Improving the quality of code in IO intensive applications through effect handlers

Sam Streef
Supervisors: Casper Poulsen, Jaro Reinders and Cas van der Rest
EEMCS, Delft University of Technology, The Netherlands

June 19th, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Effect handler orient programming (EHOP) is a recently proposed programming paradigm that aims to provide a high-level abstraction in code. Using this paradigm, programmers are able to define operations as an effect, which are implemented by an effect handler. Functions can then use effects, allowing the effect operations to be used in this function. Depending on the effect handler that handles the effect, an effect can have different functionality. In this research, EHOP is compared to the traditional functional programming paradigm on readability, maintainability, modularity and performance for IO intensive applications. The comparison is carried out by two programming experiments that each apply one of the programming paradigms to create an HTTP server. The comparison of these programs show that EHOP improves the readability, maintainability and modularity but decreases the performance in response time and memory. The conclusion of this case study is therefore that EHOP affects IO intensive applications in a negative manner due to its performance overhead.

1 Introduction

Programming languages have been continuously evolving to manage complexity and maintainability of code for programmers [1]. Chatley et al. [1] assign this as both the cause and effect of increasing applications of programming to new problems over the past decades. Consequently, new programming paradigms should be explored to improve complexity and maintainability management and to furthermore solve new problems.

An example of managing complexity and maintainability in a language is by the high-level abstraction of memory management. Languages such as C allow a programmer to allocate and free memory, often resulting in a good performance. However, memory allocation, when not applied correctly, is known to introduce bugs and security vulnerabilities [1]. Therefore, many languages such as Java and C# have a build-in memory management system that removes the concern for programmers having to manage their memory allocation. Consequently, programmers do not have to concern themselves with the irrelevant details of memory allocation every time they write a program, making programming less prone for introducing bugs or vulnerabilities and code less complex in terms of readability.

A recently proposed programming paradigm that aims to provide a high-level abstraction is *effect handler oriented programming* (EHOP). Using EHOP, a programmer is able to define an operation with its own side-effects. This is similar to a side-effect in the *functional programming* paradigm, where a side-effect is an operation that often requires to mutate state, values or interact with the outside world.

The operations or effects that a programmer defines in EHOP are implemented orthogonal from application code by effect handlers. An effect handler is an implementation of an operation. The core concept of effect handlers is that they can be changed or swapped without having to change the application code that uses the related effects. According to Hillerström [2], effect handlers can be implemented and used together with other effect handlers. Combining these makes effect handlers particularly useful. It allows for the use of multiple effects at once, whilst maintaining orthogonality in code implementation.

To illustrate how effects and effect handlers could be used, the following example is provided. Assume that an application requires logging, then for this application a *Logging* effect can be defined. The effect, or operation, can be a function called *log* that takes a string as input and then outputs it. The only concern of the programmer in the application code is having this function available to its environment. The implementation details are hidden by an effect handler for this *log* operation. As such, the effect handler can be implemented in different ways without having to change the application code or operation signature. A concrete example in *Haskell* using the *Polysemy* [3] library can be found in Listing 1 and 2.

```haskell
-- Effect
data Logging m a where
    Log     :: String -> Logging m ()

-- Polysemy function generating effect operations
makeSem ''Logging

-- Effect handlers
type LoggingHandler = Sem '[Logging, Embed IO] ()
                   -> Sem '[Embed IO] ()

runConsoleLogging :: LoggingHandler
runConsoleLogging = interpret $ embed . \case
    Log s   -> putStr s

runFileLogging :: LoggingHandler
runFileLogging = interpret $ embed . \case
    Log s   -> appendFile "log.txt" s
```

Listing 1: Example of defining a *Logging* effect and its respective effect handlers. As can be seen in the type signature of the handlers, each handler consumes the *Logging* effect from the effect row by evaluating its operation.

Currently, there are a few languages that include built-in support for effect and effect handlers. Two of these being *Koka* [4] and *Frank* [5], which both have a native support for defining effects and effect-handlers. These languages, however, are limited in terms of functionality other than being able to make use of EHOP. Such functionalities include parallel processing and asynchronous IO operations. Furthermore, there is support for EHOP in other languages by the use of li-

```haskell
program :: Sem '[Logging, Embed IO] ()
program = do
    log "Hello "
    log "World!"

main :: IO ()
main = program           -- [Logging, Embed IO]
    & runConsoleLogging  -- [Embed IO]
    & runM               -- []

main' :: IO ()
main' = program          -- [Logging, Embed IO]
    & runFileLogging     -- [Embed IO]
    & runM               -- []
```

Listing 2: Example of using a *Logging* effect in application code. The effects are handled by effect handle functions defined in Listing 1.

braries. An example is *Haskell* [6], a popular functional programming language, that can apply EHOP by using a library named *Polysemy*.

The general question that this paper will try to answer is how the use of effect handler oriented programming affects the modularity, readability, maintainability and performance of code in input-output intensive applications when compared to traditional[1] implementations. Answering this question will reveal how usable and useful EHOP is in this particular application. Providing a positive result would mean that EHOP could help developers reduce complexity in code. Consequently, it could improve software development and possibly be embedded in more programming languages. Alternatively, a negative result could indicate the limitations of EHOP as a paradigm such as its applicability and usefulness. The research question can furthermore be divided into individual sub-questions, that can together answer the general question.

For a qualitative analysis of the general question, this paper will try to answer the following individual questions:

1. Does using EHOP improve the *readability* of code compared to the traditional approach of programming?
2. Does using EHOP improve the *modularity* of code compared to the traditional approach of programming?
3. Does using EHOP improve the *maintainability* of code compared to the traditional approach of programming?

For a quantitative analysis of the general question, this paper will try to answer the following individual questions:

1. Does using EHOP improve *response time* compared to a traditional implementation?
2. Does using EHOP increase the *memory usage* compared to a traditional implementation?

[1] Traditional in a way of not applying EHOP.

The contributions of this paper are:

- an example of how EHOP can be used in a HTTP server implementation.
- an exploration of applicability of effect handlers in IO intensive applications.
- a qualitative comparison between EHOP and the traditional functional programming paradigm.
- a quantitative comparison between EHOP and the traditional functional programming paradigm.

Finally, this paper follows a structure of the chronological order in which this research is carried out. Starting with Section 2, the methodology of this case study will be explained by defining the type of experiments, evaluation and comparison methods. Furthermore, the experimental work will be described in Section 3. In Section 4, the results of the experiments will be shown and compared. Following is Section 5, where the results, limitations, future work and reproducibility of this research will be discussed. Lastly, the conclusions will be discussed in Section 6.

2 Methodology

To answer the aforementioned questions, two programming experiments are performed, evaluated and compared.

2.1 Defining the experiments

The experiments are aimed to be carried out in an identical way except for the programming paradigm that is used. Both experiments involve programming a simple HTTP server library from scratch, i.e. implementing the HTTP protocol on top of a TCP socket and supplying an interface to setup request handlers that handle HTTP requests when running the HTTP server.

They are performed using *Haskell* [6] as their programming language. Although *Koka* [4] and *Frank* [5] provide built-in support for effects and effect-handlers, they are not considered to be usable in their current state for this case study. *Koka*, for instance, is limited in its IO operations since it cannot process a stream of data asynchronously. There are functionalities that allow for asynchronicity and multi-threading, however they are still in experimental phase. Moreover, *Frank* is a fairly recent language that is not anywhere near *Koka* in its functionality. Hence, it is not suitable for this case study as of now, but might be in the future. Consequently, *Haskell* [6] is the better choice to carry out this case study. Since *Haskell* does not have effects and effect-handlers built-in like *Koka* does, the experiment with EHOP makes use of the *Polysemy* [3] library. This library allows for effects and effect-handlers to be defined and used in *Haskell*.

2.2 Evaluation of the experiments

After the execution of the experiments, the resulting programs are qualitatively compared and individually evaluated

on performance. The comparison includes a qualitative analysis of readability, maintainability and modularity. These metrics can be complex to measure since they are often subjective and can be defined differently depending on the programmer. It should be noted that there are metrics that try to provide objective measures for these qualitative metrics. However, for this research they are considered to be too time consuming or not applicable for the EHOP paradigm. A description of these metrics and limitations can be found under Limitations in the Discussion.

Nevertheless, this paper will try to objectively identify the differences in these qualitative metrics between the experiments. It will do so by providing examples and arguments to why one approach is better over the other. Aiding these examples and arguments are the principles for API design described by Bloch [7]. As Bloch mentions, API design is very similar to programming applications. Hence, they can be applied to the qualitative analysis of the experiments. The most important principles applicable to this analysis that are:

- "API should do one thing and do it well" [7, p.13]
- "API should be as small as possible but no smaller" [7, p.14]
- "Implementation should not impact the API" [7, p.15]

Moreover, the performance of each experiment is measured by the response time of requests and the overall memory usage of the application. To measure the response time of the HTTP server, 100 POST requests are sent with an equal size in payload. The server will have an endpoint registered that reverses the data of the incoming request and sends it back. This procedure is done 20 times per benchmark, with 5 different benchmarks that differ in payload size. For each benchmark, the total time of performing 100 sequential requests is measured and divided by 100 to get the average response time per request. Furthermore, it should be noted that each server will make use of buffering, to limit the traffic of the TCP socket. It is set to 512 bytes to ensure that the response time of the requests is not too short to measure.

Finally, the overall memory usage of each application is measured. This is achieved by running the executables with Runtime System options[2] (RTS). For each executable they are run by using the "+RTS -h" option, which produces a .hp file that can be converted into a graph. The results of this analysis includes the total memory allocation in bytes over the time the program has run. This procedure is executed 5 times, whilst a 100 benchmarks that each send a 100 POST requests with 100 bytes, are run each time. These POST requests are, like the response time measurement, send to an endpoint that reverses the payload contents.

2.3 Comparing the experiments

Combining the previously mentioned metrics, the experiments are compared with the qualitative results of the individual evaluations. Additionally, the quantitative metrics are compared to see if there is an overhead in performance when using EHOP.

[2]https://ghc.gitlab.haskell.org/ghc/doc/users_guide/runtime_control.html

3 Applying EHOP in IO intensive applications

The goal of applying EHOP in IO intensive applications is to reduce complexity and allow for better readability, maintainability and modularity. However, in order to apply EHOP there needs to be an idea of where and how it can be applied in an HTTP server.

First, HTTP servers deal with communication that is carried out over the TCP network layer. TCP is a protocol that allows for reliable bidirectional communication by sending packets. Whenever a client sends a message over TCP, this message can be split up in packets and received sequentially. Therefore, in order to parse a complete HTTP request that might have been split up into separate packets, they need to be buffered. A useful effect to achieve this buffering is the $State$ effect. The $State$ effect can be used to store data about the state of parsing, the bytes that were received and a construction of the HTTP request. With this state, intermediate parsing can be done while still missing data and the data can be easily stored and fetched when needed.

Second, servers often require monitoring to see the status of the server or requests. This is often done by logging, either to file or to the console. Such behaviour can be expressed by a $Logging$ effect where console or file logging can be implemented by different effect handlers and swapped out by changing the handler.

Third, in order for a server to process requests, request handlers are needed. Usually these consist of functions that take a request and produce a response. The server then routes the right handler for a HTTP method and path combination. These handlers must be stored and accessed during run time. For these operations, a key-value store could be used, which is a special type of $State$ effect. In this store, handlers of the type $Request \rightarrow Response$ can be stored and used by the server.

Finally, an important functionality of HTTP servers is being able to serve static content. This content can be content such as HTML, CSS or JavaScript files. Since these files are stored on disk they need to be read through an IO operation in order to serve a client with its content. However, in a testing environment it might be desirable to not interact with the machine the tests are running on. Therefore, this reading of files can be handled as an effect and two different effect handlers can be defined for production and development. In production, the $FileReading$ effect can interact with the filesystem of the server, whilst in development the effect can return a constant string value.

Experimental work

The HTTP network layer can be build on the TCP network layer. Consequently, a TCP socket is used in both the exper-

iments in this research. Each experiment starts with the *Network* library that provides such a socket. The boilerplate code that each experiment starts with is identical and only needs a function with type *Socket* $\rightarrow$ *IO a*. Therefore, this function is the difference in implementation and is compared between both experiments.

For each experiment, the functionality of each HTTP server is the same and similar in implementation other than its paradigm. Such functionalities include:

- buffering of incoming request data that is sent by packets of X bytes
- parsing the incoming data as an HTTP request
- registering request handlers for GET and POST requests
- setting a file source to serve static content
- logging of incoming requests

Data types, parsers and basic functions are reused in both experiments to keep their implementation similar for comparison whilst using different paradigms. The code to these experiments is stored on a GitHub repository[3].

4 Results

The results in this research are the comparison of the previously mentioned experiments that were carried out. These experiments have been compared both qualitatively and quantitatively.

4.1 Qualitative analysis

The experiments in this research are compared and analyzed by readability, maintainability and modularity. The following subsections each individually describe the positive and negative aspects that EHOP brings based on its metric.

Readability

An example in which EHOP improves the readability of code in the HTTP server experiment is $State$. In the traditional functional programming paradigm, state is threaded through functions. However, using a $State$ effect, the state operations are available to any function that uses the same $State$ effect without having to thread it with explicit bind or do notations. A simplified example from the EHOP experiment showcasing this can be found in Listing 3.

Another aspect on which EHOP improves readability is the explanation of functionality. Functions that use effects have their effect incorporated in their type signature. From this, a programmer can argue more about its functionality based on its effects. For instance, from a function that uses the IO effect, a programmer can deduce that it most likely interacts with the environment. Accordingly, in Listing 3 a developer could deduce that the program might log and manage state. In

[3]https://github.com/sstreef/research-project

```
f :: Sem '[Logging, State (Maybe String)] ()
f = do
    Logging.log (...)
    (maybeString :: Maybe String) <- State.get
    ...
```

Listing 3: Example of a benefit of using EHOP. Two different effects can be used in a single do notation. In a non EHOP approach, state operations would not be able to be sequenced with an IO operation like this. This example has been simplified by omitting unnecessary variables and operations by replacing them by dots.

the traditional paradigm, a programmer has to check the function, and possibly its implementation and declaration, inside a function as its type is not explicit unless annotated. However, this type signature can become more complicated when the number of effects increases, making it harder to understand what functionalities are embedded into the program. Nevertheless, for an HTTP server application this is not the case, as it uses a limited amount of effects. Therefore, in these experiments EHOP adds more value to understanding functionality, which follows one principle of Bloch, "Functionality should be easy to explain" [7, p. 13]. A similar argumentation can be given for the part where effects are ran by effect handlers. They give a good indication on which effects are present, but can increase complexity as the number of effects increase. An example presenting this can be found in Listing 4.

```
resolve :: HTTPRequest -> IO HTTPResponse
resolve req = (do
      raiseRequestHandlingEffect serverSetup
      chain req [resolveRequest, resolveFileRequest])
    & runRequestHandling
    & evalState Nothing
    & runKVStorePurely Map.empty
    & runFileReadingIO
    & runConsoleLogger
    & runM
    >>= snd
```

Listing 4: Example of effect handlers indicating the functionality in a program, taken from the EHOP experiment. From these particular effect handlers it can be deduced that it might log, read files and manage state without having to look at the functions in the program. Consequently, it improves the accessibility and readability of the code.

Contrary to EHOP adding visibility over functionality of code, it can also be hidden. For instance, some effects might require effects, such as state, in their implementation. However they are not interesting to the user of the main effect. Moreover, the ability to interface them could introduce complexity or opportunity for errors by manipulating them. *Polysemy* [3] allows programmers to raise effects on a program, i.e. add new effects dynamically to a program. This is useful for hiding the internal effects of an effect for a programmer

by only having the main effect in the program signature. Its internal effects can then be raised dynamically before they need to be ran by effect handlers when running the main effect. This ensures that the programmer cannot use or manipulate the internal effects. Listing 5 shows the signature of such a raising function from the experiment. Before raising, the program can only interface the $RequestHandling$ operations and after raising, the internal effects can be ran for the $RequestHandling$ effect to be ran. Being able to hide the effects is a concrete example of "*Implementation Should Not Impact API*" [7, p.15] where implementation details are hidden.

```
raise :: Sem (RequestHandling : r) a
      -> Sem (RequestHandling
        : HTTPStaticFilePathState
        : HTTPHandlerStore : FileReading
        : Logging : Embed IO : r) a
```

Listing 5: A function signature taken from the EHOP experiment where the function raises effects on a program to hide the implementation details of the $RequestHandling$ effect. Reducing complexity for the user of the program and removing the opportunity of introducing errors by not being able to use the internal effects.

Maintainability

An advantage of using EHOP is the ease in which the implementation of an effect can be swapped with little effort. For testing this means that a module or function that uses a particular effect can be tested in a more controlled environment by swapping certain effect handlers for more predictable or simple handlers. An example from the EHOP experiment that shows this is the $FileReading$ effect. In the application code this effect is run by the $runFileReadingIO$ handler, which reads a file from disk. This handler has an IO side-effect that is unpredictable since it depends on the file system, which can differ from machine to machine. Therefore, it would be more desirable to use an effect handler that does has predictable behaviour. In this experiment this is achieved by using a different effect handler, called $runFileReadConstant$, which returns a fixed string and does not do any IO operations. The same procedure for ensuring that effects have predictable behaviour can be applied to the $Logging$ effect in this experiment. Rather than having an effect handler that logs using an IO operation, such as $runConsoleLogger$ or $runFileLogger$, it could just discard its content and do nothing. Nevertheless, a traditional paradigm could similarly achieve this by creating functions similar to effect operations and changing their implementations or by using monads to encapsulate different behaviours. However, EHOP allows for a much more simple and easier to use approach. Using effect handlers, the effects of a function can be lifted dynamically based on its context. For instance, having different effect behaviour in a production or development environment without having to change the application code that uses the effect.

A second improvement that EHOP provides is adding new operations to existing effects will not break the application code structurally. Consequently, this means that as long as the operations signatures stay the same and the operations are not removed in an effect, the effect will remain compatible with code that use previous versions of the effect. This allows effects to be backwards compatible with older code, even when adding new operations or changing the implementation of operations.

Modularity

From the experiments, multiple cases were discovered on which effects and effect handlers improve the modularity in code. First, using effect handlers a programmer can define an effect and use this effect in its program without having an effect handler ready to run it. As long as the program does not require evaluating the effect, it is a valid program. This allows for code to be written independently of the implementation of effects. Second, effects and effect handlers that are already defined can be easily extended with extra operations. Adding new effect operations does not impact the application code that is already using an effect as mentioned in the previous Subsection.

4.2 Quantitative analysis

The performance of the http servers produced by the two experiments were measured by means of response time and memory characteristics. The measurements were performed on a single machine of which the specifications can be found in Table 1.

Operating system	Windows 10
Processor	Intel Core i7-8750H
RAM	Samsung M471A2K43CB1-CTD
System type	64-bit

Table 1: The hardware and operating system specifications of the machine running the quantitative analysis.

In Figure 2 the average response time of each HTTP server handling POST requests with different amounts of data can be seen. The results in this figure were produced by the method described in the Methodology Section. All the data from the response time benchmarks can be found in Appendix A.1. The results show that the response time of the HTTP server using effect handlers is overall slightly slower whilst processing POST requests with different payload size.

Furthermore, the results of the total memory used by each application of the experiments can be found in Figure 2. The results were created according to the method describe in the Methodology Section. All the data from the total memory usage benchmarks can be found in Appendix A.2. The results indicate that over the 5 benchmarks that were run, EHOP always has a memory overhead compared to the traditional paradigm.

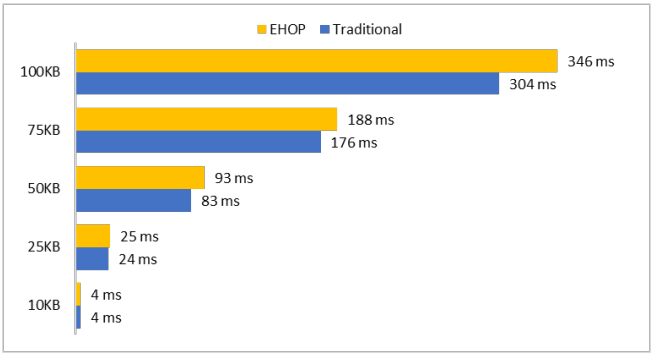

Figure 1: The response time of a POST request where the sent data is sent back reversed. Each result is respective to a different payload size and is based on the average of 20 runs where 100 requests were sent each run. This figure shows that using EHOP on average adds a small overhead to the response time compared to a traditional paradigm.

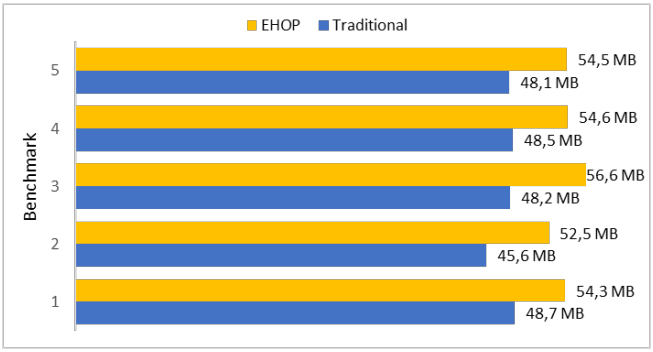

Figure 2: The total memory usage for running 10000 POST requests where the sent data is sent back reversed. Ran 5 times with a payload of 100 bytes for each request. The figure shows that EHOP has a memory overhead compared to the traditional paradigm which therefore outperforms EHOP with memory usage.

5 Discussion

The results of comparing both experiments show that EHOP provides improvement on multiple areas.

First, the readability is overall increased when compared to using a traditional paradigm. EHOP allows for easier recognition of what a function does by looking at the effects or effect handlers applied. Moreover, effects allow for less lines of code when using state since effects can be combined in a function. Furthermore, it can hide implementation details of effects by dynamically raising the effects that an effect requires. This reduces complexity within the function by hiding unnecessary effects.

Second, the results of the experiments showed that effects make code more maintainable. Effect handlers allow for a simple separation of production and development functionality by being able to run effect handlers conditionally and dynamically. Consequently, it allows for testing functions with effects using different effect handler implementations than production code. Moreover, using effects introduces backwards compatibility in code. Operations can be added and implementations can be changed without breaking application code structurally.

Third, EHOP can make improve the modularity of code by using an effect as a bridge between application code and effect handlers. A developer is able to define an effect and write type correct application code without an effect handler implementation. Moreover, effects can be extended without breaking unchanged application code.

Finally, the quantitative results show that EHOP adds an overhead on both the response time and memory usage. This could be due to *Polysemy* being a library and therefore not being as optimized as native alternatives for EHOP. However, this would have to be verified by performing these experiments in a language such as *Koka*.

The results of the performance indicate that EHOP might not be useful for IO intensive applications. Since IO intensive applications are performance driven, especially HTTP servers where any overhead in response time is extremely noticeable and undesirable. Qualitative improvements such as readability, maintainability and modularity might therefore not out way the costs in performance.

5.1 Limitations

This case study focuses on the effect of EHOP on IO intensive applications and only performs experiments on programming an HTTP server. Consequently, it is possible that it did not capture all cases for IO intensive applications in which EHOP might be useful. Moreover, all experiments were carried out in Haskell, which does not have built-in support for effects and effect handlers. Therefore, the quantitative results might show an overhead in performance. Furthermore, the experiments were carried out by a single subject on a single machine. This might show a bias in programming based on the experience of the developer. Additionally, the machine that ran the quantitative results could show bias by physical constraints and its operating system. Finally, as mentioned in the Methodology Section, this research took a less objective and numerical approach of qualitatively analyzing code. As a result of the limited time in which this research was carried out, metrics that try to objectively analyze code were not used. Metrics such as the Halstead metrics [8] for readability would take up too much time as it requires counting variables and methods by hand since there are no tools for this available for Haskell. Furthermore, the Cyclomatic Complexity, coupling and cohesion of code was considered, however they were deemed to be not usable with the *Polysemy* library or applicable in the time frame given for this research.

5.2 Future work

Future improvements and extensions could be carried out to improve the validity and quality of the results. From the aforementioned limitations multiple future work could be carried out. First, this case study could be performed with a

different application for its experiments. This could, for instance, include *serial communication* or implementing lower network protocols such as UDP and TCP. Second, the experiments could be carried out in a different programming language that has built-in support for EHOP, for example *Koka*. However, it should be noted that as of writing this paper, *Koka* is limited in its IO capabilities. Third, the qualitative analysis of both experiments could be performed, by using more objective metrics. Such metrics include the Halstead metrics [8], Cylomatic Complexity and coupling cohesion. This way the conclusion of the results could be more objectively based. Finally, the quantitative evaluation of experiments could be reproduced more often and on different machines to take an average and come to a more trustworthy result.

5.3 Ethics & reproduction

The experiments, evaluation and processing of results in this research have been carried out by a single entity, namely the author. No other subjects were involved in this research and therefore nobody other than the author could be negatively impacted by the results of this study.

Moreover, the results of this case study come from the execution of two experiments that involved programming. The resulting code of each experiment can be found in a GitHub repository[4] and is publicly accessible for reference and reproduction. The quantitative analysis was performed on a single machine, however performed multiple times. It should be noted that the quantitative results may differ based on the system that runs the programs. It is inevitable that these memory and time measurements are susceptible to the conditions of the machine that runs it. Nonetheless, multiple runs of quantitative analysis showed similar behaviour as can be seen in Appendix A, where all quantitative results that were gather are available.

6 Conclusion

This research aimed to answer how EHOP affects the modularity, readability, maintainability and performance of code in IO intensive applications when compared to a traditional approach. The results were formed by analyzing the qualitative and quantitative properties of two functionally identical HTTP servers. Each created using a different paradigm, namely the EHOP and traditional paradigm.

The qualitative analysis of both programs showed the effect of EHOP on the readability, maintainability and modularity of the code. These results indicated that EHOP improves the code on each of these. First, the readability of code increases by making use of effects and effect handlers. Effect signatures in functions allow a developer to deduce the possible functionality from the effects that it uses. Moreover, effects decrease the lines of code used when using monads such as state, by being able to combine and use effects in a single function. Second, EHOP can improve the maintainability of code by being able to dynamically change effect behaviour with effect handlers. This allows for more simple and easy testing of functions that use effects. Moreover, using effects introduces backwards compatibility in code since operations can be added and implementations can be changed without breaking existing application code. Third, the modularity of code can improved by introducing effects. Application code can be written type correctly with only an effect defined, allowing the effect handler to be independently added later on in development. Additionally, effects can be extended with operations without influencing application code, making it an independent and more modular improvement.

Furthermore, the quantitative analysis of the program exposed the overhead in memory and response time of the EHOP HTTP server. Both performed worse than the traditional paradigm by adding an overhead in response time and memory.

Concluding, EHOP can qualitatively improve code in aspects such as readability, maintainability and modularity, however it has an overhead in run-time and memory performance. For IO intensive applications, performance is considered to be very important which EHOP seems to only degrade. Therefore, according to the result of this particular case study, the effect that EHOP has on IO intensive applications is considered to be more negative. However, more IO intensive applications should be analyzed to support, as well as being programmed in different languages that have a built-in support for EHOP to support this result.

[4] https://github.com/sstreef/research-project

A Quantitative results

A.1 Response time benchmarks

The following tables contain all data collected from running POST requests on both applications that were developed in the two experiments. Each row in a table represents a single benchmark that ran 100 POST requests against an HTTP server. Each table contains 20 benchmarks and is different in the amount of data that was sent per POST request.

75 KB payload			
Total time (s)		Average time (s)	
EHOP	Traditional	EHOP	Traditional
19,037	18,410	0,190	0,184
18,770	17,855	0,188	0,179
18,831	18,431	0,188	0,184
19,685	16,883	0,197	0,169
21,987	17,503	0,220	0,175
19,901	18,650	0,199	0,186
18,063	18,017	0,181	0,180
18,160	16,940	0,182	0,169
18,413	16,869	0,184	0,169
18,289	17,288	0,183	0,173
18,187	17,272	0,182	0,173
18,510	19,705	0,185	0,197
18,219	16,904	0,182	0,169
18,777	17,098	0,188	0,171
18,366	18,876	0,184	0,189
18,496	17,280	0,185	0,173
18,658	17,103	0,187	0,171
18,405	17,311	0,184	0,173
18,595	17,129	0,186	0,171
18,356	16,830	0,184	0,168

100 KB payload			
Total time (s)		Average time (s)	
EHOP	Traditional	EHOP	Traditional
32,751	30,115	0,328	0,301
34,887	29,220	0,349	0,292
34,733	29,196	0,347	0,292
34,739	28,989	0,347	0,290
33,479	29,057	0,335	0,291
33,690	29,641	0,337	0,296
33,891	29,692	0,339	0,297
33,061	29,906	0,331	0,299
34,059	29,545	0,341	0,295
35,008	29,524	0,350	0,295
35,294	29,864	0,353	0,299
35,615	29,990	0,356	0,300
35,110	29,251	0,351	0,293
35,430	29,054	0,354	0,291
36,956	29,792	0,370	0,298
34,635	29,114	0,346	0,291
34,045	33,186	0,340	0,332
34,967	39,599	0,350	0,396
34,970	31,594	0,350	0,316
33,788	32,164	0,338	0,322

50 KB payload			
Total time (s)		Average time (s)	
EHOP	Traditional	EHOP	Traditional
9,389	8,450	0,094	0,084
8,996	8,297	0,090	0,083
9,474	8,168	0,095	0,082
9,338	8,182	0,093	0,082
9,518	8,459	0,095	0,085
9,489	8,203	0,095	0,082
9,100	8,242	0,091	0,082
9,304	8,548	0,093	0,085
8,632	8,209	0,086	0,082
9,456	8,353	0,095	0,084
9,581	8,981	0,096	0,090
9,021	8,171	0,090	0,082
9,292	8,309	0,093	0,083
8,811	8,375	0,088	0,084
8,692	8,146	0,087	0,081
9,836	8,238	0,098	0,082
9,117	8,030	0,091	0,080
9,681	8,281	0,097	0,083
9,435	8,322	0,094	0,083
9,754	8,242	0,098	0,082

25 KB payload			
Total time (s)		Average time (s)	
EHOP	Traditional	EHOP	Traditional
2,471	2,330	0,025	0,023
2,431	2,630	0,024	0,026
2,578	2,552	0,026	0,026
2,902	2,645	0,029	0,026
2,576	2,311	0,026	0,023
2,390	2,282	0,024	0,023
2,386	2,491	0,024	0,025
2,398	2,435	0,024	0,024
2,348	2,401	0,023	0,024
2,400	2,378	0,024	0,024
2,360	2,373	0,024	0,024
2,668	2,374	0,027	0,024
2,424	2,359	0,024	0,024
2,441	2,514	0,024	0,025
2,348	2,539	0,023	0,025
2,323	2,311	0,023	0,023
2,443	2,372	0,024	0,024
2,467	2,222	0,025	0,022
2,372	2,334	0,024	0,023
2,345	2,333	0,023	0,023

10 KB payload			
Total time (s)		Average time (s)	
EHOP	Traditional	EHOP	Traditional
0,429	0,435	0,004	0,004
0,399	0,402	0,004	0,004
0,390	0,346	0,004	0,003
0,386	0,401	0,004	0,004
0,388	0,395	0,004	0,004
0,481	0,420	0,005	0,004
0,427	0,374	0,004	0,004
0,399	0,382	0,004	0,004
0,416	0,417	0,004	0,004
0,401	0,337	0,004	0,003
0,399	0,366	0,004	0,004
0,374	0,366	0,004	0,004
0,372	0,389	0,004	0,004
0,404	0,364	0,004	0,004
0,398	0,355	0,004	0,004
0,392	0,371	0,004	0,004
0,390	0,400	0,004	0,004
0,420	0,384	0,004	0,004
0,381	0,417	0,004	0,004
0,407	0,383	0,004	0,004

A.2 Memory benchmarks

The following table contains all data collected from running POST requests on both applications that were developed in the two experiments. Each row in the table represents a single benchmark that ran 100 POST requests 100 times. The data collected is the amount of bytes that were allocated by running the server and sending requests.

Total memory allocation (Bytes)	
EHOP	Traditional
54.293.642	48.657.341
52.476.075	45.563.386
56.597.580	48.186.481
54.555.494	48.490.021
54.485.438	48.095.525

References

[1] R. Chatley, A. Donaldson, and A. Mycroft, "The Next 7000 Programming Languages," *Lecture Notes in Computer Science*, pp. 250–282, 2019.

[2] D. Hillerström, "Foundations for Programming and Implementing Effect Handlers," Ph.D. dissertation, The University of Edinburgh, 2021.

[3] "Polysemy: higher-order, no-boilerplate monads." [Online]. Available: https://github.com/polysemy-research/polysemy

[4] D. Leijen, "Koka: Programming with Row Polymorphic Effect Types," *Electronic Proceedings in Theoretical Computer Science*, vol. 153, pp. 100–126, 2014.

[5] S. Lindley, C. McBride, and C. McLaughlin, "Do be do be do," *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.

[6] S. Marlow, "Haskell 2010 Language Report," https://www.haskell.org/onlinereport/haskell2010/, 2010.

[7] J. Bloch, "How to Design a Good API and Why it Matters," https://research.google.com/pubs/archive/32713.pdf, 2007.

[8] M. Halstead, *Elements of software science (Operating and programming systems series)*. Elsevier, 1977.