Efficient Compilation of Algebraic Effects and Handlers

Matija Pretnar, Amr Hany Saleh,
Axel Faes, Tom Schrijvers

KU LEUVEN

Eff

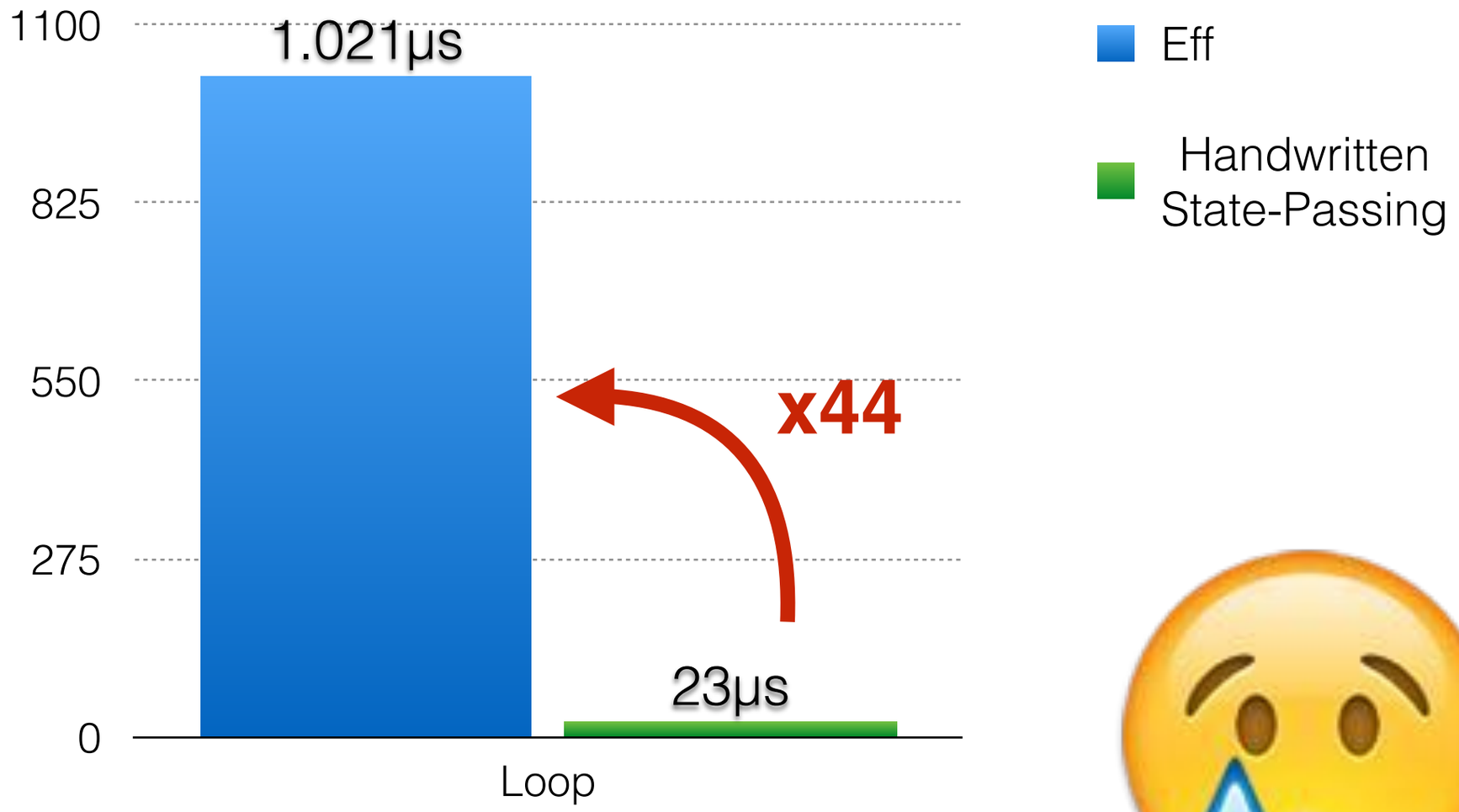
Eff

```
effect Get: unit → int
effect Put: int → unit

let rec loop n =
  if n = 0 then ()
       else (#Put (#Get () + 1); loop (n - 1))

let state_handler =
  handler | #Get () k → (fun s → k s s)
          | #Put s' k → (fun _ → k () s')
          | val y     → (fun s → s)

let state n =
  (with state_handler handle loop n) 0
```

Basic Compilation to OCaml

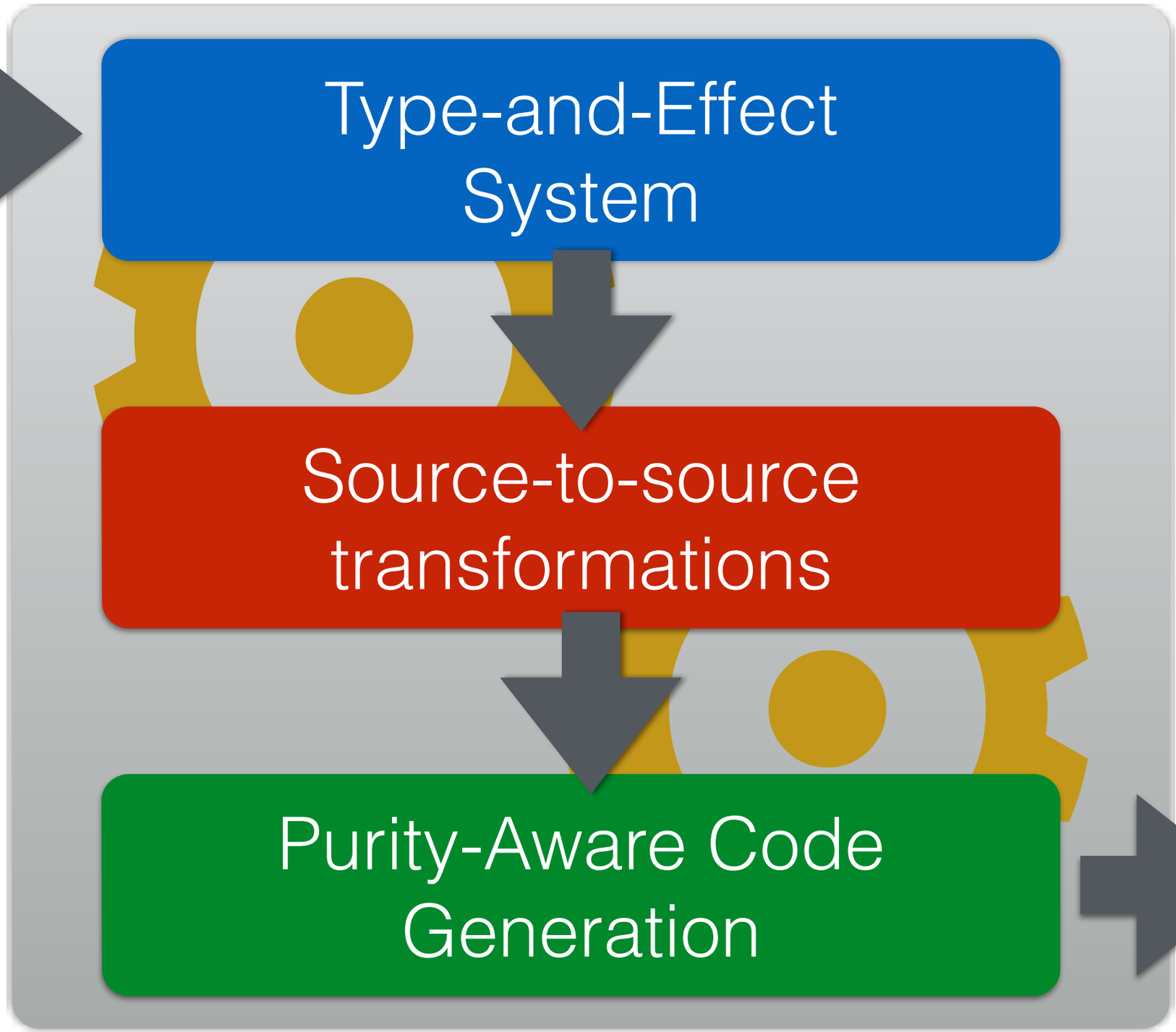
```
let rec loop n =
  _var_1 (* = *) n >>= fun f →
  f 0              >>= fun b →
  (if b then value ()
   else (effect Get ()     >>= fun s  →
         _var_14 (* + *) s >>= fun g  →
         g 1               >>= fun s1 →
         effect Put s1     >>= fun _  →
         _var_16 (* - *) n >>= fun h  →
         h 1               >>= fun n1 →
         loop n1)) ;;
```

Runtime


1100
825
550
275
0
1.021µs
23µs
x44
Loop
Eff
Handwritten
State-Passing

Optimising Compilation

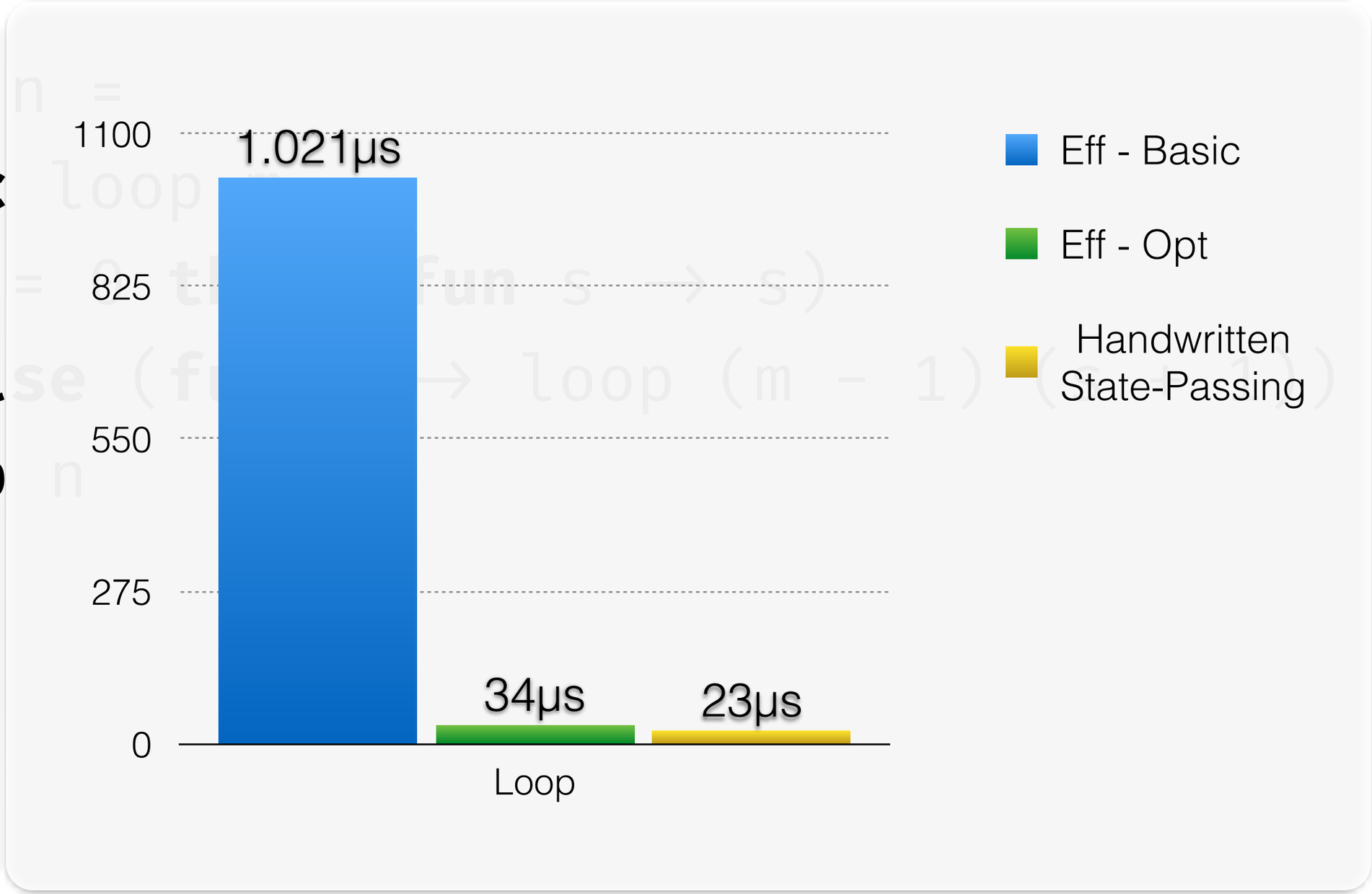
Eff



OCaml

Result

```
let state
  (let rec
     if m
     el
  in loop
) 0
```



Desugaring to Effy

Effy: Core Language

BE

inert expressions

DO

active computations

$$
\begin{array}{rcl}
\text{value } v & ::= & \mathsf{k} \\
& \mid & \mathtt{fun}\ x \mapsto c \\
& \mid & (\mathtt{handler}\ \mathtt{val}\ x \mapsto c \mid ocs) \\
\text{expression } e & ::= & x \\
& \mid & v \\
\text{operation cases } ocs & ::= & \mathtt{nil} \\
& \mid & (\mathsf{op}\, x\, k \mapsto c \mid ocs) \\
\text{computation } c & ::= & \mathtt{val}\ e \\
& \mid & \mathsf{op}\, e \\
& \mid & \mathtt{do}\ x \leftarrow c_1\ ;\ c_2 \\
& \mid & \mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2 \\
& \mid & \mathtt{with}\ e\ \mathtt{handle}\ c \\
& \mid & e_1\ e_2
\end{array}
$$

Type & Effect System

expression type A

int

computation type A ! {op$_1$,…,op$_n$}

int ! {Get,Put}

Source-to-source Transformations

3 Kinds of Transformations

Supporting Normalisation

Handler Reductions

Function Specialisation

1

Supporting Normalisation

β-reduction

monad laws

AppSub

$$\frac{}{(\mathsf{fun}\ x \mapsto c)\ e \rightsquigarrow c[e/x]}$$

DoVal

$$\frac{}{\mathsf{do}\ x \leftarrow \mathtt{val}\ e\ ;\ c \rightsquigarrow c[e/x]}$$

DoOp

$$\frac{}{\mathsf{do}\ x \leftarrow (\mathsf{do}\ y \leftarrow \mathsf{op}\, e\ ;\ c_1)\ ;\ c_2 \quad \rightsquigarrow \quad \mathsf{do}\ y \leftarrow \mathsf{op}\, e\ ;\ (\mathsf{do}\ x \leftarrow c_1\ ;\ c_2)}$$

Handler Reduction (1/3)

$$\textsc{HandleVal}\quad \frac{h = \texttt{handler val}\ x \mapsto c_v \mid ocs}{\texttt{with}\ h\ \texttt{handle}\ (\texttt{val}\ e) \rightsquigarrow c_v[e/x]}$$

$$\textsc{HandleOp}\quad \frac{h = \texttt{handler val}\ x \mapsto c_v \mid ocs \qquad (\mathsf{op}\ y\ k \mapsto c_{\mathsf{op}}) \in ocs}{\texttt{with}\ h\ \texttt{handle}\ (\mathsf{op}\ e) \rightsquigarrow c_{\mathsf{op}}[e/y, (\texttt{fun}\ x \mapsto c_v)/k]}$$

static evaluation

2

Handler Reduction (2/3)

HandleDo

$$\frac{h_1 = \texttt{handler val}\ y \mapsto c_v \mid ocs \qquad c_{v_2} = \texttt{with}\ h_1\ \texttt{handle}\ c_2 \qquad h_2 = \texttt{handler val}\ x \mapsto c_{v_2} \mid ocs}{\texttt{with}\ h_1\ \texttt{handle}\ (\texttt{do}\ x \leftarrow c_1\ ;\ c_2) \rightsquigarrow \texttt{with}\ h_2\ \texttt{handle}\ c_1}$$

Handler Reduction (3/3)

$$\textsc{HandlePure}\quad \frac{h = \texttt{handler val}\ x \mapsto c_v \mid ocs \qquad \mathit{pure}(c, h)}{\texttt{with}\ h\ \texttt{handle}\ c \rightsquigarrow \texttt{do}\ x \leftarrow c\ ;\ c_v}$$

where

$$\frac{\Gamma \vdash c : A\ !\ \Delta \qquad\qquad \Delta \cap \overline{\mathsf{op}} = \emptyset}{\mathit{pure}(c, \texttt{handler val}\ x \mapsto c_v \mid \overline{\mathsf{op}\ x\ k \mapsto c_{\mathsf{op}}})}$$

2

Function Specialisation

recursive function

```
let rec go n = go (#Next n)
in with handler | val x       → x
                | #Next n k → if n > 100 then n
                              else k (n + 1)
handle go 0
```

handling function call

$$\texttt{with}\ h\ \texttt{handle}\ f\ e \quad \rightsquigarrow \quad \texttt{let rec}\ f'\ x = \texttt{with}\ h\ \texttt{handle}\ c_f\ \texttt{in}\ f'\ e$$

2

Function Specialisation

```
let rec go n = go (#Next n)
in let rec go' n =
     with handler | val x       → x
                  | #Next n k → if n > 100 then n
                                else k (n + 1)
     handle go (# Next n)
in go' 0
```

with h handle $f\ e$ $\rightsquigarrow$ let rec $f'\ x$ = with h handle c_f in $f'\ e$

2

Function Specialisation

```
let rec go n = go (#Next n)
in let rec go' n =
     if n > 100 then n
       else with … handle go (n + 1)
   in go' 0
```

2

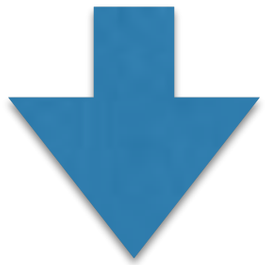
Function Specialisation

```
let rec go' n =
  if n > 100 then n
    else go' (n + 1)
in go' 0
```

2

Generalised Function Specialisation

non-tail recursive functions



selective CPS

Purity-Aware Code Generation

Naive
Code Generation

$$[\![A \mathbin{!} \Delta]\!] \quad = \quad [\![A]\!]\ \mathit{computation}$$

free monad

Purity-Aware Code Generation

plain OCaml

type

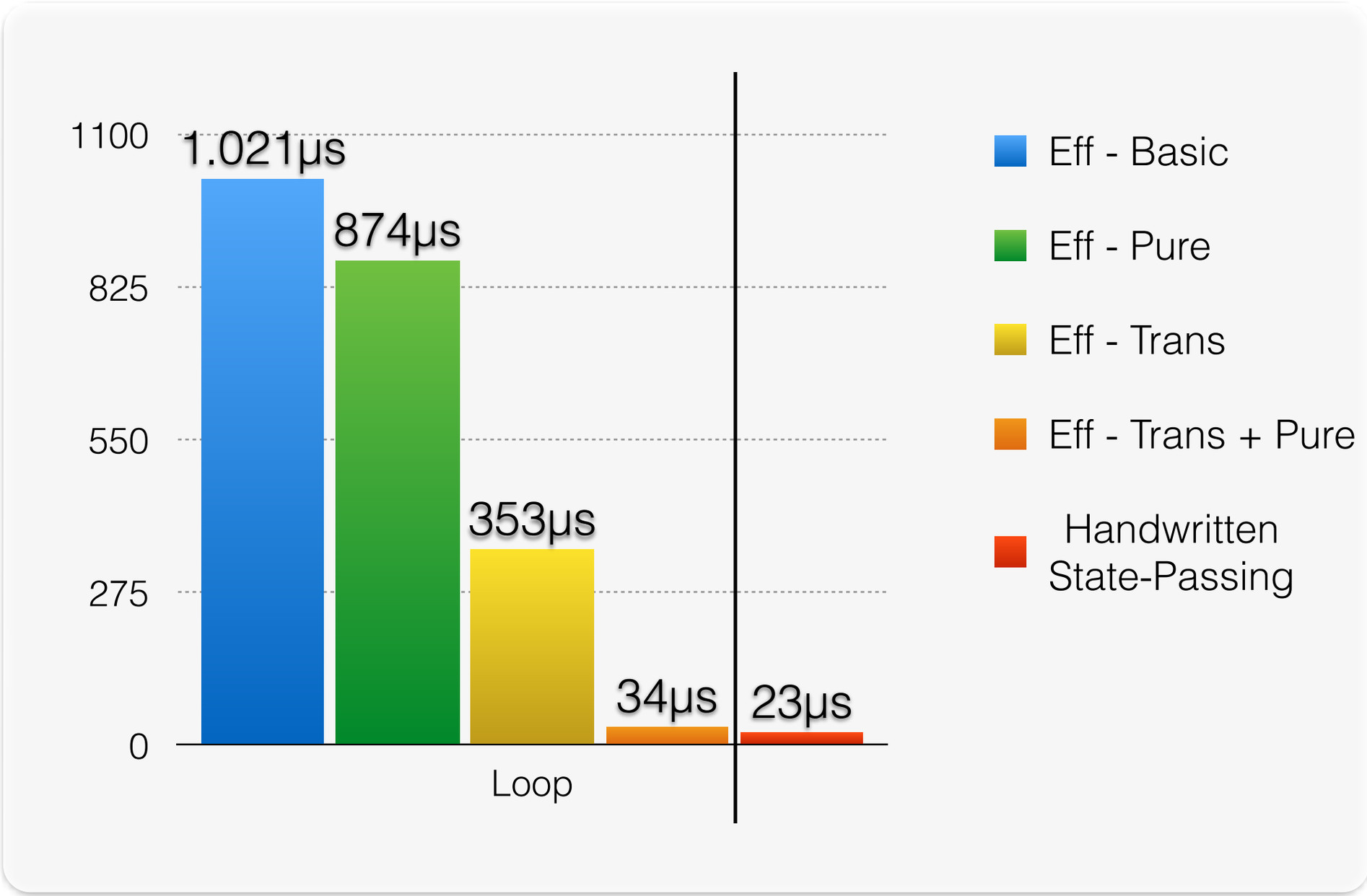
$$[\![A \mid \Delta]\!] = \begin{cases} [\![A]\!] & , \Delta = \emptyset \\ [\![A]\!]\ computation & , \Delta \neq \emptyset \end{cases}$$

free monad

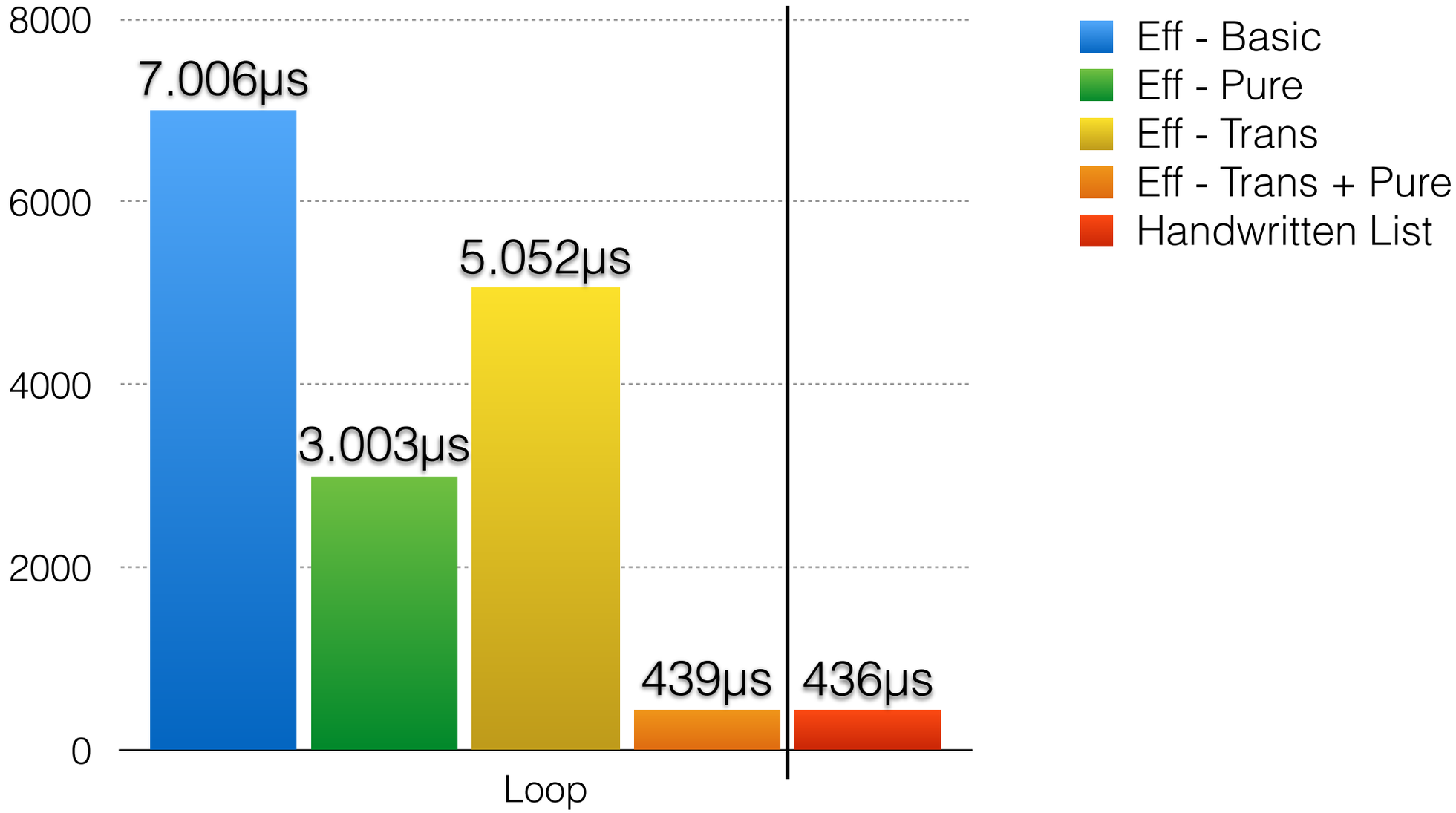
Similar technique applied in [Leijen'17]

Evaluation

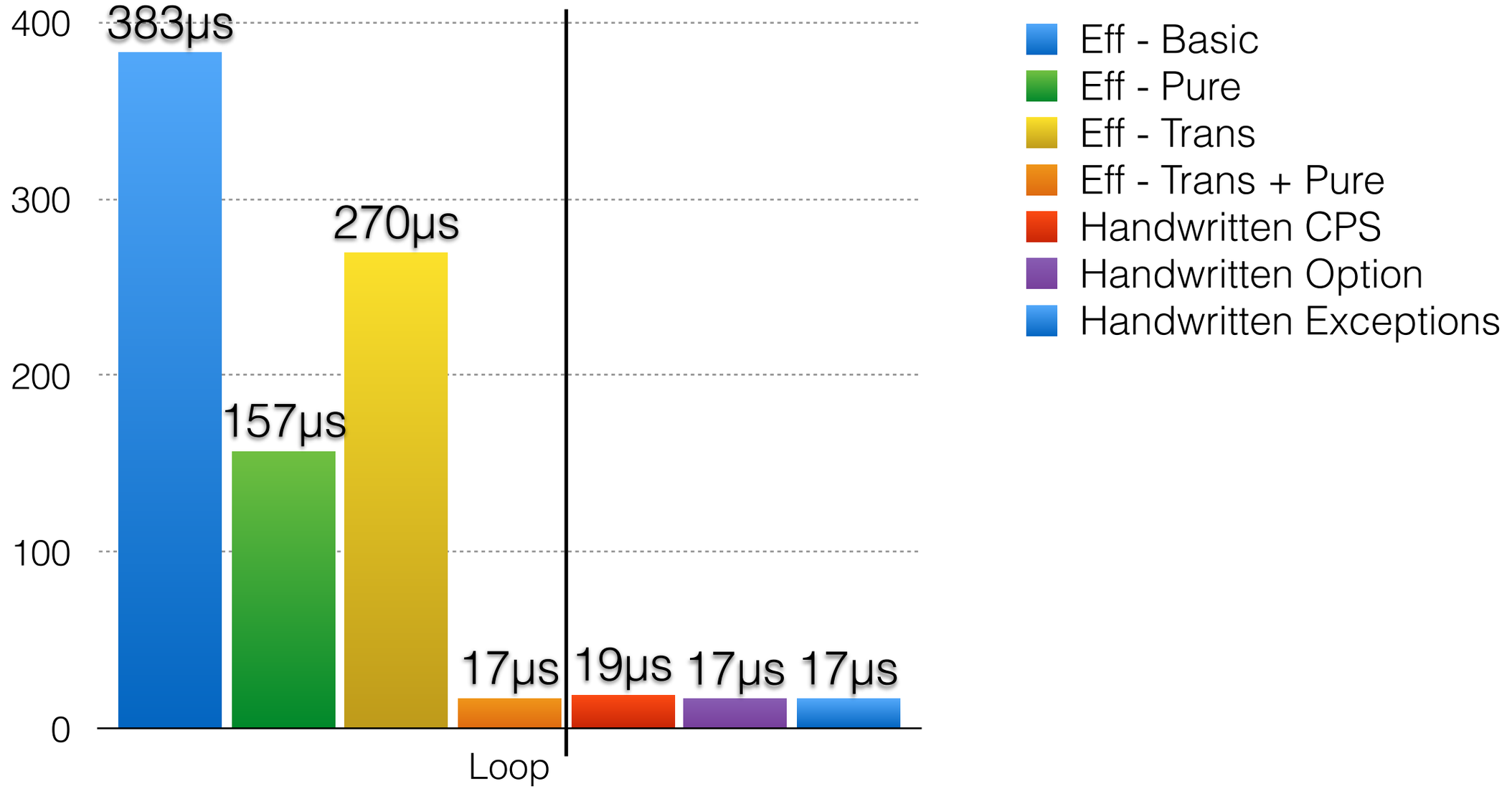
Stateful Loop



8-Queens (All Sols)



8-Queens (1 Sol)



Summary

Summary

- ✦ Algebraic effects and handlers are **slow** with **naive compilation**
- ✦ …but **optimising compilation** can make them (almost as) **fast** as hand-written code
- ✦ source-to-source transformations + purity awareness are key

Related Work

- ✦ Leijen: compiler for Koka
- ✦ Kammar et al., Kiselyov et al. : Haskell, OCaml libraries/encodings
- ✦ Hillerström et al., Dolan et al.: OCaml runtimes
- ✦ Wu & Schrijvers: fusion in Haskell

Future Work

- ✦ dealing with more advanced code patterns

Questions?