Building JIT Compilers for Dynamic Languages with Low Development Effort

(and *relatively* good performance)

Baptiste Saleil
Université de Montréal
Montréal, Québec, Canada
baptiste.saleil@umontreal.ca

Marc Feeley
Université de Montréal
Montréal, Québec, Canada
feeley@iro.umontreal.ca

Université
de Montréal

Introduction

- Project started in 2013
- Basic Block Versioning (BBV)
  - *Simple and Effective Type Check Removal through Lazy Basic Block Versioning*
    Chevalier-Boisvert & Feeley - ECOOP 2015
  - Simple technique, a single compilation pass
- Explore ideas on Basic Block Versioning
- Explore simple and efficient implementations

Goals

- Implement BBV using a simple architecture
  - AST to machine code generally presented as the **simplest** approach
  - Why not for an optimizing JIT ?

  → Do not use any intermediate representation

- Use optimizations for good performance
  - Without complexifying the architecture
  - With simplicity in mind

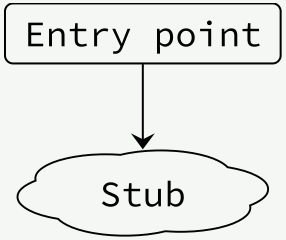
  → limit the use of static analysis

- What performance ?

Basic Block Versioning

Basic Block Versioning

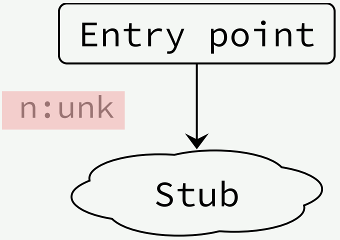
```
(define (abs n)
  (if (< n 0)
      (* n -1)
      n))
```



```
(abs -42)
(abs -3.14)
```

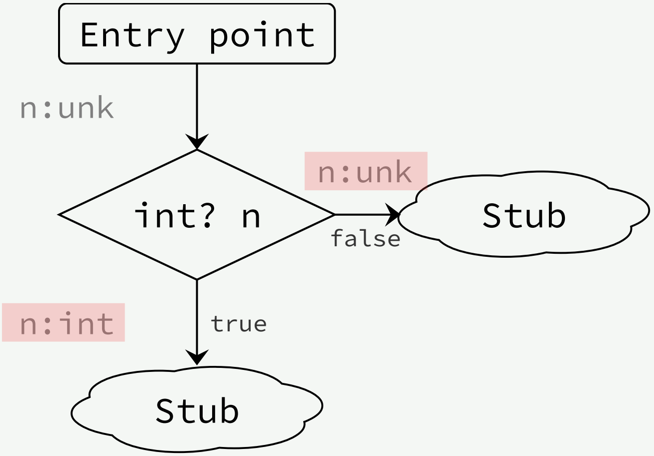
Basic Block Versioning

```
(define (abs n)
  (if (< n 0)
      (* n -1)
      n))

(abs -42)
(abs -3.14)
```



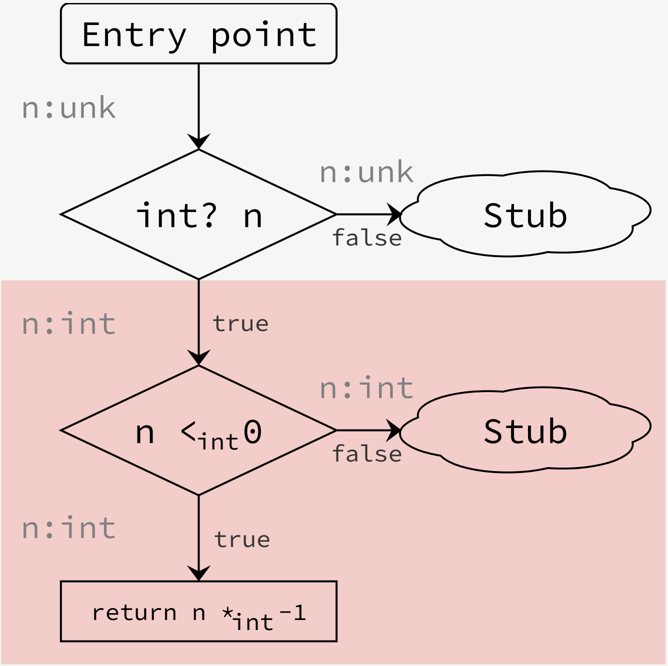
Basic Block Versioning

```
(define (abs n)
  (if (< n 0)
      (* n -1)
      n))

(abs -42)
(abs -3.14)
```



Basic Block Versioning

```
(define (abs n)
  (if (< n 0)
      (* n -1)
      n))

(abs -42)
(abs -3.14)
```

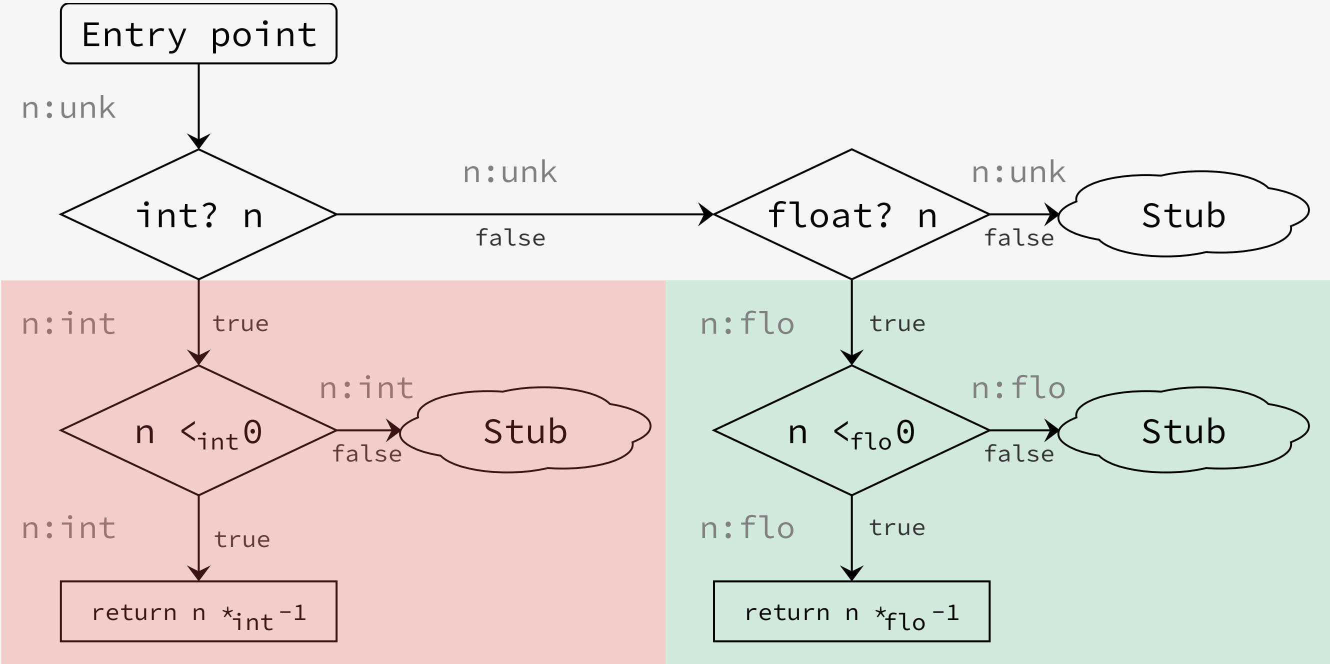


- Specialized for n:int
- No more type check

Basic Block Versioning

```
(define (abs n)
  (if (< n 0)
      (* n -1)
      n))
```

```
(abs -42)
(abs -3.14)
```

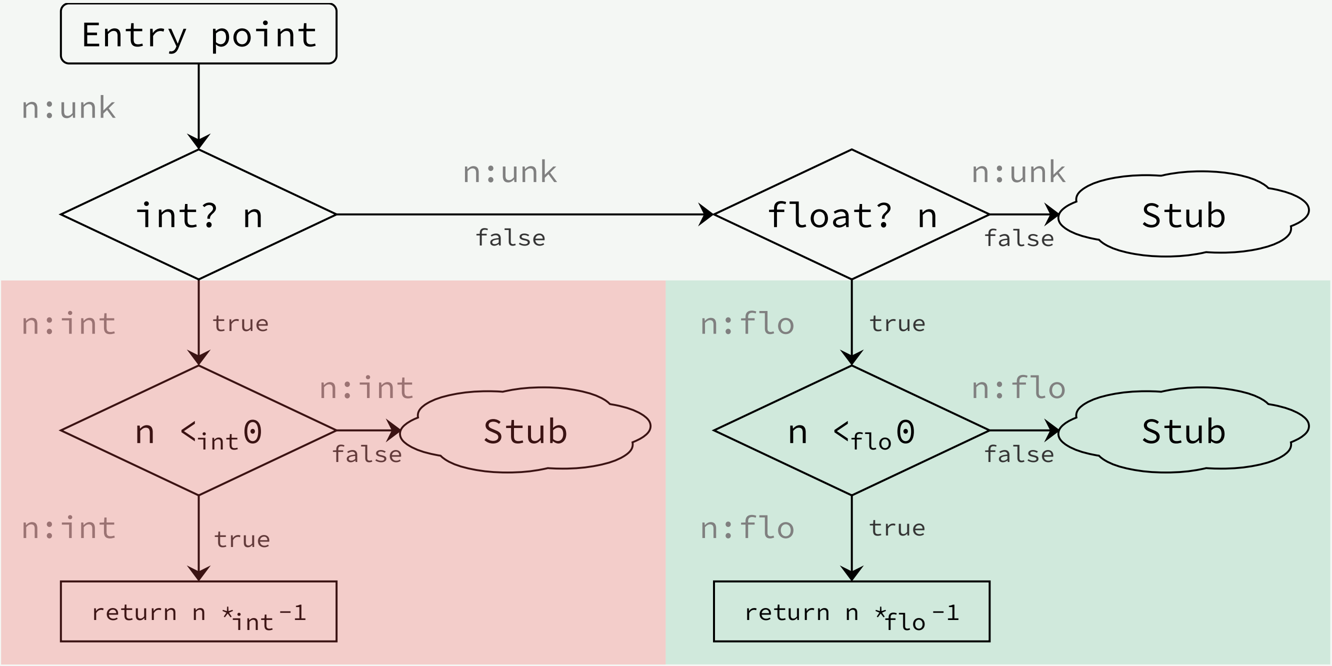


- Specialized for n:int
- No more type check

- Specialized for n:flo
- No more type check

Basic Block Versioning

```
(define (abs n)
  (if (< n 0)
      (* n -1)
      n))

(abs -42)
(abs -3.14)
```



- Specialized for n:int
- No more type check

- Specialized for n:flo
- No more type check

1. Lazy compilation of basic blocks
2. With code specialization

Apply BBV on AST

Example grammar

```
<E>    ::= <Id>
         | <Cst>
         | (set! <Id> <E>)
         | (if <E> <E> <E>)

<Id>   ::= [a-z]+

<Cst>  ::= <Int>
         | <Bool>

<Int>  ::= [0-9]+

<Bool> ::= #t
         | #f
```

Code generation from AST

```
(define (gen-expr expr)

  (match expr

    (,c when (constant? c)
     (gen-instr `(push ,c)))

    (,v when (variable? v)
     (gen-instr `(push ,v)))

    ((set! ,v ,E1) when (variable? v)
     (gen-expr E1)
     (gen-instr `(store ,v))
     (gen-instr `(push #f)))

    ((if ,E1 ,E2 ,E3)
     (gen-expr E1)
     (let ((instr1 (gen-instr `(iffalse ???))))
       (gen-expr E2)
       (let ((instr2 (gen-instr `(goto ???))))
         (comefrom instr1)
         (gen-expr E3)
         (comefrom instr2))))

    (else
     (error "unknown" expr))))
```

Code generation from AST

```
(define (gen-expr expr)

  (match expr

    (,c when (constant? c)          } e.g. push 42
     (gen-instr `(push ,c)))

    (,v when (variable? v)          } e.g. push n
     (gen-instr `(push ,v)))

    ((set! ,v ,E1) when (variable? v)
     (gen-expr E1)
     (gen-instr `(store ,v))
     (gen-instr `(push #f)))

    ((if ,E1 ,E2 ,E3)
     (gen-expr E1)
     (let ((instr1 (gen-instr `(iffalse ???))))
       (gen-expr E2)
       (let ((instr2 (gen-instr `(goto ???))))
         (comefrom instr1)
         (gen-expr E3)
         (comefrom instr2))))

    (else
     (error "unknown" expr))))
```

Code generation from AST

```
(define (gen-expr expr)

  (match expr

    (,c when (constant? c)
     (gen-instr `(push ,c)))

    (,v when (variable? v)
     (gen-instr `(push ,v)))

    ((set! ,v ,E1) when (variable? v)
     (gen-expr E1)
     (gen-instr `(store ,v))
     (gen-instr `(push #f)))

    ((if ,E1 ,E2 ,E3)
     (gen-expr E1)
     (let ((instr1 (gen-instr `(iffalse ???))))
       (gen-expr E2)
       (let ((instr2 (gen-instr `(goto ???))))
         (comefrom instr1)
         (gen-expr E3)
         (comefrom instr2))))

    (else
     (error "unknown" expr))))
```

```
      push 42
e.g.  store n
      push #f
```

Code generation from AST

```
(define (gen-expr expr)

  (match expr

    (,c when (constant? c)
     (gen-instr `(push ,c)))

    (,v when (variable? v)
     (gen-instr `(push ,v)))

    ((set! ,v ,E1) when (variable? v)
     (gen-expr E1)
     (gen-instr `(store ,v))
     (gen-instr `(push #f)))

    ((if ,E1 ,E2 ,E3)
     (gen-expr E1)
     (let ((instr1 (gen-instr `(iffalse ???))))
       (gen-expr E2)
       (let ((instr2 (gen-instr `(goto ???))))
         (comefrom instr1)
         (gen-expr E3)
         (comefrom instr2))))

    (else
     (error "unknown" expr))))
```

e.g.

```
push n
iffalse 3
push 1
goto 2
push 0
```

Code generation from AST

```
(define (gen-expr expr)

  (match expr

    (,c when (constant? c)
     (gen-instr `(push ,c)))

    (,v when (variable? v)
     (gen-instr `(push ,v)))

    ((set! ,v ,E1) when (variable? v)
     (gen-expr E1)
     (gen-instr `(store ,v))
     (gen-instr `(push #f)))

    ((if ,E1 ,E2 ,E3)
     (gen-expr E1)
     (let ((instr1 (gen-instr `(iffalse ???))))
       (gen-expr E2)
       (let ((instr2 (gen-instr `(goto ???))))
         (comefrom instr1)
         (gen-expr E3)
         (comefrom instr2))))

    (else
     (error "unknown" expr))))
```

Example:

```
(gen-expr (read))
(gen-instr `(return))
```

Code generation from AST

```
(define (gen-expr expr)

  (match expr

    (,c when (constant? c)
     (gen-instr `(push ,c)))

    (,v when (variable? v)
     (gen-instr `(push ,v)))

    ((set! ,v ,E1) when (variable? v)
     (gen-expr E1)
     (gen-instr `(store ,v))
     (gen-instr `(push #f)))

    ((if ,E1 ,E2 ,E3)
     (gen-expr E1)
     (let ((instr1 (gen-instr `(iffalse ???))))
       (gen-expr E2)
       (let ((instr2 (gen-instr `(goto ???))))
         (comefrom instr1)
         (gen-expr E3)
         (comefrom instr2))))

    (else
     (error "unknown" expr))))
```

Example:

```
(gen-expr (read))
(gen-instr `(return))
```

✗ Lazy compilation

✗ Code specialization

JIT Compilation from AST

```
(define (gen-expr expr cont)

  (match expr

    (,c when (constant? c)
     (lambda ()
       (gen-instr `(push ,c))
       (cont)))

    (,v when (variable? v)
     (lambda ()
       (gen-instr `(push ,v))
       (cont)))

    ((set! ,v ,E1) when (variable? v)
     (let ((scont (lambda ()
                    (gen-instr `(store ,v))
                    (gen-instr `(push #f))
                    (cont))))
       (gen-expr E1 scont)))

    ((if ,E1 ,E2 ,E3)
     (let* ((stub-false (make-stub E3 cont))
            (stub-true  (make-stub E2 cont))
            (ccont (lambda ()
                     (gen-instr `(iffalse ,stub-false))
                     (gen-instr `(goto ,stub-true)))))
       (gen-expr E1 ccont)))

    (else
     (error "unknown" expr))))
```

JIT Compilation from AST

```
(define (gen-expr expr cont)

  (match expr

    (,c when (constant? c)
     (lambda ()
       (gen-instr `(push ,c))
       (cont)))

    (,v when (variable? v)
     (lambda ()
       (gen-instr `(push ,v))
       (cont)))

    ((set! ,v ,E1) when (variable? v)
     (let ((scont (lambda ()
                    (gen-instr `(store ,v))
                    (gen-instr `(push #f))
                    (cont))))
       (gen-expr E1 scont)))

    ((if ,E1 ,E2 ,E3)
     (let* ((stub-false (make-stub E3 cont))
            (stub-true  (make-stub E2 cont))
            (ccont (lambda ()
                     (gen-instr `(iffalse ,stub-false))
                     (gen-instr `(goto ,stub-true)))))
       (gen-expr E1 ccont)))

    (else
     (error "unknown" expr))))
```

JIT Compilation from AST

```
(define (gen-expr expr cont)

  (match expr

    (,c when (constant? c)
     (lambda ()
       (gen-instr `(push ,c))
       (cont)))

    (,v when (variable? v)
     (lambda ()
       (gen-instr `(push ,v))
       (cont)))

    ((set! ,v ,E1) when (variable? v)
     (let ((scont (lambda ()
                    (gen-instr `(store ,v))
                    (gen-instr `(push #f))
                    (cont))))
       (gen-expr E1 scont)))

    ((if ,E1 ,E2 ,E3)
     (let* ((stub-false (make-stub E3 cont))
            (stub-true  (make-stub E2 cont))
            (ccont (lambda ()
                     (gen-instr `(iffalse ,stub-false))
                     (gen-instr `(goto ,stub-true)))))
       (gen-expr E1 ccont)))

    (else
     (error "unknown" expr))))
```

JIT Compilation from AST

```
(define (gen-expr expr cont)

  (match expr

    (,c when (constant? c)
     (lambda ()
       (gen-instr `(push ,c))
       (cont)))

    (,v when (variable? v)
     (lambda ()
       (gen-instr `(push ,v))
       (cont)))

    ((set! ,v ,E1) when (variable? v)
     (let ((scont (lambda ()
                    (gen-instr `(store ,v))
                    (gen-instr `(push #f))
                    (cont))))
       (gen-expr E1 scont)))

    ((if ,E1 ,E2 ,E3)
     (let* ((stub-false (make-stub E3 cont))
            (stub-true  (make-stub E2 cont))
            (ccont (lambda ()
                     (gen-instr `(iffalse ,stub-false))
                     (gen-instr `(goto ,stub-true)))))
       (gen-expr E1 ccont)))

    (else
     (error "unknown" expr))))
```

Example:

```
(gen-expr
  (read)
  (lambda ()
    (gen-instr `(return))))
```

JIT Compilation from AST

```
(define (gen-expr expr cont)

  (match expr

    (,c when (constant? c)
     (lambda ()
       (gen-instr `(push ,c))
       (cont)))

    (,v when (variable? v)
     (lambda ()
       (gen-instr `(push ,v))
       (cont)))

    ((set! ,v ,E1) when (variable? v)
     (let ((scont (lambda ()
                    (gen-instr `(store ,v))
                    (gen-instr `(push #f))
                    (cont))))
       (gen-expr E1 scont)))

    ((if ,E1 ,E2 ,E3)
     (let* ((stub-false (make-stub E3 cont))
            (stub-true  (make-stub E2 cont))
            (ccont (lambda ()
                     (gen-instr `(iffalse ,stub-false))
                     (gen-instr `(goto ,stub-true)))))
       (gen-expr E1 ccont)))

    (else
     (error "unknown" expr))))
```

Example:

```
(gen-expr
  (read)
  (lambda ()
    (gen-instr `(return))))
```

✓ Lazy compilation

✗ Code specialization

Code specialization from AST

```
(define (gen-expr expr cont)

  (match expr

    (,c when (constant? c)
     (lambda (ctx)
       (gen-instr `(push ,c))
       (let ((type (if (integer? c) 't_int 't_bool)))
         (call-cont cont (ctx-push ctx type)))))

    (,v when (variable? v)
     (lambda (ctx)
       (gen-instr `(push ,v))
       (let ((type (ctx-get-type ctx v)))
         (call-cont cont (ctx-push ctx type)))))

    ((set! ,v ,E1) when (variable? v)
     (let ((scont
             (lambda (ctx)
               (gen-instr `(store ,v))
               (gen-instr `(push #f))
               (let* ((type (ctx-top ctx))
                      (ctx  (ctx-pop ctx))
                      (ctx  (ctx-push ctx 't_bool))
                      (ctx  (ctx-set-type ctx v type)))
                 (call-cont cont ctx)))))
       (gen-expr E1 scont)))

    ...

    (else
     (error "unknown" expr))))
```

Code specialization from AST

```
(define (gen-expr expr cont)

  (match expr

    (,c when (constant? c)
     (lambda (ctx)
       (gen-instr `(push ,c))
       (let ((type (if (integer? c) 't_int 't_bool)))
         (call-cont cont (ctx-push ctx type)))))

    (,v when (variable? v)
     (lambda (ctx)
       (gen-instr `(push ,v))
       (let ((type (ctx-get-type ctx v)))
         (call-cont cont (ctx-push ctx type)))))

    ((set! ,v ,E1) when (variable? v)
     (let ((scont
             (lambda (ctx)
               (gen-instr `(store ,v))
               (gen-instr `(push #f))
               (let* ((type (ctx-top ctx))
                      (ctx  (ctx-pop ctx))
                      (ctx  (ctx-push ctx 't_bool))
                      (ctx  (ctx-set-type ctx v type)))
                 (call-cont cont ctx)))))
       (gen-expr E1 scont)))

    ...

    (else
     (error "unknown" expr))))
```

Code specialization from AST

```
(define (gen-expr expr cont)

  (match expr

    (,c when (constant? c)
     (lambda (ctx)
       (gen-instr `(push ,c))
       (let ((type (if (integer? c) 't_int 't_bool)))
         (call-cont cont (ctx-push ctx type)))))

    (,v when (variable? v)
     (lambda (ctx)
       (gen-instr `(push ,v))
       (let ((type (ctx-get-type ctx v)))
         (call-cont cont (ctx-push ctx type)))))

    ((set! ,v ,E1) when (variable? v)
     (let ((scont
             (lambda (ctx)
               (gen-instr `(store ,v))
               (gen-instr `(push #f))
               (let* ((type (ctx-top ctx))
                      (ctx  (ctx-pop ctx))
                      (ctx  (ctx-push ctx 't_bool))
                      (ctx  (ctx-set-type ctx v type)))
                 (call-cont cont ctx)))))
       (gen-expr E1 scont)))

    ...

    (else
     (error "unknown" expr))))
```

Code specialization from AST

```
...

((if ,E1 ,E2 ,E3)
 (let ((ccont (lambda (ctx)
                (let ((type (ctx-top ctx))
                      (ctx  (ctx-pop ctx)))
                  (if (and (not (eq? type 't_bool))
                           (not (eq? Type 't_unk)))
                      (call-cont (gen-expr E2 cont) ctx)
                      (let ((ctx (ctx-pop ctx))
                            (stub-false (make-stub E3 cont ctx))
                            (stub-true  (make-stub E2 cont ctx)))
                        (gen-instr `(iffalse ,stub-false))
                        (gen-instr `(goto ,stub-true))))))))
   (gen-expr E1 ccont)))

...
```

Code specialization from AST

- The `call-cont` function is used to manage versions
- Generate a version for a given context
- Use a cache to reuse existing versions
- Example:

```
(define (call-cont cont ctx)
  ;; check if a version exists for this context
  (let ((version (get-version cont ctx)))
    (if version
        ;; if a version exists, jump to it
        (begin (gen-instr `(goto ,version))
               version)
        ;; else, generate it and add it to the cache
        (let ((version (gen-version cont ctx)))
          (add-version! cont ctx version)
          version))))
```

Code specialization from AST

- Compilation process with CPS
  - ✓ Lazy compilation

- Use of contexts and call-cont
  - ✓ Code specialization

➡ **BBV**

Code specialization from AST

- Compilation process with CPS
  - ✓ Lazy compilation
- Use of contexts and call-cont
  - ✓ Code specialization

→ ~~BB~~V

(but without basic blocks)

Optimizations

Compilation Context

- Used to specialize generated code

- In LC, we use a *virtual stack* to represent values in the current frame

- Local variables are mapped to indexes in the virtual stack

- A specialized version for each type combination

- Easy to map to the execution stack
  - e.g. `(int   int    char   bool)`
    `([sp+0] [sp+1] [sp+2] [sp+3])`

Register allocation

- Greedy register allocation is easy from the virtual stack
- Allocate the next available register when a type is added
- If no register is available, spill a variable
  - e.g. Spill the variable associated to the older type
- Temporaries values are automatically removed (and their registers are freed)
  - e.g.
    ```
    (int   int   char bool)
    ( r1    r3    r2  [sp+0])
    ```

`Specialize code with register allocation`

Constant propagation & folding

- Constants can be propagated through the context
- Add the value next to the type, do not allocate a register
- Use context information for *constant folding*
  - e.g.

    ```
    (int  int:2  char bool:#f)
    ( r2   #f     r1    #f   )
    ```

➡ `Specialize code with constants`
`(+ interprocedural)`

Boxing / Unboxing

- Solution 1: Local / Global CSE and static analysis

- Solution 2: BBV
  - Unbox a variable when BBV discovers its type
    - e.g. type checks
  - Box a variable when we lost its type
    - e.g. maximum versions reached

- Solution 3: Use BBV on specific types
  - And rely on the box representation for the others

  → less effort

Boxing / Unboxing

- ~~Solution 1: Local / Global CSE and static analysis~~

- ~~Solution 2: BBV~~
  - ~~Unbox a variable when BBV discovers its type~~
    - ~~e.g. type checks~~
  - ~~Box a variable when we lost its type~~
    - ~~e.g. maximum versions reached~~

- Solution 3: Use BBV on specific types
  - And rely on the box representation for the others

  → less effort

Boxing / Unboxing (B/U)

- Tagging
  - Integers: *almost* free *B/U*
  - Memory allocated objects: free B/U (e.g. displacement on x86_64)
  - Floating point numbers: **memory allocated !**

  → **Solution 1: Use tagging, and BBV on floats only**

- NaN-Boxing
  - Integers: additional cost
  - Memory allocated objects: additional cost
  - Floating point numbers: **free**

  → **Solution 2: Use NaN-Boxing and BBV on memory allocated objects and integers only**

Boxing / Unboxing (B/U)

- Tagging
  - Integers: *almost* free *B/U*
  - Memory allocated objects: free B/U (e.g. displacement on x86_64)
  - Floating point numbers: **memory allocated !**

  → Solution 1: Use tagging, and BBV on floats only

  → Less effort
- ~~NaN-Boxing~~
  - ~~Integers: additional cost~~
  - ~~Memory allocated objects: additional cost~~
  - ~~Floating point numbers: **free**~~

  ~~**→ Solution 2: Use NaN-Boxing and BBV on memory allocated objects and integers only**~~

More ?

- Using the compiler design:
  - No analysis for tail position detection
  - Inline if condition

- Using BBV:
  - Bounds-checking elimination
  - …

- But optimizations requiring extensive static analysis are difficult to apply
  - e.g. Loop-invariant code motion

Results

LC

- JIT compiler for Scheme (Subset of R5RS)
    - No *call/cc*
    - Limited *eval*
    - *character*, *boolean*, *integer*, *float*, *pair*, *vector*, *f64vector*, *string*, *symbol*, *closure*

- Research tool for 2013 → 2018
    - Tagging / NaN-Boxing
    - Intraprocedural / Interprocedural BBV
    - …

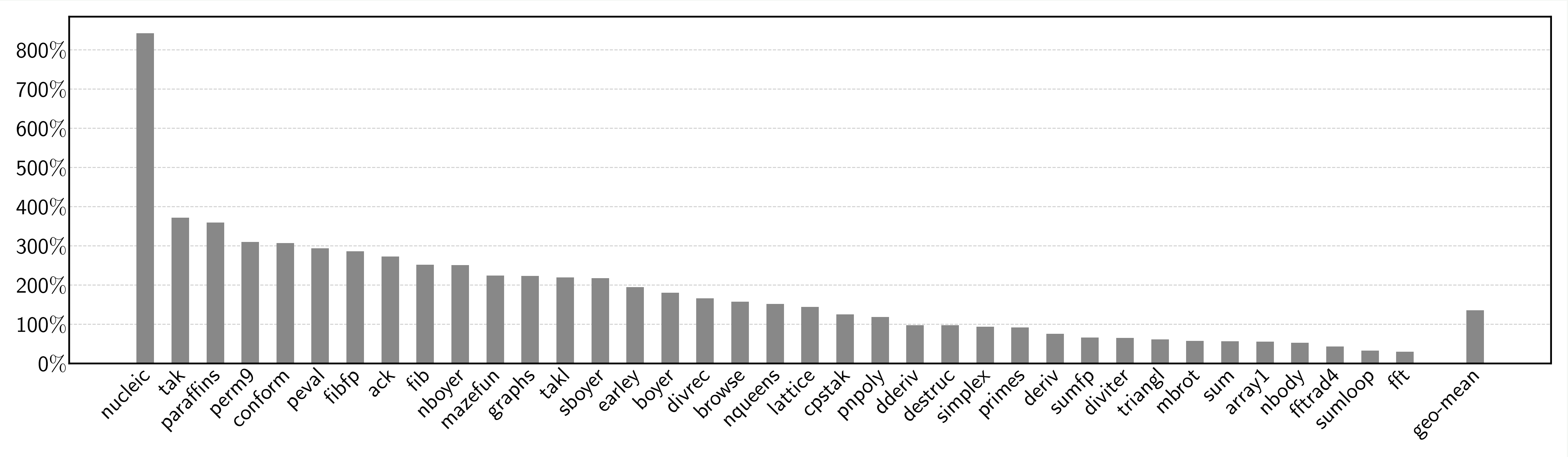
- Built on top of Gambit
    - X86 assembler
    - Frontend
    - GC

Results

- A lot of type checks removed (~70% on average)
- Almost all boxing/unboxing operations removed on floats (>95% on float benchmarks)
- Generated code is 2.25x faster than naive compilation
  (execution time only)
- Generated code is 1.52x faster than gambit
  (execution time only)

Results

- Execution is 1.35x faster than Pycket
  (execution time, compilation time, and GC time)



Pycket relative to LC (optimized mode)

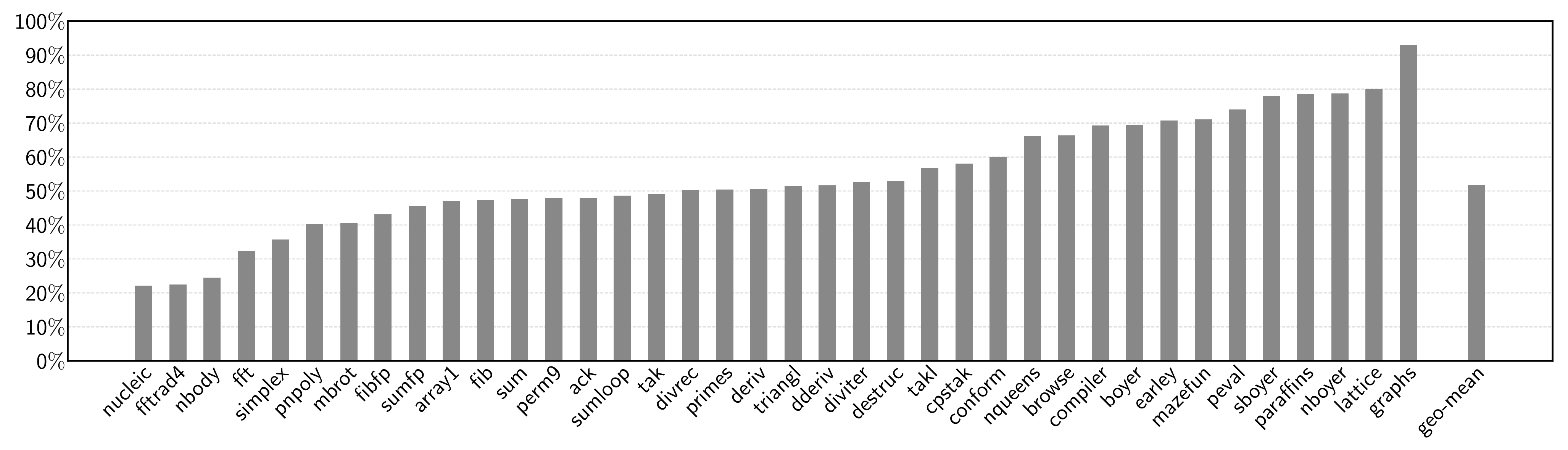
Conclusion

- ☺ We can build optimizing JIT compilers using simple architectures and simple techniques
- ☺ Limiting the architecture still allows using classical optimizations
- ☺ Relatively good performance

- ☹ Cannot easily add optimizations requiring extensive static analysis
- ☹ Cannot compete with multi-level state-of-the-art JIT compilers

➡ **Good choice in resource-limited contexts**

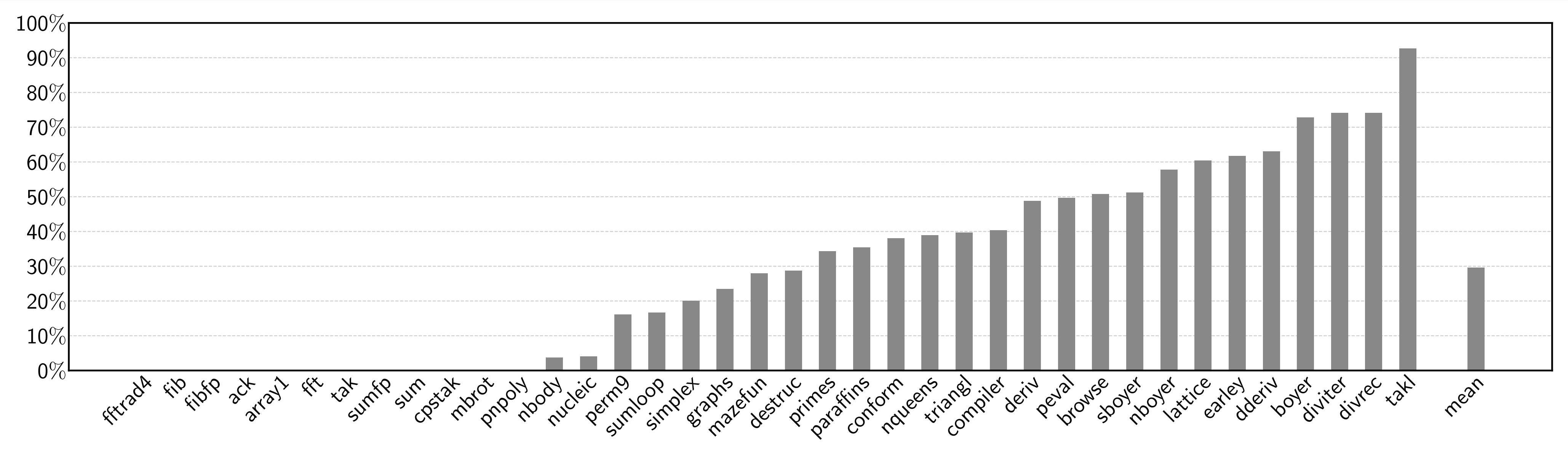
Code size



LC – optimized mode relative to naive mode

- **~50% smaller in optimized mode on average**

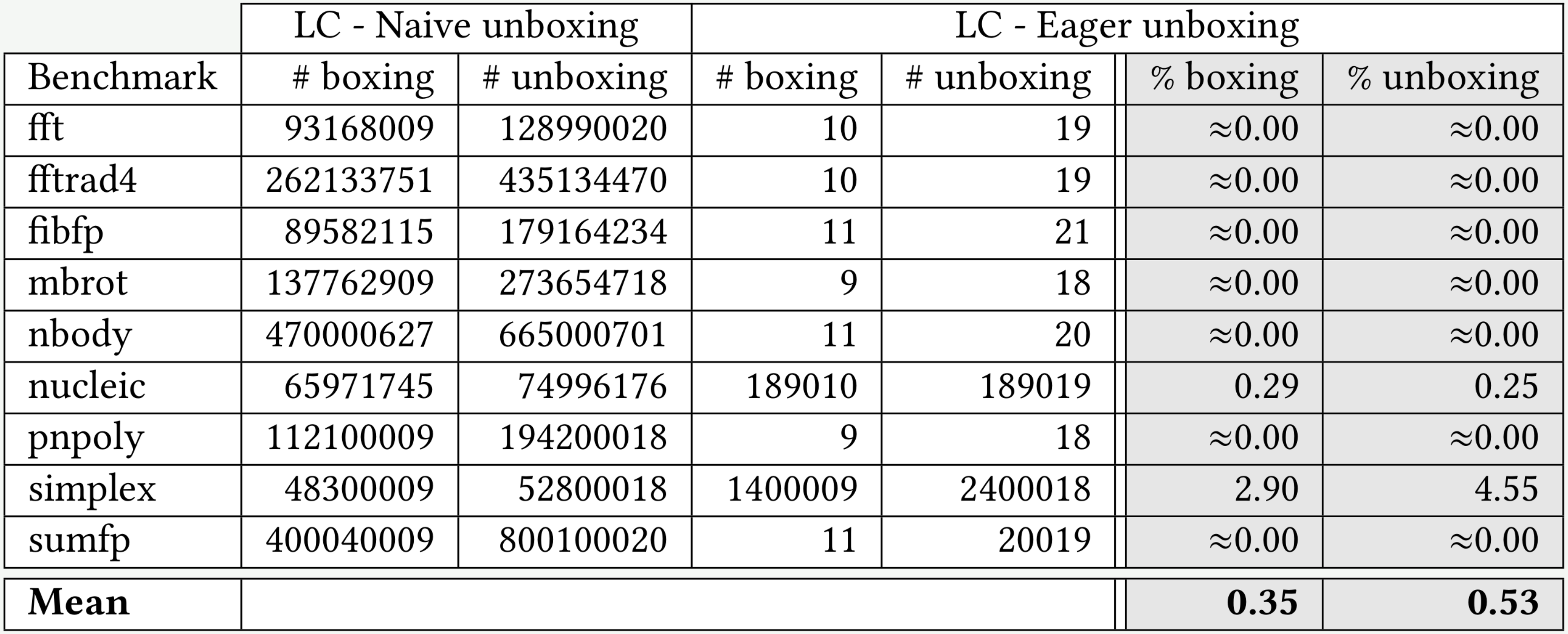
Executed Type checks



LC – optimized mode relative to naive mode

- **~70% fewer checks executed in optimized mode on average**
- **~100% fewer checks executed for 12 benchmarks**

Boxing / Unboxing operations



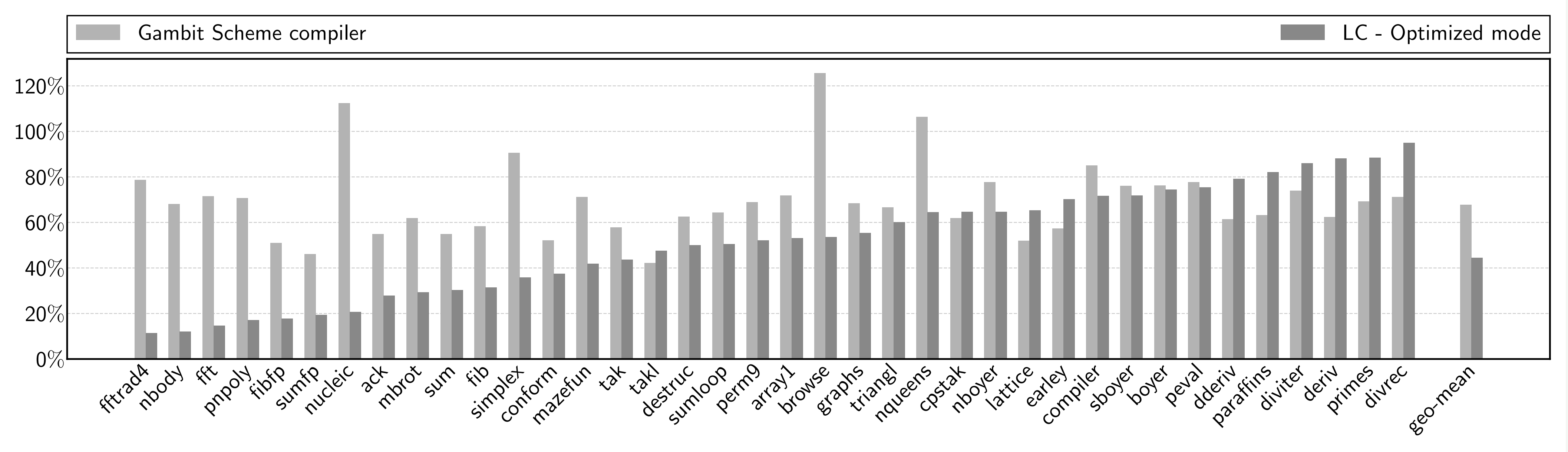
	LC - Naive unboxing		LC - Eager unboxing			
Benchmark	# boxing	# unboxing	# boxing	# unboxing	% boxing	% unboxing
fft	93168009	128990020	10	19	≈0.00	≈0.00
fftrad4	262133751	435134470	10	19	≈0.00	≈0.00
fibfp	89582115	179164234	11	21	≈0.00	≈0.00
mbrot	137762909	273654718	9	18	≈0.00	≈0.00
nbody	470000627	665000701	11	20	≈0.00	≈0.00
nucleic	65971745	74996176	189010	189019	0.29	0.25
pnpoly	112100009	194200018	9	18	≈0.00	≈0.00
simplex	48300009	52800018	1400009	2400018	2.90	4.55
sumfp	400040009	800100020	11	20019	≈0.00	≈0.00
Mean					**0.35**	**0.53**

Number of executed boxing / unboxing operations

- **Almost all boxing/unboxing operations removed**
- **Worst case is still ~95% fewer operations executed**

Execution time (LC vs LC naive)

(no compilation, no GC)

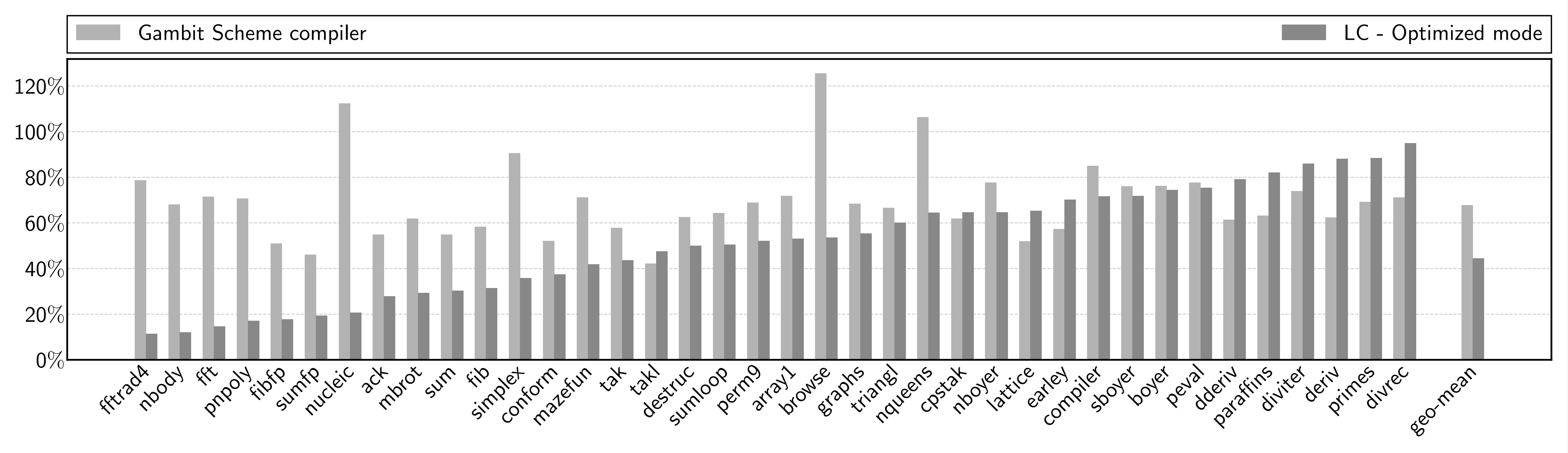


Gambit & LC (optimized mode) relative to LC (naive mode)

- 2.25x faster with LC in optimized mode (vs LC in naive mode)
- No slower benchmark

Execution time (LC vs Gambit)

(no compilation, no GC)

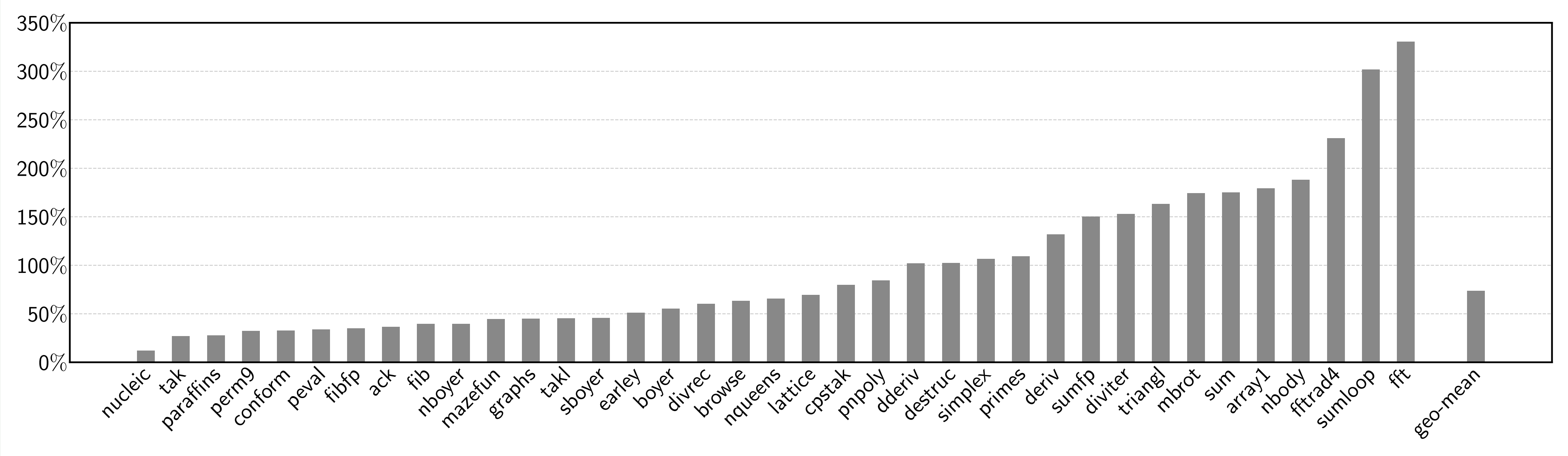


Gambit & LC (optimized mode) relative to LC (naive mode)

- **1.52x faster with LC on average**

Execution time (LC vs Pycket)

(with compilation and GC)



LC (optimized mode) relative to Pycket

- **1.35x faster with LC on average**

Constant propagation: Example

```
(define (type-mask n m)
  (+ (if (fixnum? m) 1 0)
     (if (fixnum? n) 2 0)))

(type-mask 10 #f)
(type-mask (read) (read))
```

- **No code generated for the addition**