Simple and Effective Type Check Removal through Lazy Basic Block Versioning

Maxime Chevalier-Boisvert
joint work with Marc Feeley

ECOOP - July 8th, 2015

Université de Montréal

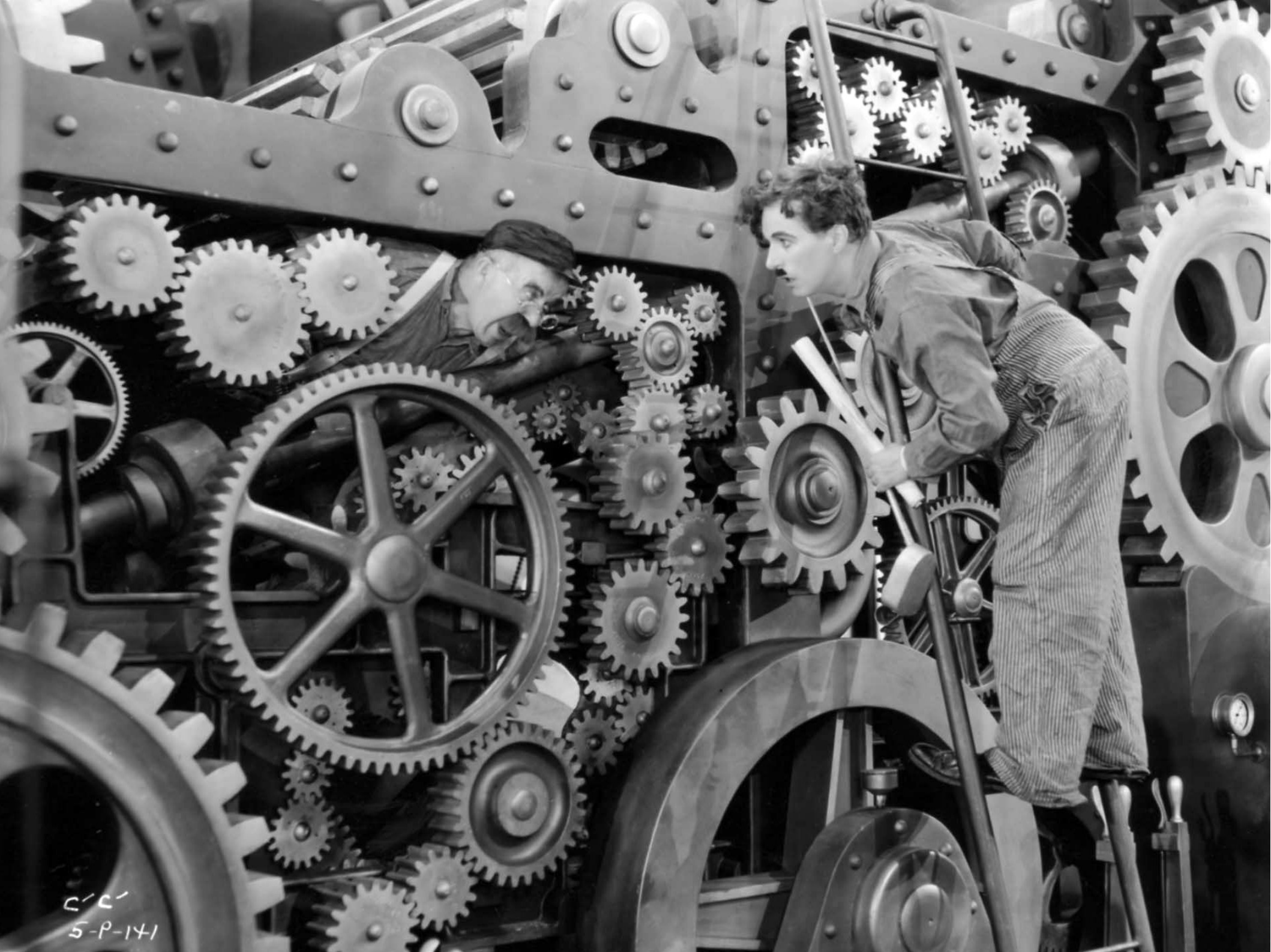
Introduction

- 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

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
```


C-C-
5-P-141

```
// 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);
    }

    …
}
```

Basic Block Versioning

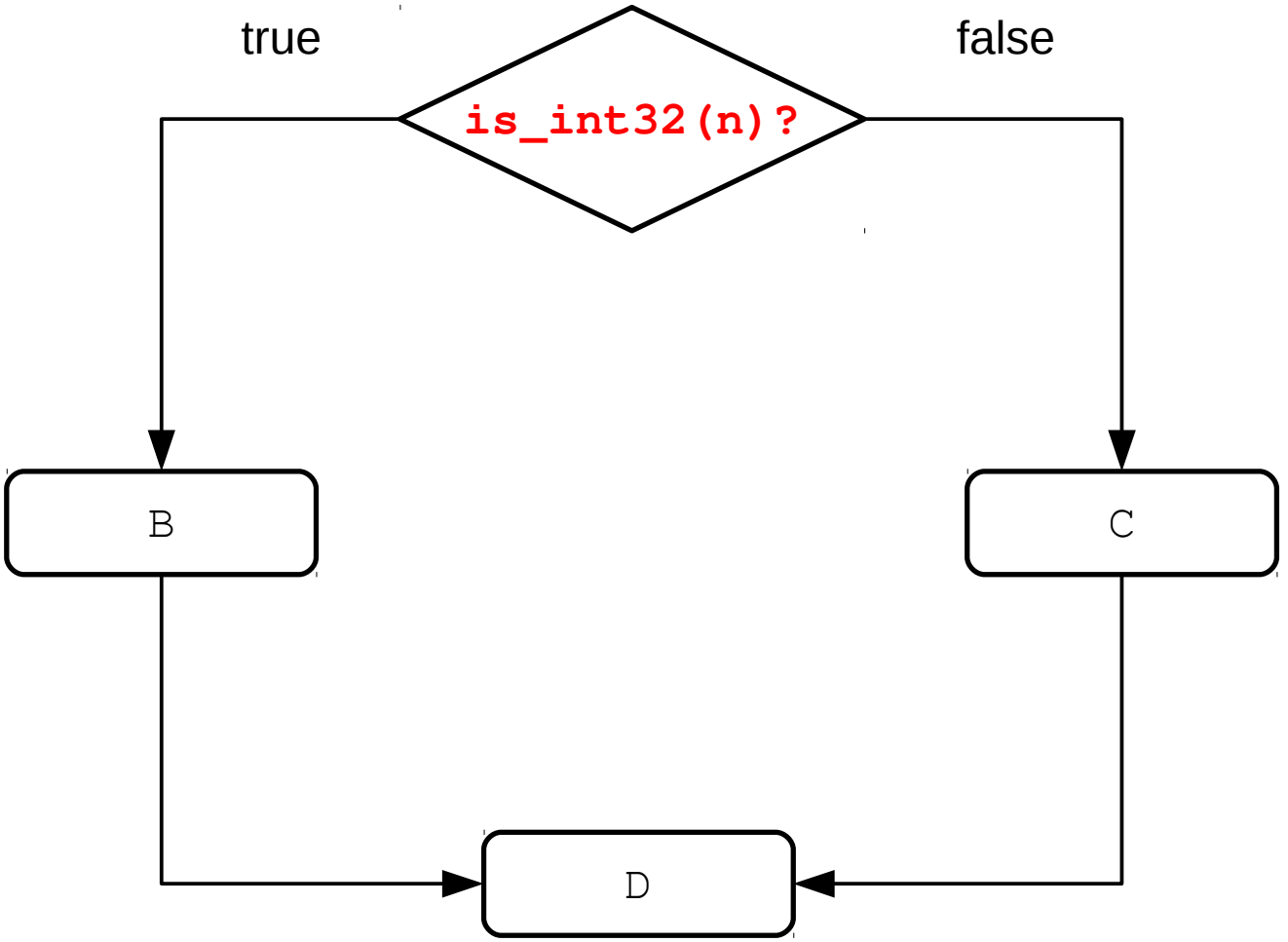
Basic Block Versioning

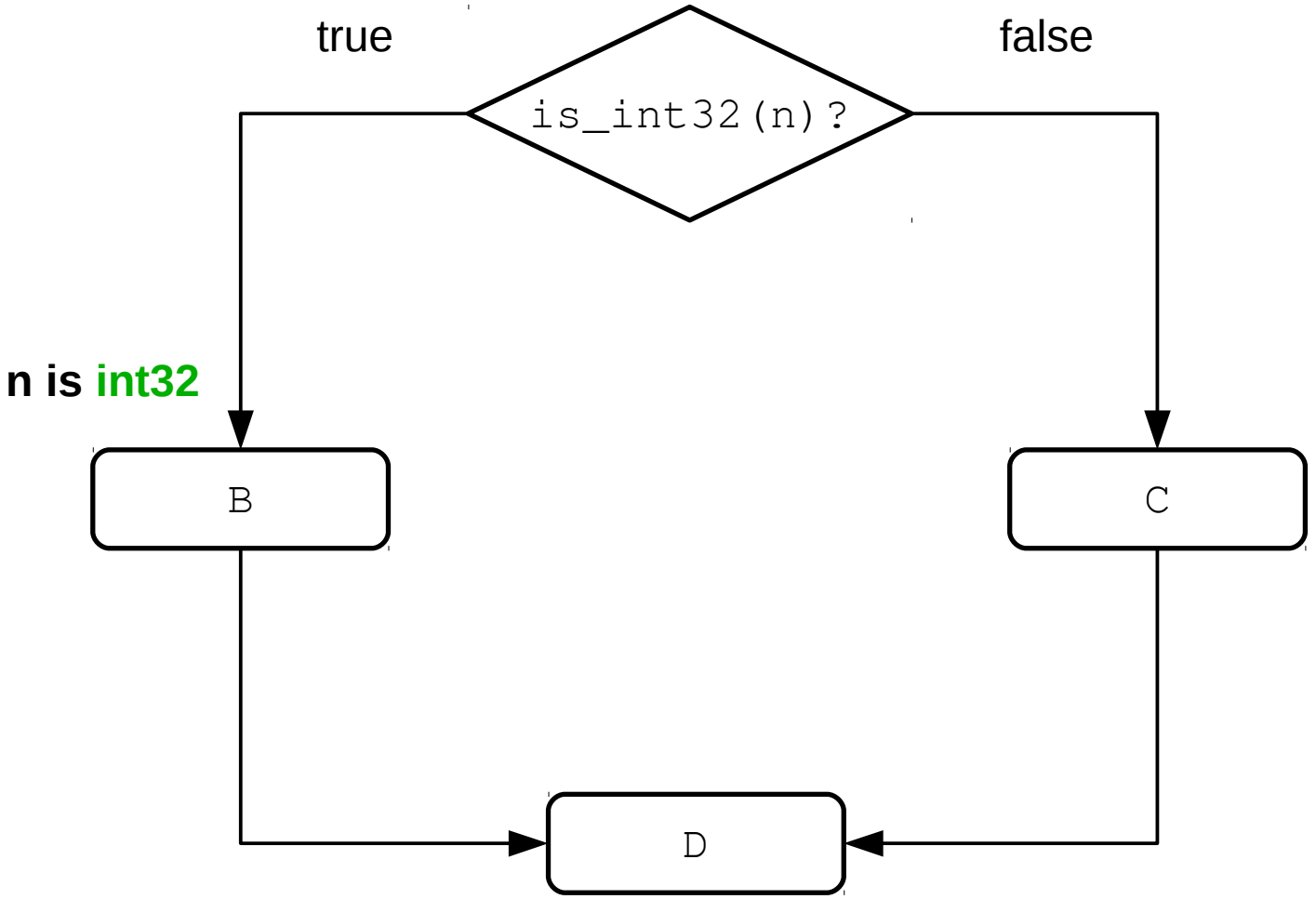
- Similar to trace compilation, procedure cloning
  - Lower granularity: basic blocks
- As we compile code, accumulate facts
  - Type tests extract type information (type tags)
  - Propagate known type tags
- Specialize blocks based on live var. types
  - May compile multiple versions of blocks

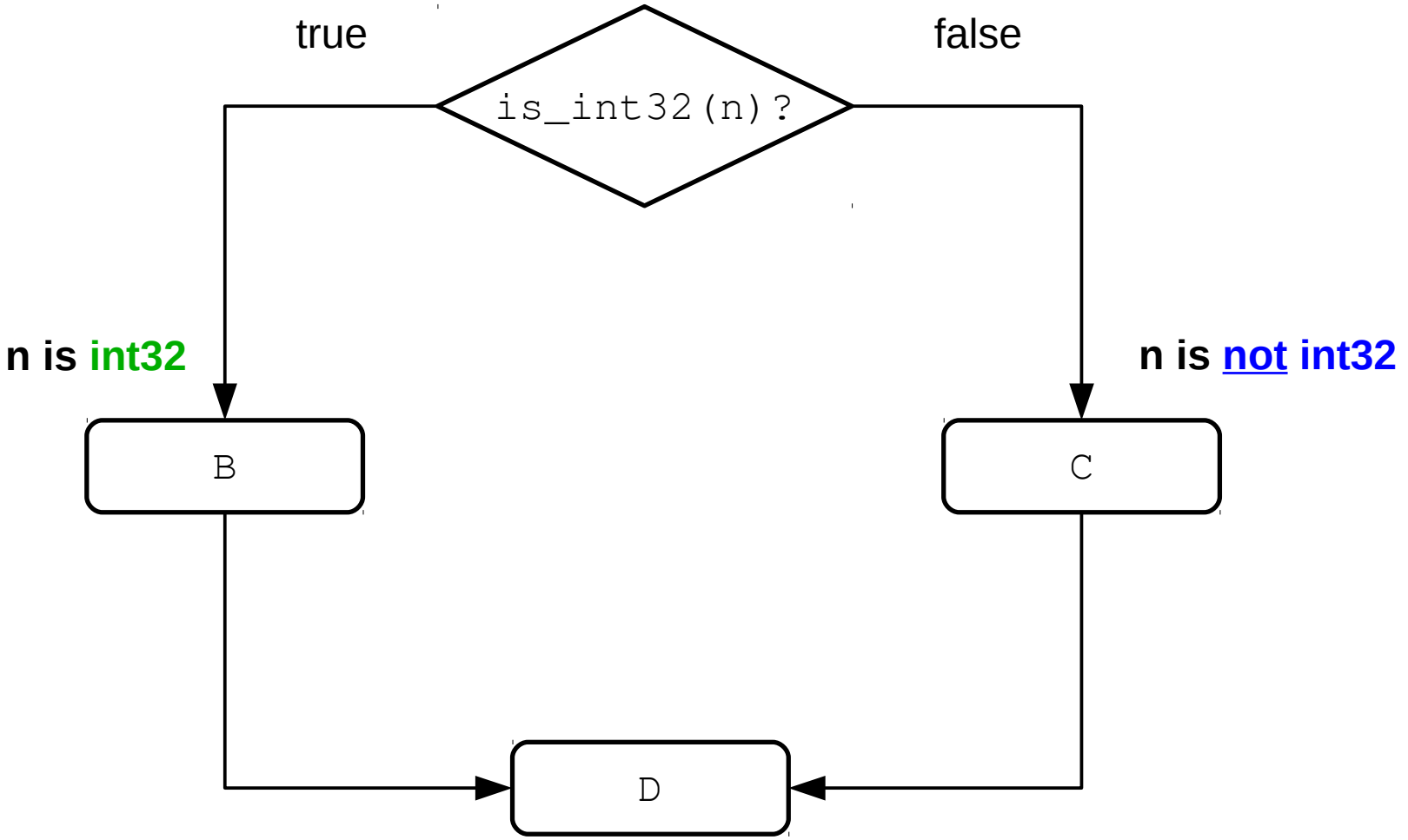
Type Tags

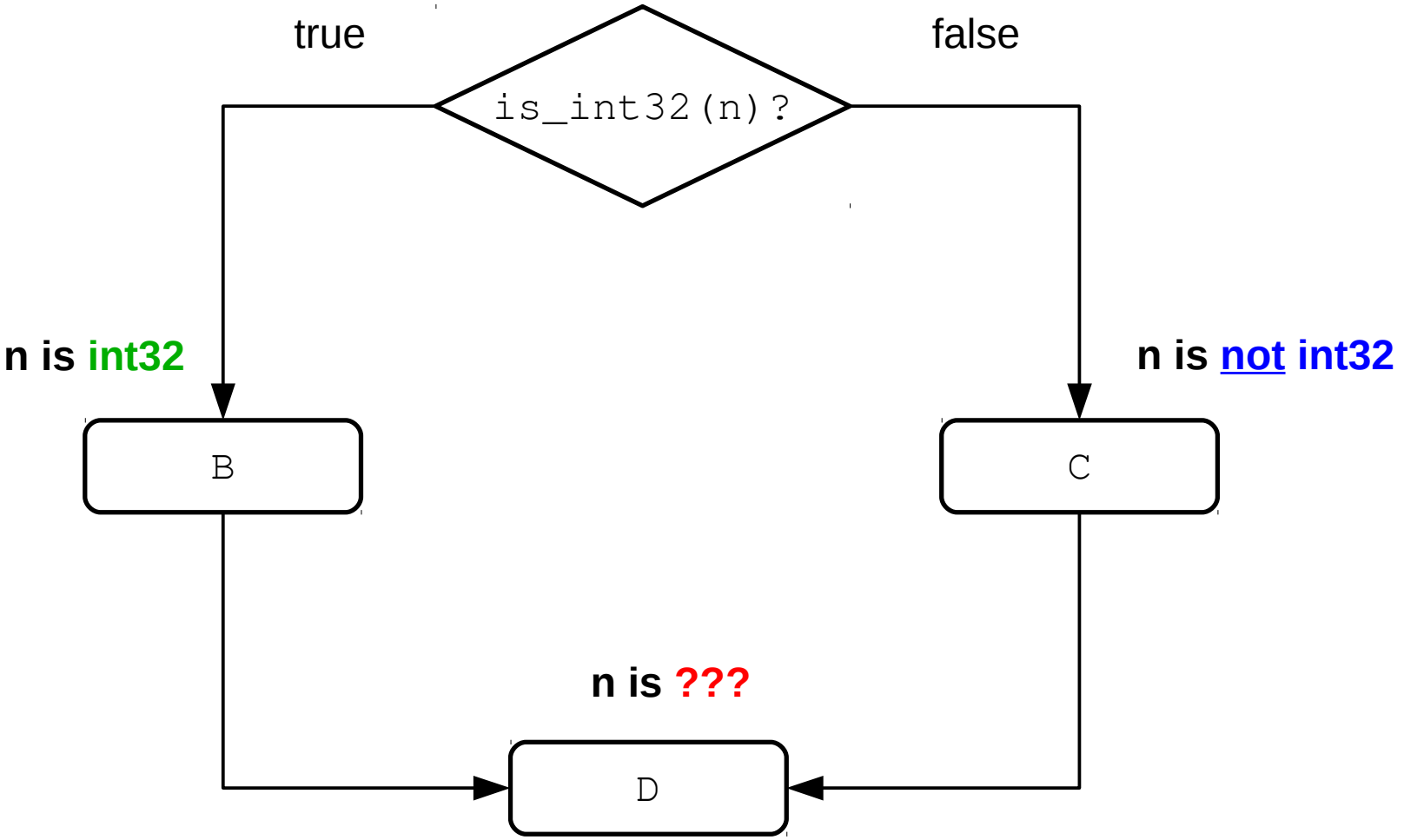
Tag	Description
int32 *	32-bit integer
float64	64-bit floating-point value
string	Immutable JS string
const	true, false or null
object	Plain JS object
array	JS array
function	JS function/closure

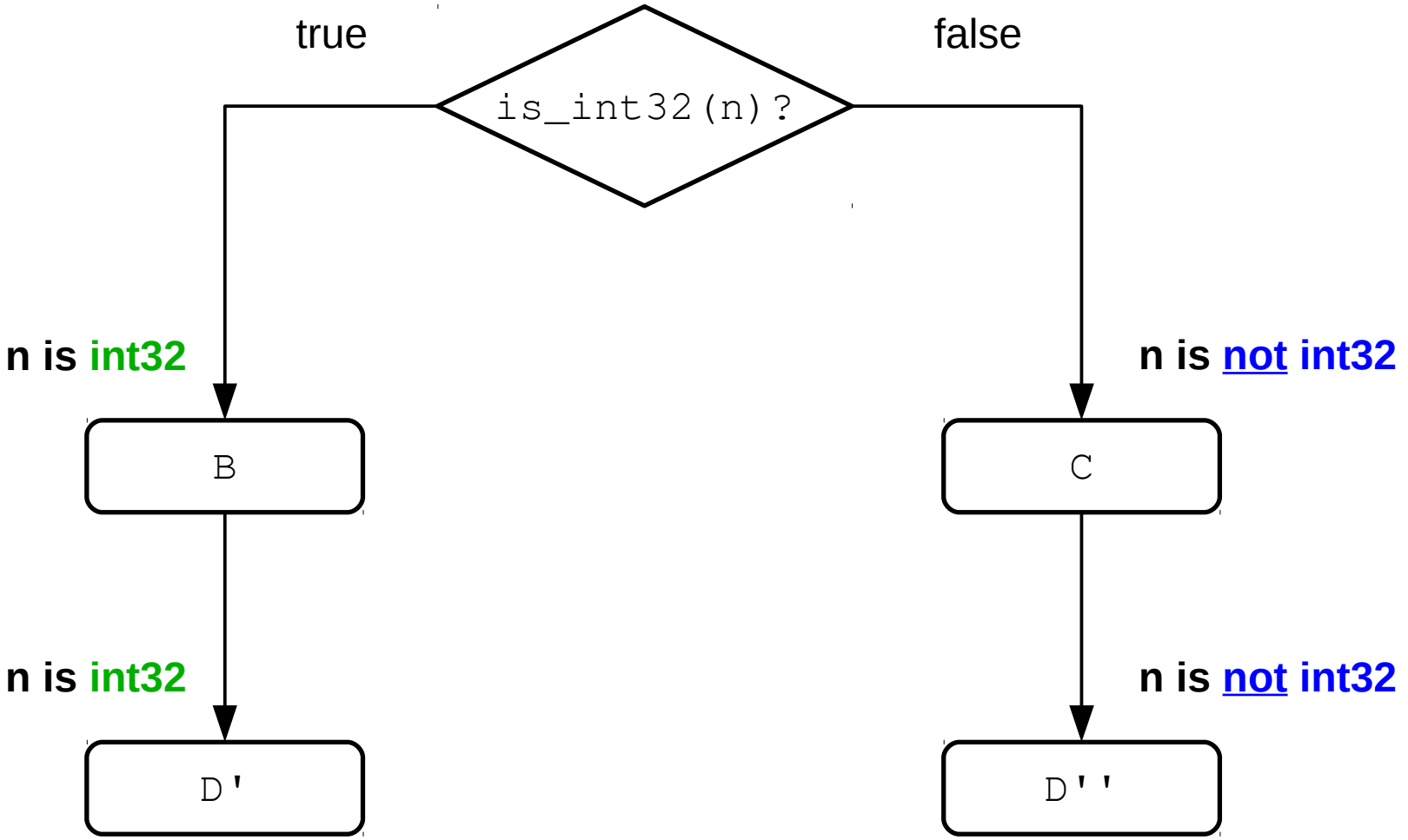
```
if (is_int32(n)) // A
{
    // B
    ...
}
else
{
    // C
    ...
}

// D
...
```


true
false
is_int32(n)?
B
C
D


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


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


true
false
is_int32(n)?
n is int32
n is not int32
B
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 = 0xABCDEF;
for (var i = 0; i < 600000; i++)
    v = v & i;
```

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

```
var v = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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 = 0xABCDEF;
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

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

Will this Explode?

- Obvious criticism: version explosion
- KISS: hard limit on versions per block
  - Guaranteed upper bound on code size
- After limit, look for best match
  - Minimize loss of type information
- We can do even better
  - Generate only versions needed at run-time

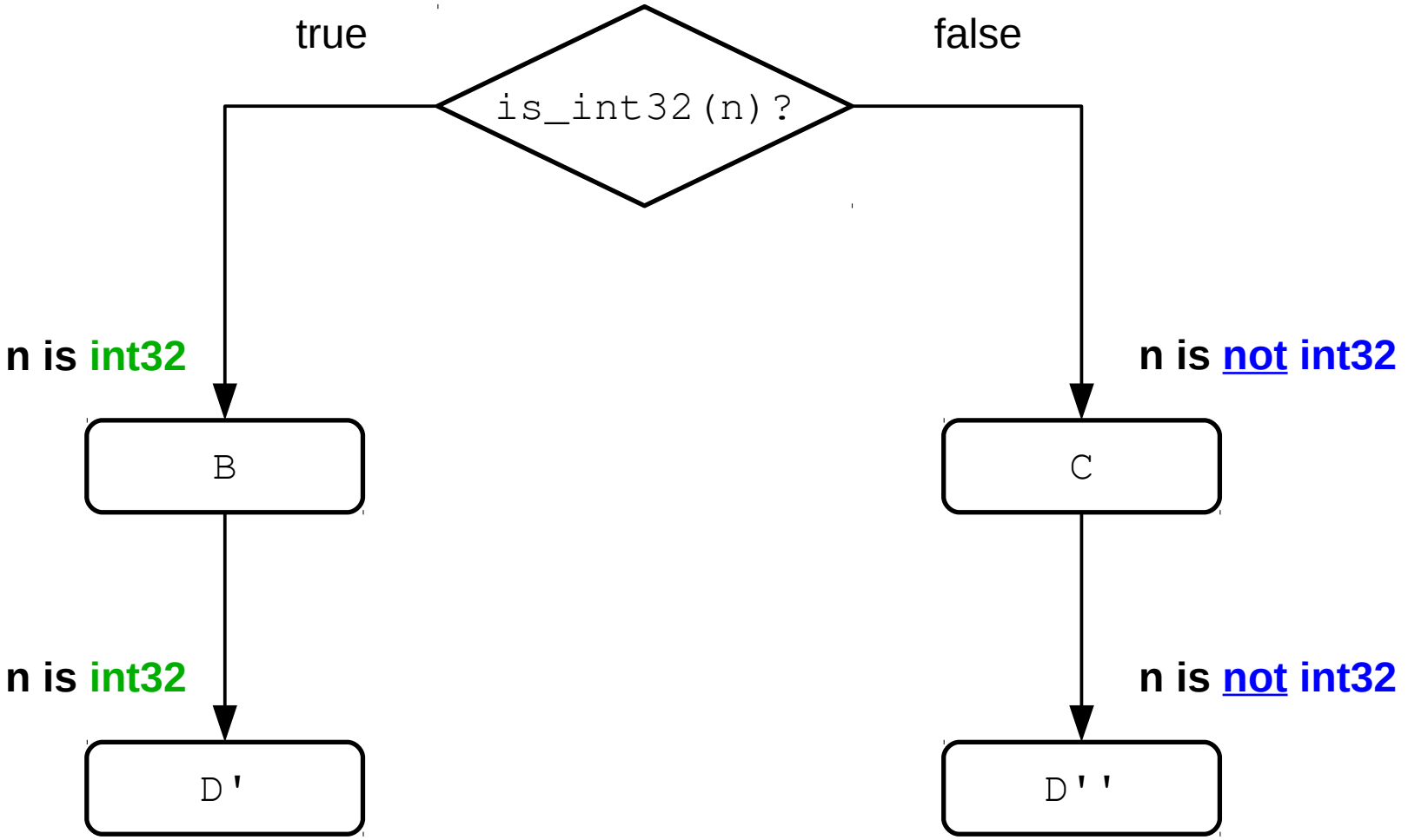
Lazy Basic Block Versioning

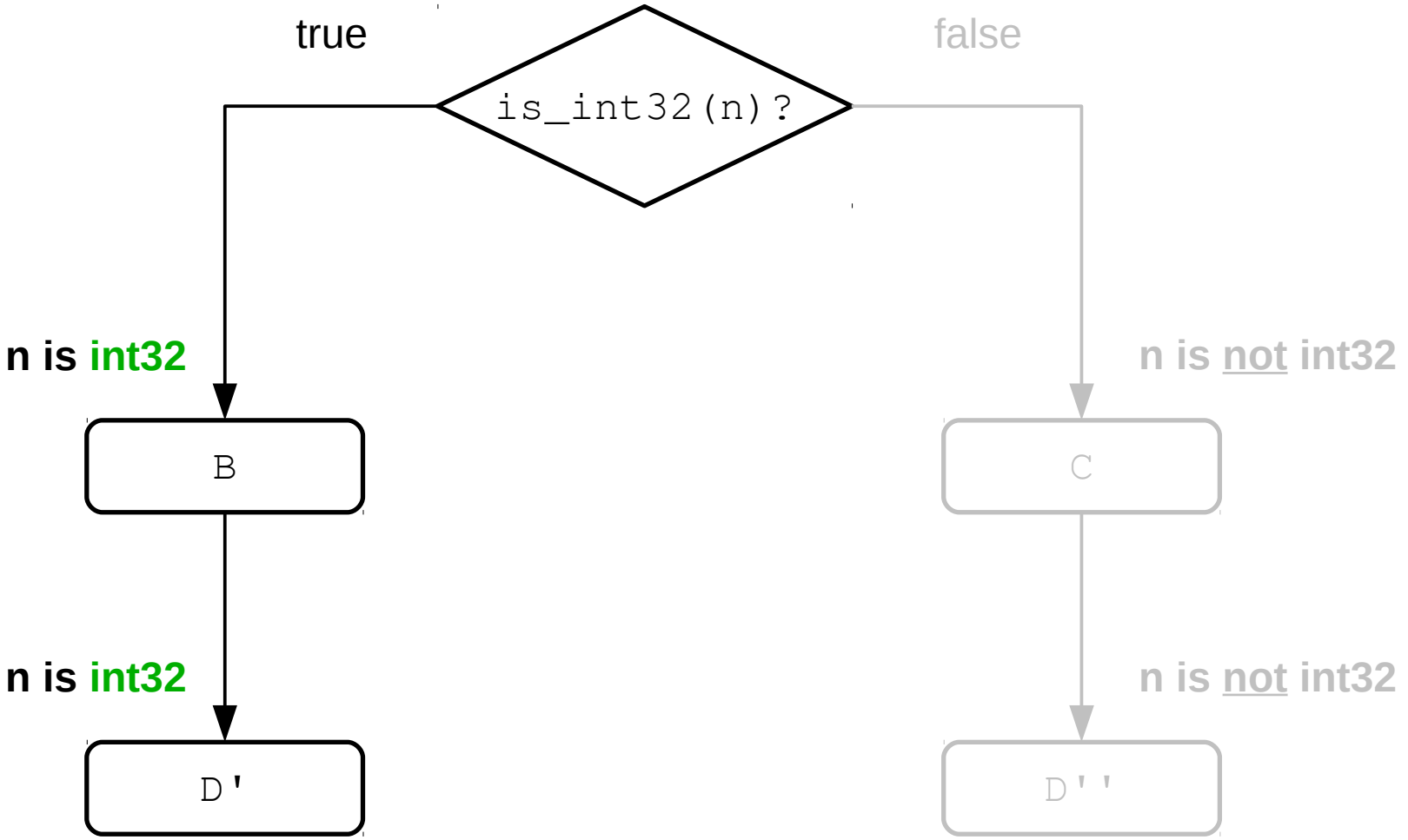
Eager Versioning is Impractical

- Generating versions eagerly is problematic
  - Expand CFG over all type combinations
  - Simply doesn't scale
- Predict what will be executed (heuristics)
  - Must remain conservative, overestimate
  - Generate versions that will never be used
  - Guess wrong, generate irrelevant versions

Lazy Basic Block Versioning

- Compile blocks lazily: when first executed
  - Only for types seen at run-time
  - The program's behavior drives versioning
  - Interleave compilation and execution
- 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''

Ceci n'est pas un tracing JIT

- 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 direction unknown, resume execution
  - Jumping to a stub resumes compilation
- Write code linearly and patch it

Incremental Codegen Example

```
function sumInts(n) // sum ints in [0,n[
{
    var sum = 0;
    for (var i = 0; i < n; i++)
        sum += i;

    return sum;
}
```

```
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);
```

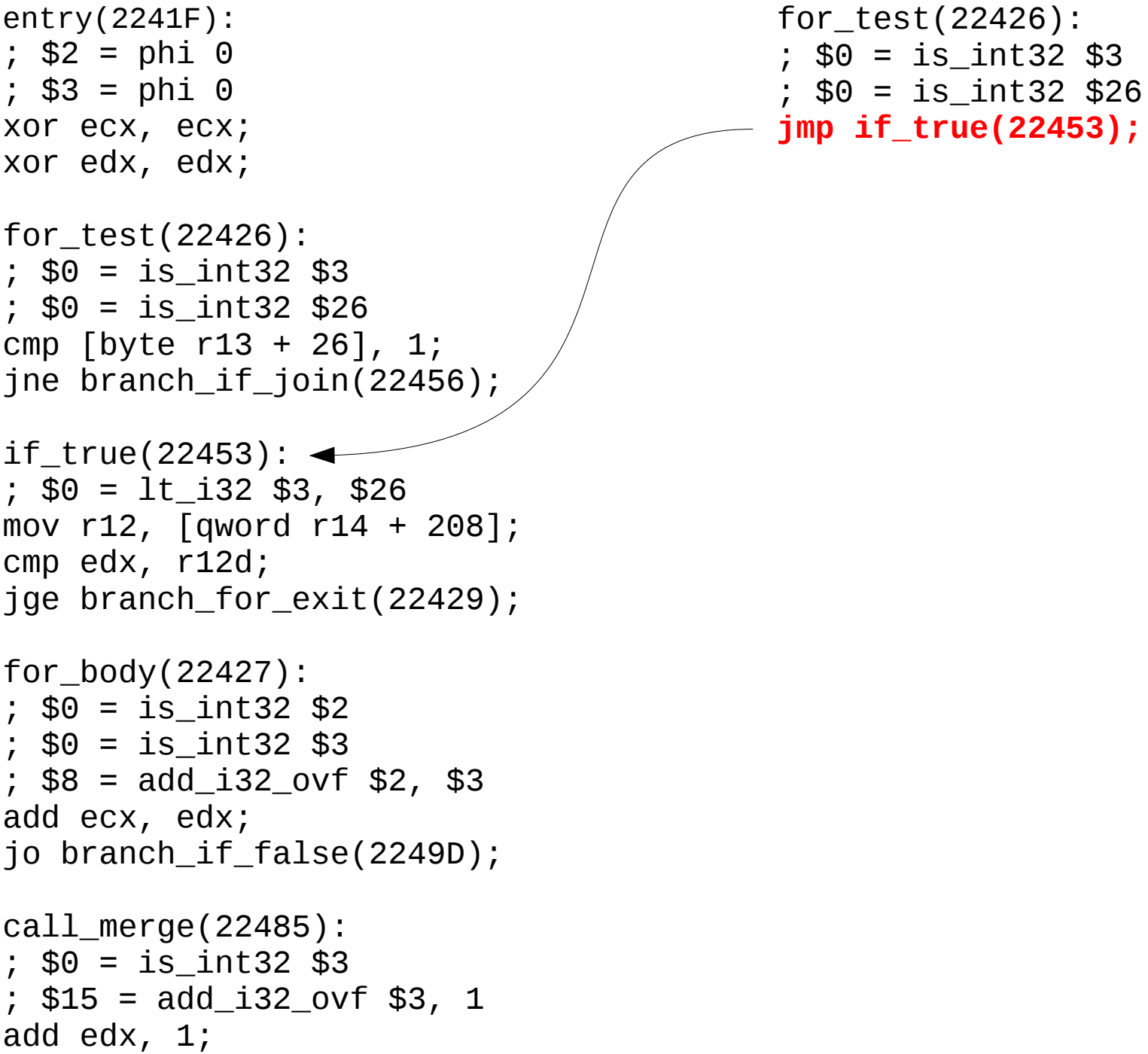
```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

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);

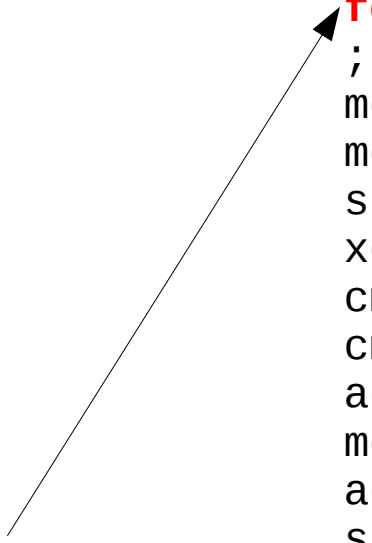
```
entry(2241F):
; $2 = phi 0
; $3 = phi 0
xor ecx, ecx;
xor edx, edx;

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;
```

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

    return sum;
}

sumInts(600);
```

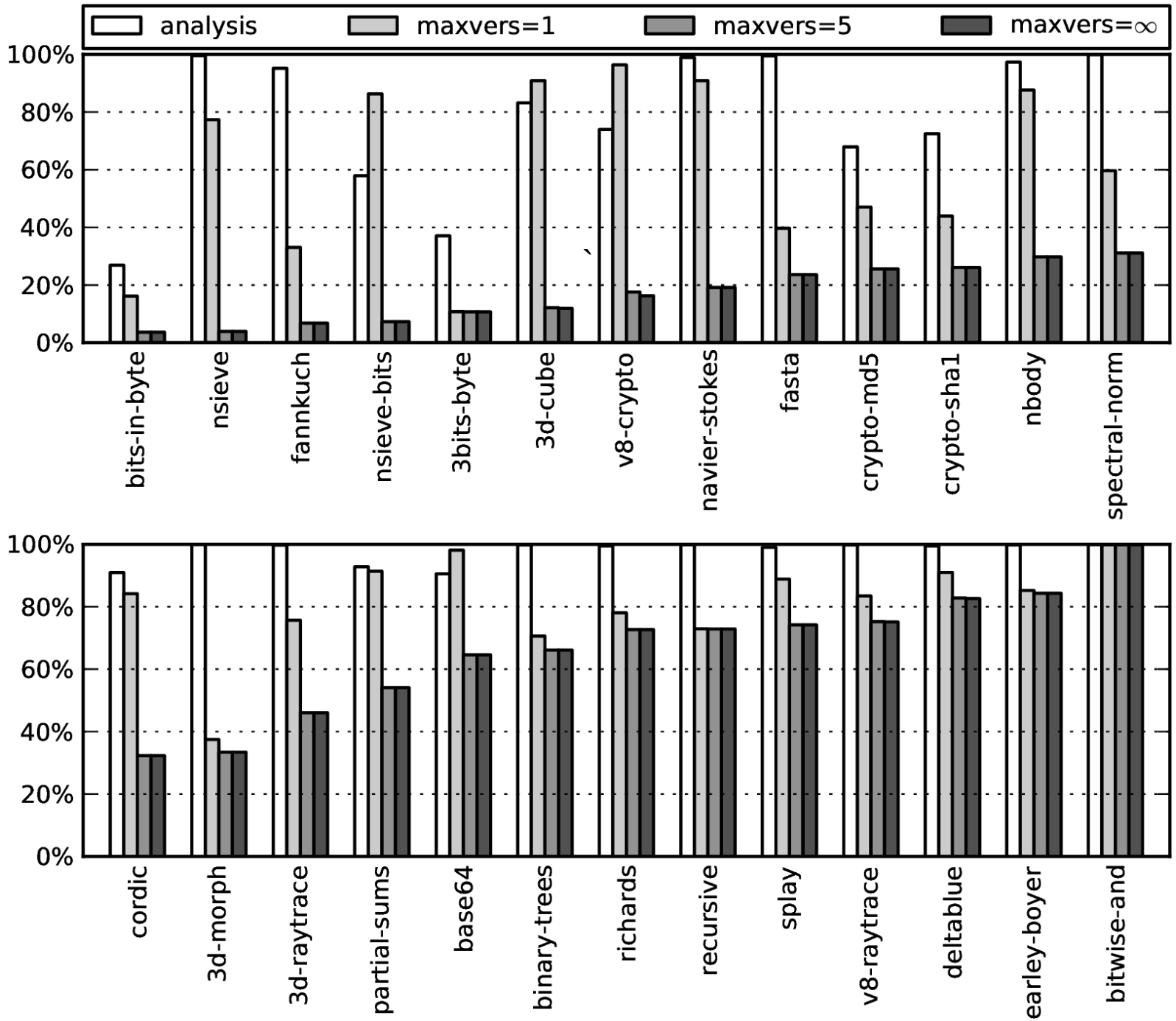
```
function sumInts(n) // n is ???
{
    var sum = 0;
    for (var i = 0; i < n; i++)
        sum += i;

    return sum;
}

sumInts(600);
```

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

    return sum;
}

sumInts(600);
```

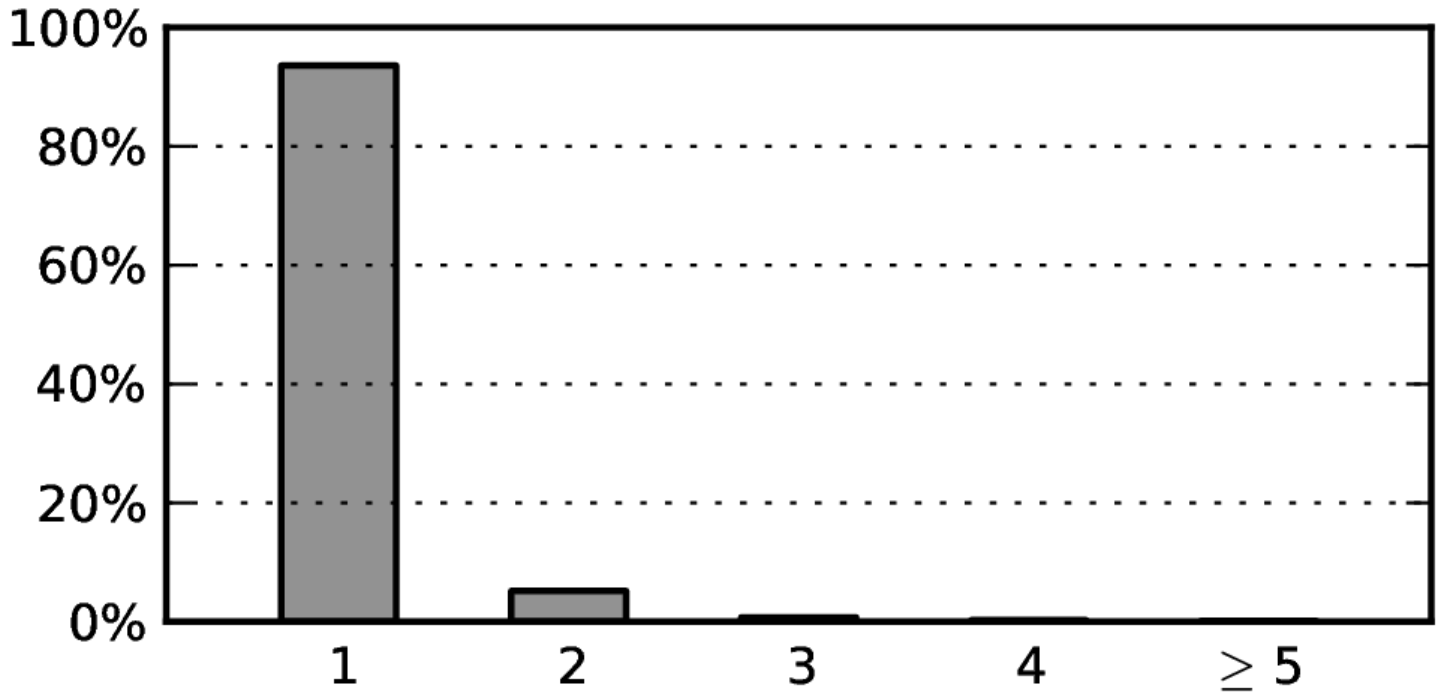
Experimental Results

Experimental Setup

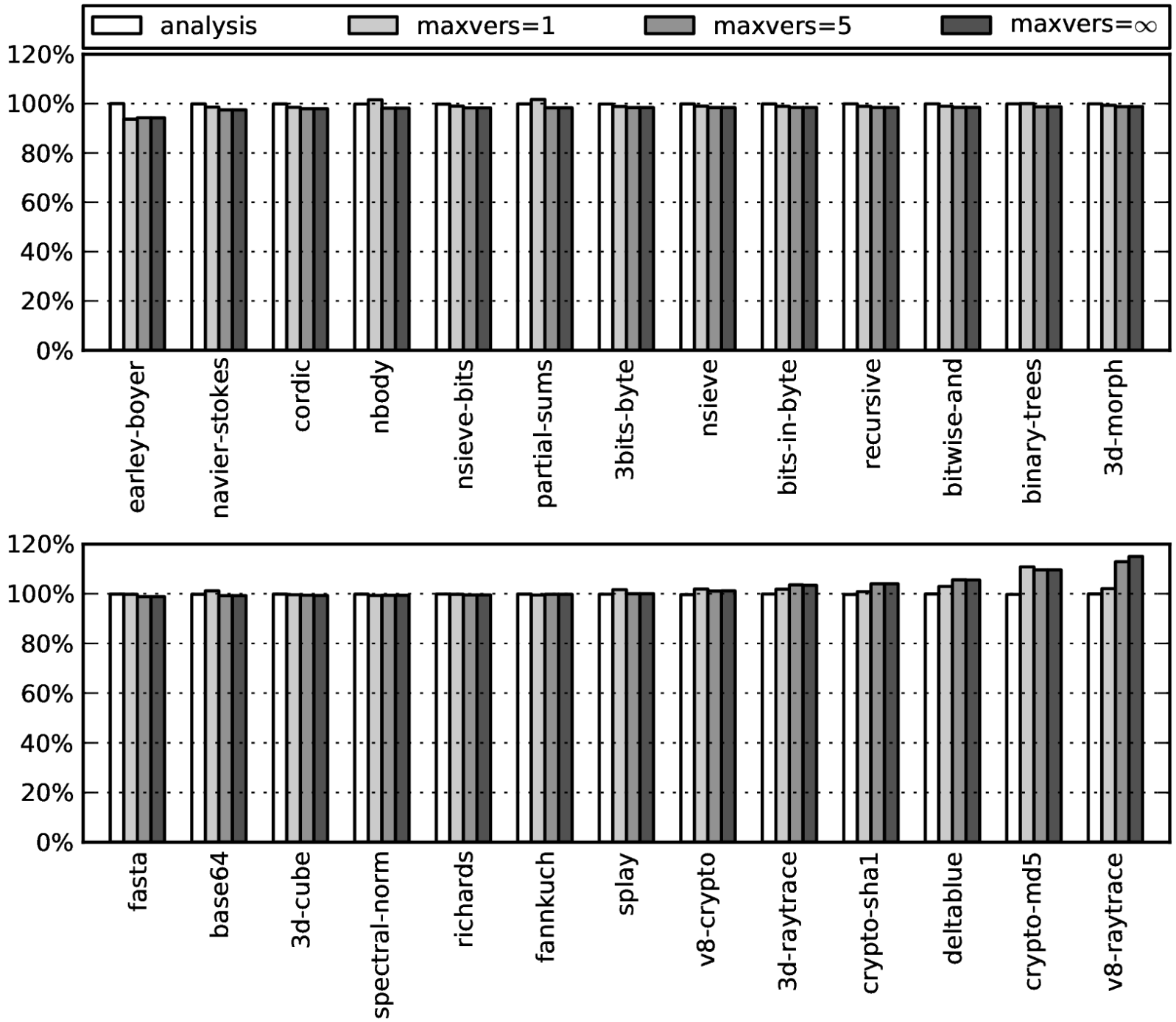
- 26 benchmarks from SunSpider and V8 suites
  - Excluded two which Higgs could not run
  - Excluded RegExp benchmarks
- Questions to answer:
  - Proportion of type tests eliminated by BBV
  - Impact on generated code size
  - Impact on execution time
  - Impact on compilation time
  - BBV vs fixed-point type analysis
  - BBV vs trace compilation



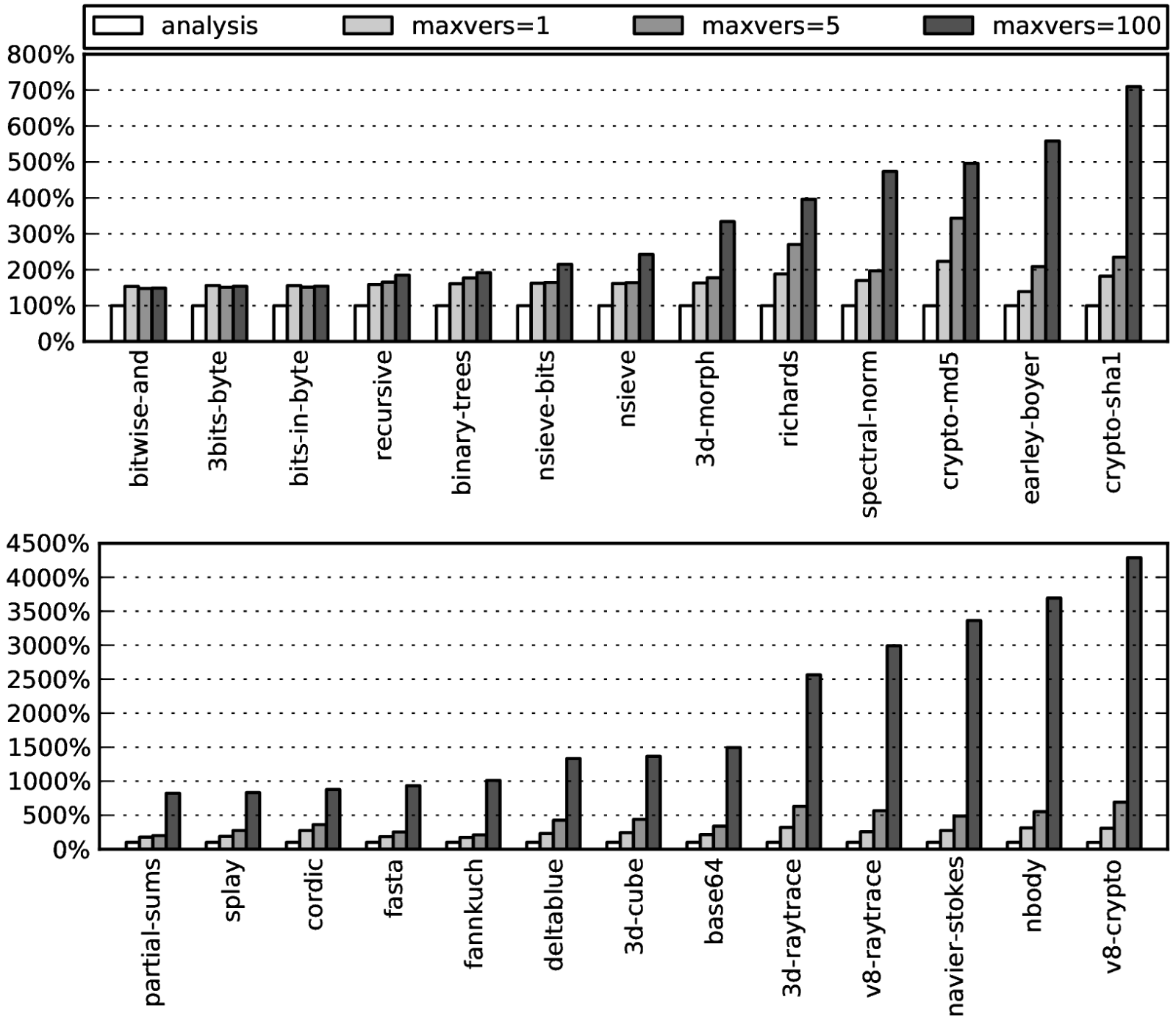
Dynamic counts of type tests executed using the representation analysis and lazy basic block versioning with various version limits (relative to baseline)



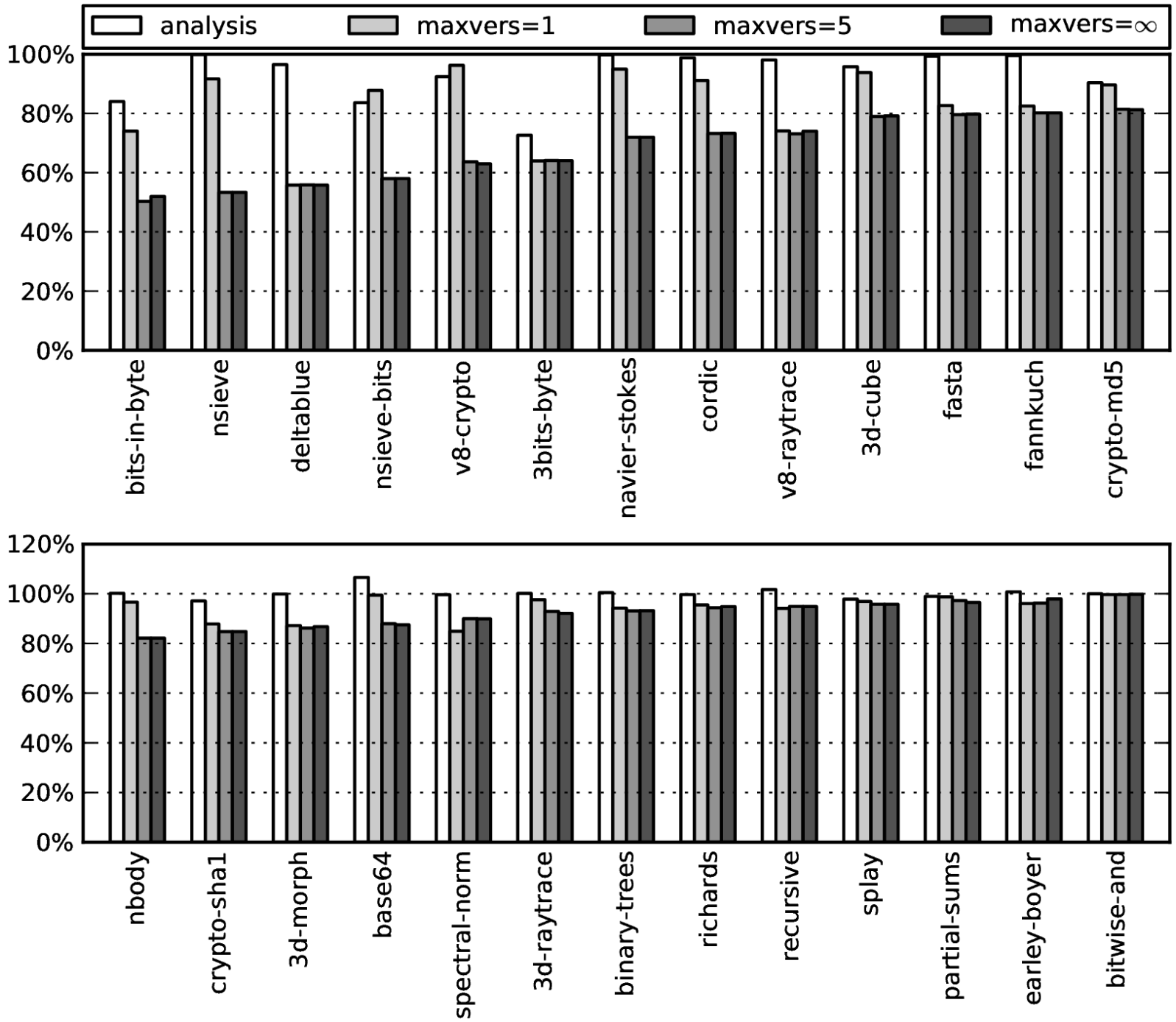
Relative occurrence of block version counts
(bucket counts)



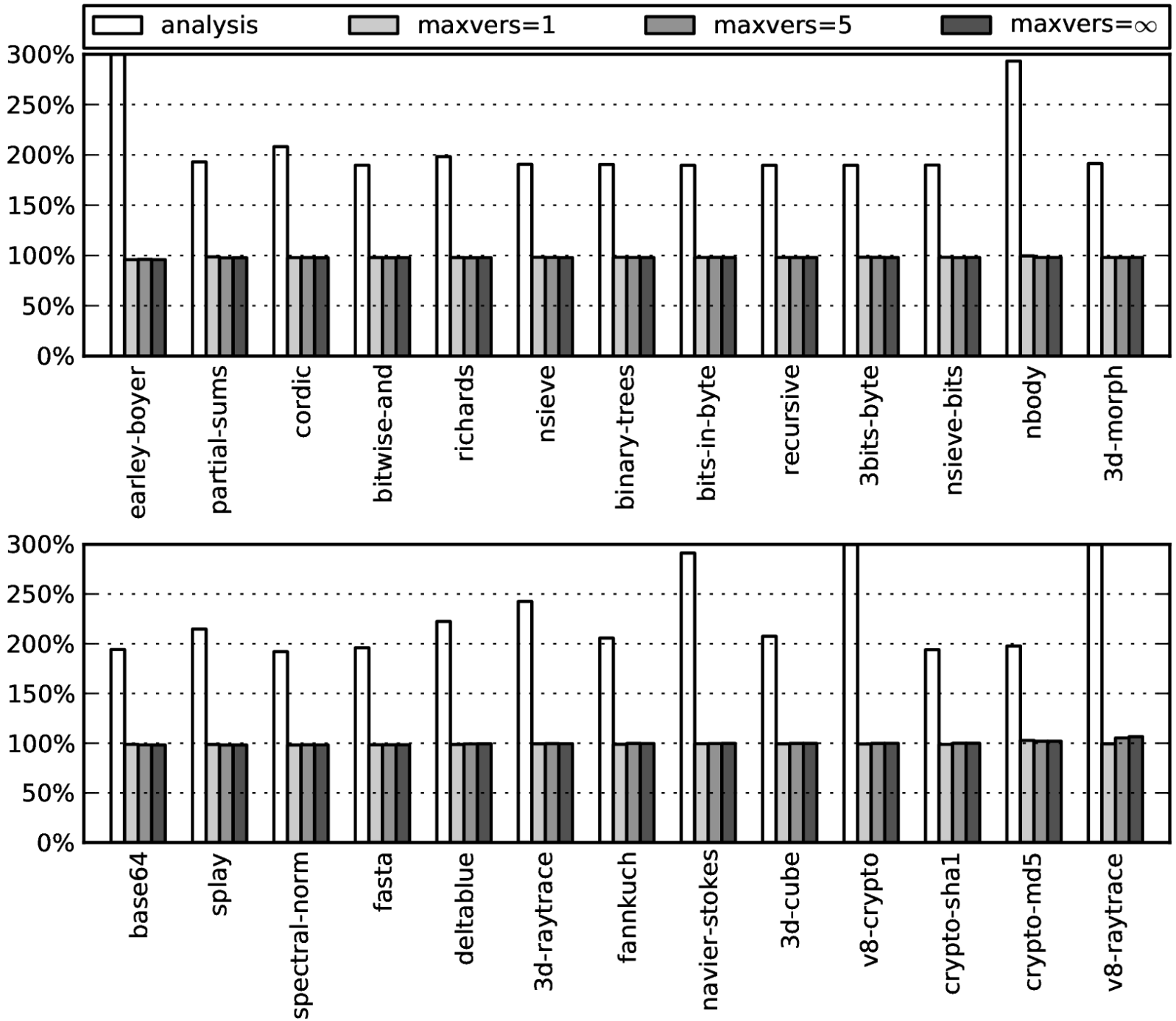
Code size for various block version limits (relative to baseline)



Code size with eager basic block versioning (relative to baseline)

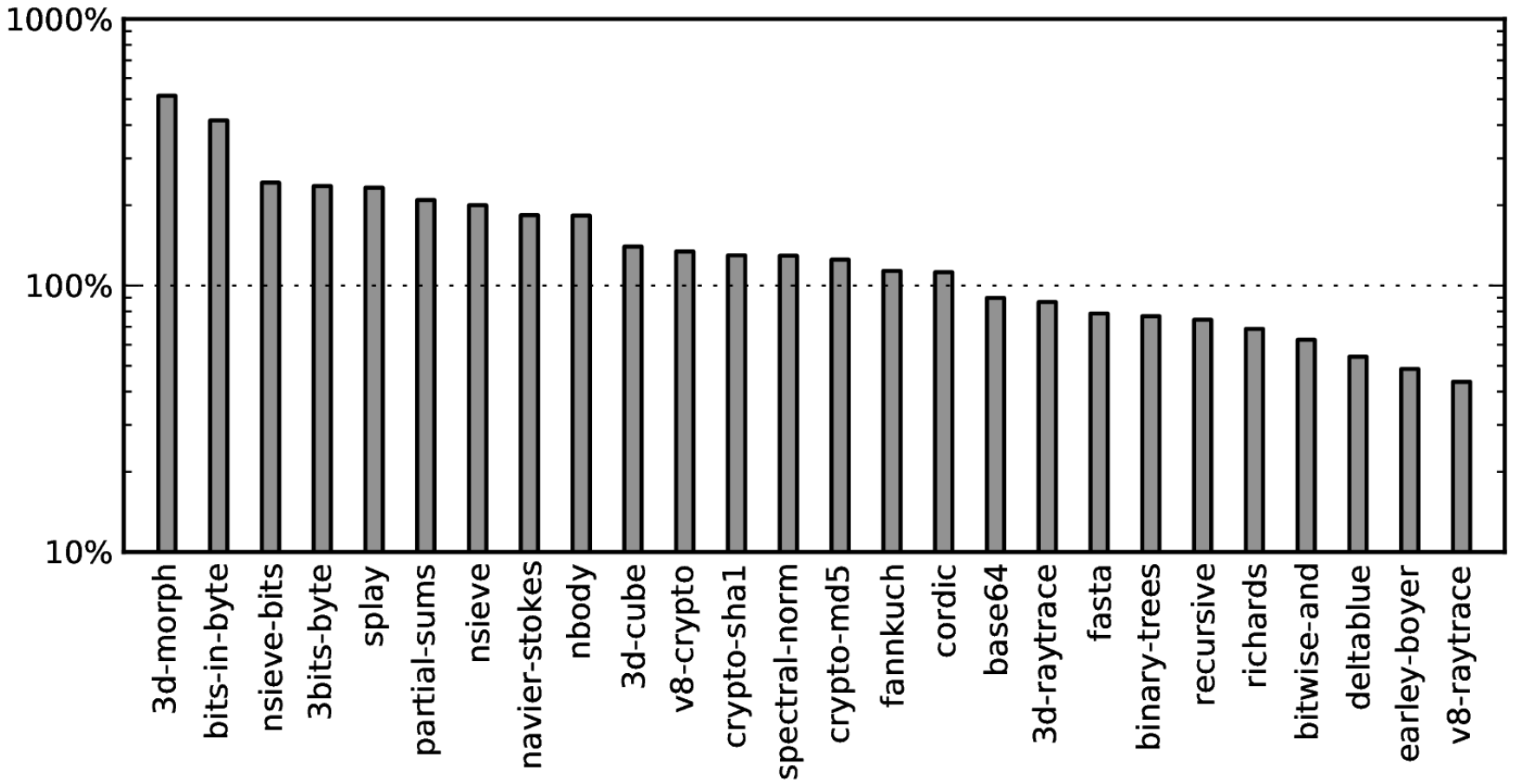


Execution time for various block version limits (relative to baseline)

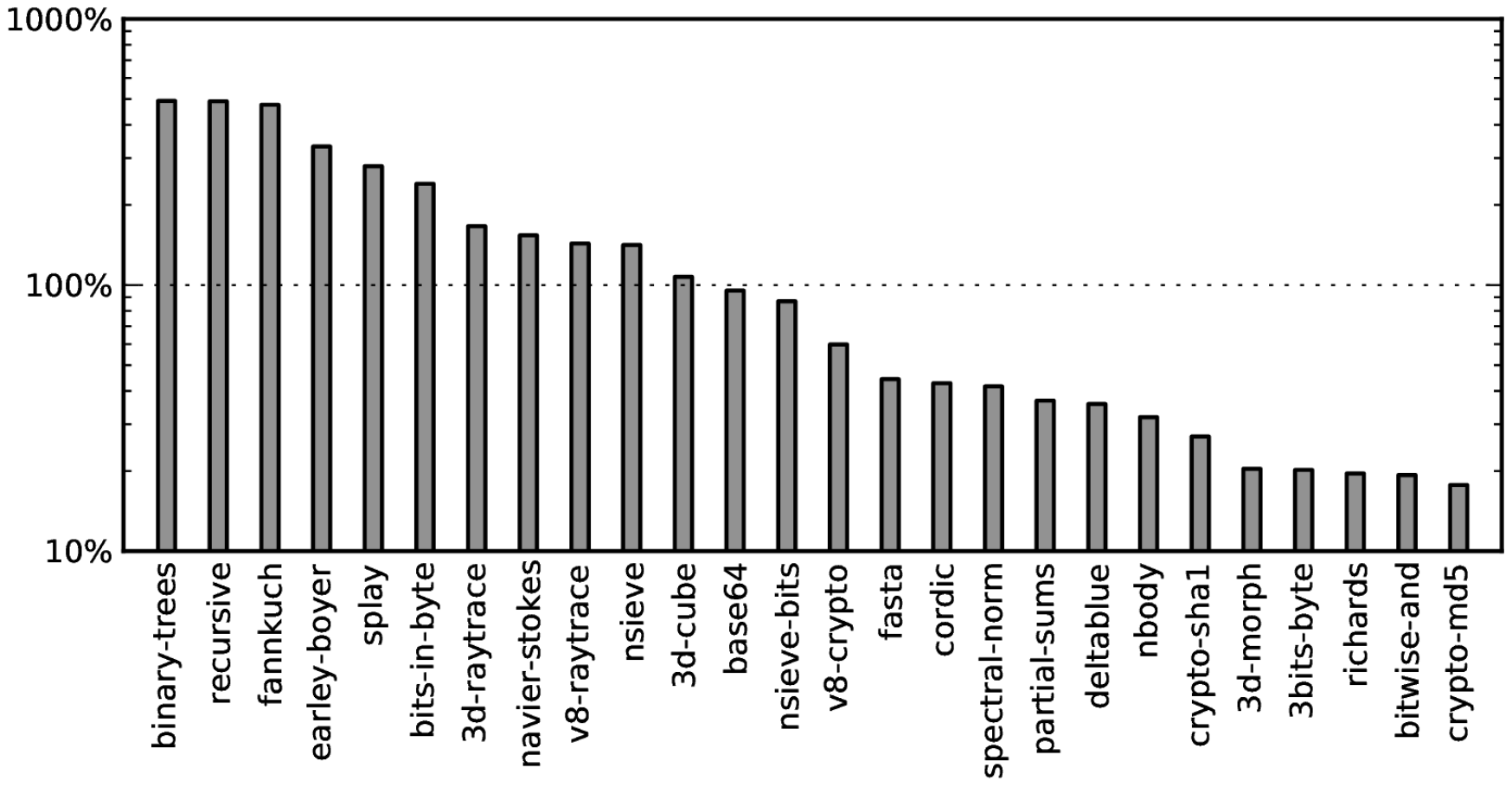


Compilation time for various block version limits (relative to baseline)

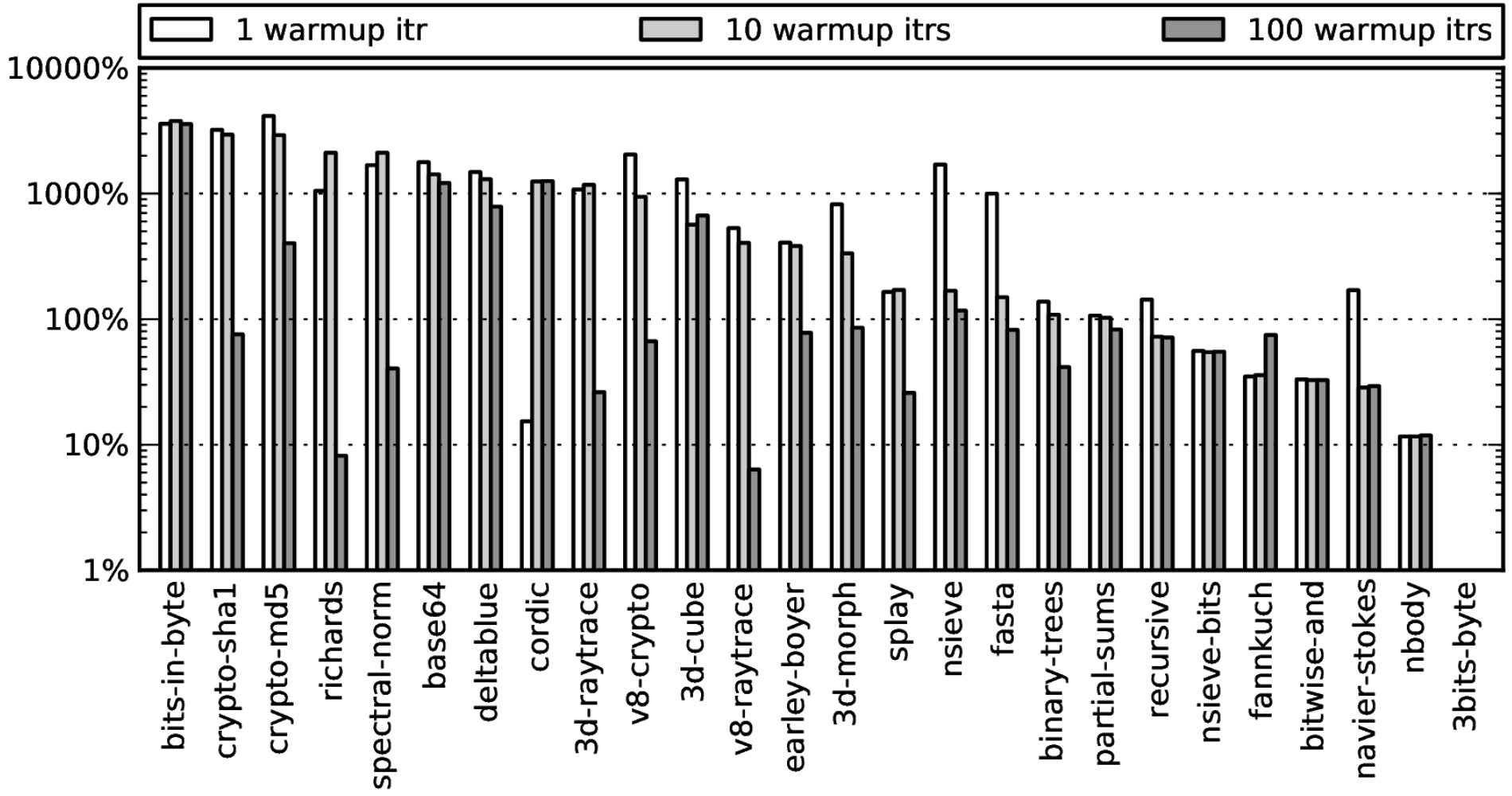
Comparative Performance



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



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



Speedup relative to Truffle/JS (log scale, higher is better)

Future Work

Typed Object Shapes

- Working with local variable types only
  - No awareness of obj property types
  - In JS, global variables are part of global obj
- Proposed solution:
  - Track object shapes as part of BBV
  - Encode property types in object shapes
  - Shape of an object gives us property types

Interprocedural Versioning

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

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)