Tracing Just-in-Time Compilation for Effects and Handlers

MARCIAL GAISSERT, University of Tübingen, Germany
CF BOLZ-TEREICK, Heinrich-Heine-Universität Düsseldorf, Germany
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

Effect handlers are a programming language feature that has recently gained popularity. They allow for non-local yet structured control flow and subsume features like generators, exceptions, asynchronicity, etc. However, implementations of effect handlers currently often sacrifice features to enable efficient implementations. Meta-tracing just-in-time (JIT) compilers promise to yield the performance of a compiler by implementing an interpreter. They record execution in a trace, dynamically detect hot loops, and aggressively optimize those using information available at runtime. They excel at optimizing dynamic control flow, which is exactly what effect handlers introduce. We present the first evaluation of tracing JIT compilation specifically for effect handlers. To this end, we developed RPython-based tracing JIT implementations for Eff, Effekt, and Koka by compiling them to a common bytecode format. We evaluate the performance, discuss which classes of effectful programs are optimized well and how our additional optimizations influence performance. We also benchmark against a baseline of state-of-the-art mainstream language implementations.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Control structures**.

Additional Key Words and Phrases: Effect Handlers, Tracing JIT

ACM Reference Format:
Marcial Gaißert, CF Bolz-Tereick, and Jonathan Immanuel Brachthäuser. 2025. Tracing Just-in-Time Compilation for Effects and Handlers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 307 (October 2025), 49 pages. https://doi.org/10.1145/3763085

1 Introduction

In the last decade, a number of different programming language features have (re-)emerged that help programmers to structure control flow. Examples include asynchronous programming, event-based programming, generators, fibers, coroutines, and more. One such feature are effect handlers [Plotkin and Pretnar 2009, 2013]. Effect handlers have been shown expressive enough to subsume the aforementioned control-flow features [Bračevac et al. 2018; Dolan et al. 2017; Leijen 2016, 2017a; Plotkin and Pretnar 2013]. They are also high-level enough to admit a simple typing discipline and encourage structured programming. By using effect signatures as interfaces, programmers separate the use-site of effects (*e.g.*, reading from the console) from its concrete implementations (*e.g.*, performing I/O or reading inputs off a provided string). While their origin lies in programming language theory, effect handlers are gaining interest both theoretically and practically. They are implemented not only in a variety of research languages (such as Eff [Bauer and Pretnar 2015], Koka [Leijen 2017b], Frank [Lindley et al. 2017], Effekt [Brachthäuser et al. 2020], Helium [Biernacki et al. 2019], and more), but also practical general-purpose languages like OCaml 5 [Sivaramakrishnan et al. 2021], Scala [Kagami 2023; Odersky 2023], Unison [Unison Computing 2025], and WebAssembly [Phipps-Costin et al. 2023] began integrating effects and handlers.

Authors' Contact Information: Marcial Gaißert, marcial.gaissert@uni-tuebingen.de, University of Tübingen, Germany; CF Bolz-Tereick, cfbolz@gmx.de, Heinrich-Heine-Universität Düsseldorf, Germany; Jonathan Immanuel Brachthäuser, jonathan.brachthaeuser@uni-tuebingen.de, University of Tübingen, Germany.

ACM 2475-1421/2025/10-ART307
https://doi.org/10.1145/3763085

One way to view effect handlers is that they generalize exception handlers. However, they are not to be used only in exceptional cases, but as a general way to structure non-local control flow, both in the small (*e.g.*, to break out of a loop) and in the large (*e.g.*, to model the architecture of an entire application). It is thus important to implement them efficiently to reduce the trade-off between modularity and performance. This is an active area of research, with two kinds of approaches:

- ahead-of-time optimization using either
  - *rewrite rules* [Karachalias et al. 2021b; Pretnar et al. 2017; Sieczkowski et al. 2023], or
  - *continuation-passing style* [Müller et al. 2023; Schuster et al. 2020], and
- runtime systems [Alvarez-Picallo et al. 2024; Ma et al. 2024; Sivaramakrishnan et al. 2021].

While prior work on optimizing handlers already achieves good performance, it suffers from typical problems with ahead-of-time compilation. Schuster et al. [2020] require that concrete effect handlers and their nesting order are statically known. Müller et al. [2023] lift these restrictions, but neither support first-class functions, polymorphic recursion, nor separate compilation. Karachalias et al. [2021b] (and similarly Sieczkowski et al. [2023]) use rewrite rules based on explicit effect coercions [Saleh et al. 2018]. Rewrites can also only be applied when handlers are known and their applicability is thus restricted by separate compilation. Languages like OCaml 5 [Sivaramakrishnan et al. 2021] and Lexa [Ma et al. 2024] offer specialized runtime systems for effect handlers. For efficiency, it is common to restrict continuations to be *one-shot*, that is, resumed at most once [Bruggeman et al. 1996; Dolan et al. 2015; Ma et al. 2024; Sivaramakrishnan et al. 2021]. A linear use of continuations allows efficient stack-switching strategies but rules out other use cases of effect handlers, *e.g.*, backtracking search [Leijen 2016] or probabilistic programming [Nguyen et al. 2023].

Some of these limitations could be resolved by a just-in-time (JIT) compiler. Moving optimizations to runtime, this naturally supports separate compilation. Optimizations that require involved static analysis to be performed ahead-of-time (*e.g.*, knowing the effect handler for a given effect operation), or are unsound in the general case, could be implemented with more light-weight analysis at runtime. At runtime, *all* information about effect handlers is available, and assumptions can be guarded. Yet, no JIT compiler has been developed with the goal of optimizing effects and handlers.

While prior work on JIT compilation has shown that it is possible to optimize exceptions into direct jumps [Paleczny et al. 2001; Würthinger et al. 2013], inline generators [Zhang et al. 2014], and efficiently support undelimited continuations [Bauman et al. 2015a, 2017], there exists no work on the more general construct of effect handlers. Here, we—to the best of our knowledge, for the first time—explore the applicability of JIT compilation techniques for optimizing effect handlers. Since the area of JIT compilation is large and developing a JIT compiler requires significant engineering effort, we start with *tracing JIT compilers* [Bala et al. 2000; Bolz et al. 2009] by investigating their effectiveness for effect handlers. In particular, we pose the following research questions:

- **RQ 1**: How does tracing JIT compilation compare with existing ahead-of-time optimizing implementations of effect handlers?
- **RQ 2**: What are classes of effectful programs that tracing JIT compilation can optimize well? What are the limitations of the approach?
- **RQ 3**: How can we optimize the performance of tracing JIT compilation for effect handlers?
- **RQ 4**: Are there differences in how well tracing JIT compilation performs for different variations of effect handlers?
- **RQ 5**: Does JIT-compiling effect handlers impact the performance of programs that do not use effects?

To find first answers to these questions, we implemented a bytecode interpreter in the RPython framework [Bolz et al. 2009], which gives rise to a (meta-)tracing JIT compiler. We further modified three existing research languages with effect handlers to target this bytecode format: Eff, the first language with effect handlers, featuring *dynamically scoped* effect handlers [Plotkin and Pretnar 2013], implemented by a dynamic search at runtime. Effekt, a language with *lexically scoped effects* and handlers, based on a capability-passing transformation [Brachthäuser et al. 2020]. Koka, a language with both dynamically scoped effect handlers and *named handlers*, based on an evidence-passing transformation [Leijen 2017b; Xie and Leijen 2021]. We believe these three to be representative of a range of languages with effect handlers as they support different variations of effect handlers and follow different approaches to implementing them. To study JIT compilation for all three, and to minimize the bias of translating one paradigm to another, our bytecode format directly supports the necessary features for both dynamic and lexical effect handlers. We evaluate our implementation both *qualitatively*, identifying and describing representative examples, and *quantitatively*, measuring the performance on benchmark programs.

Summary of our findings. Based on our implementation, our experience with it, and the results of our analysis, we can indeed confirm that tracing JIT compilation appears to be a viable implementation technique for effect handlers.

- **RQ 1 (Comparison with AOT)** Common optimizations implemented by AOT optimizing compilers for effect handlers (*e.g.*, optimizing tail-resumptive handlers) emerge automatically (Sections 2, 6.1 and 6.2). In a JIT, being able to speculate, these optimizations did not come with a loss of expressivity of the source language (*e.g.*, disallowing multi-shot resumptions or separate compilation). Due to standard optimizations implemented by RPython, the abstraction overhead of effect handlers can be eliminated in many use cases. We can improve performance further by applying specific optimizations (Section 4.1). Benchmark results comparing our implementation to AOT optimizing compilers show a competetive performance both over most existing implementations of effect handlers and state-of-the-art language implementations for programs with control effects (Section 6.1 and 6.5).
- **RQ 2 (Advantages and Limitations)** Effect handlers that use the continuation in a one-shot and tail manner, or as exceptions, are optimized very well by the JIT (Section 6.2). Those that are still one-shot, but resume in a non-tail position, are optimized well, but incur costs for allocating the additional stack frames. Based on RPython, our implementation inherits common limitations and problems of tracing JIT compilers, including performance cliffs. Effect handlers, allowing complex control-flow patterns, further amplify some of these problems. Handlers with complex patterns of resumes, or effect operations handled by many different handlers, can potentially lead to a large number of bridges and traces, duplicated continuations (tails) [Gal et al. 2009], and overly long traces (Section 6.2).
- **RQ 3 (Optimizations)** Standard optimizations for using the stack context to detect false loops (Section 4.1.4) and specializing certain dynamically sized data structures (Section 4.1.3) have a positive effect. Additional optimizations for prompt search (Section 4.1.2) and loop start points (Section 4.1.1) have a minor effect and only help with specific benchmarks.
- **RQ 4 (Variations of Effects)** Most differences in the performance are minor in nature. The adjustment of evidence vectors in Koka can lead to allocations when changing the handler context. On the other hand, the tracking of handlers in evidence vectors in Koka can help to avoid traversing deep stacks in some cases. The prompt search in Eff tends to generate more guards, as it is harder for the JIT to reason about them (Section 6.4).
- **RQ 5 (Baseline)** The implemented JIT is similar in performance to PyPy for our set of baseline benchmarks, but is outperformed by some state-of-the-art JIT compilers (V8, LuaJIT) by a factor of 2–3x. For control-effect-heavy benchmarks, though, we outperform state-of-the-art language implementations like V8.

In summary, this paper makes the following contributions:

- The first implementation of a JIT compiler optimizing effects and handlers at runtime; it supports three source languages with different variations of effect handlers (Section 4).
- A common bytecode format that directly supports dynamic and lexical effect handlers, as well as a description of how to translate three different notions of effect handlers to the common virtual machine (Section 3).
- An internal performance evaluation comparing the three variations of effect handlers in our unified VM (Sections 5 and 6.4); this is the first time such a comparison is possible.
- An external performance evaluation, comparing against existing implementations of effect handlers (Sections 5 and 6.1) and state-of-the-art language runtimes (Section 6.5).
- A description of implemented optimizations (Section 4.1) as well as an ablation study to evaluate their effectiveness (Section 6.3).

The following section demonstrates our approach by example (Section 2), before we present technical details (Sections 3 and 4), evaluate the effectiveness of tracing JIT compilation for effect handlers (Section 5), and discuss our findings (Section 6).

2 JITting Effects by Example

This section aims to give a high-level overview of our approach. We provide a short example-driven introduction to programming with effect handlers in Eff and then walk through the different steps of evaluating the example in our tracing JIT implementation.

2.1 Example: Push Streams with Effects and Handlers

Our running example is push streams, adapted from Kiselyov et al. [2017], using the effect Emit.

```
effect Emit: int → unit
```

Similar to generators in Python, we can use the Emit effect to emit integer values, *e.g.* temperatures. On the left, the function generateMeasurements implements a push-stream producer of integers.

```
let generateMeasurements n =
  let rec loop t =
    if t > 0
      then
        perform (Emit (measure t));
        loop (t - 1)
      else ()
  in loop n
```

```
let filter p b =
  handle
    b ()
  with
  | effect (Emit x) k →
      if p x then perform (Emit x) else ();
      k ()
  | _ → ()
```

The example on the right shows the definition of the standard combinator filter on push streams. Here, the implementation of filter receives a function parameter b, which can use the Emit effect. It handles the effect by checking the condition on the current value. If this succeeds, we then yield the value; otherwise, we do not. In both cases, we resume at the original call site of Emit by calling the continuation k. Being able to resume the computation by calling the continuation is the feature that distinguishes effect handlers from (non-resumeable) exception handlers.

Function count on the left implements another push-stream consumer that counts yielded values. Before resuming, we remember to perform the addition 1 + □ by pushing it onto the call stack.

```
let count b = handle b () with
  | effect (Emit x) k → 1 + (k ())
  | _ → 0
```

```
let countTooBig b =
  count (fun () →
    filter isValueTooBig (fun () → b ()))
```

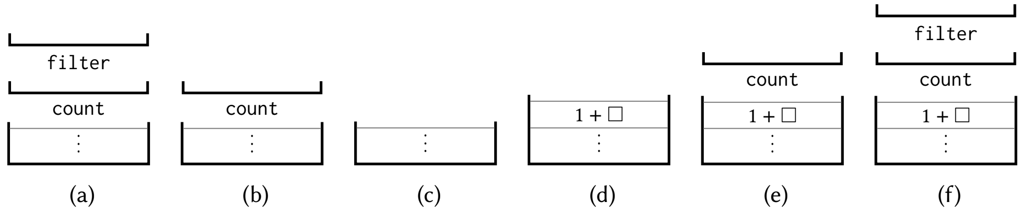


Fig. 1. Sketch of stack shapes during the execution of the loop. Some frames have been omitted. Prompts are annotated as the name of the function installing them. Stacks grow upwards.

We can use the above combinators to construct a stream-processing pipeline (on the right). Function `countTooBig` consumes a stream of integers by filtering values above a certain threshold and counting them. It handles the `Emit` effect in its parameter `b`. Finally, we compose the producer and consumer to compute the number of measurements that exceed the threshold.

```
countTooBig (fun () → generateMeasurements n)
```

2.2 Runtime Intuition

Operationally, effects and handlers manipulate the runtime call stack. Calling an effect operation like `Emit`, captures the current continuation and binds it to `k`. Operationally, we unwind the call stack up to the corresponding handler (*i.e.*, the `handle ... with ...`). To support this search for the correct handler, some language implementations push a marker on the normal call stack for each handler. Instead, here we represent the call stack as a stack-of-stacks, which we refer to as the *metastack* [Danvy and Filinski 1990; Dybvig et al. 2007; Schuster and Brachthäuser 2018].

Figure 1 illustrates the structure of the call stack for evaluating one iteration of our example:

```
countTooBig (fun () → generateMeasurements n)
```

Subfigure 1a represents the call stack before evaluating the call to the effect operation `Emit` in `loop`:

```
let rec loop t = ... ⓐ perform (Emit (measure t))  ...
```

At that point in the program execution, `count` and `filter` have both installed effect handlers, which correspond to a separate stack segment each. Each stack segment is labeled with the function that installed it. To evaluate the call to `Emit`, we check whether the current stack segment contains a handler for `Emit`, remove that segment, and enter the handler installed by `filter` (Subfigure 1b).

```
let filter ... = ... if p x then ⓑ perform (Emit x) else () ...
```

Here, we assume that we evaluated the conditional `p x` and now have to re-emit value `x`. This unwinds the stack once more and transfers control to the handler installed by `count` (Subfigure 1c):

```
let count ... = handler ... | effect (Emit x) k → ⓒ 1 + (k ())
```

Before resuming, we push a frame to remember the addition of 1 (Subfigure 1d).

```
let count ... = handler ... | effect (Emit x) k → 1 + ( ⓓ k ())
```

We then resume computation after the last call to `Emit` by pushing the captured stack segment back to the metastack (Subfigure 1e).

```
let filter ... = ...; ⓔ k ()
```

Finally, we resume computation to the very first call to `yield` within `loop` (Subfigure 1f).

```
let rec loop t = ... perform (Emit (measure t)); ⓐ loop (t - 1)  ...
```

When we encounter an effect handler, we create a new empty stack and push it onto the metastack before evaluating the handled statement. Each element of the metastack thus directly corresponds to one effect handler. We call the number and order of these the *stack shape*. Capturing the continuation amounts to repeatedly popping stack segments off the metastack until the correct handler is reached. Resuming the continuation amounts to pushing stack segments back onto the metastack.

2.3 BC – Bytecode for Effect Handlers

As illustrated in the previous subsection, pushing and popping stacks occurs frequently when evaluating a program that uses effect handlers. The cost of these operations has a significant impact on the overhead of effect handlers. Before we describe how our JIT compiler performs optimizations, we first sketch how the above code translates to our common bytecode BC. As we will see formally in Section 3, BC features labeled blocks and jumps, manages local variables using explicit register management instructions (for copying, swapping, and dropping/deleting), and explicitly manages the metastack by pushing and popping individual frames as well as whole stack segments.

2.3.1 Example: Translation of `loop`*.* To get acquainted with the bytecode format and informally relate effect handling with corresponding stack operations in bytecode, we now walk through the running example translated to BC. We start by inspecting the simplified[1] translation of `loop`:

```
loop(t, emit_tag):                          loopContMeasure(ret, t):
  gt ← primitive ">"(t, 0);                   emit_tag ← primitive getGlobal("Emit");
  if gt then loopThen(gt, t) else             emit_h ← get dynamic emit_tag;
  return (unit())                             push loopContPerform(t);
                                              emit_h.emit(ret)

loopThen(gt, t):                            loopContPerform(ret, t):
  push loopContMeasure(t);                    t ← primitive "-"(t, 1);
  jump measure(t)                             emit_tag ← primitive getGlobal("Emit");
                                              jump loop(t, emit_tag)
```

The recursive function `loop` translates into four blocks. Block `loop` corresponds to the entrypoint of the loop, while `loopThen` represents the then-branch of the conditional. Block `loopContMeasure` is the continuation at the call to `measure` and `loopContPerform` is the continuation at the effect operation `Emit`. Continuations receive the returned value as first argument followed by closure arguments. In entrypoint `loop`, we check the loop condition and jump to `loopThen` if it is satisfied. Here, we push the continuation `loopContMeasure`, with the current counter value `t` and then call function `measure`, which we omit. Once we return from `measure`, we continue executing at `loopContMeasure`, where we have two values available: `ret`, the measurement, and `t`, the loop counter. Now, to perform an effect, we need to know which handler implementation to use. In Eff, effect handlers are dynamically scoped, similar to how exceptions are dynamically scoped in languages like JavaScript. That is, at runtime we perform a search for the closest handler for a given effect. Our implementation assigns a globally unique *prompt* [Felleisen 1988] to each effect and allocates the handler implementation on the stack, labeled with that prompt. To find the handler, we first load the global prompt for `Emit`, and then retrieve the current dynamic binding for this prompt using `get dynamic`. This entails searching the stack for the prompt and returning the value stored there. We then invoke this handler implementation. Later, it will resume and we continue the loop in `loopContPerform`, decrementing the counter and jumping back for another iteration.

[1] The actual code is more complicated due to curried function applications in Eff, which we don't remove statically, and semantics-preserving simplifications. Also, we summarized reordering of arguments before jumps to a jump-with-arguments.

2.3.2 Example: Translation of `filter`*.* The handler that handled the effect in `loop`, was installed by the function `filter`. Thus, calling it leads us into its implementation:

```
| effect (Emit x) k →
    if p x then perform (Emit x) else ();
    k ()
```

In the source code, we have access to the captured continuation as `k`. In our implementation, this means that we need to unwind and capture the stack, then execute the actual code of the operation, and restore the stack when calling `k`. This handler is translated as follows (again, simplified):

```
filterEmit(x, pred):
  emit_tag ← primitive getGlobal("Emit");
  k ← shift emit_tag;
  push filterEmitContPred(x, k);
  pred.apply(x)

filterEmitContPred(ret, x, k):
  push filterEmitContIf(k);
  if ret then filterEmitThn(ret, x) else
  return(unit())
```

```
filterEmitThn(ret, x):
  emit_tag ← primitive getGlobal("Emit");
  emit_h ← get dynamic emit_tag;
  emit_h.emit(x)

filterEmitContIf(ret, k):
  r ← unit();
  push stack k;
  return (r);
```

The four blocks are: the entrypoint `filterEmit`, the continuation `filterEmitContPred` of the call to `p`, the then-branch `filterEmitThn`, and `filterEmitContIf` corresponding to the continuation `□; k ()`. In the entrypoint, we first retrieve the prompt for `Emit` from the global variable. The next instruction `shift emit_tag` unwinds and captures the part of the metastack up until and including the stack labeled with the prompt (installed by `filter` prior to executing the body), and stores the captured stacks in register `k`. The next instructions push the continuation on the stack and proceed with calling the predicate, which is translated to a closure `pred` with a single operation `apply`. Upon returning from the predicate, we continue with the conditional in `filterEmitContPred`. If the predicate holds, we continue execution in `filterEmitThn`. This will invoke the currently installed handler like we did in `loop`. Finally, we return to block `filterEmitContIf`, which reinstalls the captured stack `k` and resumes execution after the call to `Emit` in `loop` by returning to it.

2.3.3 Example: Translation of `count`*.* Finally, the handler implementation in `count` translates to the two blocks below. Block `countEmit` is the entrypoint, and `countEmitCont` corresponds to the continuation `1 + □`. As for every effect operation, we start by capturing the correct part of the metastack using `shift`, push a frame for `1 + □`, and finally resume using `push stack` and `return`.

```
countEmit(x):
  emit_tag ← primitive getGlobal("Emit");
  k ← shift emit_tag;
  r ← unit();
  push countEmitCont();
  push stack k;
  return (r)
```

```
countEmitCont(ret):
  r ← primitive "+"(ret, 1);
  return (r)
```

2.3.4 Summary. Translating our running example to bytecode illustrates how handling of effects is split into two parts: selecting the correct handler implementation and capturing the correct continuation. The former is achieved in Eff by storing the handler object on the stack, next to the

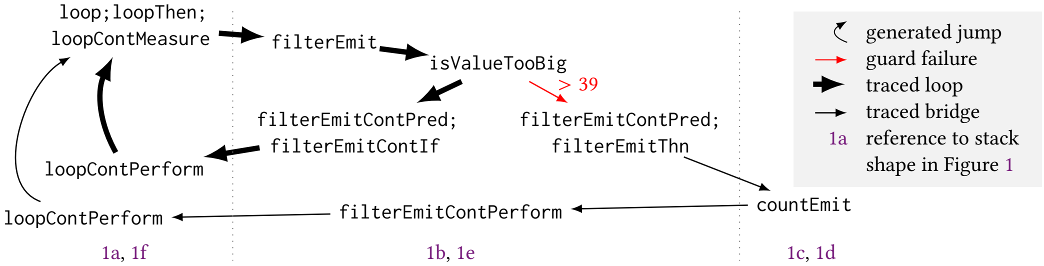


Fig. 2. Sketch of the traced loop and bridge. Nodes are bytecode blocks, sometimes combined, names refer to the numbers in the translations above. Edges are traced control flow.

prompt, and looking it up dynamically. As we will later see, this is the part where the different implementations differ the most. The latter is achieved by translating handlers to use `shift` which captures the continuation up to and including a specific prompt. To resume, the captured continuation is pushed back onto the metastack using instruction `push stack ptr`.

2.4 Tracing JIT for Effects and Handlers

Having gathered an operational intuition about effect handlers and their translation to our bytecode format, we now describe how our tracing JIT compiler optimizes the execution of this program. In particular, we demonstrate how it removes the indirection introduced by effects and handlers.

2.4.1 Tracing. Generally, a tracing JIT compiler starts in interpreted mode where it dispatches on the bytecode instructions and interprets them one by one. At the same time, it maintains counters for certain program positions and eventually reaches a threshold when evaluating a loop multiple times [Bolz et al. 2009; Cuni 2010; The PyPy Project 2025]. In our example, it first does so within the execution of the `loop` function, before the primitive operation >. When the threshold is reached, it then records the instructions executed in the loop. Here, it does so for the case where the emitted value does not exceed 39 °C. This is statistically likely, since this is more common in the test data.

2.4.2 Compiling. After the JIT compiler has traced the loop (**bold** in Figure 2), it will optimize the resulting straight-line trace and generate native machine code, which will be run directly on subsequent iterations. On each potential branch in the trace, a *guard* is emitted, checking that the current iteration still corresponds to the traced control flow [Bolz et al. 2009; Cuni 2010]. The effect call `perform (Emit x)` in `loop` corresponds to the optimized trace on the left in Figure 3. Importantly, the annotations in `< >` should be read as comments; they do not have any runtime semantics. We can notice that calling the effect operation and capturing the continuation do not result in any generated machine instructions. The implementation is specialized to the specific handler calling `isValueTooBig`, which in this example checks if the temperature value is smaller than 39 °C. This is split into two guards here due to the implementation of generic comparison, which could be changed to generate just one guard. To resume the continuation, we execute the part of the trace on the right, which, again, merely continues with traced blocks. More notable is what we do not see: allocations. Although our bytecode interpreter uses immutable linked lists for stacks and metastacks, and thus would allocate on each push or resume, thanks to the allocation removal performed by the JIT [Bolz et al. 2011], we do not allocate anything inside the hot loop!

2.4.3 Bridges. The loop will run as long as all guards still hold, *i.e.*, as long as all values are smaller than our limit of 39 °C. But eventually, while executing the traced loop, this guard may fail.

```
tracedLoop1(p1,i1,p2):
  ...
  < loopContMeasure, filterEmit >
  < isValueTooBig >
  i3 = int_eq(i2, 39)
  guard_false(i3) ⇒ filterEmitContIf [i1]
  i4 = int_lt(i2, 39)
  guard_true(i4) ⇒ filterEmitThn [i1]
  < filterEmitContPred >
  ...
```

```
  ...
  < filterEmitContIf >
  < loopContPerform >
  i5 = int_sub(i1, 1)
  < loop >
  jump tracedLoop1(p1,i5,p2)
```

Fig. 3. Summary of traced and optimized loop code. Unrelated parts are marked with ... and blocks we go through in < >. All other lines are generated machine code instructions.

```
tracedBridge1: // at filterEmitThn
  ... // guards checking stack shape and handler
  < bridge end >
  p6 = new Stack([[countEmitCont]], p7)
  p5 = new MetaStack(p6, p3, p4)
  ...(p6,p5) // reify inner stack segment
  jump prelude of tracedLoop1(...)
```

When this happens, we have to switch back to an interpreted version again. In fact, we will switch to the even slower fallback interpreter that can retrieve all the local data from the executing native code [Bolz et al. 2009]. However, each time this happens, another counter is incremented and eventually, we start tracing a *bridge* out of the guard, create a new, optimized, natively compiled program for this case, and connect the existing loop to it [Cuni 2010]. This bridge is illustrated by the non-bold edges in Figure 2. In the bridge, the effect call `Emit x` results in code similar to above, but with additional guards and allocations at the end of the loop. Specifically, we can see one allocation for the new stack frame pushed in `sum` and one for the changed stack segment in the metastack. While these escape the loop and have to be allocated, there is no allocation necessary for the first stack segment as it is kept separately [Hillerström et al. 2017] and did not change. We do, however, reify the inner stack segment for the prelude of `tracedLoop`$_1$, which starts inside the call to >. After that, we will keep executing native code, even when our limit of 39 °C is exceeded. When we eventually exit the program, an additional loop will be generated which executes all the 1 + □ frames we pushed.

2.5 Section Conclusion

In this section, we have introduced effect handlers and provided an intuition of their operational semantics in terms of operations on the metastack. We then introduced our bytecode format BC by example and showed how our tracing JIT implementation can remove almost all of the overhead introduced by effect handlers. Instructions that remain fall into three categories: guards that are required to check whether the trace is still valid, operations unrelated to the use of effect handlers, and reifying operations at the boundaries between bridges, loops, and interpreted code. While all examples so far have been in Eff, we also developed language implementations for Effekt and Koka. The major difference between these languages is how we find the handler to execute for a given effect operation. In Effekt, instead of binding handlers dynamically, handlers are directly passed as capabilities. Every handler introduces and closes over a fresh prompt, which results in handlers that are bound lexically [Biernacki et al. 2019; Brachthäuser et al. 2020; Zhang and Myers 2019]. In Koka, we maintain a global evidence vector [Xie and Leijen 2021], which, for each effect currently allowed, stores both the handler implementation and the associated prompt. This—in comparison

Program syntax:

Program	P	$::=$	$\overrightarrow{B_i}$	programs
Blocks	B	$::=$	$\{\ l(\overrightarrow{x_i})\ :\ s\ \}$	basic blocks
Statements	s	$::=$	$\cdots$	see Appendix A.1
normal control flow				
		$\vert$	$\mathsf{jump}\ l$	jumps
		$\vert$	$\mathsf{push}\ l\ (\overrightarrow{x_i});\ s$	pushing frames
		$\vert$	$\mathsf{return}\ (\overrightarrow{x_i})$	returning
control operators				
		$\vert$	$x_o \leftarrow \mathsf{new\ stack}\ l(\overrightarrow{x_i})\ @\ x_p;\ s$	creating stacks
		$\vert$	$\mathsf{push\ stack}\ x;\ s$	pushing stacks
		$\vert$	$x_o \leftarrow \mathsf{shift}\ x_p;\ s$	capturing stacks
		$\vert$	$x_o \leftarrow \mathsf{new\ stack}\ l(\overrightarrow{x_i})\ @\ x_p\ \mathsf{with}\ x_b;\ s$	create stack with dynamic binding
		$\vert$	$x_o \leftarrow \mathsf{get\ dynamic}\ x_p;\ s$	accessing dynamic bindings
VTables	V	$::=$	$\{\overrightarrow{m_i\ \mapsto\ l_i}\}$	virtual tables
Labels and tags	l, t, m			
Variables	x			
Values	v	$::=$	M	continuations (see below)
		$\vert$	$\cdots$	see Appendix A.1

Stack syntax:

Stacks	k	$::=$	$\#$	empty stack
		$\vert$	$l(\overrightarrow{v_i}) :: k$	stack frame
Metastacks	M	$::=$	$\circ$	empty metastack
		$\vert$	$k\ @\ v\ \mapsto v\ :::\ M$	stack with prompt v

Fig. 4. Excerpt of the syntax of the bytecode format BC, relevant to express effect handlers.

with the Eff implementation—corresponds to simulating dynamic binding via mutability. The optimized traces for both Effekt and Koka look similar to the ones for Eff, given input programs ported to those languages, as we will see in Section 6.2, with differences described in Section 6.4.

3 The Bytecode Format BC

To implement a single tracing JIT compiler for our selection of source languages, we translate them to a common bytecode format BC. In this section, we briefly present BC and hint at the operational semantics of selected language constructs, not only to provide a mathematical intuition, but because it very closely describes our implementation in RPython. The bytecode format was designed to translate different forms of effect handlers in a way that is close to their standard implementation, that is, it tries to be intentionally unsurprising for the most part. It is designed for a register machine with an unbounded number of registers, where programs are split into blocks, which start at the points we can jump or return to. The simplified syntax of BC is shown in Figure 4; for the full description of BC and its abstract machine semantics, as well as further explanations, we refer to Appendix A.1.

3.1 Delimited Control

To support control operators and to capture parts of the stack as the continuation, our stack is segmented. As usual [Danvy and Filinski 1990; Dybvig et al. 2007], the *metastack* is a linked list of *stacks*, which itself is a linked list of *stack frames*. The syntax of stacks and metastacks is formally defined in Figure 4. Metastacks M are constructed using ::: and $\circ$, where every element is annotated with two values: (1) a prompt that is used to select specific parts of the stack (resp. continuation) to capture, and (2) the current dynamic value (explained in Section 3.2). Stacks k act like “normal” program stacks, where # denotes the empty stack and :: pushes a frame, remembering label l and closure $\overrightarrow{v_i}$. *e.g.*, the stack from Figure 1e would be denoted as $\#@\mathit{Emit} \mapsto h$ ::: `countEmitCont()` :: ...$@v_0 \mapsto v_1$ ::: $\circ$. The two-level structure allows us to directly iterate or move the stack segments between prompts, without going over individual frames. As we will see in Section 4, this immutable nested structure of linked lists also matches our implementation.

BC includes statements to express (multi-prompt) delimited control operators [Dybvig et al. 2007]. Roughly, we can translate the usual multi-prompt variant of reset_0 and shift_0 [Danvy and Filinski 1989; Shan 2004] to our bytecode instructions as follows:

$$
\begin{array}{lll}
\llbracket\ \mathrm{reset}_0(p)\ \{\ \mathit{term}\ \}\ \rrbracket & \llbracket\ \mathrm{shift}_0(p)(x_c)\ \{\ \mathit{term}\ \}\ \rrbracket & \llbracket\ \mathrm{resume}\ x_c\ (\overrightarrow{x_i})\ \rrbracket \\
\quad =\quad x_p \leftarrow \llbracket p \rrbracket; & \quad =\quad x_p \leftarrow \llbracket p \rrbracket; & \quad =\quad \text{push stack}\ \llbracket x_c \rrbracket; \\
\qquad\quad x_k \leftarrow \text{new stack}\ l()\ @\ x_p;^{*} & \qquad\quad \llbracket x_c \rrbracket \leftarrow \text{shift}\ x_p; & \qquad\quad \text{return}\ (\overrightarrow{\llbracket x_i \rrbracket}) \\
\qquad\quad \text{push stack}\ x_k; & \qquad\quad \llbracket \mathit{term} \rrbracket & \\
\qquad\quad \llbracket\ \mathit{term}\ \rrbracket & &
\end{array}
$$

*: where $\{\ l(\overrightarrow{x_i})\ :\ \text{return}\ (\overrightarrow{x_i})\ \}\ \in P$ with l fresh and the x_i appropriate for the return type

Here we see that reset, which delimits a computation, is decomposed into creating a fresh stack and pushing that stack onto the current metastack. Similarly, resuming a continuation is decomposed into pushing it and returning to it. Decomposing standard control operators into smaller instructions simplifies their implementation and is more flexible in some cases. For instance, by not (immediately) returning after push stack, we could support bidirectional handlers [Zhang et al. 2020].

We can use new stack to create new stack segments with a given first frame, shift to capture the part of the stack up until including some prompt, and push stack to (re)install stack segments onto the current stack. A more detailed explanation can be found in Appendix A.1.1.

3.2 Dynamic Binding

Effect handlers traditionally can be summarized as *“dynamic binding plus delimited control”*. So far, we have only seen the aspect of BC that models delimited control. In general, dynamic binding can be implemented in terms of delimited control [Kiselyov et al. 2006] – a technique commonly used to describe the semantics of effect handlers [Forster et al. 2017; Hillerström et al. 2017; Kammar et al. 2013]. To avoid the indirection and the potential overhead of an encoding, here we refrain from doing so and directly implement dynamic binding. Specifically, we support *immutable* dynamic variables, which are conceptually allocated on the stack. Such a binding can be installed by using a variant of new stack that also takes a value to be bound to the prompt, and can be looked up using the special get dynamic instruction. Further details can be found in Appendix A.1.2.

3.3 Other Constructs

The calculus contains additional standard constructs, for creating objects and calling methods (without inheritance), register management, conditional and unconditional jumps, stack management, constructing data, and pattern matching, which are defined in Appendix A.1. This also includes a formal definition of how we resolve labels l to blocks B in the obvious way.

3.4 Source Languages

Figure 5 gives a brief overview of how effect handlers in the three source languages are translated to BC. In Eff, effect handlers are dynamically scoped like exception handlers in many languages. To achieve this, we have a global prompt per effect and store the handler on the stack, next to the prompt. In contrast, in Effekt, handlers are bound lexically and implemented by passing capabilities

	Eff	Effekt	Koka
Declaration	`global Ef ⟵ [[fresh prompt]]`		`global evv ⟵ ref([])`
Handler definition		`p ⟵ [[fresh prompt]]`	`p ⟵ [[fresh prompt]]`
and installation	`h ⟵ new { op() ⇒`	`cap ⟵ new { op() ⇒`	`h ⟵ new { op(p) ⇒`
	`k ⟵ shift Ef`	`k ⟵ shift p`	`k ⟵ shift p`
	`// handler body`	`// handler body`	`// handler body`
	`}`	`}`	`}`
			`w`$_{\emptyset}$ `⟵ evv`
			`ev ⟵ Ev("Ef",p,h,evv)`
			`evv ⟵ [[copy evv]]`
			`evv ⟵ [[add ev to evv]]`
	`s ⟵ new stack @ Ef with h`	`s ⟵ new stack @ p`	`s ⟵ new stack @ p`
	`push stack s`	`push stack s`	`push stack s`
	`// ...`	`// ... pass down cap`	`// ...`
			`evv ⟵ w`$_{\emptyset}$
Resuming	`push stack k`		
	`return x`		
Effect invocation	`h ⟵ get dynamic Ef`		`ev ⟵ evv at [[static idx]]`
	`h.op()`	`cap.op()`	`[[ev.h]].op([[ev.p]], ev)`

Fig. 5. High-level overview over the different translations in BC-like pseudocode. Simplified. Partially reordered within cells to highlight similarities.

that close over the prompt. Here, we generate a fresh prompt each time we install a handler. In Koka, handlers are scoped dynamically but stored in a global evidence vector. Again, we generate a fresh prompt for each handler, use it to delimit the continuation, and associate it with the handler in the evidence vector, as described in Xie and Leijen [2021]. This also allows for masks/lifts [Biernacki et al. 2017; Convent et al. 2020]. The translations are described in more detail in Appendix A.3.

4 Implementation

To implement the three languages using our JIT, we reuse the frontend of the respective language implementation and add a backend that generates a common core representation (MCore), which we then transform through various mostly-standard compiler phases, to finally emit the bytecode format described here. The implemented phases on MCore are (in order): some light desugaring, an ANF-like transformation (repeated later to re-establish the postcondition), dealiasing (also repeated later), closure conversion, lambda lifting, and transformation of recursive bindings. After the (now trivial) transformation to a simple assembly format, we then perform register allocation (without spilling), number the resulting blocks, and remove some obviously unnecessary pushes introduced by the translation. After this, we emit BC in a straightforward JSON encoding.

The JIT was implemented by writing a bytecode interpreter in RPython and using the RPython toolkit to generate a tracing JIT from this. For simplicity of implementation, we do not actually encode BC into a binary format, but represent it as a JSON file, which is parsed and loaded up-front. The implementation of the interpreter closely follows the abstract machine semantics outlined above. One of the key decisions in the implementation of our interpreter is that almost all runtime structures, like the call stack, are immutable. For the stack, this means that it is a singly-linked list with immutable frames and immutable list structure. While this might seem nonstandard for a bytecode interpreter, it turns out to be a key enabler for JIT optimizations in the RPython toolkit. This way, the JIT compiler can easily specialize on whole structures, while only checking the address of the roots of the relevant structures. In addition to the implementation of an interpreter, the RPython framework requires annotations for code positions where multiple code paths might merge, and in positions where we might start to trace a loop [Bolz et al. 2009]. As in previous tracing JITs [Bala et al. 2000], we potentially enter the JIT at all syntactical backwards jumps in the code, which ensures that all loops will eventually be traced. This is a relatively simple condition that performs reasonably well. However, we further optimized places where the JIT compiler starts tracing in two ways (Section 4.1.1 for additional JIT entrypoints and 4.1.4 for stack context).

4.1 Implemented Optimizations

While a naive implementation of the bytecode interpreter already performs quite well, we implement multiple optimizations to improve its performance (evaluated in Section 5.3).

4.1.1 Additional JIT Entry Points. RPython requires us to mark positions in the implementation where it can start tracing. Each loop should go through at least one of those points. An easy and often used heuristic is to do this on every syntactical backwards transfer of control. For breaking all loops, and thus all hot ones, it would be enough to just allow entering the JIT on those control transfer operations. However, RPython will only virtualize allocations and dereferences within one loop or bridge, but not across bridges. That is, objects that escape the loop need to be allocated at the end of every iteration, hindering RPython's allocation removal [Ardö et al. 2012; Bolz et al. 2011]. It is thus generally important to cut the loop *before* values are allocated and *after* they are used. Specifically, to avoid allocations of stack frames, we additionally enter the JIT before each `push` instruction. This way the frame allocated by `push` will be part of the trace and can be virtualized. This is particularly important for sequences of `push` – `return` instructions: if we start tracing inside of a non-tail function, we always have to re-push, and thus allocate, its return frame in every iteration. If we start tracing before, we fully remove the allocation.

4.1.2 Fast-check for Prompt Equality by Code Position. Prompts are implemented as pointers, that is, we use the memory allocator to generate fresh values. This is a simple implementation and usually, RPython can reason about those allocations much better than, e.g., about a counter. When generating prompts dynamically, however, there could be an unlimited number of different prompts encountered for the same handler. When searching for a prompt on the stack, we compare each installed prompt with the one we are searching for. When tracing this, we generate a specific trace for the exact prompts we encountered during trace generation, thus specializing to the exact sequence of prompts. This is overly specific, though. We note that, while there can be arbitrarily many prompts at runtime, two prompts can only ever be equal if their definition site agrees. Thus, to generate loops that are more general, we instead first check the definition sites of the prompts when tracing and compare these in a respective guard. The trace generated then is still valid as long as the *definition sites* of skipped prompts are the same, which limits the number of possible values statically. On the flip side, this generates slightly more operations in the trace.

4.1.3 Specializing Data, Object, and Stack Frames. Closures, stack frames, data values, and objects must internally store an environment. Since RPython does not support data types of dynamic size, and the sizes of these environments are unknown when compiling the JIT interpreter, environments are represented using arrays in RPython. This leads to the allocation of an array for the field values, which is inefficient for common cases like cons cells in lists. To address this, we apply a standard technique and generate specialized variants for common cases [Bauman et al. 2015a], specifically for: *(a)* stack frames with up to 13 fields, *(b)* data with up to 6 fields, and *(c)* objects with up to 11 fields. In these specialized variants, all fields are stored directly in the object, eliminating the indirection through an array. To avoid dynamic dispatch when accessing fields in these specialized versions, we inform the JIT of the correct specialization using information from the program code. At each use site, we can determine which specialized version to use by storing the frame type indexed by the return program counter (for stacks), storing environment sizes in the virtual table (for objects), or computing the variant based on the match clause (for data).

4.1.4 Stack Context for False Loop Detection.

RPython would usually detect a loop whenever it reaches the same (smaller) program counter value, that is, the same bytecode instruction, again. If we now, *e.g.*, call a library function twice in a row, we will potentially trace a *false loop* that will never execute more than one iteration. This problem is well known for tracing JITs, and can be alleviated by considering (some part of) the calling context [Hayashizaki et al. 2011]. Thus, we also consider the target of the topmost stack frame when determining the program location. Taking one frame into account seems to be a good trade-off for the programs we evaluated this on, but this is configurable in the implementation, and other choices are better for some of the benchmarks.

4.1.5 Separate Compilation.

We support separate compilation and dynamic code loading. To do so, we relocate newly loaded blocks into the existing list of program blocks and update internal references during loading. Since this changes the list of program blocks, the program is no longer constant in principle. However, we make sure that the JIT will speculatively treat the list of program blocks as immutable and constant. To maintain correctness, this means all generated traces need to be invalidated when loading new code. This optimization means we can optimize separately-compiled code just as well as other code. In fact, all benchmarks in Koka are compiled separately by module, including standard library functions to implement handlers. Also, for Eff, we compile the benchmark code separately from the wrapper code that parses the input and prints the result. However, it also leads to a quite significant startup time for Koka, which starts off loading a big standard library and running all static initializers. This could be improved using standard techniques like lazy-loading, or a more optimized loading mechanism, but this is out-of-scope for this paper.

4.2 Limitations

There are a few noteworthy restrictions of our current implementation for the different languages. First, our implementation of Eff does not fully support `finally`-clauses, although the bytecode could. Additionally, the programming language Koka features a few additional concepts that we currently do not (reliably) support to keep the implementation effort manageable. Some of these concepts are related to effect handlers, such as shallow resumptions and special mask operations [Convent et al. 2020]. We conjecture that these additional features could be implemented without changes to the bytecode format. Koka also supports additional features, unrelated to effect handlers, such as special operations for optimizing tail-recursion modulo context [Leijen and Lorenzen 2023]. Again, these features were not needed for the benchmarks, and we do not support them at the moment.

5 Performance Evaluation

To evaluate the effectiveness of tracing JIT compilation for effect handlers, we first will evaluate the performance against other implementations of the same languages, and among our implementations. Then, we will benchmark against a selection of state-of-the-art language implementations, and finally conduct an ablation study to evaluate the influence of our optimizations.

5.1 Benchmark Descriptions

Most of the control-effect benchmarks are from a community-maintained benchmark suite [Hillerström et al. 2023], also described in Appendix A.2. However, to better explore certain aspects of the implementations, we added a few additional benchmarks: *(1)* **unused-handlers** is a variation of the **countdown** benchmark from the community benchmark suite, designed to explore the case where the handler is separated from the call site by multiple (unused) handlers. To achieve this, we insert multiple (unused) handlers for a yield effect between the inner loop and the state handler. This is similar in intent and structure to a benchmark used by Kiselyov and Ishii [2015]. *(2)* **to-outermost-handler** is very similar to **unused_handlers**, but makes it so the unused handlers that separate the call site from the handler are also for the same state effect, and defined at the same program position. This is meant to potentially confuse heuristics based on handler type and definition site. We were unable to implement this benchmark in Eff, since it requires a way to skip handlers. In Koka, we use masking to achieve this, while in Effekt, we add a higher-order function with an appropriate type (not mentioning the effect). *(3)* **multiple-handlers** is designed to simulate the case where one occurrence of an effect operation is handled by different handlers at different points in the program execution. It defines a `generator` function which emits/yields values using an effect. This is subsequently handled by three different handlers in three different calls, that compute the square sum, the sum, and the number of emitted values, respectively. *(4)* **counter** (koka only) is a variant of **countdown** where the effect operation is explicitly annotated as linear (`fun` instead of `ctl`, also see Brachthäuser and Leijen [2019]). It was taken from the standard set of benchmarks provided with the Koka language implementation. *(5)* **startup** is a benchmark implementing a constant 0 function and is meant to evaluate startup times for the different implementations.

Finally, for evaluating direct-style performance, we used a subset of the benchmarks from [Marr et al. 2016], excluding the macro benchmarks, also described in Appendix A.2. For our backends, we only implemented those in Effekt. In contrast to the benchmarks provided there, we do not use the harness but a simple wrapper to run complete runs with a given number of iterations. In this way, the methodology can be the same as for the other benchmark results reported in this paper. Apart from minor changes to support this (*e.g.* exports), the benchmarks remain unchanged.

5.2 Benchmarking Methodology

The benchmarks were run using `hyperfine` version 1.19.0 [Peter 2024] on NixOS 25.05 (Linux 6.12.16) with a x86_64 Intel i7-8550U CPU. For both our modified implementations and the other cases we used Eff version 5.1 (at `130709b9`), Effekt v0.19.0 (at `acb9c982`), and Koka version 3.1.2 (at `3b2083d4`). For the external benchmarks, we used Ocaml 5.2.1, Python 3.12.9 and PyPy 7.3.17 (for Python 3.10.14), Lua 5.2.4 and LuaJIT 2.1.1713773202, and Node 22.14.0 (with V8 12.4.254.21). The benchmarks were also run on M1, for which the results can be found in Appendix A.7.

All benchmarks were first run once to check their output, as well as potential errors. This also sorted out benchmarks that run for more than 90 seconds, which are reported as >90s based on this. The other timings are of full program runs, which includes the time for loading the bytecode, as well as tracing and code generation in the JIT, but not the time to compile them to bytecode. Each was run for at least 20 runs or 6 seconds[2]. The reported numbers are the arithmetic mean over those runs. Before each set of runs, one warmup run was executed to fill potential disk caches.

As a summary number, we also report the geometric mean slowdown compared to the JIT implementation. This "geomean slowdown" reported in the results is the geometric mean of the slowdown for the cases where both the implementation in question and the base implementation ran successfully, not including those that timed out. The base implementation is the respective JIT implementation for comparisons within a language and the Effekt JIT implementation otherwise.

5.3 Benchmark Results

5.3.1 Internal performance evaluation. Figure 6 shows the timings of different backends for the three languages on the control-effect benchmarks. Here, the geometric mean slowdowns are

[2] This includes some of the overhead of benchmarking, so the resulting times add up to slightly less.

	Eff			Effekt				Koka		
	JIT	Ocaml	Ocaml*	JIT	LLVM	JS	ML	JIT	C	JS
countdown	0.695	13.156	0.110	0.257	1.322	1.759	**0.105**	0.351	12.904	2.001
counter	—	—	—	—	—	—	—	**0.417**	8.248	1.304
fibonacci-recursive	10.324	62.564	**1.548**	8.399	2.779	33.609	2.090	4.688	13.627	8.397
generator	**0.504**	3.772	3.078	0.754	6.368	8.856	✗	0.756	> 90.000	30.176
handler-sieve	5.301	39.965	14.196	4.696	**1.813**	3.427	✗	5.119	9.948	≡↯
iterator	0.150	4.694	1.401	**0.022**	0.325	0.456	0.148	0.275	1.880	0.939
multiple-handlers	0.963	16.593	—	**0.891**	1.511	1.436	1.807	1.122	48.402	5.508
nqueens	0.900	✗	0.432	1.465	1.224	2.445	**0.164**	0.934	30.148	3.970
parsing-dollars	0.532	27.038	1.607	0.520	4.469	1.169	**0.289**	0.771	22.949	12.624
product-early	0.818	7.833	1.049	0.516	0.537	2.094	**0.465**	0.776	33.661	3.985
resume-nontail	0.206	2.092	0.213	0.180	0.175	OOM	**0.174**	0.429	31.017	≡↯
startup	0.005	0.002	0.002	0.005	0.002	0.034	0.002	0.239	**0.002**	0.064
to-outermost-handler	—	—	—	2.375	1.361	1.772	✗	**0.777**	17.633	3.332
tree-explore	0.551	1.808	**0.286**	0.378	0.621	2.143	0.469	0.662	3.305	1.210
triples	0.315	0.857	0.159	0.163	0.288	1.913	**0.075**	0.755	44.092	5.161
unused-handlers	4.875	> 90.000	—	1.544	1.317	1.792	✗	**0.353**	12.743	1.970
geomean slowdown	1.000	7.830	0.962	1.000	1.508	3.507	0.696	1.000	11.322	4.242

Fig. 6. Runtimes of the benchmarks. Time in seconds. Fastest in **bold**. ≡↯ marks stack overflows, — unimplemented benchmarks and ✗ failing compilations. Geometric mean slowdown is relative to JIT.

w.r.t. the JIT implementation of the same language. We compare against the `plain-ocaml` backend of Eff ("Ocaml"), as well as the version at `oopsla-2021Artifact`, which should agree with the artifact [Karachalias et al. 2021a] ("Ocaml*"), the LLVM-based ("LLVM") and JavaScript ("JS") backends of Effekt, the discontinued [Brachthäuser 2024] ML backend of Effekt ("ML"), and the C ("C") and JavaScript ("JS") backends of Koka. For Eff, the JIT implementation is faster than the `plain-ocaml` backend using the same Eff version in all but **startup**. The Eff `plain-ocaml` backend from the OOPSLA artifact [Karachalias et al. 2021a] is similar overall, being faster than the JIT for **countdown**, **fibonacci-recursive**, **nqueens**, **tree-explore** and **triples**, and slower in the others. The relative standard deviation was less than 3% for all benchmarks on Eff. Note that the **nqueens** benchmark for the newer Eff is not shown since the code generated by Eff resulted in a type error, which we were unable to fix. Neither could we run our added benchmarks **multiple-handlers** and **unused-handlers** on the old version, due to an assertion failure in the Eff compiler. For Effekt, the ML backend is fastest overall by a factor of almost 2x, and fastest in all but the Benchmarks that aren't supported by it and **iterator**, **multiple-handlers** and **tree-explore**. In all of those three benchmarks and **generator**, JIT is the fastest implementation. In **handler-sieve**, **to-outermost-handler** and **unused-handlers**, the LLVM backend is faster than the JIT. The ML backend does not support **handler_sieve**, **to_outermost_handler**, and **unused_handlers**. The relative standard deviation was less than 3% for all benchmarks on Effekt except **generator** for LLVM and JavaScript (4% resp. 3.6%), **nqueens** for JavaScript (4.5%) and **parsing-dollars** for LLVM (4%). For Koka, the JIT backend is fastest in all benchmarks, but has a high startup time. In the results measured on M1 (see Figure 20 in the appendix), though, the C backend outperforms the JIT for **fibonacci-recursive**. The JavaScript backend also often outperforms the Koka C backend on those benchmarks. The relative standard deviation was less than 4% for all benchmarks on Koka except for **multiple-handlers** (4.5%), **nqueens** (5.9%) and **parsing-dollars** (4.6%) on JS and **parsing-dollars** for JIT (6.3%).

5.3.2 *External performance evaluation.* Figure 7a shows the results for the subset of the benchmarks from Marr et al. [2016] for direct-style code. Here, we compare against the subset of other

(a) direct style	Effekt	JS	Lua		Python	
	JIT	V8	LuaJIT	Lua	CPython	PyPy
bounce	0.816	0.412	0.898	14.286	13.256	**0.410**
list-tail	0.570	**0.545**	1.209	9.100	8.356	2.335
mandelbrot	0.184	0.087	**0.051**	0.488	1.192	0.154
nbody	0.203	0.071	**0.057**	1.598	1.787	0.198
permute	2.464	**0.752**	0.862	20.946	24.204	2.160
queens	4.021	**0.675**	0.950	12.261	11.730	1.266
sieve	1.119	0.460	**0.455**	5.154	11.077	0.647
storage	0.710	**0.325**	1.779	5.544	5.865	0.929
towers	1.821	**1.370**	1.883	33.920	27.371	4.481
geomean slowdown	1.000	0.437	0.632	7.755	9.215	0.984

(b) control effects	Eff	Effekt	Koka	JS	OCaml 5	Python	
	JIT	JIT	JIT	V8	OCaml 5	CPython	PyPy
countdown	0.695	**0.257**	0.351	11.188	5.829	> 90.000	10.463
fibonacci-recursive	10.324	8.399	4.688	4.002	**1.473**	54.703	10.694
generator	**0.504**	0.754	0.756	28.535	1.203	41.976	13.544
handler-sieve	5.301	**4.696**	5.119	≡↯	9.894	≡↯	> 90.000
iterator	0.150	**0.022**	0.275	1.274	0.621	3.451	0.297
multiple-handlers	0.963	**0.891**	1.122	8.443	—	21.479	1.756
parsing-dollars	0.532	**0.520**	0.771	11.064	4.322	> 90.000	2.648
product-early	0.818	0.516	0.776	11.530	**0.238**	≡↯	5.413
resume-nontail	0.206	**0.180**	0.429	—	0.464	—	—
startup	0.005	0.005	0.239	0.031	**0.002**	0.020	0.090
geomean slowdown	1.419	1.000	2.233	13.611	2.213	21.846	8.269

Fig. 7. Runtimes of external benchmarks. Time in seconds. Lower is better, fastest in **bold**. ≡↯ marks stack overflows and — unimplemented benchmarks. Geometric mean slowdown is relative to Effekt JIT.

languages, for which implementations of the benchmarks existed already, *i.e.* the V8 JavaScript implementation [Google 2025; OpenJS Foundation 2025], the CPython and PyPy implementations of Python [Bolz et al. 2009; Python Software Foundation 2025], and the LuaJIT and default implementations of Lua [Ierusalimschy et al. 2024; Pall 2025]. The geometric mean slowdown is w.r.t. Effekt JIT. Overall, V8 is the fastest measured implementation, faster than our JIT by a factor 2.3x. In each benchmark, either V8 or LuaJIT are faster than our JIT backend. Our JIT is faster than PyPy in **list-tail**, **storage** and **towers**, and slower by a larger factor in **bounce**, **queens** and **sieve**. For the other benchmarks, the differences between our JIT and PyPy are below 20%. The relative standard deviation for these benchmarks is below 4% except for **towers** on Effekt (6.5%) and **queens** (7.2%) and **permute** for LuaJIT. **permute** had a large relative standard deviation of 112% for LuaJIT.

Figure 7b shows the benchmarking results for our implementations compared with other implementations of the control-effect benchmarks. We include Ocaml 5 [Sivaramakrishnan et al. 2021], for which the benchmarks were implemented in the benchmark suite already. To further anchor our results to existing state-of-the-art language implementations, we implemented a subset of the community benchmark suite using existing language features like generators and exceptions, where possible in JavaScript and Python. The JavaScript and Python versions were constructed by translating the benchmarks from the community benchmark suite that only use effect handlers in the exception-like (*i.e.* zero-shot) or generator-like (*i.e.* one-shot and tail) form. To do this, we used the standard encoding (similar to [Alvarez-Picallo et al. 2024]), using message objects to distinguish multiple effect operations and re-yielding where appropriate. Deviating from this, we manually removed obvious overhead by not using message objects if there is only one generator-like

	JIT	no 4.1.1	no 4.1.2	no 4.1.3	no 4.1.4	more 4.1.4	none
Eff	1.000	0.930	1.000	1.111	1.610	0.991	1.967
Effekt	1.000	0.899	1.004	1.000	1.497	0.871	2.076
Koka	1.000	0.999	0.994	1.125	2.114	1.094	2.366

Fig. 8. Geomean slowdowns of the benchmarks on jit with different optimizations disabled or changed. Summarized from timings per benchmark in Figure 19 in the appendix.

effect reaching a certain handler, translating tail-recursive functions to loops, and using standard constructs (as `for ... of ...` in JavaScript) where their semantics matches exactly.

In the geometric mean, the Effekt JIT is fastest, followed by the JIT for Eff and Koka as well as Ocaml5, which is similar to the JIT for Koka. The others are slower by a significant amount. This is also visible in the individual benchmarks. Ocaml5 is fastest in **fibonacci-recursive** (which does not use handlers) and **product-early**. For all other effect-handler benchmarks (not counting **startup**), one of the JIT backends is fastest, in all cases but **generator**, the Effekt one. For the additional values not in Figure 6, the relative standard deviations are below 4%.

5.4 Influence of Optimizations

Finally, Figure 8 shows the geometric means over the slowdowns on the control-effect benchmarks when disabling certain (or all) optimizations in the JIT. The individual data for the benchmarks can be found in the appendix in Figure 19. The biggest slowdown can be seen for disabling contexts for false loop detection (Section 4.1.4), while disabling additional JIT entry points (Section 4.1.1) makes the benchmarks faster in the geometric mean, and comparing labels by definition site first (Section 4.1.2) does not show a significant effect overall. For specialization (Section 4.1.3), or using more context (Section 4.1.4), the influence seems to depend on the source language.

6 Discussion

Given our empirical results from Section 5, we will now discuss our answers to the research questions posed in the introduction.

6.1 Comparison with AOT (RQ 1)

> *How does tracing JIT compilation compare with existing ahead-of-time optimizing implementations of effect handlers?*

To answer this question, we will first look into how the JIT compares to the other backends for each of the languages in turn.

6.1.1 Eff. The JIT backend is faster than the Eff implementation at the same version using the `plain-ocaml` backend in all benchmarks. The Eff version from the artifact by Karachalias et al. [2021b] is significantly faster than the Eff version we based our modifications on. This is because of changes in the translation[3]. For **countdown** and **fibonacci-recursive**, the Eff JIT implementation is also significantly slower than the other JIT implementations, which is discussed in Section 6.4. For the other benchmarks (except **startup**), the faster Eff implementation is faster by at most a factor of 2.1x. Overall, the JIT is comparable in performance to the faster `plain-ocaml` backend.

6.1.2 Effekt. The JIT is faster than all other evaluated Effekt implementations in 4 cases, but is outperformed by the MLton backend for most benchmarks. However, it is worth noting that the whole-program optimizing MLton backend relies on fully monomorphizing stack shapes [Müller

[3] See https://github.com/matijapretnar/eff/issues/86#issuecomment-1493388587 for more details.

et al. 2023] and thus fundamentally cannot support the whole Effekt language, which is why there are no values reported for **generator**, **handler_sieve**, **to_outermost_handler**, and **unused_handlers**. It is also why this implementation was discontinued [Brachthäuser 2024]. If we exclude the MLton backend from the comparison, the JIT backend is fastest in more than half of the benchmarks. It also is always within a factor of 3.03x of the LLVM backend, while outperforming it by an order of magnitude in some cases. Overall, the JIT is outperformed by the more restrictive MLton backend, but is slightly more performant than the LLVM backend.

6.1.3 Koka. The JIT is faster than the other Koka implementations for all control-effect benchmarks except **startup**. Note, however, that, on x86 the C backend was significantly slower for us than on M1 (Figure 20 in the appendix). There, it is faster on **fibonacci-recursive**, which does not use effect handlers. Thus, for effect-handlers, the JIT clearly outperforms both Koka backends.

6.1.4 Conclusion. In summary, the tracing JIT implementation compares competitively with existing AOT implementations of effect handlers. Implementations that restrict programs to a subset of possible effect handler usages, or few with extensive optimizations, can outperform our current implementation significantly in some cases, but most are significantly slower. As we will see in Section 6.2, many standard optimizations for effect handlers emerge automatically. Tracing JIT compilation thus seems a good fit for implementing languages with effect handlers when focusing on the performance of control-effect-heavy code.

6.2 Advantages and Limitations (RQ 2)

> *What are classes of effectful programs that tracing JIT compilation can optimize well?*
> *What are the limitations of the approach?*

To answer this question, we will first split the benchmarks by their usage of effect handlers, more precisely by the way the captured continuation is used:

(1) It may not be used at all (zero-shot), as is the case in **product_early**. This case also occurs in **nqueens**, **parsing-dollars** and **triples**, which also include some of the other uses.
(2) It may be called once in tail postion (one-shot and tail) as in **countdown**, **handler-sieve**, **iterator**, **multiple-handlers**, and **parsing-dollars**.
(3) It may be called once in non-tail position (one-shot, non-tail) as in **resume-nontail**.
(4) Finally, it may be called multiple times (multi-shot), as is the case for **nqueens**, **tree-explore** and **triples**.

The other part of the implementation of effect handlers, the dynamic dispatch, can be optimized out in all cases. For a detailed explanation of why this is the case, we refer to Appendix A.5.

6.2.1 Case 1: zero-shot. In the zero-shot case, **nqueens**, **parsing-dollars** and **triples** are unlikely to be dominated by the performance of the zero-shot continuation. For **product-early**, the JIT is slightly faster than the general AOT implementations, and outperformed slightly by more optimized ones like the ML implementation.

Inspecting the traces for **product-early**, we find that the generated loop iterates over the list, checking that the value is not 0, and pushes stack frames according to the values. This part is unrelated to the effect handler and continuation capture. Additionally, some entry bridges and bridges are generated, one of which is for the case where the number is 0, in each of the backends. In all three backends, this bridge inserts guards for the current stack and removes the topmost stack segment, without copying it. Sadly, due to the position where the loop is split, it allocates stack frames for the next benchmark iteration.

In general, as described in more detail in Appendix A.4.1, if the captured continuation is not called and thus does not escape the trace, the continuation capture is optimized out in the JIT trace.

6.2.2 Case 2: one-shot and tail. In **iterator**, the JIT outperforms all other implementations significantly. For all backends, the JIT traces one loop, that can be optimized to just a simple counting loop for Effekt and Koka, while for Eff, some checks for the stack-shape stay in the optimized loop (*cf.* Section 6.4). The benchmarking result for Koka is dominated by startup time (cmp. **startup**). In **countdown**, JIT can also outperform most language implementations, except for the better Eff `plain-ocaml` backend and the Effekt ML backend, as well as the direct Ocaml 5 implementation. In Koka however, annotating the one-shot and tail nature helps the performance of the non-JIT implementations (**counter** is the annotated version). In **parsing-dollars**, the only implementation faster than the JIT implementations is the Effekt ML implementation. In all three JIT base implementations, one optimized loop is generated—that does not contain any operations leftover from the implementation of effect handlers—, accompanied by some bridges and entry bridges. In **handler-sieve**, the stack shape is highly dynamic, which is why it cannot be evaluated with the MLton backend for Effekt. The LLVM backend is faster than the JIT here, by a factor of 2.6x. For the other languages, the JIT outperforms all other implementations, with the Koka JavaScript backend leading to a stack overflow. The JIT traces three loops in all backends, which are a loop entering the handler and re-throwing, one resuming back through all handlers, and one returning out through all handlers at the end. Thus, resuming is split from capturing the continuation and some allocations for the handler do occur, which hurts JIT performance.

In general, again, no additional overhead for the continuation capture occurs when the continuation is resumed just once in tail position and this resumption is within the same trace. This is described in more detail in Appendix A.4.2.

6.2.3 Case 3: one-shot, nontail. The benchmark **resume-nontail** was specially constructed for this case. Performance of the JIT is very close to the faster Eff `plain-ocaml` backend and the Effekt ML and LLVM backends respectively, where the ML backend is the fastest implementation. For Koka, it outperforms the C backend, while the JavaScript backend stack overflows. In all three backends, three loops are generated for this benchmark, where one of those traces through the handler call, and the inner loop, and contains allocations for the additional stack frame. The second loop traces through the returns through the new stack. Finally, in all three backends one of those loops is duplicated with a different split point and context.

Again, this case is such that, at least conceptually, we need to "insert" some new stack frame(s) above the prompt on the stack. This means that due to our linked stack representation, without any knowledge about those frames, we will have to allocate at least those new frames. Since every stack is delimited by a prompt and resuming always reinstalls the delimiter, we never have to (re-)allocate the individual stack frames in the prefix or suffix, but only the additional frames [Ploeg and Kiselyov 2014], and the outer-level linked list of the captured continuation. We have seen an example for this case in Section 2.

6.2.4 Case 4: multi-shot. The prototypical example for this case is backtracking search or nondeterminism. Here, the results are mixed. Note that some implementations do not support this usage of effect handlers to implement other cases efficiently [Sivaramakrishnan et al. 2021] or only support it under special circumstances [Ma et al. 2024].

The JIT is outperformed by a significant margin for **nqueens** by the better Eff `plain-ocaml` backend, and the Effekt ML and LLVM backends. For **tree-explore**, it is outperformed by the old Eff `plain-ocaml` backend but can outperform even the Effekt MLton backend. For **triples**, it is just about a factor of 2x from the old Eff `plain-ocaml` backend and the Effekt MLton backend. In summary, it *can* get reasonable performance in those cases, but it can also be significantly slower, in not-easily-predictable ways.

Resuming the continuation multiple times, the JIT compiler will in general generate multiple loops as well as potential bridges for alternative paths. As an example, the **triples** benchmark from the community benchmark suite [Hillerström et al. 2023] generates—depending on the source language—between 10 and 13 loops as well as between 17 and 30 bridges. Here, we can start to see a downside of tracing JIT compilation: Because we always inline the whole continuation, we duplicate it for all variants of a loop for different control flow paths. Especially for multi-shot handlers, which naturally have a complex control flow, this will lead to duplicated copies of parts of the code. While this tail duplication [Gal et al. 2009] potentially allows more specialized and thus efficient traces, it also means the JIT compiler spends more time compiling and uses more memory.

6.2.5 Capturing Large Continuations. The way our JIT optimizes stack capturing means that it effectively specializes to the stack shape of the captured continuation. Because of this, in the case where this shape becomes large enough, our implementation will run into the following issue: During tracing, when walking the metastack, each inspected stack segment contributes to the length of the unoptimized trace. Eventually, we reach an internal limit of RPython. In this case, no optimized loop will be generated, which results in a significant performance cliff. This case did not occur in the benchmarks, though. The issue stems from our decision to always specialize the code to the specific metastack shape, by unrolling the loop that captures the continuation. It would be possible not to specialize, *e.g.* based on some heuristic, at the cost of generating less efficient code in those cases.

6.2.6 Conclusion. In summary, if the captured continuation is used at most once, and within the same loop, the JIT can remove the stack operations, with minor overhead to extend and rebuild the stack structure if doing so in a non-tail manner. It sometimes struggles with multiple resumptions, where performance can degrade in some cases, amplifying the general problem of tail duplication for tracing JITs. Also, very large contiuations can be a problem (not further explored here).

6.3 Optimizations (RQ 3)

How can we optimize the performance of tracing JIT compilation for effect handlers?

Overall, the optimizations improve benchmarks by about a factor of 2x in the geometric mean, although this varies widely by benchmark and language implementations. While the optimizations improve the runtime by more than an order of magnitude for some benchmarks, like **iterator** for Eff and Effekt, for others there is almost no effect, *e.g.* **countdown** for Effekt. Overall, even this simplest implementation provides a reasonable performance in many cases.

6.3.1 Stack context for false loop detection (Section 4.1.4). When looking at the data from the ablation study, using the first stack frame to distinguish program positions (Section 4.1.4) and thus preventing false loops has a clear positive effect overall, independent of the source language. There are cases where it has a minor negative effect, though. In particular, **unused-handlers** and **tree-explore** on Eff are faster with this optimization turned off. This is due to a loop being unnecessarily duplicated because it gets used in different contexts. Using more context (i.e., two frames) has a minimal positive effect for Eff overall, and a slightly stronger positive influence for Effekt, but a very minor negative effect for Koka.

6.3.2 Specializing Data, Object, and Stack Frames (Section 4.1.3). Specializing the data, object and stack frame representations does have a small positive influence for Eff, Koka, and most benchmarks for Effekt. For **to-outermost-handler** and **unused-handlers** in Effekt, it has a negative effect, which makes the positive influence on the other benchmarks disappear in the geometric mean.

6.3.3 Additional JIT Entry Points (Section 4.1.1). Overall, this optimization does not improve the results in the geometric mean. For most benchmarks, it does not show any significant change at all. It does, however, speed up **multiple-handlers** in Eff and Effekt by a factor of 2x resp. 1.7x. It also has a slight positive effect for **handler-sieve** in those languages. However, **fibonacci-recursive**, **to-outermost-handler** and **unused-handlers** get significantly faster for those languages when disabling this optimization. For Koka, all differences from this optimization are relatively small.

6.3.4 Fast-check for Prompt Equality by Code Position (Section 4.1.2). This does not have a major impact on the performance for any of the benchmarks measured. This might be due to the fact that in the small micorbenchmarks used here, only very few different prompts occur, or prompts freshly generated in every iteration are within the trace, and can be readily reasoned about by the JIT.

6.3.5 Conclusion. Standard optimizations for tracing JIT compilers like using the context for false loop detection (Section 4.1.4) or specializing dynamically sized objects for common sizes (Section 4.1.3) do have a positive effect here, too. Further optimizations can help with specific benchmarks, but come with costs in other cases. Selectively applying them based on heutristics might be a way to further improve results. Also, other optimizations could, of course, be explored.

6.4 Variations of Effects (RQ 4)

> *Are there differences in how well tracing JIT compilation performs for different variations of effect handlers?*

The largest difference in benchmarks between the different JITs is in the **iterator** benchmark. Here, the Koka variant becomes dominated by the significant startup time (see subsection 6.4.3), which also explains the slightly larger difference for **resume-nontail** for Koka. The Eff JIT is by no means slow for **iterator**, but still less optimal than the Effekt variant, due to some additional guards left in for checking the stack shape (subsection 6.4.1), which also affects **countdown**. **triples** in our Koka implementation generates significantly more allocations in the traces, which at least in part are due to managing evidence vectors (subsection 6.4.4). **fibonacci-recursive** is significantly slower for Eff than the other languages on the JIT. This seems to be due to all function calls in Eff being curried (subsection 6.4.2). **unused-handlers** is significantly slower for Eff than for Effekt, which is in turn much slower than the Koka variant. The Eff implementation walks the stack twice — once for fetching the handler, and once for capturing the continuation. Koka does not capture the — one-shot and tail — continuation at all in this benchmark, and can use the evidence to directly get the handler. This is also why **to-outermost-handler** is faster in the Koka JIT backend. In all other cases, the implementations are within a factor of two of each other.

6.4.1 Prompt search for Eff*.* For Eff, some additional guards are left in for checking the stack shape for **iterator** and **countdown**. The additional guards originate from the fact, that the RPython JIT can more readily reason about the more locally allocated prompts in the Effekt and Koka implementations than about the global ones in the Eff implementation (*cf.* Figure 5). Because of the structure of our bytecode, it might walk the stack a second time to capture the continuation. This could be aleviated by introducing a combined instruction to do both.

6.4.2 Curried functions for Eff*.* As noted above, for **fibonacci-recursive**, Eff is significantly slower due to currying. All function applications are translated curried, and in **fibonacci-recursive**, the partially applied function escapes in one of the loops, which leads to additional allocations. In the other benchmarks, we did not observe any additional operations due to the currying, though, as the allocation is close to the invocation, and easily optimized when both are in the same trace.

6.4.3 Startup time for Koka*.* Koka has a significantly higher startup time than our other JIT implementations. This is due to it loading the extensive separately compiled standard library and running all static initializers therein. The simple approach to dynamic code loading we take (Section 4.1.5), while allowing us to optimize the code well, also might have an influence here.

6.4.4 Representation of evidence vectors in Koka*.* As noted before, we represent the evidence vectors in Koka as linked lists. This, together with the nested structure of the evidence, means that if the evidence vector changes in a trace and the changed evidence vector escapes, we have to generate a non-negligible amount of allocations. Because evidence vectors are sorted, this also potentially not only includes added handlers, but also the prefix of the vector.

6.4.5 Conclusion. While in some cases, the variant of handlers we implement, can have a significant influence on the performance, as for the prompt search in Eff (subsection 6.4.1) or the evidence encoding in Koka (subsection 6.4.4), by and large the differences only occur in specific instances or have a comparatively small effect. Possibly, specific optimizations would be able to remove those overheads. To summarize, JIT compilation seems to be a viable approach for various kinds of effect handlers, where lexical and evidence-based semantics do have slight advantages for some cases.

6.5 Baseline (RQ 5)

> *Does JIT-compiling effect handlers impact the performance of programs that do not use effects?*

As we have seen in Section 5.3, the Effekt JIT implementation is slower than some start-of-the-art language implementations by a factor of about 2.3x. Some of this is likely to be due to our implementation being significantly less optimized than industrial-grade JIT compilers. The performance being overall similar to PyPy, which also is implemented using RPython, further hints into this direction—though both show different strengths and weaknesses. Our implementation outperforms PyPy for the benchmarks **list-tail** and **storage**, in both of which a recursive function is central to the benchmark. This might be due to PyPy not being particularly optimized for more functional programs; Ocaml 5, for example, significantly outperforms the JIT for **fibonacci-recursive** from the effect-handlers benchmark suite.

For the control-effect benchmarks, optimizing for effects does pay of: Even when the translation is an idiomatic use of standard language features, like for **iterator** in JavaScript, our JIT can outperform top-of-the-line industry JITs like V8, and is faster in almost all of the benchmarks. This shows a clear potential for further optimizing control effects in mainstream language implementations.

6.6 Threats to Validity

Due to effect handlers being a relatively new language construct, there are not yet larger programs written using them. Thus, the quantitative evaluation has to rely on microbenchmarks, and might not generalize to future real-world uses of effect handlers. Also, analyzing our particular implementation means that our findings are limited to tracing just-in-time compilation, and might not generalize. Implementation decisions unrelated to the implementation of effect handlers might—and in some cases, do—have a non-negligible influence on the performance results. We tried to detect and aleviate these cases, but some such issues might remain.

7 Related Work

In this section, we discuss prior work related to efficient compilation of effect handlers, as well as other JIT compilation approaches for specific control effects, such as exceptions.

7.1 Abstract Machines for Control Operators

Similar to Pycket [Bauman et al. 2015b], we apply meta-tracing JIT compilation to (almost) off-the-shelf interpreters. Our abstract machine is thus, intentionally, mostly standard. Following Dybvig et al. [2007], the most significant difference to Hillerström et al. [2020], Biernacki et al. [2015], and Fujii and Asai [2021] is that we support *multiple prompts*. These facilitate the implementation of lexical effect handlers [Brachthäuser et al. 2020]. The biggest difference to Dybvig et al. [2007] is that we implement $\texttt{shift}_0$ rather than `control` (the continuation contains the delimiter), since this corresponds closely to *deep* effect handlers [Forster et al. 2017; Kammar et al. 2013].

7.2 Efficient AOT Compilation of Effect Handlers

To the best of our knowledge, no prior work on JIT compilation of effects and handlers exists. Most closely related is the work on ahead-of-time (AOT) compilation.

Pretnar et al. [2017] and later Karachalias et al. [2021b] define source-to-source rewrite rules for effect handlers, which forms the basis of the Eff [Plotkin and Pretnar 2013] language. Their rewrites push down effect handlers until they meet the corresponding effect operation in which case the handling can be reduced statically.

Schuster et al. [2022] give a translation for lexical effect handlers to iterated continuation-passing style [Danvy and Filinski 1990]. Their control operator takes *evidence*, which measures the distance between the definition site of the effect handler and the callsite of an effect. In contrast, our control operators are parametrized by prompts and we search for the correct handler by comparing prompts. Schuster et al. [2020] show that under certain assumptions all abstractions related to effect handling can be statically reduced. Müller et al. [2023] expands on the work of Schuster et al. [2022] and makes the necessary connection between lexical scoping and subregioning evidence. Using this technique, they report excellent performance. However, their implementation relies on whole program optimizations performed by MLton [Matthew Fluet [n. d.]; Weeks 2006] and as such does neither support higher-rank types nor effect-polymorphic recursion [Müller et al. 2023], that is, recursive calls under an additional handler.

Both approaches, the one of Schuster et al. [2020] resp. Müller et al. [2023] and Karachalias et al. [2021b] will most likely not be (fully) applicable in a setting with separate compilation, while JIT compilation approaches are naturally well suited for separate compilation.

Recently, Ma et al. [2024] proposed a new efficient runtime system for supporting effect handlers, for which they report very good performance. However, their approach heavily restricts the use of multi-shot continuations. Specifically, a continuation can only be resumed multiple times if it consists of exactly one stack segment.

7.3 JIT Compilation of Control Effects

The Java Hotspot Server compiler tries to directly connect `throw` statements with the surrounding exception handlers and replaces them by a direct jump [Paleczny et al. 2001]. This typically works after inlining has provided enough context for a matching exception handler to be found. Stadler et al. [2009] copy captured stack frames lazily upon returning from them, however they do not describe support for optimizing capture and resume with the help of the JIT compiler. In contrast, continuation capture is often free in our approach, since we heap-allocate immutable frames, which the JIT compiler can reason about and optimize.

The language implementation framework Truffle uses exceptions [Würthinger et al. 2013] to implement non-local control flow transfer. Again, after inlining those transfers are optimized to be efficient jumps. Zippy is a Python implementation using Truffle. Zippy uses inlining to optimize Python generator execution across `yield` expressions and replaces them with a jump if inlining can produce enough context to allow the resulting control flow to be analyzed at runtime [Zhang et al. 2014]. If the transformation succeeds, the execution of producer and consumer code gets fused and the overhead of using a generator is fully removed.

Pycket is a Racket implementation using the RPython framework [Bauman et al. 2015a, 2017]. Pycket is a direct implementation of a CEK machine [Felleisen and Friedman 1986] and uses RPython to make this approach efficient, similar to the implementation described in the present paper. The implementation approach of using the CEK machine makes first-class continuations efficient and the RPython JIT compiler can often replace the invocation of captured continuations into direct jumps, given enough inlined context.

Python has two main ways to support lightweight threads: asynchronous functions using `yield` and `greenlet` [Rigo et al. 2011], a third-party library that implements thread switching by stack copying. To the best of our knowledge the PyPy Python implementation [Bolz et al. 2009], also based on RPython, does *not* optimize either alternative in any way. For asynchronous functions yield points always terminate traces. For greenlets, the low-level stack switching is intransparently hidden behind a function call the JIT has no knowledge about. This way producer/consumer loops are optimized independent of each other and the stack switching cost is not removed.

8 Conclusion and Future Work

In this work, we started to explore JIT compilation as an implementation technique for effect handlers. Specifically, we investigated how tracing JIT compilation compares to existing ahead-of-time optimizing implementations, which classes of effectful programs can be optimized well by our JIT, how specific optimizations influence the performance, which differences there exist between the different variations of effect handlers, and how baseline performance is influenced. To this end, we translated three different source languages to a common bytecode format and implemented an interpreter using the RPython meta-tracing just-in-time toolkit. Our JIT implementation does not consistently outperform other heavily optimized implementations, but often provides competitive performance. In the JIT setting, common optimizations, implemented by other AOT compilers, emerge automatically and do not need to be implemented manually. Uses where the continuation is resumed at most once can be optimized well. Standard optimizations for tracing JIT compilers help, while our more specific optimizations only have a minor impact on most benchmarks. The differences for different variations of effect handlers are minor. Managing the evidence vector for Koka can incur overhead, but also help to avoid expensive stack search in at least one instance. The JIT can better reason about the lexical prompts in Effekt than the dynamic ones in Eff. For direct-style code, our implementation is outperformed by state-of-the-art language implementations, while outperforming their implementations of control-effects significantly.

One natural direction for future work would be to investigate other approaches of JIT compilation, such as rewrite based AST interpreters [Würthinger et al. 2013], or hand-rolled method-based JIT compilers, to evaluate whether and which of our results generalize to those settings. There are also many opportunities for additional optimizations on top of the current approach. For example, it might be interesting to explore heuristics for when to specialize to the particular handler, which we always do right now. Additionally, our approach benefits from new optimizations and solutions for common problems in the area of tracing JIT compilation, like tail duplication [Chevalier-Boisvert and Feeley 2015; Gal et al. 2009]. Last but not least, while we already support three different effect handler languages, it would be interesting to support additional features and variations of effect handlers. This may include support for masks / lifts [Biernacki et al. 2017; Convent et al. 2020] or shallow handlers [Hillerström and Lindley 2018], which make the currently used handler more dynamic and thus, make JITting particularly interesting. In this direction, it would also be interesting to explore more dynamic languages with effect handlers like Shonky [McBride 2016], and implementations of effects for dynamic languages like JavaScript.

9 Data-Availability Statement

There is an artifact [Gaißert et al. 2025] which contains the example programs, the modified versions of the Eff, Effekt, and Koka compilers (both source and binaries for the benchmarking system), the common part of the compilation pipeline, as well as the implementation of the RPython-based just-in-time compiler. This also includes (references to) the exact versions of the other implementations benchmarked, and the code of the benchmarks. It also contains some tooling to simplify compiling, executing and benchmarking the various variants on the given benchmarks or custom programs. Finally, it contains the JIT trace logs and the raw measurement data.

Acknowledgments

The work on this project was supported by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) – project number DFG-448316946.

References

Mario Alvarez-Picallo, Teodoro Freund, Dan R. Ghica, and Sam Lindley. 2024. Effect Handlers for C via Coroutines. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 358 (Oct. 2024), 28 pages. doi:10.1145/3689798

Håkan Ardö, Carl Friedrich Bolz, and Maciej Fijałkowski. 2012. Loop-aware optimizations in PyPy's tracing JIT. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 63–72. doi:10.1145/2384577.2384586

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/349299.349303

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. doi:10.1016/j.jlamp.2014.02.001

Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015a. Pycket: A Tracing JIT for a Functional Language. *SIGPLAN Not.* 50, 9 (aug 2015), 22–34. doi:10.1145/2858949.2784740

Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015b. Pycket: a tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 22–34. doi:10.1145/2784731.2784740

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 54 (oct 2017), 24 pages. doi:10.1145/3133878

Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. 2015. A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations. *ACM Trans. Program. Lang. Syst.* 38, 1, Article 2 (Oct. 2015), 25 pages. doi:10.1145/2794078

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2, POPL, Article 8 (Dec. 2017), 30 pages. doi:10.1145/3158096

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (Dec. 2019), 29 pages. doi:10.1145/3371116

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Allocation Removal by Partial Evaluation in a Tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Austin, Texas, USA) *(PEPM '11)*. Association for Computing Machinery, New York, NY, USA, 43–52. doi:10.1145/1929501.1929508

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. Association for Computing Machinery, New York, NY, USA, 18–25. doi:10.1145/1565824.1565827

Jonathan Immanuel Brachthäuser. 2024. *A Brief History of Effekt for Fellow Researchers*. https://effekt-lang.org/evolution

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020). doi:10.1145/3428194

Jonathan Immanuel Brachthäuser and Daan Leijen. 2019. *Programming with Implicit Values, Functions, and Control*. Technical Report MSR-TR-2019-7. Microsoft Research.

Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. 2018. Versatile Event Correlation with Algebraic Effects. *Proc. ACM Program. Lang.* 2, ICFP, Article 67 (July 2018), 31 pages.

Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 99–107. doi:10.1145/231379.231395

Maxime Chevalier-Boisvert and Marc Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 101–123. doi:10.4230/LIPIcs.ECOOP.2015.101

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo Bee Doo Bee Doo. *Journal of Functional Programming* 30 (2020), e9. doi:10.1017/S0956796820000039

Antonio Cuni. 2010. *High performance implementation of Python for CLI/.NET with JIT compiler generator for dynamic languages.* Dottorato di Ricerca in Informatica. Università degli Studi di Genova.

Olivier Danvy and Andrzej Filinski. 1989. A functional abstraction of typed contexts. *DIKU Rapport 89/12, DIKU, University of Copenhagen* (1989).

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the Conference on LISP and Functional Programming* (Nice, France). ACM, New York, NY, USA, 151–160.

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Concurrent system programming with effect handlers. In *Proceedings of the Symposium on Trends in Functional Programming*. Springer LNCS 10788.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective concurrency through algebraic effects. In *OCaml Workshop*.

R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. 2007. A Monadic Framework for Delimited Continuations. *Journal of Functional Programming* 17, 6 (Nov. 2007), 687–730. doi:10.1017/S0956796807006259

Matthias Felleisen. 1988. The Theory and Practice of First-class Prompts. In *Proceedings of the Symposium on Principles of Programming Languages* (San Diego, California, USA). ACM, New York, NY, USA, 180–190.

Matthias Felleisen and Daniel P. Friedman. 1986. Control Operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*. Elsevier (North-Holland), Amsterdam, 193–217.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the Expressive Power of User-defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Proc. ACM Program. Lang.* 1, ICFP, Article 13 (Aug. 2017), 29 pages.

Maika Fujii and Kenichi Asai. 2021. Derivation of a Virtual Machine For Four Variants of Delimited-Control Operators. In *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 195)*, Naoki Kobayashi (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:19. doi:10.4230/LIPIcs.FSCD.2021.16

Marcial Gaißert, CF Bolz-Tereick, and Jonathan Immanuel Brachthäuser. 2025. *Artifact of the paper 'Tracing Just-in-time Compilation for Effects and Handlers'*. doi:10.5281/zenodo.16901452

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 465–478. doi:10.1145/1542476.1542528

Google. 2025. *V8 JavaScript engine*. https://v8.dev

Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. 2011. Improving the Performance of Trace-Based Systems by False Loop Filtering. *SIGARCH Comput. Archit. News* 39, 1 (mar 2011), 405–418. doi:10.1145/1961295.1950412

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 415–435.

Daniel Hillerström, Sam Lindley, Bob Atkey, and KC Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *Formal Structures for Computation and Deduction (LIPIcs, Vol. 84)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Daniel Hillerström, Filip Koprivec, and Philipp Schuster (benchmarking chairs). 2023. Effect handlers benchmarks suite. (2023). https://github.com/effect-handlers/effect-handlers-bench

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *Journal of Functional Programming* 30 (2020), e5. doi:10.1017/S0956796820000040

Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo. 2024. *The Programming Language Lua*. https://www.lua.org

Natsu Kagami. 2023. *Implement delimited continuations primitives*. https://github.com/scala-native/scala-native/pull/3286

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the International Conference on Functional Programming* (Boston, Massachusetts, USA). ACM, New York, NY, USA, 145–158.

Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. 2021a. *Compiler and replication of results: "Efficient Compilation of Algebraic Effect Handlers"*. https://doi.org/10.5281/zenodo.5497862

Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. 2021b. Efficient Compilation of Algebraic Effect Handlers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 102 (oct 2021), 28 pages. doi:10.1145/3485479

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 285–299. doi:10.1145/3009837.3009880

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the Haskell Symposium* (Vancouver, BC, Canada). ACM, New York, NY, USA, 94–105. doi:10.1145/2887747.2804319

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited Dynamic Binding. In *Proceedings of the International Conference on Functional Programming* (Portland, Oregon, USA). ACM, New York, NY, USA, 26–37.

Daan Leijen. 2016. *Algebraic Effects for Functional Programming*. Technical Report. MSR-TR-2016-29. Microsoft Research technical report.

Daan Leijen. 2017a. Structured Asynchrony with Algebraic Effects. In *Proceedings of the Workshop on Type-Driven Development* (Oxford, UK). ACM, New York, NY, USA, 16–29.

Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 486–499. doi:10.1145/3093333.3009872

Daan Leijen and Anton Lorenzen. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proc. ACM Program. Lang.* 7, POPL, Article 40 (Jan. 2023), 30 pages. doi:10.1145/3571233

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the Symposium on Principles of Programming Languages* (Paris, France). ACM, New York, NY, USA, 500–514. doi:10.1145/3009837.3009897

Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. Lexical Effect Handlers, Directly. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 330 (Oct. 2024), 29 pages. doi:10.1145/3689770

Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS'16)*. ACM, 120–131. doi:10.1145/2989225.2989232

Matthew Fluet. [n. d.]. *MLton*. https://mlton.org [Last access: 21-10-2023].

Conor McBride. 2016. *Shonky*. https://github.com/pigworker/shonky

Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2023. From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 255 (oct 2023), 30 pages. doi:10.1145/3622831

Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2023. *From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers*. Extended Technical Report. University of Tübingen, Germany. https://se.informatik.uni-tuebingen.de/publications/mueller23lift.

Minh Nguyen, Roly Perera, Meng Wang, and Steven Ramsay. 2023. Effects and Effect Handlers for Programmable Inference. *arXiv preprint arXiv:2303.01328* (2023).

Martin Odersky. 2023. *Strawman: Suspensions for algebraic effects*. https://github.com/scala/scala3/pull/16739

OpenJS Foundation. 2025. *Node.js — Run JavaScript Everywhere*. https://nodejs.org/en

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*. USENIX Association, Monterey, CA.

Michael Pall. 2025. *The LuaJIT Project*. https://luajit.org

David Peter. 2024. *hyperfine. A command-line benchmarking tool*. https://github.com/sharkdp/hyperfine [Last access: 29-07-2025].

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. 7, OOPSLA2, Article 238 (oct 2023), 26 pages. doi:10.1145/3622814

Atze van der Ploeg and Oleg Kiselyov. 2014. Reflection Without Remorse: Revealing a Hidden Sequence to Speed Up Monadic Reflection. In *Proceedings of the Haskell Symposium* (Gothenburg, Sweden) *(Haskell '14)*. ACM, New York, NY, USA, 133–144.

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *European Symposium on Programming*. Springer-Verlag, 80–94. doi:10.1007/978-3-642-00590-9_7

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). doi:10.2168/LMCS-9(4:23)2013

Matija Pretnar, Amr Hany Shehata Saleh, Axel Faes, and Tom Schrijvers. 2017. *Efficient compilation of algebraic effects and handlers*. Technical Report. Department of Computer Science, KU Leuven; Leuven, Belgium.

Python Software Foundation. 2025. . https://www.python.org/

Armin Rigo, Christian Tismer, and Jason Madden. 2011. *greenlet: Lightweight concurrent programming*. https://greenlet.readthedocs.io/en/latest/ [Last access: 21-08-2025].

Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. 2018. Explicit Effect Subtyping. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, Switzerland, 327–354. doi:10.1007/978-3-319-89884-1_12

Philipp Schuster and Jonathan Immanuel Brachthäuser. 2018. Typing, Representing, and Abstracting Control. In *Proceedings of the Workshop on Type-Driven Development* (St. Louis, Missouri, USA). ACM, New York, NY, USA, 14–24. doi:10.1145/3240719.3241788

Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A Typed Continuation-Passing Translation for Lexical Effect Handlers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 566–579. doi:10.1145/3519939.3523710

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proc. ACM Program. Lang.* 4, ICFP, Article 93 (Aug. 2020), 28 pages. doi:10.1145/3408975

Chung-chieh Shan. 2004. Shift to control. In *Proceedings of the 5th workshop on Scheme and Functional Programming*. 99–107.

Filip Sieczkowski, Mateusz Pyzik, and Dariusz Biernacki. 2023. A General Fine-Grained Reduction Theory for Effect Handlers. *Proc. ACM Program. Lang.* 7, ICFP, Article 206 (Aug. 2023), 30 pages. doi:10.1145/3607848

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. doi:10.1145/3453483.3454039

Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. 2009. Lazy continuations for Java virtual machines. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*. ACM, New York, NY, USA, 143–152.

The PyPy Project. 2025. *PyJitPl5*. https://rpython.readthedocs.io/en/latest/jit/pyjitpl5.html [Last access: 21-08-2025].

Unison Computing. 2025. *Introduction to Abilities: A Mental Model*. https://www.unison-lang.org/docs/fundamentals/abilities/

Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 1. doi:10.1145/1159876.1159877

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. doi:10.1145/2509578.2509581

Ningning Xie and Daan Leijen. 2021. Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C. *Proc. ACM Program. Lang.* 5, ICFP, Article 71 (aug 2021), 30 pages. doi:10.1145/3473576

Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. 2014. Accelerating Iterators in Optimizing AST Interpreters. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 727–743. doi:10.1145/2660193.2660223

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages. doi:10.1145/3290318

Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. 2020. Handling Bidirectional Control Flow. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 139 (Nov. 2020), 30 pages. doi:10.1145/3428207

A Appendix

A.1 Complete formal presentation of BC

Program Syntax:

Program	P	::=	$\overrightarrow{B_i}$	programs
Blocks	B	::=	$\{\ l(\overrightarrow{x_i}) :\ s\ \}$	basic blocks
Block references	$B^?$	::=	l	block labels
		|	B	concrete blocks
Statements	s	::=	$x \leftarrow v;\ s$	literals / constants
		|	$\overrightarrow{x_i} \leftarrow$ primitive $f(\overrightarrow{x_j});\ s$	primitive calls
		|	jump $B^?$	jumps
		|	if x_1 then $B^?(\overrightarrow{x_i})$ else s	conditional jumps
		|	push $B^?\ (\overrightarrow{x_i});\ s$	pushing frames
		|	return $(\overrightarrow{x_i})$	returning
metastacks				
		|	$x_o \leftarrow$ shift $x_p;\ s$	capturing stacks
		|	$x_o \leftarrow$ get dynamic $x_p;\ s$	accessing dynamic bindings
		|	$x_o \leftarrow$ new stack $B^?(\overrightarrow{x_i})$ @ $x_p;\ s$	creating stacks
		|	$x_o \leftarrow$ new stack $B^?(\overrightarrow{x_i})$ @ x_p with $x_b;\ s$	creating stacks with dynamic binding
		|	push stack $x;\ s$	pushing stacks
objects				
		|	$x \leftarrow$ new $V(\overrightarrow{x_j});\ s$	creating objects
		|	$x.m(\overrightarrow{x_i})$	invoking methods
data				
		|	$x \leftarrow t(\overrightarrow{x_i});\ s$	constructing data
		|	x match $\{\overrightarrow{t(\overrightarrow{x_i})\ \Rightarrow\ B^?}\}$	pattern matching
registers				
		|	$x_1 \leftarrow x_2;\ s$	copying
		|	$x_1\ \leftrightarrow\ x_2;\ s$	swapping
		|	drop $x;\ s$	dropping
VTables	V	::=	$\{\overrightarrow{m_i\ \mapsto B^?_i}\}$	virtual tables
Labels and tags	l, t, m			
Variables	x			
Values	v	::=	$(V,\ \overrightarrow{v})$	object
		|	M	continuations (see below)
		|	$(t,\ \overrightarrow{v})$	data value
		|	null	null value
		|	$\cdots$	*values of base types (not shown here)*

Fig. 9. Syntax of the bytecode format BC.

The main text only showed the parts of BC that are especially important for the translation of effect handlers. Here, we will present the full language, with the syntax shown in Figure 9.

A.1.1 Details for Delimited Control. In Section 3.1, we only hinted at how the instructions for delimited control are to be interpreted. We will now look into them in detail.

Creating a new stack. The instruction new stack creates a new stack segment marked with a *prompt* given by x_p and a single frame with return address l and locals $\overrightarrow{x_i}$. It constructs the resulting (meta)stack and assigns it to register x_0. Always installing a frame provides us with the invariant

Stack syntax:

$$
\begin{array}{llcll}
\text{Stacks} & k & ::= & \# & \text{empty stack} \\
 & & \mid & l(\overrightarrow{v_i}) :: k & \text{stack frame} \\
\text{Metastacks} & M & ::= & \circ & \text{empty metastack} \\
 & & \mid & k \mathbin{@} v \mapsto v \ ::: \ M & \text{stack with prompt } v
\end{array}
$$

Abstract machine syntax:

$$
\begin{array}{llcll}
\text{Abstract machine} & m & ::= & \langle\ P \mid X \mid s \mid M\ \rangle & \text{abstract machine tuple} \\
 & & \mid & \langle\ P \mid X \mid s \mid M \looparrowleft (v_p)\ M\ \rangle & \text{(unwinding)} \\
 & & \mid & \langle\ P \mid X \mid s \mid M \looparrowright M\ \rangle & \text{(rewinding)} \\
 & & \mid & \langle\ P \mid X \mid s \mid (v_p)\ ?\ M \mid M\rangle & \text{(lookup)} \\
\text{Register file} & X & ::= & \{\overrightarrow{x_i \mapsto v_i}\} &
\end{array}
$$

Fig. 10. Syntax of the abstract machine for BC.

that the inner stack level is never empty for metastacks passed around or being installed. We use the notation $X\{\ x \mapsto v\ \}$ to denote updating the register x to v in X.

$$
\begin{array}{ll}
\textit{(new stack)} & \langle\ X \mid x \leftarrow \text{new stack } l(\overrightarrow{x_j}) \mathbin{@} x_p;\ s\ \rangle \\
 & \quad \longrightarrow\ \langle X\{\ x \mapsto l(\overrightarrow{v_j}) :: \# \mathbin{@} v_p \mapsto \mathit{null}\ :::\ \circ\} \mid s\ \rangle \text{ if } v_j = X(x_j),\ v_p = X(x_p)
\end{array}
$$

Pushing a stack. Installing a delimiter amounts to creating a stack and pushing it, which we can achieve with the push stack instruction:

$$
\textit{(push stack)}\quad \langle\ X \mid \text{push stack } x;\ s \mid M\rangle \ \longrightarrow\ \langle\ X \mid s \mid M' \looparrowright M\rangle \quad \text{if } M' = X(x)
$$

To copy the stack segments to the current stack, push stack switches to a different abstract machine state, for which we have two evaluation rules that install the metastack segment by segment.

$$
\begin{array}{lrcl}
\textit{(rewind)} & \langle\ X \mid s \mid k' \mathbin{@} v_p \mapsto v_b\ :::\ M' \looparrowright M\rangle & \longrightarrow & \langle\ X \mid s \mid M' \looparrowright k' \mathbin{@} v_p \mapsto v_b\ :::\ M\ \rangle \\
\textit{(rewind 0)} & \langle\ X \mid s \mid \circ \looparrowright M\rangle & \longrightarrow & \langle\ X \mid s \mid M\ \rangle
\end{array}
$$

Capturing the continuation. The instruction shift captures the stack segments up to and including the prompt we pass it:

$$
\textit{(shift)}\quad \langle\ X \mid x_o \leftarrow \text{shift } x_p;\ s \mid M\rangle \ \longrightarrow\ \langle\ X \mid s \mid \circ \looparrowleft (v_p)\ M\rangle \quad \text{if } v_p = X(x_p)
$$

It is dual to push stack and also switches to a special abstract machine state. As long as we do not encounter the prompt we are searching for, we move the topmost stack segment to the captured continuation and continue, effectively reversing the order of stacks in the captured continuation.

$$
\begin{array}{ll}
\textit{(unwind} \neq\textit{)} & \langle\ X \mid s \mid M \looparrowleft (v_p)\ k \mathbin{@} v_q \mapsto v_b\ :::\ M'\ \rangle \\
 & \quad \longrightarrow\ \langle\ X \mid s \mid k \mathbin{@} v_q \mapsto v_b\ :::\ M \looparrowleft (v_p)\ M'\ \rangle \quad \text{if } v_p \neq v_q
\end{array}
$$

Once we find the correct prompt, we also move this last stack segment and store the final captured continuation in a register:

$$
\begin{array}{ll}
\textit{(unwind 0)} & \langle\ X \mid s \mid M' \looparrowleft (v_p)\ k' \mathbin{@} v_p \mapsto v_b\ :::\ M\rangle \\
 & \quad \longrightarrow\ \langle\ X\{\ x \mapsto k' \mathbin{@} v_p \mapsto v_b\ :::\ M'\ \} \mid s \mid M\rangle
\end{array}
$$

As seen above, to resume the continuation, we use the push stack instruction to rewind it back onto the current metastack.

A.1.2 Details for Dynamic Binding. Here, we will describe the details of how the dynamic binding described in Section 3.2 works.

Introducing a binding. To introduce a new binding, we include a separate form of new stack which additionally binds a value x_b at a given prompt:

$$\begin{array}{ll} \textit{(new stack with bind)} & \langle\ X \mid x \leftarrow \mathsf{new\ stack}\ l(\overrightarrow{x_j}) \mathbin{@} x_p\ \colorbox{lightgray}{$\mathsf{with}\ x_b$}\,;\ s\ \rangle \\ & \longrightarrow \langle X\{\ x \mapsto l(\overrightarrow{v_j}) :: \# \mathbin{@} v_p \mapsto \colorbox{lightgray}{v_b}\ :::\ \circ\} \mid s\ \rangle \\ & \quad \text{if } v_j = X(x_j),\ v_p = X(x_p),\ v_b = X(x_b) \end{array}$$

This instruction behaves just like *(new stack)* but stores the value of the binding in the second field of a metastack segment. To introduce the binding in the current scope, we need to also push the newly created stack. Note that we can use the same prompt as described in Subsection 3.1 to capture the delimited continuation up to this point. This is useful, as for effect handlers, we will often want to install both a dynamic binding for the handler and a prompt to delimit the continuation.

Accessing a binding. To access the dynamic binding associated with a given prompt, we can now use the get dynamic instruction, which looks very similar to shift:

$$\begin{array}{ll} \textit{(get dynamic)} & \langle\ X \mid x_o \leftarrow \mathsf{get\ dynamic}\ x_p;\ s \mid M\ \rangle \\ & \quad \longrightarrow \langle\ X \mid x_o \leftarrow \mathsf{get\ dynamic}\ x_p;\ s \mid \colorbox{lightgray}{$(v_p)\ ?$}\ M \mid \colorbox{lightgray}{M}\ \rangle \quad \text{if } v_p = X(x_p) \end{array}$$

It also uses two rules to implement searching the stack. Note that these are very similar to the *unwind* rules in Section 3.1, but instead of capturing the continuation, we leave the stack as-is and retrieve the value associated with the prompt. Together, these instructions allow us to dynamically bind values to prompts on the stack.

$$\begin{array}{lll} \textit{(lookup} \neq\textit{)} & \langle\ X \mid s \mid (v_p)\ ?\ k \mathbin{@} v_q \mapsto v_b\ :::\ M' \mid M\ \rangle & \\ & \quad \longrightarrow \langle\ X \mid s \mid (v_p)\ ?\ M' \mid \colorbox{lightgray}{M}\ \rangle & \text{if } v_p \neq v_q \\ \textit{(lookup}\ 0\textit{)} & \langle\ X \mid x_o \leftarrow \mathsf{get\ dynamic}\ x_p;\ s \mid (v_p)\ ?\ k' \mathbin{@} v_p \mapsto v_b\ :::\ M' \mid M \rangle & \\ & \longrightarrow \langle\ X\{\ x_o \mapsto \colorbox{lightgray}{v_b}\ \} \mid s \mid \colorbox{lightgray}{M}\ \rangle & \end{array}$$

A.1.3 Objects. We encode all kinds of function, closure, and object types using two constructs:

$$\begin{array}{llll} s & ::= & x \leftarrow \mathsf{new}\ V(\overrightarrow{x_j});\ s & \text{object creation} \\ & \mid & x.m(\overrightarrow{x_i}) & \text{method invocation} \end{array}$$

with the following evaluation rules:

$$\begin{array}{llll} \textit{(new)} & \langle\ X \mid x \leftarrow \mathsf{new}\ V(\overrightarrow{x_j});\ s\ \rangle & \longrightarrow\ \langle\ X\{x \mapsto (V, \overrightarrow{v_j})\} \mid s\ \rangle & \text{if } v_j = X(x_j) \\ \textit{(invoke)} & \langle\ X \mid x.m(\overrightarrow{x_i})\ \rangle & \longrightarrow\ \langle\ \{\overline{x_k \mapsto v_k}\} \mid s'\ \rangle & \\ & \multicolumn{3}{l}{\text{if } \overrightarrow{v_k} = \overline{X(x_i)},\ \overrightarrow{v_j} \text{ and } \{\ l(\overrightarrow{x_k}) :\ s'\ \} = V(m) \text{ and } X(x) = (V, \overrightarrow{v_j})} \end{array}$$

Note that there are no constructs for inheritance or similar concepts, and there are no classes. Also note that the formalization matches methods by index for simplicity; the actual implementation supports names, which simplifies separate compilation and debugging.

A.1.4 Stack management and normal control flow. As noted earlier, within the stack segments, we have a normal stack. We can use push to add a new stack frame here:

$$\begin{array}{ll} \textit{(push)} & \langle\ X \mid \mathsf{push}\ l(\overrightarrow{x_j});\ s \mid k \mathbin{@} v_1 \mapsto v_2\ :::\ M \rangle \\ & \quad \longrightarrow \langle\ X \mid s \mid l(\overrightarrow{v_j}) :: k \mathbin{@} v_1 \mapsto v_2\ :::\ M\ \rangle \quad \text{if } v_j = X(x_j) \end{array}$$

This will install a new frame on the stack with a label l, which will be jumped to on the next return. It also allows us to explicitly save some values, which will be passed to the code at l alongside the returned value(s).

To simulate a common calling convention for functions, we can follow this with a `jump`, which we could also use on its own for tail calls:

$$\textit{(jump)}\quad \langle\ X \mid \mathsf{jump}\ \{l(\overrightarrow{x_j}) :\ s\} \mid M\ \rangle \ \longrightarrow\ \langle\ X|_{\overrightarrow{x_j}} \mid s \mid M\ \rangle$$

Here, we simply update the currently evaluated statement. Formally, we also restrict the register file to the registers explicitly declared for this block. To pass arguments, we would have to put them into registers agreed on by convention.

Now, if we want to return to the frames we pushed earlier, we would use the return instruction:

$$\begin{aligned}&\textit{(ret)}\quad \langle\ X \mid \mathsf{return}(\overrightarrow{x_i}) \mid \{l(\overrightarrow{x_j}) :\ s\}(\overrightarrow{v_k}) :: k \mathbin{@} v_1 \mapsto v_2 ::: M\,\rangle\\ &\qquad\longrightarrow\ \langle\ \{\overrightarrow{x_j \mapsto v_j}\} \mid s \mid k \mathbin{@} v_1 \mapsto v_2 ::: M\ \rangle\\ &\quad\text{for } \overrightarrow{v_j} = \overrightarrow{X(x_i)},\ \overrightarrow{v_k}\end{aligned}$$

We can return multiple values, which are combined with the locals from the stack frame to get the new register values. If we reach the bottom of a stack segment, we remove the prompt and return to the next one:

$$\textit{(underflow)}\quad \langle\ X \mid \mathsf{return}(\overrightarrow{x_i}) \mid \# \mathbin{@} v_1 \mapsto v_2 ::: M\,\rangle \ \longrightarrow\ \langle\ X \mid \mathsf{return}(\overrightarrow{x_i}) \mid M\ \rangle$$

A.1.5 Other constructs. To write meaningful programs, the bytecode, of course, also supports many additional standard features, like conditional jumps, some instructions for register management, data values, and some instructions for register management. Also, labels are mapped to blocks using substitution. These are modelled straghtforwardly in the full syntax (Figure 9) and semantics (Figure 11) but we won't describe them in detail here.

A.1.6 Parts not modeled here. The calculus does not model some concerns, which are supported by the implementation. The first is backtrackable local mutable state, with a region element attached to each stack segment, which is out-of-scope for this paper. Another is dynamic code loading: We allow a running program to load new blocks from a file at runtime, which also can have static initializers. We leave those features out of the formalization to simplify the already sizeable calculus, and focus on the features needed to describe the main features of this work. Also, the implementation already partially support some variants of the presented instructions in preparation for future work (Section 8).

Reductions without program context:

$$\textit{(const)} \quad \langle\ X \mid x \leftarrow v;\ s \mid M\ \rangle \ \longrightarrow\ \langle\ X\{x \mapsto v\} \mid s \mid M\ \rangle$$

$$\textit{(prim)} \quad \langle\ X \mid \overrightarrow{x_i} \leftarrow \text{primitive}\, f(\overrightarrow{x_j});\ s \mid M\ \rangle \ \longrightarrow\ \langle\ X\{\overrightarrow{x_i \mapsto v_i}\} \mid s \mid M\,\rangle \quad \text{if}\ f(\overrightarrow{X(x_j)}) = \overrightarrow{v_i}$$

Register Management

$$\textit{(copy)} \quad \langle\ X \mid x_1 \leftarrow x_2;\ s \mid M\ \rangle \ \longrightarrow\ \langle\ X\{x_1 \mapsto v\} \mid s \mid M\,\rangle \qquad \text{if}\ v = X(x_2)$$

$$\textit{(swap)} \quad \langle\ X \mid x_1 \leftrightarrow x_2;\ s \mid M\ \rangle \ \longrightarrow\ \langle\ X\{x_1 \mapsto v_1,\ x_2 \mapsto v_2\} \mid s \mid M\ \rangle \quad \text{if}\ v_1 = X(x_1),\ v_2 = X(x_2)$$

$$\textit{(drop)} \quad \langle\ X \mid \text{drop}\ x;\ s \mid M\,\rangle \ \longrightarrow\ \langle\ X|_{\text{dom}(X)-\ x} \mid s \mid M\,\rangle \qquad \text{if}\ v = X(x_2)$$

Data

$$\textit{(cons)} \quad \langle\ X \mid x \leftarrow t(\overrightarrow{x_i});\ s \mid M\ \rangle \ \longrightarrow\ \langle\ X\{x \mapsto (t, \overrightarrow{v_i})\} \mid s \mid M\ \rangle \qquad \text{if}\ v_i = X(x_i)$$

$$\textit{(match)} \quad \langle\ X \mid x\ \text{match}\ \{\ \overrightarrow{t_c(\overrightarrow{x_{ic}}) \ \Rightarrow\ B_c^?}\ \} \mid M\ \rangle \ \longrightarrow\ \langle\ X\{\overrightarrow{x_{ic} \mapsto v_i}\} \mid \text{jump}\ B_c^? \mid M\ \rangle$$
$$\text{if}\ t_c = t\ \text{and}\ X(x) = (t, \overrightarrow{v_i})$$

Conditional Jumps

$$\textit{(if t)} \quad \langle\ X \mid \text{if}\ x_1\ \text{then}\ \{l(\overrightarrow{x_i'}) :\ s_1\}(\overrightarrow{x_i})\ \text{else}\ s_2 \mid M\ \rangle \ \longrightarrow\ \langle\ \{\overrightarrow{x_i' \mapsto v_i}\} \mid s_1 \mid M\ \rangle \quad \text{if}\ v_i = X(x_i)\ \text{and}\ X(x_1) = \top$$

$$\textit{(if f)} \quad \langle\ X \mid \text{if}\ x_1\ \text{then}\ \{l(\overrightarrow{x_i'}) :\ s_1\}(\overrightarrow{x_i})\ \text{else}\ s_2 \mid M\ \rangle \ \longrightarrow\ \langle\ X \mid s_2 \mid M\ \rangle \qquad \text{if}\ X(x_1) = \bot$$

We use $f|_X$ as the restriction of a function f to a domain X (remove all other register values).

Handling of program context:

$$\textit{(Pcong)} \quad \langle\ P \mid X \mid s \mid M\ \rangle \ \longrightarrow\ \langle\ P \mid X' \mid s' \mid M'\ \rangle$$
$$\text{if}\ \langle\ X \mid s \mid M\ \rangle \longrightarrow \langle\ X' \mid s' \mid M'\ \rangle$$

$$\textit{(Pcong} \looparrowright\textit{)} \quad \langle\ P \mid X \mid s \mid M_1 \looparrowright M_2\ \rangle \ \longrightarrow\ \langle\ P \mid X' \mid s' \mid M_1' \looparrowright M_2'\ \rangle$$
$$\text{if}\ \langle\ X \mid s \mid M_1 \looparrowright M_2\ \rangle \longrightarrow \langle\ X' \mid s' \mid M_1' \looparrowright M_2'\ \rangle$$

$$\textit{(Pcong} \looparrowleft\textit{)} \quad \langle\ P \mid X \mid s \mid M_1 \looparrowleft_v M_2\ \rangle \ \longrightarrow\ \langle\ P \mid X' \mid s' \mid M_1' \looparrowleft_v M_2'\ \rangle$$
$$\text{if}\ \langle\ X \mid s \mid M_1 \looparrowleft_v M_2\ \rangle \longrightarrow \langle\ X' \mid s' \mid M_1' \looparrowleft_v M_2'\ \rangle$$

$$\textit{(lookup B)} \quad \langle\ P \mid X \mid \Sigma[l] \mid M\ \rangle \ \longrightarrow\ \langle\ P \mid X \mid \Sigma[B] \mid M\ \rangle \qquad \text{if}\ B = \{\ l(\overrightarrow{x_i})\ :\ s\ \} \in P$$

where

$$\begin{array}{lcl}
\text{Label contexts}\ \Sigma & ::= & \text{jump}\ \Box \\
 & | & \text{if}\ x_1\ \text{then}\ \Box(\overrightarrow{x_i})\ \text{else}\ s \\
 & | & \text{push}\ \Box(\overrightarrow{x_i});\ s \\
 & | & x_o \leftarrow \text{new stack}\ \Box(\overrightarrow{x_i})\ @\ x_p;\ s \\
 & | & x_o \leftarrow \text{new stack}\ \Box(\overrightarrow{x_i})\ @\ x_p\ \text{with}\ x_b;\ s \\
 & | & x \leftarrow \text{new}\ \{\overrightarrow{m_i \mapsto B_i},\ m_j \mapsto \Box,\ \overrightarrow{m_k \mapsto B_k^?}\}(\overrightarrow{x_j});\ s
\end{array}$$

Fig. 11. Remaining abstract machine semantics for the JIT bytecode.

A.2 Benchmark descriptions

Figures 12 and 13 give concise descriptions of the control effect benchmarks from [Hillerström et al. 2023] resp. [Marr et al. 2016].

`countdown` *(input: 200M)*. Counts down in a loop using effect operations to decrement a state.

`fibonacci_recursive` *(input: 42)*. Standard recursive computation of Fibonacci numbers.

`generator` *(input/tree height: 25)*. Generates a complete binary tree (DAG representation), traverses it in-order, converts it to an explicit stream structure (with thunks), and sums the values.

`handler_sieve` *(input: 60k)*. Computes the sum of all primes up to the input using nested handlers for trial division (one per prime).

`iterator` *(input: 40M)*. Emits an ascending range of values and then takes their sum.

`nqueens` *(input: 12)*. Solves the N-Queens problem using brute-force search and a `Pick`/`Fail` effect.

`parsing_dollars` *(input: 20k)*. Reads an input via an effect, counts the dollars per line and emits it, outputs the sum until the first non-dollar, non-newline character.

`product_early` *(input: 100k)*. Computes the product of a linked list non-tail recursively, aborts early on 0.

`resume_nontail` *(input: 10k)*. Repeatedly performs an effect in a loop in non-tail position, and resumes in a non-tail position.

`tree_explore` *(input/tree height: 16)*. In a full binary tree, computes the maximal result of reducing a binary operation over all possible paths.

`triples` *(input: 300)*. Computes the hash sum of strictly decreasing triples that sum up to a target number.

Fig. 12. Short descriptions of the benchmarks, adapted from Hillerström et al. [2023].

`bounce` *(iterations: 10k)*. Simulation of a box with bouncing balls.

`list-tail`*(iterations: 10k)*. List creation and traversal.

`mandelbrot`*(input: 500)*. Classic Mandelbrot computation.

`nbody`*(input/simulation steps: 250k)*. Classic n-body simulation of solar system.

`permute`*(iterations: 10k)*. Generate permutations of an array.

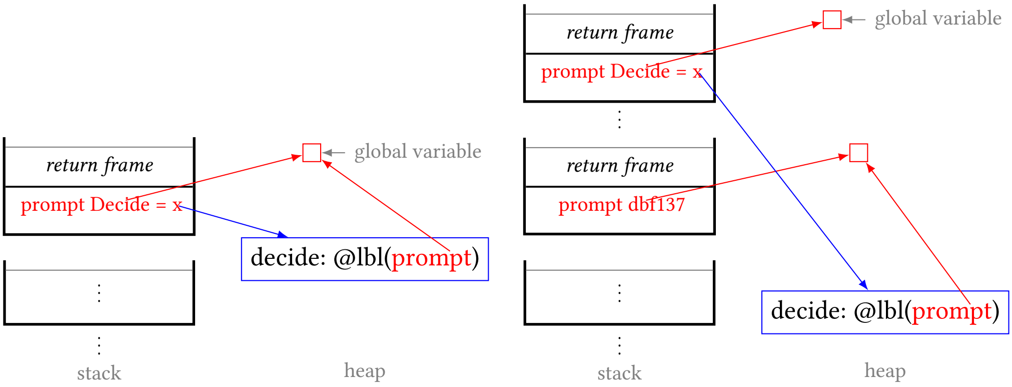
`queens`*(iterations: 10k)*. Solver for eight queens problem.

`sieve`*(iterations: 10k)*. Sieve of Eratosthenes.

`storage`*(iterations: 2k)*. Tree of arrays to stress GC.

`towers`*(iterations: 10k)*. Towers of Hanoi.

Fig. 13. Short descriptions of the benchmarks from Marr et al. [2016]



(a) Single handler. Prompts are merged.

(b) Handler for multiple effects.

Fig. 14. How handlers are modelled for Eff on the stack for a single installed handler. @lbl refers to the handler implementation (including shift). The prompt is in red, the handler implementation in blue.

A.3 Source Language translations in detail

We will now take a look at the source languages we support, describing how those languages' effect handlers map to the constructs in BC. Figure 5 already provided an overview over the different translations in BC-like pseudocode. The following subsections will describe the translations in more detail, using concrete bytecode.

A.3.1 Eff. In Eff, handlers can be defined separately from where they are installed. Consider the following Eff program snippet:

```
effect Decide : bool;;

let enumerate = handler
  | effect Decide k → k true + k false
  | x → x + 1

let foo () = if perform Decide then 0 else 1

let run () = with enumerate handle foo ()
```

To translate this, we first generate a global prompt for each handler, in the static initializer, respectively at the program entry point:

```
  x₁ ← "Decide";
  x₂ ← primitive freshlabel();
  _ ← primitive setGlobal(x₁, x₂)
```

Let's now look at the translation of the handler. Since the same handler could be used in multiple places, we translate a handler into the equivalent of a higher-order function. The translation is the following:

```
  enumerate(x_0):
    x_1 ← new { decide ↦ @206 }();
    x_2 ← "Decide";
    x_2 ← primitive getGlobal(x_2);
    x_1 ← new stack @215() @ x_2 with x_1;
    push stack x_1;
    x_0.apply()
```

Here, we construct a handler value as an object with methods for the operations,

then install the previously generated prompt with the handler as a dynamic binding — in Eff, handlers are bound dynamically. Finally, we invoke the parameter, which corresponds to the body of the `handle ... with ...`. This conceptually leaves us with a stack and heap as sketched in figure 14a.

The (slightly simplified) implementation of the handler is:

```
@206():
  x_1 ← "Decide";
  x_1 ← primitive getGlobal(x_1);
  x_0 ← shift x_1;
  jump @205  // actually a match on unit
@205(x_0):
  n_0 ← 1;
  push @204(x_0);
  jump @199
@199(n_0, x_0):
  push stack x_0;
  return n_0
```

Here, we first capture the appropriate part of the stack using shift, and then call the implementation of the operation. Then, we execute the handler operation, and finally push the captures stack part back on the (potentially changed) stack, and return into it, effectively resuming the continuation.

Installing the handler for a subprogram is now as simple as calling the above function, passing the body as a closure:

```
run(x_0): // actually, there is still a match on unit first
  x_0 ← new { apply ↦ 230 }();
  x_1 ← "Decide";
  x_1 ← primitive getGlobal(x_1);
  push @233();
  jump enumerate
```

In `foo`, we can now perform the `Decide` operation as follows:

```
@223():
  x_0 ← "Decide";
  x_0 ← primitive getGlobal(x_0);
  x_0 ← get dynamic x_0;
  x_1 ← unit();
  push @222();
  x_0.decide(x_1)
```

Here, we first get the prompt for the effect. Then, we retrieve the handler implementation using `get dynamic`, and finally call the operation on it. Note that `get dynamic` and `shift` both search

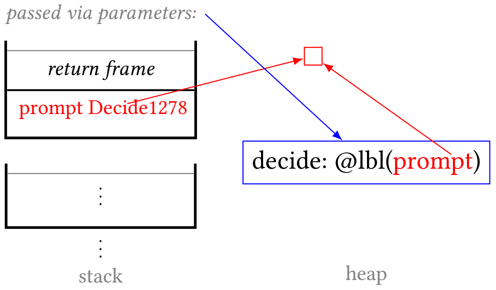


Fig. 15. How handlers are modelled for Effekt on the stack for a single installed handler. @lbl refers to the handler implementation (including shift). The prompt is in red, the handler implementation in blue.

for the same prompt on the stack, which could be optimized further at the cost of a more complex calculus, merging the two operations. Practically, this does not make a difference after a trace was optimized, as those instructions closely follow each other and the second search will be optimized out, safe for the very unlikely case that tracing starts at the invocation of the handler (after `get dynamic`).

For handlers that handle multiple operations, this scheme becomes slightly more involved. Now, we install multiple prompts: One for each of the handlers, for the dynamic binding, and an additional (fresh) one to shift to. The resulting structure of the stack is sketched in Figure 14b.

A.3.2 Effekt. In Effekt, effect handlers are bound lexically, while still capturing stack dynamically. To support this, we generate new prompt labels for each handler. This means that there are no static labels which we need to generate at program startup or dynamic code loading.

Consider the following Effekt program:

```
interface Decide {
  def decide(): Bool
}

def foo() = {
  if (do decide()) { 0 } else { 1 }
}

def main() = try {
  1 + foo()
} with Decide {
  def decide() = resume(true) + resume(false)
}
```

Installing and defining a handler are the same syntactic construct, so we do not take the indirection via a higher-order function. The handler definition is then translated to the following:

```
main():
  x₀ ⟵ primitive freshlabel();
  x₁ ⟵ new stack @3() @ x₀;
  push stack x₁;
  x₀ ⟵ new { decide ↦ @7 }(x₀);
  // inlined definition of foo
```

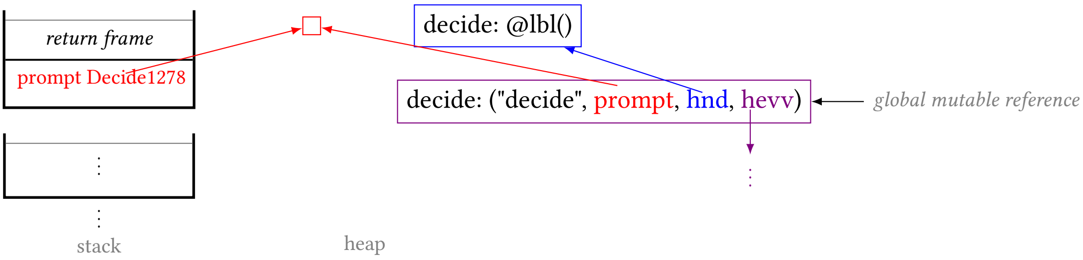


Fig. 16. How handlers are modelled for Koka on the stack for a single installed handler. @lbl refers to the handler implementation (excluding shift). The prompt is in red, the handler implementation in blue, and the evidence vector in purple. The evidence vector is a linked list, simplified here for aesthetics.

Here, we first create a fresh prompt, and directly create and install a new stack with it. Also, we instantiate a capability, which holds the handler implementation and closes over the prompt. Then, we directly continue with the handler body. The stack at this point is sketched in Figure 15. Note that there are no return clauses needed, so the return frame installed does not do any computation (but forwarding the result).

When later, in foo (which happens to have been inlined by the compiler here), we execute the effect operation, we simply call the correct method on the capability passed down to us:

```
// inlined definition of foo:
  push @14();
  jump @2
@2(x₀):
  x₀.decide()
@14(n₀):
  // continues after call to decide in foo
```

A.3.3 Koka. In Koka, effect handlers are implemented using a global mutable evidence vector, which stores both the handler implementations and labels of the currently in-scope handlers, sorted by name. [Xie and Leijen 2021] We implement those vectors as singly-linked lists in our backend, which sometimes causes more allocations than an array would, but simplifies the implementation.

Consider the following Koka program:

```
effect decide
  ctl decide(): bool

fun foo(): decide int
  if decide() then 0 else 1

fun main(): int
  with handler
    return(x)       1 + x
    ctl decide()
      resume(True) + resume(False)
  foo()
```

To install the handler for decide in main, we generate the following:

```
main():
  n_0 ⟵ 3; // statically computed index in the evidence vector
  x_0 ⟵ new { apply ↦ @82 }(); // handler implementation
  x_0 ⟵ "std/core/hnd/Clause_0"(x_0);
  x_0 ⟵ "@Hnd-decide"(n_0, x_0);
  x_1 ⟵ new { apply ↦ @86 }(); // return clause
  x_2 ⟵ new { apply ↦ @94 }(); // handler body
  x_3 ⟵ "minimal/@tag-decide";
  x_3 ⟵ primitive getGlobal(x_3);
  jump "std/core/hnd/@hhandle-vm"
```

Here, `"std/core/hnd/@hhandle-vm"` is an internal standard library function implemented as:

```
fun @hhandle-vm( tag:htag<h>, h : h<e,r>, ret: a ⟶ e r, action : () ⟶ e_1 a ) : e r
  val w_0 = evv-get()
  val m   = fresh-marker()
  val ev = Ev(tag,m,h,w_0)
  val w_1 = evv-insert(w_0,ev)
  evv-set(w_1)
  val res = @reset-vm(m,ret,cast-ev_0(action))
  evv-set(w_0)
  res
```

which is translated to bytecode in a straightforward way. Here, we first get the current evidence vector from a global variable, and generate a fresh prompt (using the BC-primitive `freshlabel`). We then generate a new evidence record with the given contents. Most notably, this contains the prompt and a pointer to the handler implementation passed to us. We then insert this new evidence into the current evidence vector, and update it. In the end, we restore the old evidence vector and return the result.

The call to `@reset-vm` handles the control-flow component of installing the handler, which is the following:

```
"std/core/hnd/@reset-vm"(x_0,x_1,x_2):
  x_0 ⟵ new stack @0141(x_1) @ x_0;
  push stack x_0;
  x_2.apply()
@0141(x_0, x_1):
  x_1.apply(x_0)
```

This is almost the usual definition of reset0 as described in Section [#sec-delimcc-std], but with a frame that calls the return clause. Also, because of the structure of the Koka library for handlers, we do not inline this code at the callsite and thus pass in the body as a function object.

Above, we just jumped over the handler implementation and `Clause`$_0$ constructor. What `@82` does is to wrap the actual handler implementation in a call to `@yield-to-vm`:

```
noinline fun @yield-to-vm( m : marker<e_1,r>, clause : (b ⟶ e_1 r) ⟶ e_1 r ) : e_1 b
  val w_0 = evv-get()
  val r = @yield-to-prim-vm(m, clause)
  evv-set(w_0)
  r
```

This stores (and later restores) the evidence vector around the handler, and then calls the actual handler implementation via `@yield-to-prim-vm`. which is translated as:

```
"@yield-to-prim-vm"(x₀,x₁):
  x₀ ← shift x₀;
  x₀ ← new { apply ⇒ @208 }(x₀);
  x₁.apply(x₀)
@208(x₀, x₁): // resume
  push stack x₁;
  return x₀
```

In foo, we then can perform the decide operation using the following code:

```
foo():
  n₀ ← 0;
  push @43(n₀);
  jump "getCurrentEvv" // evv-get()
@43(x₀, n₀):
  x₀ ← primitive promotePtr(x₀); // assume x₀ is static
  push @42();
  jump "elt"
@42(x₀):
  push @38(); // rest of foo
  [[x₀.hnd.implementation.get]].apply([[x₀.marker]], x₀) // yield-to missing
```

Here, @42 is actually doing a series of pattern matches, whose effect we summarized within [[]]. There are two standard library functions called here: - getCurrentEvv is a helper function which simply returns the current evidence vector (which is stored in a global mutable variable) - elt which returns the *n*th element of an evidence vector.

Once we have the correct entry of the evidence vector, all we have to do is to call the handler stored inside it.

Note that the actual generated code by our Koka implementation is more complicated than presented here, because function calls are more involved for separate compilation (they include a symbol lookup). Also, we simplified the control flow by inlining jumps and push/return pairs to improve readability.

```
effect Exc : unit

let inner () = perform Exc

let outer () = handle
    inner ()
  with
    | effect Exc k → ()
    | x → x

let rec loop n =
  if n ≥ 0
  then (outer (); loop (n - 1))
  else ()
```

(a) Eff code.

```
loop1(i53):
  guard_not_invalidated ⇒ loop [i53]
  < repetition bookkeeping >
  i55 = int_eq(i53, 0)
  guard_false(i55) ⇒ then [i53]
  i56 = int_lt(i53, 0)
  guard_false(i56) ⇒ else [i53]
  < call outer, install handler >
  < invoking body, then handler >
  < capture the continuation >
  < returning, repetition bookkeeping >
  i58 = int_sub(i53, 1)
  jump loop1(i58)
```

(b) Optimized loop.

Fig. 17. Example for zero-shot continuation capture.

A.4 Detailed Traces for Discussion

As further detail for the discussion in Section 6.2, here, we show the traces for minimal examples of the respective usage of continuations, first in Eff, and then summarized in all three languages.

A.4.1 Case 1: zero-shot. To look into this case more detail, consider the Eff example code in Figure 17a. When we now repeatedly execute `outer` in a loop, it gets traced by the JIT. The resulting optimized loop for Eff looks like shown in Figure 17b, with source code positions selectively annotated in < > using natural language. The trace for Effekt looks the same, and the one for Koka merely contains an additional guard on the evidence vector. Except for this guard in Koka, there are no operations generated for capturing the continuation, since it will never escape the loop and the escape analysis detects this fact. Note that this does not include guards generated to check that the current stack has the appropriate shape. The guard for Koka is necessary since evidence is stored in a global mutable reference and could thus, in principle, have been modified. To summarize, in all three backends, capturing the continuation has been completely optimized away for this example.

A.4.2 Case 2: one-shot and tail. For the one-shot and tail case, let's again look into the trace for a simplified example in slightly more detail. Ideally we can elide capturing the continuation at all, since it is effectively left in place. Let's look at the example in Eff in Figure 18.

We moved the handler definition out of the loop compared to the previous case, so we can only observe the invocation in the loop traces. Again, on the right, we provide a summary of the optimized trace. The only part of the continuation capture left is an addition and a guard. Those are introduced by first checking labels by definition site as described in Section 4.1.2. Note that checking if we are still under the same handler is necessary for correctness, since we might execute the same loop under a different handler later. Thus, modulo a single addition instruction, those loops are as good for this case as we could hope to get. There is no code generated to copy the continuation. Furthermore, in our implementation we did not need to special-case this scenario, for none of the source languages. The necessary checks have automatically been inserted by RPython and the optimization is sound by construction. As in the previous case, the traces for the other languages look the same, except that even the guard for Koka can be hoisted out of the loop.

```
effect Fun : unit

let inner () = perform Fun

let rec loop n =
  if n ≥ 0
    then (inner ();
          loop (n - 1))
    else ()

let outer n = handle
    loop n
  with
    | effect Fun k ⟶ k ()
    | x ⟶ x
```

(a) Eff code

```
loop1(i65):
  guard_not_invalidated ⇒ loop [i65]
  < check loop conditioon >
  i67 = int_eq(i65, 0)
  guard_false(i67) ⇒ then [i65]
  i68 = int_lt(i65, 0)
  guard_false(i68) ⇒ else [i65]
  < call inner >
  < invoke handler >
  < capture continuation >
  < call continuation >
  < update loop variable >
  i70 = int_sub(i65, 1)
  jump loop1(i70)
```

(b) Optimized loop

Fig. 18. Example for one-shot continuation capture with tail resumption.

A.4.3 Examples in other languages. For brevity, we only showed the examples in Section 6 for Eff. In this appendix, we show the variants for the other two languages.

Continuation Capture (zero-shot). The example in all three languages (Eff on the left, Effekt in the middle, and Koka on the right) here is:

```
effect Exc : unit

let inner () = perform Exc

let outer () = handle
    inner ()
  with
    | effect Exc k ⟶ ()
    | x ⟶ x

let rec loop n =
  if n ≥ 0
  then (outer ();
        loop (n - 1))
```

```
interface Exc {
  def exc(): Unit
}

def inner() = {
  forceNonpure()
  do exc()
}

def outer() = try {
  inner()
} with Exc {
  def exc() = ()
}

def run(n: Int): Int = {
  if (n > 0) {
    outer()
    run(n - 1)
  } else { 0 }
}
```

```
effect exc
  ctl exc(): unit

noinline fun inner(): exc unit
  exc()

fun outer()
  with handler
    return(x)    x
    ctl exc()
      ()
  inner()

fun run(n: int)
  if n ≥ 0
    then
      outer()
      run(n - 1)
    else 0
```

which generate the following optimized traces (again in the same order for the languages):

```
loop1(i53):
  guard_not_invalidated
      ⇒ loop [i53]
  < repetition bookkeeping >
  i55 = int_eq(i53, 0)
  guard_false(i55)
      ⇒ then [i53]
  i56 = int_lt(i53, 0)
  guard_false(i56)
      ⇒ else [i53]
  < call outer >
  < install handler >
  < invoking body >
  < invoke handler >
  < capture continuation >
  < returning >
  < repetition bookkeeping >
  i58 = int_sub(i53, 1)
  jump loop1(i58)
```

```
loop1(i31):
  guard_not_invalidated
      ⇒ loop [i31]
  < repetition bookkeeping >



  i33 = int_gt(i31, 0)
  guard_true(i33)
      ⇒ else [i31]
  < call outer >
  < installing the handler >
  < call inner >
  < invoke exc capability >
  < capture the continuation >
  < returning >
  < repetition bookkeeping >
  i35 = int_sub(i31, 1)
  jump loop1(i35)
```

```
loop1(i49, p52):
  guard_not_invalidated
      ⇒ loop [i49]
  < called outer >
  < install handler, modify evv >
  p54 = getfield_gc(p52, tag)
  guard_value(p54, ConstPtr(ptr86))
      ⇒ with handler [i49]
  < set evv, install prompt >
  < invoke body, get evidence >
  guard_nonnull_class(p52, Data__0)
      ⇒ with handler [i49]
  < invoke handler >
  < capture the continuation >
  < restore evidence >
  < repetition bookkeeping >
  i57 = int_sub(i49, 1)
  i59 = int_ge(i57, 0)
  guard_true(i59) ⇒ else [i49]
  jump loop1(i57, p52)
```

Continuation Capture (one-shot and tail). The example in all three languages (Eff on the left, Effekt in the middle, and Koka on the right) here is:

```
effect Fun : unit

let inner () = perform Fun

let rec loop n =
  if n ≥ 0
    then (inner ();
          loop (n - 1))
    else ()

let outer n = handle
    loop n
  with
    | effect Fun k ⟶ k ()
    | x ⟶ x
```

```
interface Fun {
  def fn(): Unit
}

def inner() = {
  forceNonpure()
  do fn()
}

def loop(n: Int): Unit / Fun
= {
  if (n > 0) {
    inner()
    loop(n - 1)
  } else { () }
}

def outer(n: Int) = try {
  loop(n)
} with Fun {
  def fn() = resume(())
}
```

```
effect func
  ctl func(): unit

fun inner(): func unit
  func()

fun loop(n: int): <div,func> unit
  if n > 0
    then
      inner()
      loop(n - 1)
    else ()

fun outer(n: int)
  with handler
    return(x)    x
    ctl func()
      resume(())
  loop(n)
```

which generate the following optimized traces (again in the same order for the languages):

```
loop_1(i_65):
  guard_not_invalidated
      ⇒ loop [i_65]
  < check loop conditioon >
  i_67 = int_eq(i_65, 0)
  guard_false(i_67)
      ⇒ then [i_65]
  i_68 = int_lt(i_65, 0)
  guard_false(i_68)
      ⇒ else [i_65]
  < call inner >
  < invoke handler >
  < capture continuation >
  < call continuation >
  < update loop variable >
  i_70 = int_sub(i_65, 1)
  jump loop_1(i_70)
```

```
loop_1(i_48):
  guard_not_invalidated
      ⇒ loop [i_48]
  < check loop condition >



  i_50 = int_gt(i_48, 0)
  guard_true(i_50)
      ⇒ else [i_48]
  < call inner >
  < invoke handler >
  < capture continuation >
  < call continuation >
  < update loop variable >
  i_66 = int_sub(i_48, 1)
  jump loop_1(i66s)
```

```
loop_1(i_69):
  guard_not_invalidated
      ⇒ loop [i_69]
  < invoke handler >
  < capture continuation >
  < call continuation >
  < restore evidence >
  < update loop variable >
  i_75 = int_sub(i_69, 1)
  < check loop condition >
  i_55 = int_gt(i_75, 0)
  guard_true(i_77)
      ⇒ else [i_69]
  jump loop_1(i_75)
```

A.5 Optimizing the dynamic dispatch

The aspect that we did not discuss in the previous subsections is getting rid of the dynamic dispatch. When the JIT traces the call to an effect operation, it traces directly into the implementation used in this instance, effectively inlining it. By inserting a guard, it ensures the correctness of the generated code. While the lookup itself varies, in all three source languages the handler is called by invoking a particular method on a handler object. For a method invocation, our interpreter implementation executes the following RPython code:

```
recv = env.get_codata(op.receiver)
vtable = promote(recv.vtable)
target = vtable.get_target(op.tag, primitives)
js = [len(a) for a in op.args.regs]
env.setfrom_codata(recv, js, program)

return jump_to(target, ...)
```

In our interpreter, we marked the registers of our interpreter env as virtualized. As a result, recv will already be in a SSA register if possible, and thus likely end up in a machine register. We then check the vtable and promote it, which inserts a guard checking for equality against the value we observed during trace generation. All entries in the vtable are immutable, and thus also determined by the vtable, so the lookup for target is no longer needed, because target is a known constant. We then restore the environment from the closure value recv. Finally, we use jump_to to execute a (known) jump. In the trace, this will not generate any code. Instead, we directly continue with the implementation of the concrete handler.

If we have multiple handler implementations used in one code location, *i.e.*, in the same loop and same instruction, this can still be optimized. When the guard on vtable fails too often, we trace more code paths from this guard, effectively building an inline cache of vtables. This, however, will duplicate the rest of the loop resp. bridge, which is a well-known problem for tracing JITs [Gal et al. 2009]. Note that the number of different handler implementations is bounded statically by the number of handler definitions in the source program.

A.6 All benchmarking results for the ablation study

Figure 19 shows all timings for the ablation study.

A.7 Benchmarking results on M1

The original submission contained the benchmarking results measured on an Apple M1 running macOS 14.1 using a Darwin 23.1.0 kernel. To allow for direct comparisons with the original submission for the changes introduced since, and since they are easily available to us, we include measurements on this machine here in Figures 20, 21a and 21b. They were run on a quiet system. The results are faster overall, and there are some differences, like the Koka C backend being much faster, which might also be due to using clang[4] instead of gcc[5] for compatibility reasons. In spite of this, the conclusions made in this paper are the same for those results.

Received 2025-03-26; accepted 2025-08-12

[4] https://clang.llvm.org

[5] https://gcc.gnu.org

Eff	JIT	no 4.1.1	no 4.1.2	no 4.1.3	no 4.1.4	more 4.1.4	none
countdown	0.695	**0.388**	0.699	0.610	0.552	0.549	0.661
counter	—	—	—	—	—	—	—
fibonacci-recursive	10.324	**8.869**	10.622	12.534	12.058	10.228	15.284
generator	0.504	0.495	0.485	0.625	**0.473**	0.616	0.569
handler-sieve	**5.301**	5.607	5.442	7.051	5.469	5.583	6.215
iterator	0.150	0.109	0.155	**0.108**	2.100	0.110	2.707
multiple-handlers	0.963	1.975	1.005	1.103	4.470	**0.557**	6.030
nqueens	0.900	0.870	**0.863**	1.065	1.446	1.471	1.650
parsing-dollars	0.532	**0.354**	0.540	0.646	6.525	0.540	7.838
product-early	0.818	0.815	0.820	1.193	**0.810**	0.838	1.159
resume-nontail	0.206	0.205	0.205	0.255	**0.197**	0.206	0.237
startup	0.005	**0.005**	0.005	0.005	0.005	0.005	0.005
to-outermost-handler	—	—	—	—	—	—	—
tree-explore	0.551	0.557	0.525	0.741	**0.500**	0.754	0.670
triples	0.315	0.309	**0.296**	0.349	0.337	0.346	0.351
unused-handlers	4.875	3.793	5.110	3.833	**3.636**	3.913	5.064
geomean slowdown	1.000	0.930	1.000	1.111	1.610	0.991	1.967

Effekt	JIT	no 4.1.1	no 4.1.2	no 4.1.3	no 4.1.4	more 4.1.4	none
countdown	**0.257**	0.258	0.261	0.257	0.258	0.262	0.289
counter	—	—	—	—	—	—	—
fibonacci-recursive	8.399	**7.774**	8.487	9.875	9.065	9.547	11.076
generator	**0.754**	0.830	0.763	1.060	0.772	0.918	1.016
handler-sieve	**4.696**	5.038	4.872	7.730	4.729	5.073	7.375
iterator	**0.022**	0.025	0.022	0.022	0.806	0.031	0.975
multiple-handlers	0.891	1.518	0.901	1.020	3.670	**0.477**	3.950
nqueens	1.465	1.523	1.521	1.841	1.355	**1.085**	1.753
parsing-dollars	0.520	0.521	**0.506**	0.524	6.028	0.543	5.960
product-early	0.516	0.520	0.517	0.684	0.510	**0.505**	0.679
resume-nontail	0.180	0.180	0.179	0.230	0.292	**0.178**	0.391
startup	0.005	0.005	0.005	0.005	0.005	**0.005**	0.005
to-outermost-handler	2.375	1.443	2.483	1.455	1.422	**1.414**	2.358
tree-explore	0.378	**0.356**	0.369	0.468	0.398	0.443	0.547
triples	0.163	0.156	**0.152**	0.188	0.209	0.208	0.239
unused-handlers	1.544	0.260	1.557	0.259	0.279	**0.259**	1.740
geomean slowdown	1.000	0.899	1.004	1.000	1.497	0.871	2.076

Koka	JIT	no 4.1.1	no 4.1.2	no 4.1.3	no 4.1.4	more 4.1.4	none
countdown	**0.351**	0.355	0.367	0.368	3.617	0.368	3.639
counter	0.417	0.430	0.415	**0.414**	2.079	0.442	2.104
fibonacci-recursive	4.688	5.006	**4.399**	5.671	6.140	6.528	4.616
generator	**0.756**	0.781	0.767	0.824	1.316	0.879	1.449
handler-sieve	**5.119**	5.207	5.267	8.099	5.362	5.494	8.573
iterator	0.275	0.261	0.275	0.274	0.263	**0.260**	0.272
multiple-handlers	1.122	**1.080**	1.112	1.466	1.153	1.125	1.433
nqueens	**0.934**	0.935	0.970	1.116	1.128	1.488	1.316
parsing-dollars	0.771	0.724	**0.686**	0.763	6.810	0.690	8.701
product-early	0.776	0.766	0.778	0.912	**0.751**	0.769	0.927
resume-nontail	**0.429**	0.435	0.430	0.478	0.737	0.440	0.958
startup	0.239	0.246	0.238	0.250	0.252	**0.235**	0.245
to-outermost-handler	0.777	0.766	0.769	0.807	4.014	**0.750**	4.083
tree-explore	0.662	**0.624**	0.650	0.766	0.719	0.942	0.944
triples	**0.755**	0.780	0.764	0.866	0.937	0.865	1.076
unused-handlers	0.353	**0.352**	0.352	0.367	3.596	0.361	3.518
geomean slowdown	1.000	0.999	0.994	1.125	2.114	1.094	2.366

Fig. 19. Runtimes of the benchmarks on jit with different optimizations disabled or changed, on x86_64. Time in seconds. Fastest in **bold**. ≡↯ marks stack overflows, and — unimplemented benchmarks.

	Eff			Effekt				Koka		
	JIT	Ocaml	Ocaml*	JIT	LLVM	JS	ML	JIT	C	JS
countdown	0.196	8.109	0.128	0.437	1.017	0.893	**0.066**	0.239	3.228	0.837
counter	—	—	—	—	—	—	—	**0.317**	1.970	0.545
fibonacci-recursive	6.195	40.663	1.434	5.307	1.650	17.889	2.104	3.259	**1.301**	3.513
generator	**0.348**	2.472	2.018	0.548	4.520	3.863	—	0.508	50.532	13.917
handler-sieve	2.376	25.293	8.326	2.067	**0.379**	2.291	✗	2.121	3.837	≡↯
iterator	0.060	2.972	1.015	**0.019**	0.228	0.245	0.191	0.188	0.343	0.402
multiple-handlers	1.208	10.205	—	1.019	0.972	4.359	**0.643**	0.997	12.271	7.047
nqueens	0.500	✗	0.294	0.832	0.892	1.177	**0.110**	0.562	8.130	2.089
parsing-dollars	0.175	16.664	1.173	0.419	2.757	0.568	**0.129**	0.605	3.536	6.115
product-early	0.403	4.991	0.973	0.253	**0.215**	0.655	0.383	0.440	10.246	2.736
resume-nontail	0.122	1.492	0.229	**0.109**	0.127	OOM	0.112	0.287	9.102	≡↯
startup	0.005	0.004	0.004	0.006	0.004	0.037	0.004	0.169	**0.004**	0.052
to-outermost-handler	—	—	—	0.787	0.995	0.895	✗	**0.587**	3.992	1.300
tree-explore	0.323	1.206	0.180	0.215	0.362	0.893	**0.177**	0.393	0.824	0.571
triples	0.163	0.581	0.150	0.103	0.227	0.857	**0.043**	0.445	12.562	2.642
unused-handlers	2.129	> 90.000	—	0.438	0.994	0.896	✗	**0.240**	3.226	0.864
geomean slowdown	1.000	10.088	1.617	1.000	1.541	3.099	0.624	1.000	5.617	3.226

Fig. 20. Runtimes of the benchmarks on the different backends, on M1. Time in seconds. Fastest in **bold**. For Eff, Ocaml is the `plain-ocaml` backend, Ocaml* is the version from the OOPSLA artifact. ≡↯ marks stack overflows, — unimplemented benchmarks and ✗ failing compilations.

(a) direct style	Effekt	JS	Lua		Python	
	JIT	V8	LuaJIT	Lua	CPython	PyPy
bounce	0.408	**0.213**	0.810	7.177	8.686	0.251
list-tail	0.342	**0.330**	0.706	5.505	5.274	1.551
mandelbrot	0.138	0.075	**0.045**	0.295	0.636	0.119
nbody	0.124	0.059	**0.036**	0.753	1.102	0.152
permute	1.347	**0.450**	0.513	12.211	13.256	1.293
queens	2.051	**0.400**	0.471	7.675	6.705	0.835
sieve	0.774	0.288	**0.170**	2.979	5.568	0.402
storage	0.462	**0.196**	0.860	3.105	3.022	1.114
towers	1.161	**0.662**	0.733	18.310	19.947	2.503
geomean slowdown	1.000	0.451	0.591	7.182	8.899	1.135

(b) control effects	Eff	Effekt	Koka	JS	OCaml 5	Python	
	JIT	JIT	JIT	V8	OCaml 5	CPython	PyPy
countdown	**0.196**	0.437	0.239	6.751	10.104	65.505	7.361
fibonacci-recursive	6.195	5.307	3.259	2.433	**1.433**	30.444	6.668
generator	**0.348**	0.548	0.508	23.769	1.429	23.859	8.029
handler-sieve	2.376	**2.067**	2.121	≡↯	8.101	≡↯	> 90.000
iterator	0.060	**0.019**	0.188	0.740	1.081	1.852	0.196
multiple-handlers	1.208	1.019	**0.997**	8.029	—	11.368	1.087
parsing-dollars	**0.175**	0.419	0.605	6.177	6.450	45.294	1.481
product-early	0.403	0.253	0.440	6.620	**0.106**	≡↯	4.395
resume-nontail	0.122	**0.109**	0.287	—	0.486	—	—
startup	0.005	0.006	0.169	0.035	**0.004**	0.016	0.063
geomean slowdown	1.002	1.000	1.896	11.042	3.448	27.493	6.238

Fig. 21. Runtimes of external benchmarks, on M1. Time in seconds. Lower is better, fastest in **bold**. ≡↯ marks stack overflows and — unimplemented benchmarks. Geometric mean slowdown is relative to Effekt JIT.

Eff	JIT	no 4.1.1	no 4.1.2	no 4.1.3	no 4.1.4	more 4.1.4	none
countdown	**0.196**	0.196	0.356	0.325	0.294	0.294	0.356
counter	—	—	—	—	—	—	—
fibonacci-recursive	**6.195**	6.196	7.303	9.256	7.735	7.528	9.597
generator	0.348	0.348	0.340	0.441	**0.332**	0.395	0.414
handler-sieve	2.376	2.378	**2.273**	3.051	2.313	2.379	2.836
iterator	**0.060**	0.060	0.089	0.064	1.069	0.064	1.244
multiple-handlers	1.208	1.208	0.809	0.903	2.521	**0.582**	3.234
nqueens	**0.500**	0.500	0.513	0.638	0.880	0.866	1.364
parsing-dollars	**0.175**	0.175	0.216	0.258	3.226	0.217	3.876
product-early	**0.403**	0.409	0.414	0.694	0.412	0.408	0.671
resume-nontail	**0.122**	0.122	0.128	0.149	0.125	0.132	0.146
startup	**0.005**	0.005	0.010	0.010	0.010	0.010	0.010
to-outermost-handler	—	—	—	—	—	—	—
tree-explore	0.323	0.323	0.323	0.464	**0.310**	0.463	0.428
triples	**0.163**	0.163	0.169	0.195	0.210	0.212	0.225
unused-handlers	2.129	**2.129**	2.953	2.375	2.134	2.267	2.925
geomean slowdown	1.000	1.001	1.147	1.307	1.831	1.167	2.302

Effekt	JIT	no 4.1.1	no 4.1.2	no 4.1.3	no 4.1.4	more 4.1.4	none
countdown	0.437	0.437	0.432	**0.432**	0.436	0.432	0.435
counter	—	—	—	—	—	—	—
fibonacci-recursive	5.307	**5.307**	6.154	8.107	6.384	6.769	7.841
generator	0.548	0.549	0.552	0.722	**0.543**	0.607	0.706
handler-sieve	2.067	2.064	**1.993**	3.674	2.002	2.102	3.573
iterator	**0.019**	0.019	0.024	0.024	0.541	0.024	0.627
multiple-handlers	1.019	1.019	0.744	0.842	2.254	**0.529**	2.703
nqueens	0.832	0.832	0.830	1.100	0.864	**0.616**	1.114
parsing-dollars	0.419	0.416	**0.092**	0.092	3.294	0.093	3.817
product-early	0.253	**0.253**	0.257	0.355	0.258	0.257	0.348
resume-nontail	**0.109**	0.109	0.115	0.133	0.187	0.116	0.232
startup	0.006	**0.006**	0.010	0.010	0.010	0.010	0.010
to-outermost-handler	0.787	**0.787**	1.436	0.791	0.791	0.791	1.398
tree-explore	**0.215**	0.215	0.233	0.311	0.264	0.276	0.346
triples	**0.103**	0.103	0.106	0.125	0.147	0.132	0.164
unused-handlers	0.438	0.439	0.728	0.433	0.437	**0.433**	0.758
geomean slowdown	1.000	1.000	1.021	1.118	1.708	0.948	2.199

Koka	JIT	no 4.1.1	no 4.1.2	no 4.1.3	no 4.1.4	more 4.1.4	none
countdown	0.239	**0.239**	0.243	0.243	2.041	0.244	2.264
counter	**0.317**	0.318	0.321	0.320	1.222	0.321	1.344
fibonacci-recursive	3.259	3.257	3.027	3.276	3.715	4.232	**2.848**
generator	0.508	0.503	**0.497**	0.544	0.860	0.559	1.003
handler-sieve	2.121	2.116	2.107	3.259	**2.085**	2.091	3.441
iterator	0.188	**0.188**	0.192	0.192	0.193	0.192	0.194
multiple-handlers	**0.997**	0.997	1.001	1.164	1.005	1.008	1.159
nqueens	**0.562**	0.563	0.584	0.659	0.730	0.894	0.826
parsing-dollars	0.605	**0.602**	0.605	0.608	4.237	0.605	5.868
product-early	0.440	**0.422**	0.424	0.597	0.432	0.425	0.547
resume-nontail	**0.287**	0.287	0.292	0.313	0.505	0.296	0.637
startup	**0.169**	0.169	0.173	0.173	0.174	0.173	0.175
to-outermost-handler	0.587	**0.587**	0.590	0.591	2.224	0.591	2.451
tree-explore	**0.393**	0.393	0.431	0.488	0.477	0.542	0.581
triples	0.445	**0.444**	0.455	0.525	0.595	0.517	0.704
unused-handlers	0.240	**0.240**	0.245	0.245	2.050	0.245	2.263
geomean slowdown	1.000	0.996	1.008	1.112	1.978	1.091	2.272

Fig. 22. Runtimes of the benchmarks on jit with different optimizations disabled or changed, on M1. Time in seconds. Fastest in **bold**. ≡↯ marks stack overflows, and — unimplemented benchmarks.