This chapter is concerned with comparing the performance of my CEK interpreter and my SHADE VM with existing solutions implementing algebraic effect handlers. The comparison is split into two parts: to evaluate the performance of the CEK interpreter I compare against the Eff interpreter and against the interpreter of Multicore OCaml. The SHADE virtual machine is compared to Multicore OCaml's byte code interpreter.

At the end of this chapter a proof of concept web server is described. This is more of a qualitative evaluation and is supposed to show what can be done with effect handlers in the real-world.

Evaluation strategy and technical details. I will run multiple benchmarks each of which uses effect handlers in a different way. Each benchmark is run 10 times and on the plots the arithmetic average of the execution times is reported (the standard deviation is displayed as error bars). Data was collected using GNU's `time` command. During the measurements my laptop was connected to the power supply and I tried to minimise noise in the measurements by not running any other process (apart from a terminal) and turning off unnecessary functionalities like WiFi, Ethernet, Bluetooth, etc.

All benchmarks are executed on a Lenovo Thinkpad X1 6th gen. with an Intel Core i7-8565U CPU, 16GB of RAM using a single thread. The operating system in use was a 64bit Ubuntu 18.04.4 LTS, Linux kernel version `5.3.0-46-generic`.

Eff was compiled from its official GitHub repository at the time where the top commit had SHA1 `796900d`. [1] The version of Multicore OCaml used in the benchmarks was obtained by OPAM, the identifier of the OPAM switch was `4.06.1+multicore`.

1 Exceptions

The exception benchmark raises and catches exception-like effects (the continuation is never resumed). I must add that in the OCaml benchmark I used effects simulating exceptions rather than *actual* OCaml exceptions. Actual OCaml exceptions are implemented very efficiently in OCaml (they are not resumable at all).

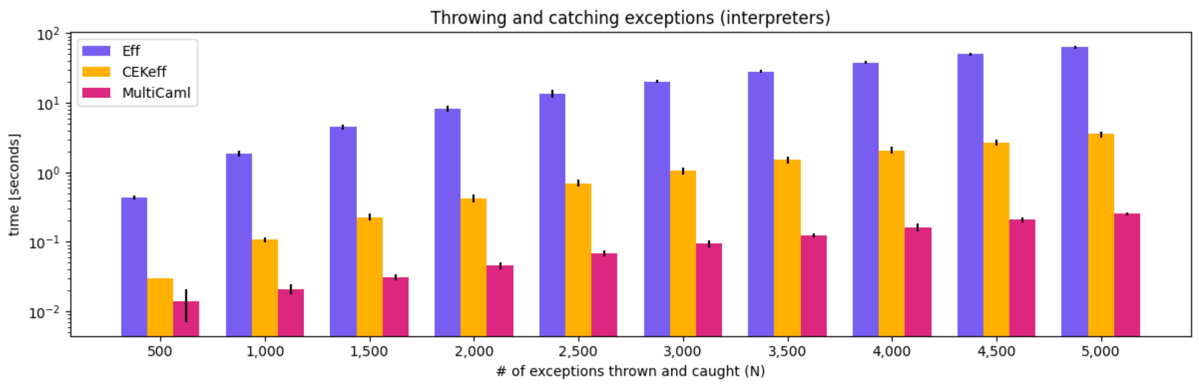


Figure 1: The CEK interpreter performs better than Eff but worse than Multicore OCaml

?? shows that the OCaml interpreter beats the CEK interpreter in this benchmark. However, we can see on **??** that the performance of SHADE VM is comparable to Multicore OCaml's VM even though the execution time for SHADE includes the compilation time to SHADEcode, but the times for Multicore OCaml do not include compilation. As I did not implement serialization for my byte code Eff programs were recompiled every time a SHADE benchmark was run. However, this extra time is less and less significant as we increase the number of exceptions.

[1]The official repository is `https://github.com/matijapretnar/eff/` at the time of writing.

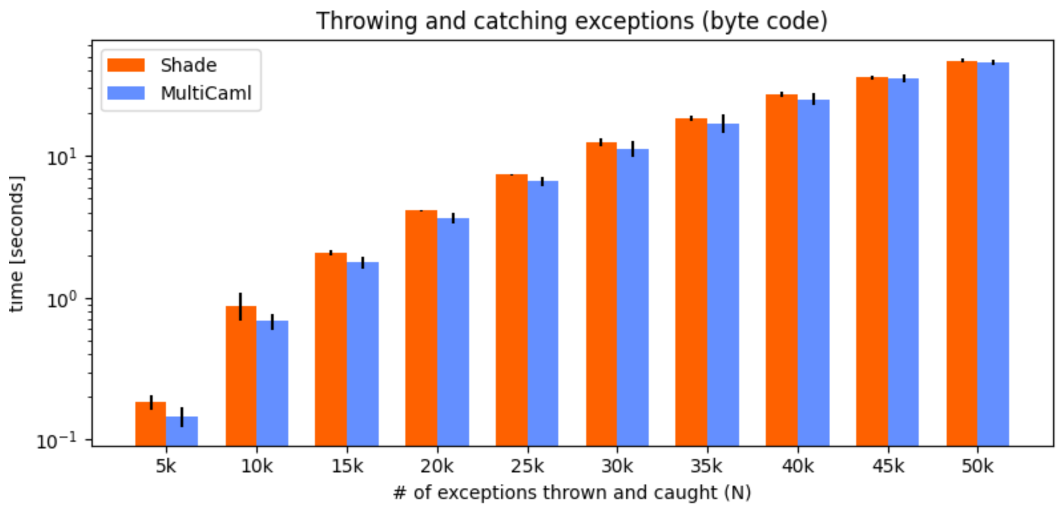


Figure 2: The peformance of SHADE is comparable to Multicore OCaml's

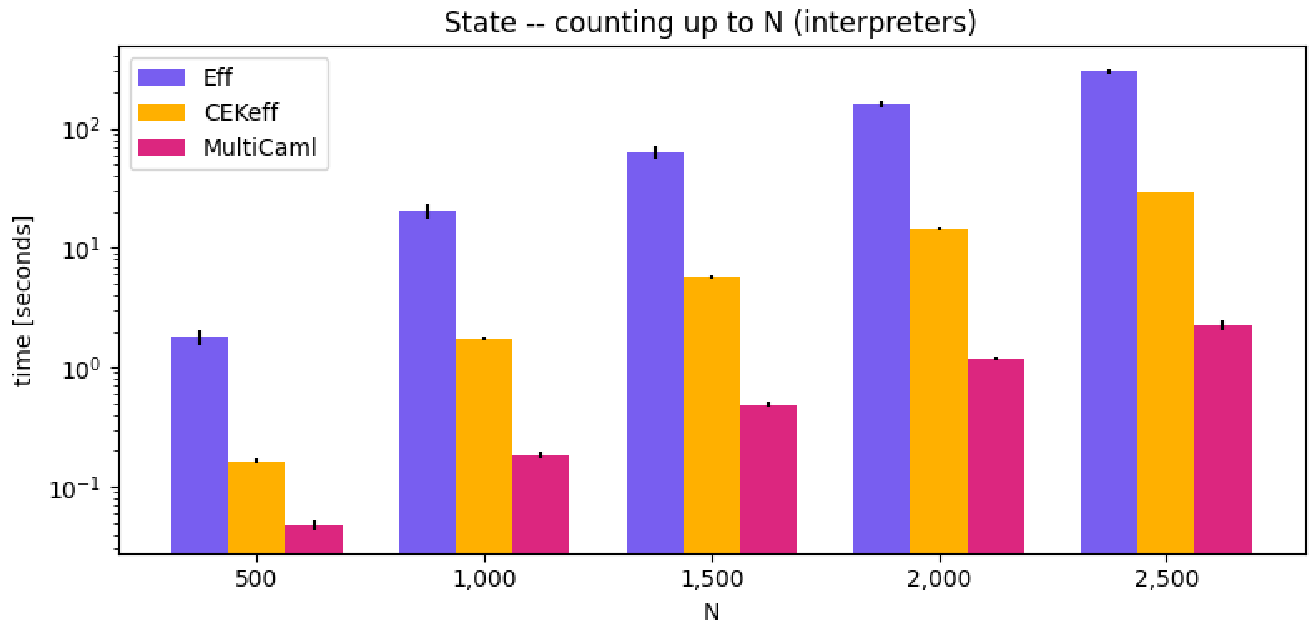
Note that as the byte code solutions are vastly more efficient than their interpreter counterparts they are compared with a larger number of exceptions. It takes around 63 seconds for the Eff interpreter to evaluate the benchmark with 5000 exceptions, whereas this takes only 3.5 seconds for the CEK interpreter. For SHADE VM the same takes around 200 milliseconds (with compilation included), which is a 300-fold improvement!

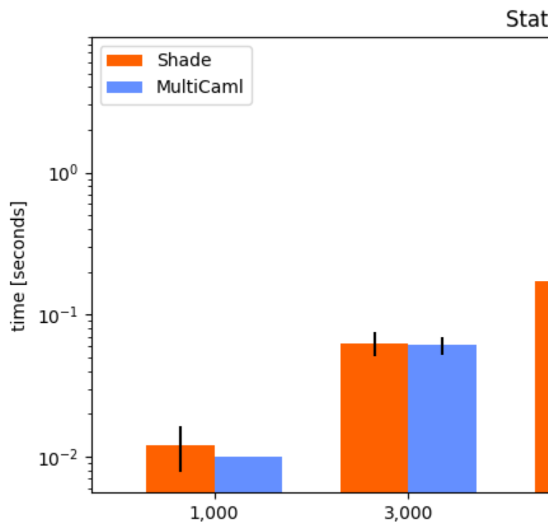
The y-axis is logarithmic and the plots show a linear slowdown when we increase the number of exceptions linearly. This suggests an $\mathcal{O}(N)$ time complexity for performing and catching N exceptions which suggests that my implementation does not contain bugs or inefficiencies that accumulate as more effects are performed in a program.

2 State and I/O

The state benchmark implements an integer state which gets incremented every time an `Incr` effect is performed. The handler of the `Incr` effect resumes a continuation exactly once. The reason why I chose such a computationally cheap operation as integer addition is that I wanted to measure the performance of performing effects and resuming continuations. As the number of performed effects N increases we can be confident that the cost of performing effects and resuming continuations will dominate the execution time and that we are measuring what we want.

.5 The CEK interpreter performs better than Eff but worse than the interpreter of Multicore OCaml [7cm][c]





.5 SHADE VM's performance is still head-to-head with Multicore OCaml [9cm][c]

Figure 3: State benchmarks

The results are displayed on **??** and **??**. The interpretation of these results is similar to the exception case although they reveal a *very important property* of the implementation.

In the exception case the size of the dump in SHADE VM remained constant. However, in this case the size of the dump grows linearly with N. As the dump is essentially a linked list of shadows residing in the heap it is reassuring to see that the performance does not degrade rapidly with N. In fact, we see an $\mathcal{O}(N)$ slowdown again, which is what one would expect from any efficient implementation (but we would not want a solution where the cost of performing an effect or resuming a continuation grows like $\mathcal{O}(N^2)$ for instance).

3 Non-determinism – Solving the N-queens problem

The N-queens [2] benchmark is a backtracking program which resumes continuations more than once. Here I do not compare against the CEK interpreter for the bigger benchmarks because due to its naïve implementation it slows down significantly after $N > 14$.

[2]The N-queens problem is concerned with finding the positions of N queens on an $N \times N$ chessboard such that no two queens attack each other.

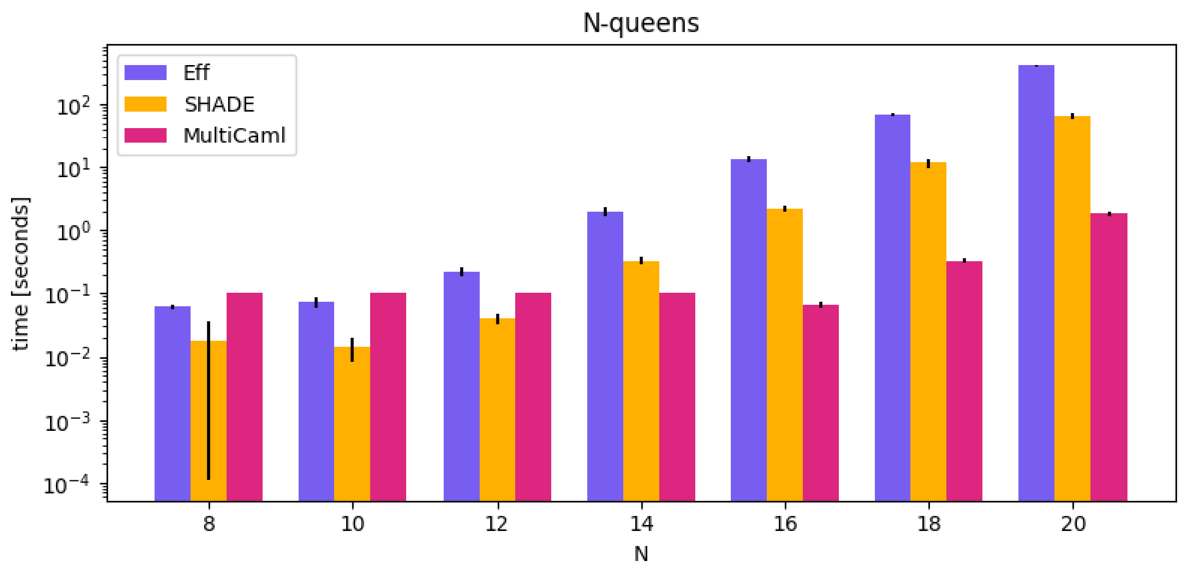


Figure 4: SHADE VM loses against Multicore OCaml here but is still more performant than the Eff interpreter

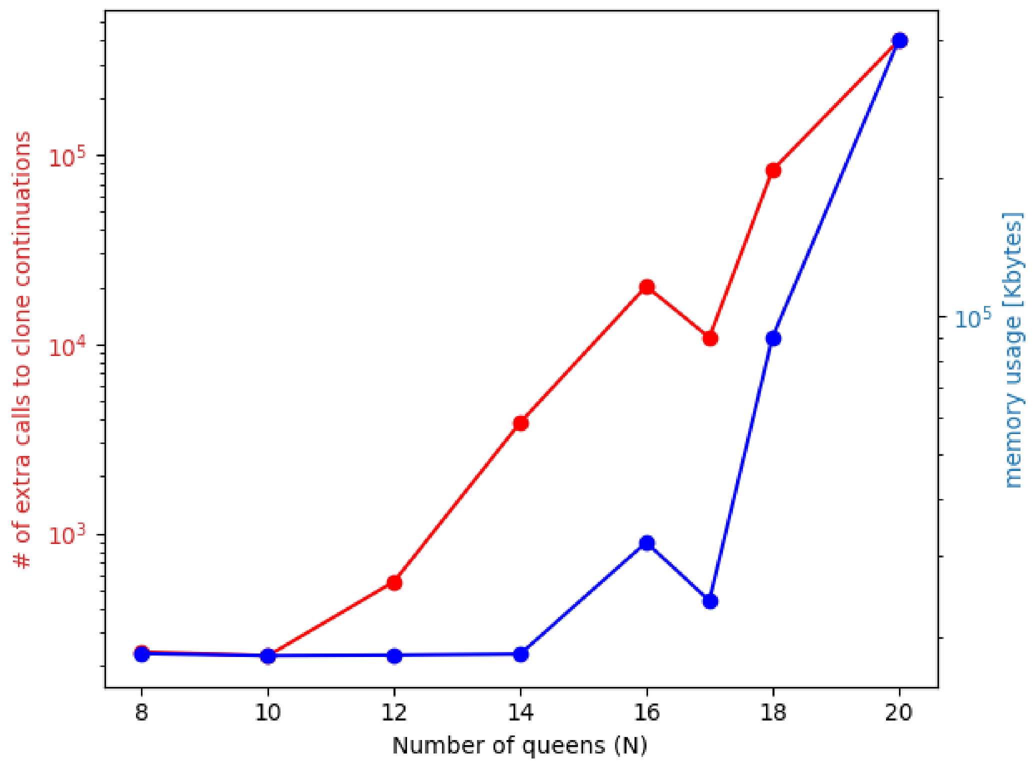


Figure 5: Investigating the effect of cloning on performance and memory usage

The performance of SHADE VM remains acceptable, although worse than the VM of Multicore OCaml as **??** shows.

The reason why Multicore OCaml performs so much better than SHADE VM was already mentioned briefly in the Implementation chapter. Multicore OCaml implements *one-shot continuations* and relies on the user to specify when a continuation should be cloned.

Listing 1: Cloning continuations must be done explicitly in Multicore OCaml

```
| effect (Decide ()) k ->
  try continue (Obj.clone_continuation k) false with
  | effect (Fail ()) _ -> continue k true
;;
```

?? reveals the extent to which this extra cloning affects SHADE VM's performance. I added a counter to my runtime system just for this benchmark to measure how many times continuations were cloned (the original benchmark did not have such a counter as I did not want performance monitoring to interfere with my results, i.e., the two metrics were measured in separate runs). We see that there is a good correlation between the number of extra clone calls and the memory used in the program. It is also possible to determine the average size of a continuation from the raw data. It turns out that the typical size of continuations is between 1 and 2 Kbytes (based on the bigger test cases, i.e., where N is 16, 18 or 20).

3.1 Vulnerability of the evaluation

A subtlety must be pointed out here. The N-queens benchmark traverses a huge search space of all N-queen configurations on an $N \times N$ chessboard. The order in which we inspect the configurations **does** matter here.

If one were to repeat this experiment one would have to ensure that their backtracking strategy is the same if they wish to compare their results to this evaluation. I used the same strategy in all my measurements and although this is a popular benchmark I do not compare my results with benchmarks from other papers as the papers often do not mention what strategy they use.

An even more subtle point is that N matters too. One must not fall in the pitfall of comparing the results for cases with different sizes (**??** reveals for instance that the $N = 10$ case is actually "easier" than the $N = 8$ case) as N does not represent the complexity of the problem, it is merely the identifier of a big search space. Interestingly, the $N = 8$ and $N = 10$ cases are nearly always reported in papers whereas the $N = 17$ is consistently left out as it is an obvious outlier.

4 Concurrency – Hello Online World!

Cooperative multitasking can be implemented with algebraic effects and their handlers too. We can realise *green threads*[3] as **unit** → **unit** functions which can be `Spawn`ed and which are capable of `Yield`ing control. The reader might correctly suspect that these operations will correspond to algebraic effects with the following types:

Listing 2: Effects for implementing a Hello World webserver

```
(* Concurrency for green threads *)
effect Yield : unit -->> unit;;
effect Spawn : (unit -> unit) -->> unit;;

(* Asynchronous effects in SHADE *)
effect HTTPHello : int -->> unit;;
effect Accept : unit -->> int;;
```

I built in two additional asynchronous effects in the runtime system of SHADE VM: `Accept` and `HTTPHello`. Both effects use non-blocking Unix system calls under the hood. `Accept` can accept an incoming network connection on port 8080 and returns the new port number that can be used for further communication with a client. The `HTTPHello` effect takes a port number and sends a "Hello World!" HTTP message using an already established connection.

Now we can implement two types of green threads: the function `start_server` implements the main thread of the web server: it accepts connections on the port 8080. If there is a new connection then a new green thread is spawned for that connection, otherwise control is given to other threads.

[3]Green threads are lightweight user space threads usually scheduled by runtime systems rather than the operating system.

Listing 3: Main green thread of the web server

```
let start_server () =
  let rec run () =
    let port = perform Accept () in
    match port with
    (* Yield if no incoming connection *)
    | -1 -> perform Yield ()

    (* Spawn thread for new connection *)
    | _ -> perform Spawn (say_hello port);
    run ()
  in
    run ()
;;
```

Listing 4: Green thread handling a connection

```
let say_hello port =
  let rec run () =
    let error = perform (HTTPHello port) in

    match error with
    | 0 -> () (* On success *)
    | -1 -> () (* If connection closed *)

    (* Try again if not ready *)
    | _ -> perform Yield (); run ()

  in run
;;
```

The `say_hello` function takes a port number and *returns a green thread*: a **unit** → **unit** function that will send the “Hello World” message on the port given but only if the underlying socket is ready. If this is not the case it yields control and will retry later.

As effect handlers can simulate state it is possible to implement thread scheduling with handlers too. The web server thus can be started with something like `with thread_scheduler handle start_server ()`. Based on the work of Dolan et al. **??** I claimed in the Introduction that effect handlers give flexibility to programmers to implement their own concurrency models which suits their application the most while they reduce the complexity of runtime systems. This is a justification of that claim (it is actually a simplified experiment from their paper). We see that `thread_scheduler` could implement any kind of scheduling policy by handling the `Yield` and `Spawn` effects appropriately. The fact that one can implement a toy webserver using one’s own runtime system written from scratch in less than a year justifies the second claim (the cited paper is a result of the collaboration of 6 people).

I load tested this web server with the `wrk2` load testing tool [**?**] using 4 threads and gradually increasing the number of connections until the p90 time fell below the 5s threshold (this happened at around 100 connections as the red numbers indicate this on **??**.

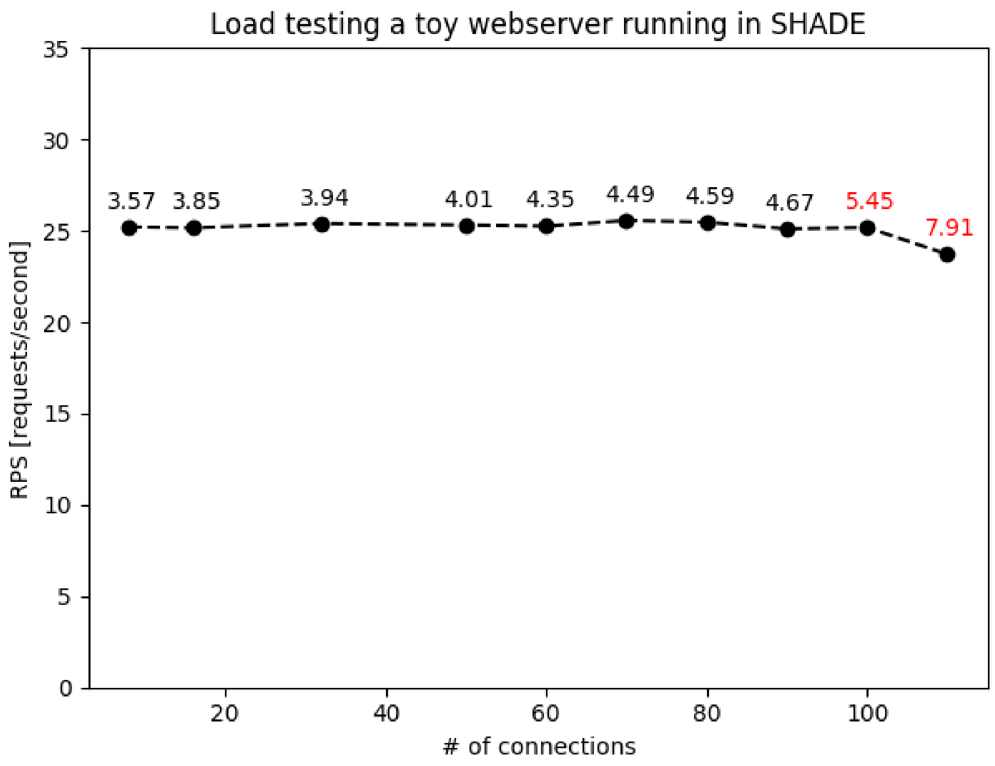


Figure 6: Although the performance of the webserver is not at all impressive, it is my own runtime system that implements it and it can still handle 100 connections with a p90 time of 5 seconds with a throughput of 25 requests per second.

5 Summary

The following sums up this chapter and reports its findings.

- The performance of the syntax-level CEK interpreter and the SHADE VM was compared to Eff and Multicore OCaml.
- My CEK interpreter was 20x quicker on the exception benchmarks and 10x quicker on the state benchmarks than the Eff interpreter. However, it was also 15x slower on both the exception and state benchmarks than the OCaml interpreter.
- My SHADE VM was consistently quicker than all the interpreters and performed well on the exception (SHADE is 2% slower) and state benchmarks (SHADE is 17% quicker) against the VM of Multicore OCaml. However, it remained around 35x slower than the VM of Multicore OCaml on the N-queens benchmark.
- The reason for this slowdown on the N-queens benchmark was determined and it was found that the cause is inherent in Eff. Eff supports multi-shot continuations, whereas Multicore OCaml only supports one-shot continuations and therefore the copy overhead in the SHADE VM is bigger.
- As a qualitative evaluation of SHADE a toy webserver was presented and load tested.