context is implicitly threaded in the IR
  - Higher-level optimizations possible
  - e.g.: property access optimizations

Code Collection & Compaction

- Some functions become unreachable
  - All compiled code is then dead
- Some block versions may become irrelevant
  - Object shape which doesn't exist anymore
- Dead machine code should be removed
  - Space is limited, compactness is good for i-cache
- In Higgs: machine code is relocatable
  - Eventually: compacting machine code GC

github.com/maximecb/Higgs

#higgsjs on freenode IRC

pointersgonewild.com

maximechevalierb@gmail.com

Love2Code on twitter

Special thanks to:

Prof. Marc Feeley
Tommy Everett @tach4n
Brett Fraley
Paul Fryzel @paulfryzel
Zimbabao Borbaki @zimbabao
Sorella @robotlolita
Óscar Toledo @nanochess
Luke Wagner (blog.mozilla.org/luke)