Compiling for Concise Code and Efficient I/O

Sebastian Ertel  
Chair for Compiler Construction  
Technische Universität Dresden  
Dresden, Germany  
sebastian.ertel@tu-dresden.de

Justus Adam  
Chair for Compiler Construction  
Technische Universität Dresden  
Dresden, Germany  
justus.adam@tu-dresden.de

Andrés Goens  
Chair for Compiler Construction  
Technische Universität Dresden  
Dresden, Germany  
andres.goens@tu-dresden.de

Jeronimo Castrillon  
Chair for Compiler Construction  
Technische Universität Dresden  
Dresden, Germany  
jeronimo.castrillon@tu-dresden.de

Abstract

Large infrastructures of Internet companies, such as Facebook and Twitter, are composed of several layers of microservices. While this modularity provides scalability to the system, the I/O associated with each service request strongly impacts its performance. In this context, writing concise programs which execute I/O efficiently is especially challenging. In this paper, we introduce Ÿauhau, a novel compile-time solution. Ÿauhau reduces the number of I/O calls through rewrites on a simple expression language. To execute I/O concurrently, it lowers the expression language to a dataflow representation. Our approach can be used alongside an existing programming language, permitting the use of legacy code. We describe an implementation in the JVM and use it to evaluate our approach. Experiments show that Ÿauhau can significantly improve I/O, both in terms of the number of I/O calls and concurrent execution. Ÿauhau outperforms state-of-the-art approaches with similar goals.

CCS Concepts   • Software and its engineering → Functional languages; Data flow languages; Concurrent programming structures;

Keywords   I/O, dataflow, concurrency

ACM Reference Format:

1 Introduction

In today’s Internet applications, I/O is a dominating factor for performance, an aspect that has received little attention in compiler research. The server-side infrastructures of internet companies such as Facebook [1], Twitter [11], and Amazon [10] serve millions of complex web pages. They run hundreds of small programs, called microservices, that communicate with each other via the network. Figure 1 shows the interactions of the microservices at Amazon. In these infrastructures, reducing network I/O to satisfy sub-millisecond latencies is of paramount importance [7].

Microservice code bases are massive and are deployed across clusters of machines. These systems run 24/7 and need to be highly flexible and highly available. They must accommodate for live (code) updates and tolerate machine and network failures without noticeable downtime. A service-oriented architecture [24] primarily addresses updates while the extension to microservices deals with failures [8]. The idea is to expose functions as a service that runs as a standalone server application and responds to HTTP requests. It is this loose coupling of services via the network that enables both flexibility and reliability for the executing software. But this comes at the cost of code conciseness and I/O efficiency. Code that calls another service over the network instead of invoking a normal function loses conciseness. The boilerplate code for the I/O call obfuscates the functionality of the program and makes it harder to maintain. The additional latency to receive the response adds to the latency of the calling service.

1.1 Code Conciseness versus I/O Efficiency

Programmers commonly use frameworks, like Thrift [3], to hide the boilerplate network code behind normal function calls. This improves code conciseness but not I/O efficiency. There are essentially two ways to improve I/O latency: by reducing the number of I/O calls and through concurrency.
To reduce the number of I/O calls in a program, one can remove redundant calls and batch multiple calls to the same service into a single one. Redundant calls occur naturally in concise code because such a code style fosters a modular structure in which I/O calls can happen inside different functions. The programmer should not have to, e.g., introduce a global cache to reuse results of previous I/O calls. This would add the optimization aspect directly into the algorithm of the application/service, making the code harder to maintain and extend, especially in combination with concurrency. Batching, on the other hand, does not necessarily reduce the amount of data transferred but does decrease resource contention. A batched request utilizes only a single HTTP connection, a single socket on the server machine and a single connection to the database. If all services batch I/O then this can substantially decrease contention in the whole infrastructure. Network performance dramatically benefits when multiple requests are batched into a single one [22].

As a simple example, consider a web blog that displays information on two panes. The main pane shows the most recent posts and the left pane presents meta information such as the headings of the most frequently viewed ones and a list of topics each tagged with the number of associated posts. The blog service is then responsible of generating the HTML content for a list of posts. Figure 2 compares the latency of this batch call with a sequential retrieval of the blog posts. The batched call benefits from various optimizations such that retrieving 19 (4 kB) blog posts becomes almost 18× faster. Note that batched retrievals with a larger latency cannot be seen in the plot because of the scale.

A concurrent program can be executed in parallel to decrease latency. When the concurrent program contains I/O calls, these can be performed in parallel even if the computation is executed sequentially. A concurrent execution of the program can fully unblock computation from I/O, but is at odds with a reduction of I/O calls. To reduce I/O calls, they need to be collected first which removes the concurrency between the calls. Not all I/O calls can be grouped into a single one, for example if they interface with different services. Such calls would now execute sequentially. In order to benefit from concurrency, a batching optimization must preserve the concurrent program structure, not only between the resulting I/O calls but also between said calls and the rest of the computation.

To introduce concurrency, the developer would have to split up the program into different parts and place them onto threads or event handlers. Threads use locks which can introduce deadlocks [18] and event-based programming suffers from stack ripping [2]. There is a long debate in the systems community which of the two approaches provides the best performance [23, 30]. Programming with futures on the other hand, needs the concept of a monad which most developers in imperative languages such as Java or even functional languages such as Clojure struggle with [12]. All of these programming models for concurrency introduce new abstractions and clutter the code dramatically. Once more, they lift optimization aspects into the application code, resulting in unconcise code.

We argue that a compiler-based approach is needed to help provide efficient I/O from concise code. State-of-the-art approaches such as Haxl [20] and Muse [17] do not fully succeed to provide both code conciseness and I/O efficiency. They require the programmer to adopt a certain programming style to perform efficient I/O. This is due to the fact that both are runtime frameworks and require some form of concurrent programming solely to batch I/O calls. Both frameworks fall short on exposing the concurrency of the application to the system they execute on.

### 1.2 Contribution

In this paper, we present a compiler-based solution that satisfies both requirements: concise code and efficient I/O. Our approach, depicted in Figure 3, builds on top of two intermediate representations: an expression language and a dataflow representation [6]. The expression language is based on the
we implemented it as a compiler and runtime system for transformations. To remove these dependencies, we define a set of transformations in the expression language which are provably semantic-preserving. This batching transformation introduces dependencies which block computation from making progress while I/O is performed. To remove these dependencies, we define transformations on the dataflow graph. To validate our approach in practice, we implemented it as a compiler and runtime system for a domain-specific language called Yauhau, that can be embedded into Clojure and supports arbitrary JVM code. For our simple blog example, our transformations improve latency by a factor of 4. To evaluate Yauhau in the context of more complex microservices, we use a tool that builds programs along the characteristics of real service implementations. For the generated services in our experiments our transformations are superior to Haxl and Muse in terms of latency.

The rest of the paper is structured as follows. Section 2 introduces the expression IR and the transformations to reduce the number of I/O calls. In Section 3, we define dataflow IR and the transformations to support a fully concurrent execution. We review related work in Section 4. Section 5 briefly describes our Yauhau implementation and evaluates it before we conclude in Section 6.

2 Reducing I/O on an Expression IR

We now present our expression IR. It is based on the call-by-need lambda calculus [4, 5], which prevents duplicated calculations. We want this for two reasons. First, our expressions can contain side-effects and duplicating them would alter the semantics of the program. Second, our intention is to minimize the computation, i.e. the I/O calls in the program expression. Figure 4 defines our IR as an expression language. In addition to the terms of the call-by-need lambda calculus for variables, abstraction, application and lexical scoping, we define combinators for conditionals, foreign function application and I/O calls. The combinator \( \text{ff}_f \) applies a function \( f \) that is not defined in the calculus to an arbitrary number of values. This addition allows to integrate code written in other languages, such as Java. It allows to integrate legacy code and makes our approach practical. For this reason, our values may either be a value in \( V_{\text{ff}} \), the value domain of the integrated language, an abstraction or a list of values. Instead of recursion, we define the well-known higher-order function \( \text{map} \) to express an independent computation on the items in a list. In Section 2.2, we argue that \( \text{map} \) is sufficient to perform I/O optimizations. The \( \text{nth} \) function retrieves individual items from a list. We facilitate I/O calls in the language with the \( \text{io} \) combinator that takes a request as input and returns a response as its result. Both values, the request and the response, are defined in \( V_{\text{ff}} \). As such, \( \text{io} \) can be seen as a foreign function call and the following semantic equivalence holds: \( \text{io} \equiv \text{fff}_\text{io} \). A request is a tuple that contains the (HTTP) reference to the service to be interfaced with and the data to be transferred. A response is a single value. To perform I/O optimizations. The \( \text{io} \) function retrieves last 10 posts of the existing posts, are defined in \( V_{\text{ff}} \). To fetch all identifiers of the existing posts, we write \( \text{io}_{\text{postIds}} \). As an example, here is the expression for our blog service:

```plaintext
let xMainPane =
  ff_runner(
    map(\lambda x.io(ff_reqPostContent(x)))
      . produce HTML
    ff_last-n(10)
      . fetch post contents
      . filter last 10 posts
    map(\lambda x.io(ff_reqPostInfo(x)))
      . fetch meta data
    io(ff_reqPostIds()))))
      . fetch ids of all posts

let xTopics =
  ff_runner(
    map(\lambda x.io(ff_reqPostInfo(x)))
      . fetch meta data
    io(ff_reqPostIds()))))
      . fetch ids of all posts

let xPopularPosts = maps(\text{io}(ff_reqPostIds))

let xLeftPane = ff_runner(ffCompose(xTopics xPopularPosts))
let xMainPane = ff_runner(ffCompose(xMainPane xLeftPane))

Figure 4. Language definition of the expression IR.
```
Two requests query the same service if their references are equal. Since we define this equality over the values instead of the types, our transformations need to produce code that finds out at runtime which I/O calls can and cannot be batched. In the remainder of this section, if not stated otherwise we transform (\(\rightarrow\)) expressions only by applying the standard axioms from the call-by-need lambda calculus \([4]\). That is, all transformations preserve the semantics of the expression. We first introduce the transformations that can reduce the number of I/O calls whose evaluation is not influenced by control flow. Afterwards, we show how even I/O calls in expressions passed to conditionals and map can participate in these transformations.

### 2.1 I/O Reduction

Intuitively, we can only reduce the number of I/O calls (to the same remote service) inside a group of I/O calls that are data independent of each other. This means that there exist no direct nor transitive data dependencies between the calls in this group.

This limits the applicability of our approach. However, we believe it is enough to cover a great number of use-cases. In future work we plan to investigate how to batch I/O calls with dependencies, effectively sending part of the computation to a different service, and not only a simple I/O request.

In order to find such a group of independent calls, we first transform the expression into a form that makes all data dependencies explicit.

**Definition 2.1.** An expression is in explicit data dependency (EDD) form, if and only if it does not contain applications and each result of a combinator or a (predefined) function is bound to a variable.

Thus, the data dependencies in the expression are explicitly defined by the bound variables \(x_1 \ldots x_n\)

\[
\text{let } x_1 = t_1 \text{ in let } x_2 = t_2 \text{ in } \ldots \text{let } x_n = t_n \text{ in } x_n
\]

Each expression \(t\) is a term with a combinator (\(\text{io}, \text{ff}, \text{if}\)) or a predefined function, e.g., \(\text{map}, \text{nth}\). For example, the construction of the main pane of the web blog in EDD form is as follows:

\[
\begin{align*}
\text{let } x_1 &= \text{ff}\_\text{reqPostIds}() \text{ in } \\
\text{let } x_2 &= \text{io}(x_1) \text{ in } \\
\text{let } f &= \lambda x\_\text{postId}. \text{let } y_1 = \text{ff}\_\text{reqPostInfo}(x\_\text{postId}) \text{ in } \\
&\quad \text{let } y_2 = \text{io}(y_1) \text{ in } y_2 \text{ in } \\
\text{let } x_3 &= \text{map}(f \_x) \text{ in } \\
\text{let } x_4 &= \text{ff}\_\text{last} (10 \_x3) \text{ in } \\
\text{let } g &= \lambda x\_\text{postInfo}. \text{let } z_1 = \text{ff}\_\text{reqPostContent}(x\_\text{postInfo}) \text{ in } \\
&\quad \text{let } z_2 = \text{io}(z_1) \text{ in } z_2 \text{ in } \\
\text{let } x_5 &= \text{map}(g \_x) \text{ in } \\
\text{let } x_6 &= \text{ff}\_\text{render}(x_5) \text{ in } x_6
\end{align*}
\]

In order to find independent groups of I/O calls in the EDD expression, we use a well-known technique called \(\text{let-floating}\) that is also used in the Haskell compiler \([25]\). Let-floating allows to move bindings either inwards or outwards for as long as the requirements for variable access are preserved, i.e., no new free variables are created. As such, let-floating is completely semantic preserving. In our case, we float each I/O binding as much inward as possible, i.e., we delay the evaluation of I/O calls until no further progress can be made without performing I/O.

\[
\begin{align*}
\text{let } x_1 = \text{ff}\_\text{reqPostIds}() \text{ in } 1. \text{mainPane} \\
\text{let } x_2 = \text{ff}\_\text{reqPostIds}() \text{ in } 1. \text{topics} \\
\text{let } x_3 = \text{io}(x_1) \text{ in } 2. \text{mainPane} \\
\text{let } x_4 = \text{io}(x_2) \text{ in } 2. \text{topics} \\
\end{align*}
\]

For the sake of brevity, we use \(e\) to denote the rest of the blog computation. At this point, we perform the final step and replace the group of I/O calls with a single call to the function \(\text{bio}\) which performs the batching and executes the I/O. As such the following semantic equivalence holds:

\[
\text{bio}(i_1 \ldots i_n) \equiv \{ \text{io}(i_1) \ldots \text{io}(i_n) \}
\]

(2)

The function is defined using two new (foreign) functions, \(\text{batch}\) and \(\text{unbatch}\):

\[
\begin{align*}
\text{bio}(x_1 \ldots x_n) &::= \text{let } y_1 \ldots m = \text{batch}(x_1 \ldots x_n) \text{ in } \\
&\quad \text{let } z_1 \ldots m = \text{map}(\lambda x. \text{io}(x) \_y_1 \ldots \_m) \text{ in } \\
&\quad \text{let } y_1 \ldots n = \text{unbatch}(z_1 \ldots m) \text{ in } y_1 \ldots n
\end{align*}
\]

We say that variable \(y_1 \ldots m\) stores the list of values that would otherwise be bound to variables \(y_1, \ldots, y_m\). Since we do not know at compile-time which calls can be batched, \(\text{batch}\) takes \(n\) I/O requests and returns \(m\) batched I/O requests with \(n \geq m\). Each batched I/O request interfaces with a different service. A batched request contains a list of references to the original requests it satisfies, i.e., the positions in the argument list of \(\text{bio}\). We assume that this information is preserved across the I/O call such that \(\text{unbatch}\) can re-associate the response to the position in the result list. That is, it preserves the direct mapping between the inputs of batch and the outputs of \(\text{unbatch}\) to preserve the equivalence defined in Equation 2. Finally, this list needs to be destructured again to re-establish the bindings:

\[
\begin{align*}
\text{let } y_1 &= \text{io}(x_1) \text{ in } \ldots \text{let } y_n &= \text{io}(x_n) \text{ in } e \\
\rightarrow \text{let } y_1 \ldots n &= \text{bio}(x_1 \ldots x_n) \text{ in } \\
\text{let } y_1 &= \text{nth}(1 \_y_1 \ldots \_n) \text{ in } \ldots \text{let } y_n &= \text{nth}(n \_y_1 \ldots \_n) \text{ in } e
\end{align*}
\]

Note that the order of I/O calls inside an I/O group is normally non-deterministic. This includes calls with and without side-effects to the same service. But I/O calls that are located in different functions may nevertheless assume an implicit order, i.e., on the side-effects occurring to the service. This fosters a modular program design by allowing to add new functionality without making these dependencies explicit. In
order to preserve the consistency of the program, we require that the service receiving the batch defines the execution order of the contained requests. For example, a service may define to always execute all side-effect-free requests before side-effecting ones.

2.2 I/O Lifting

The lambda calculus does not directly incorporate control flow. For that reason, most programming languages define at least two additional forms: conditionals and recursion. Instead of relying on the more general concept of recursion, we base our I/O lifting on the higher-order map function. The reasoning behind this decision is as follows: We cannot optimize I/O across recursive calls that strictly depend on each other. From the perspective of the I/O call that is located in a recursive function, we have to make the distinction whether the recursion is strictly sequential or not. The higher-order function scan\(^\dagger\) is a synonym for a strictly sequential recursion while map does not define an order on the computation of the results. For the rest of this paper, we assume that the abstraction passed to scan is strictly sequential and define

\[ \text{scan} \equiv \text{fold}/\text{reduce} \]

The if combinator and map function are special because they control the evaluation of the expression(s) passed to it as arguments. The if combinator may evaluate only one of the two expressions passed to it. The map function applies the expression to each value in the list. Although we transform the argument expressions into EDD form, I/O calls located inside of them cannot directly participate in the rest of the I/O-reducing transformations. To enable this, we need to extract, i.e., lift, I/O calls out of these argument expressions while still preserving the evaluation semantics. To this end we use the following two simple and semantic-preserving rewrite rules:

\[
\begin{align*}
\text{if}(\text{c } \text{io}(x) \text{ io}(x)) & \equiv \text{io}(x) \\
\text{map}(\lambda x.\text{io}(x)\ [v_1 \ldots v_n]) & \equiv \text{io}(v_1) \ldots \text{io}(v_n)
\end{align*}
\]

In general, instead of the expressions io(x) and \(\lambda x.\text{io}(x)\) on the left-hand side of these rules we could find arbitrary expressions. The challenge is to find a chain of transformations that creates a form where the expression passed to an if or a map is only an I/O call, as in rules 2 and 3.

We proceed in two steps. At first, we explain how to transform an expression into a form with three individual parts, where one of them contains solely an I/O call. Afterwards, we use this new form to extract I/O calls out of an if and a map using the rewrite rules defined above.

We start with an arbitrary expression in EDD form and use a concept called lambda lifting [15]. Figure 5 visualizes the transformations. Figure 5a shows an EDD expression in a form where the right-hand side of the \(j\)th binding applies an I/O call to the variable \(x_{j-1}\) bound in the \(j - 1\)th binding. Without loss of generality we assume that \(t_j = \text{io}(x_{j-1})\) because we can find this form via let-floating. In Figures 5b, 5c and 5d, we apply the lambda lifting transformations in order to devise an expression with three individual functions: the computation of the request \((f)\), the I/O call \((g)\), and the continuation of the computation \((h)\). To lambda lift the expression \(t_j\) that executes the I/O call in Figure 5b, we

1. create an abstraction for \(t_j\) that defines all its free variables, in that case \(x_{j-1}\), as arguments and bind this abstraction to \(g\),
\[
\text{io}(x_{j-1}) \rightarrow g = \lambda x_{j-1}.\text{io}(x_{j-1})
\]

2. apply \(g\) in place of \(t_j\) and
\[
t_j \rightarrow g \ x_{j-1}
\]

3. reconstruct the lexical scope for the rest of the computation \(e_{j+1} \ldots n\).

\[
\text{let } x_j = g \ x_{j-1} \text{ in } e_{j+1} \ldots n
\]

In Figure 5c, we lift the computation of the request \(e_{1 \ldots j-1}\) and in Figure 5d the continuation of the computation on the result of the I/O call \(e_{j+1} \ldots n\). To cover the general case, we assume that \(e_{j+1} \ldots n\) does not only contain applications to the I/O result \(x_j\) but also to all other variables \(x_1, \ldots, x_{j-1}\) in its scope. To provide them in the lexical scope for the application to \(h\), the lambda lifting of \(e_{1 \ldots j-1}\) returns them from \(f\) and rebinds them after the application. The analysis to reduce this set of free variables to the ones actually used in \(e_{j+1} \ldots n\) is straightforward and therefore left out. With this process, we effectively created an expression that captures the three parts as individual functions.

In the final step, we apply this approach to expressions passed to if and map such that we can extract the application of \(g\), i.e., the I/O call. We abstract over the specific combinator or function with a combinator \(c\) that takes a single expression as its argument. We focus solely on the applications and leave out the abstractions and the reconstruction of the lexical scope \(x_1 \ldots x_n\) from \(x_{1 \ldots j-1}\). For \(c\), we define the following semantic equivalence:

\[
\begin{align*}
\text{c(let } x_1 \ldots j-1 = f \ x_0 \text{ in } & \equiv \text{let } x_1 \ldots j-1 = c(f \ x_0) \text{ in } \\
\text{let } x_j = g \ x_{j-1} \text{ in } & \equiv \text{let } x_j = c(g \ x_{j-1}) \text{ in } \\
\text{let } x_n = h \ x_1 \ldots x_j \text{ in } x_n & \equiv \text{let } x_n = c(h \ x_1 \ldots x_j) \text{ in } x_n
\end{align*}
\]

That is, passing a single expression to \(c\) is semantically equivalent to evaluating \(c\) on each individual part of it. This holds under the premise that the individual applications of \(c\) preserve the operational semantics of the single application. When \(c\) is a conditional if, this requires that

\(^\dagger\text{scan}\) is a version of fold/reduce that emits not only the result of the last recursion step but the results of all recursion steps.

\(^2\text{We use the notation } e_{1 \ldots n}\text{ to refer to the part of the expression defined in Figure 5a that binds the results of the terms } t_1 \ldots t_n\text{ to variables } x_1 \ldots x_n.\)
the individual if applications use the same condition result bound to x-cond:

\[
\begin{align*}
&\text{let } x_1 \ldots x_{j-1} = \text{if}(x_{\text{cond}} \, \text{x}_\text{true} \, x_0) \, \text{x}_\text{false} \, x_0) \text{ in} \\
&\quad \text{let } x_j = \text{if}(x_{\text{cond}} \, \text{g}_\text{true} \, x_{j-1}) \, \text{g}_\text{false} \, x_{j-1}) \text{ in} \\
&\quad \text{let } x_n = \text{if}(x_{\text{cond}} \, \lambda \,(h_\text{true} \, x_1 \ldots x_j) \, \lambda \,(h_\text{false} \, x_1 \ldots x_j)) \text{ in } x_n \\
\end{align*}
\]

The very same concept applies to the map function. Since map returns a list instead of a single value, the variable x_j[m] denotes the list of m results where each would have been bound to variable x_j.

\[
\begin{align*}
&\text{let } x_{[1\ldots j-1]}[m] = \text{map}(\lambda x_0. \, f \, [v_1 \ldots v_m]) \text{ in} \\
&\quad \text{let } x_{j}[m] = \text{map}(\lambda x_{j-1} \, g \, x_{j-1}) \text{ in} \\
&\quad \text{let } x_{n}[m] = \text{map}(\lambda x_1 \ldots x_j \, h \, x_1 \ldots x_j \, x_{[1\ldots j]}[m]) \text{ in } x_{n}[m] \\
\end{align*}
\]

For both, if and map, the application of g reduces via beta reduction to a single I/O call. That is, it matches the left-hand side of equations 2 and 3 such that we can lift the I/O call out of the expressions. Our I/O lifting works for a single c at a time but we can apply it again to the resulting expression to handle deeply nested combinations of if and map. In case of conditionals, we assumed that \text{x}_\text{true} and \text{x}_\text{false} both have an I/O call. If either expression does not perform an I/O call, we simply add one to it that executes an empty request, i.e., does not perform real I/O and filter it out in bio. This allows optimizing the true I/O call on the other branch. Note the importance of the map function: since the same expression is applied to all m values in the list, it has the potential to reduce m I/O calls to one. Further note that our concept of lifting expressions is not restricted to I/O calls. It is more general and can lift other foreign function calls or even expressions for as long as there exists a semantic equivalence in the spirit of Equations 2 and 3.

3 A Dataflow IR for Implicit Concurrency
Our expression IR has no explicit constructs for concurrency such as threads, tasks or futures. This is on purpose: Our intention is to hide concurrency issues from the developer and let the compiler perform the optimization. The expression IR builds upon the lambda calculus, which is well-suited for concurrency. The very essence of the Church-Rosser property, also referred to as the diamond property, is that there may exist multiple reductions that all lead to the same result [4]. During reduction, a step may occur where more than a single redex (reducible expression) exists. This can only be the case when these redexes are (data) independent of each other. To capture this essential property, we lower our expression IR to a dataflow IR.

3.1 From Implicit to Explicit Concurrency
Dataflow is an inherently concurrent representation of a program as a directed graph where nodes are computations and edges represent data transfer between them. As such, we
present our dataflow IR in an abstract fashion, as an execution model for exposing concurrency. For a concrete concurrent or even parallel implementation, a compiler back-end can map this graph either to threads and queues [19] or to processors and interconnects, for example on an FPGA [29]. For this paper, we implemented the threads-and-queues version. The details are beyond the scope of this paper.

The Dataflow IR  We define our dataflow IR on the left-hand side of Figure 6. The basic dataflow elements are nodes, edges and ports. A dataflow node receives data via its input ports to perform a computation and emits the results via its output ports. Such a computation can correspond to foreign function or I/O calls, as well as to one of the special dataflow predefined value functions. Data travels through the graph via the edges where an edge originates at an output port and terminates at an input port.

An input port can only be connected to a single edge while an output port can have multiple outgoing edges. An output port replicates each emitted value and sends it via its outgoing edges to other nodes. Typical dataflow nodes dequeue one value either from one or all their input ports, perform a computation and emit a result. That is, they have a 1–1 correspondence from the input to the output, very much like a function. But dataflow nodes are free to define their own correspondence. For example, a node that retrieves a value from one or all its input ports but emits N values before retrieving the next input value has correspondence 1–N.

The opposite N–1 node retrieves N values from one or all its input ports and only then emits a single result value. This correspondence is depicted by the color of the node in our figures. In order to support this concept, a node is allowed to have state, i.e., its computation may have side-effects. We define a list of 1–1 dataflow nodes that allow control flow and enhance concurrency. The ctrl node takes a boolean and turns it into a control signal that states whether the next node shall be executed or not. If the control signal is false then the node is not allowed to perform its computation and must discard the data from its input ports. The not node implements the negation of a boolean. The sel (select) node receives a choice on its bottom input port that identifies the input port from which to forward the next data item. The ds (deterministic split) node may have any number of output ports and forwards arriving packets in a round-robin fashion. The dm (deterministic merge) node may have any number of input ports and performs the inverse operation. Additionally, we define three nodes to operate on lists. The len (length of list) computes the size of a list. The ()~> is a 1–N node that receives a list and emits its values one at a time. In order to perform the inverse N–1 operation, the ~>[] node first receives the number N on its input port on the top before it can construct the result list.

Lowering Expressions to Dataflow  From an expression in EDD form we can easily derive the corresponding dataflow graph, as shown on the right-hand side of Figure 6. Each

![Figure 6. Definition of the dataflow IR and the translation from the expression IR to the dataflow IR.](image-url)
We introduce concurrency for the function in two steps. First, we translate it into a dataflow graph with concurrent I/O calls. Then we define a translation that also unblocks the rest of the program.

The translation exploits the fact that a dataflow node can have multiple output ports while a function/combinator in our expression IR can only return a single value. Most programming languages support syntactic sugar to support multiple return values. The concept is referred to as destructuring and allows binding the values of a list directly to variables.

As of this writing, two solutions exist in the public domain. The first is the closed-source and unpublished system Stitch, which is inspired by Stitch and Haxl, the second framework in the book’s Haxl [20], is an embedded domain-specific-language (EDSL) written in Haskell. Haxl uses the concept of applicative functors to introduce concurrency in programs and to batch I/O requests at runtime. Another existing framework is the closed-source and unpublished system Stitch, which is Twitter’s solution written in Scala and influenced by Haxl. Inspired by Stitch and Haxl, the second framework in the

**3.2 Concurrent I/O**

We introduce concurrency for the function in two steps. First, we translate it into a dataflow graph with concurrent I/O calls. Then we define a translation that also unblocks the rest of the program.

The translation exploits the fact that a dataflow node can have multiple output ports while a function/combinator in our expression IR can only return a single value. Most programming languages support syntactic sugar to support multiple return values. The concept is referred to as destructuring and allows binding the values of a list directly to variables.
public domain is Muse [16, 17], written in Clojure. Muse implements a so-called free monad [27] to construct an abstract syntax tree (AST) of the program and interpret it at run-time. To enable the AST construction, a developer using Muse has to switch to a style of programming using functors and monads. While these are fundamental concepts in Haskell, this is not the case for Clojure. Hence, the compiler can not help the developer to ensure these concepts are used correctly, making Muse programs hard to write.

Both Haxl and Muse significantly improve the I/O of systems with language-level abstractions. However, in contrast to Ÿauhau, these abstractions still have to be used explicitly. Furthermore, Haxl and Muse are limited in terms of concurrency. They both don’t allow a concurrent execution of computation and I/O by unblocking computation when some I/O calls have returned.

On the side of dataflow, there exists a myriad of systems including prominent examples such as StreamIt [28], Flume [9] or StreamFlex [26]. Programs written for these engines do not perform extensive I/O. They usually comprise two I/O calls, one for retrieving the input data and another one for emitting the results. Compiler research hence focuses primarily on optimizing the dataflow graph to improve parallel execution [14]. In contrast, our focus relies on enabling efficient I/O for programs that contain a substantial amount of I/O calls spread across the dataflow graph. Our approach is intertwined with the expression IR, which contains additional semantic information. Thus, leveraging this additional information allows us to apply provably semantic-preserving transformations in the presence of nested control flow structures. To the best of our knowledge, this is unique to Ÿauhau.

5 Evaluation

To evaluate Ÿauhau in terms of I/O efficiency and compare it against Muse and Haxl, we implemented Ÿauhau by defining a macro in Clojure. The Ÿauhau macro takes code as its input that adheres to the basic forms of the Clojure language which is also expression-based. Figure 9 gives a brief overview of the language constructs. For example, the function that builds the main pane of the blog would be written as follows:

```clojure
(fn mainPane []
  (render (map #(io (reqPostContent % src1))
    (last-n 10)
    (map #(io (reqPostInfo % src2))
      (io (reqPostIds src3)))))))
```

In order to abstract over the types of the requests, we prefix (foreign) functions with req that create a certain type of request. For example, reqPostInfo takes the identifier of a post as an argument and the reference to a source (src2) to generate the corresponding request.

As a first evaluation, we implemented the blog example using Clojure, and ported it to Ÿauhau. Table 1 shows the averages and estimated standard deviations over 20 runs of this example in four variants: the baseline sequential Clojure version (seq), a Ÿauhau version without applying any transformation (base), only with the batching transformation (batch), and with both, batching and concurrency (full). We see that the overhead introduced through Ÿauhau is within the standard deviation of the experiment, while the transformations significantly improved I/O efficiency. These results affirm the benefits of batching shown in Figure 2.

Although the simple blog example already shows the benefits from batching, we expect Ÿauhau’s rewrites to shine in real-world applications where dozens of microservices interact in complex ways. Writing such microservice-based systems and obtaining real-world data is a difficult undertaking on its own, into which large companies invest person-decades. Thus, the individual services are usually intellectual property and not on the public domain. This is a general problem that extends beyond this work and hinders research in this area.

5.1 Microservice-like Benchmarks

Due to missing public domain benchmarks, we use a framework for generating synthetic microservices [15]. It generates programs that aim to resemble the typical structure of service-based applications. To build such benchmark programs, the tool generates source code out of random graphs. In this way, it can serialize the graph into comparable code in different languages.

The random graphs are based on the concept of Level Graphs, which are directed acyclic graphs where every node is annotated with an integer level. These graphs are useful to capture the structure of computation at a high level of abstraction. We use l(v) to denote the level of the node v. A level graph can only contain an edge (v, w) if l(v) > l(w). Levels aim to capture the essence of data locality in a program. The basic concept behind the random level graph generation, thus, lies in having non-uniform, independent probabilities of obtaining an edge in the graph. In our benchmarks,

![Figure 9. Mapping the terms of the Clojure-based Ÿauhau language to our expression IR.](https://tud-ccc.github.io/yauhau-doc/)

<table>
<thead>
<tr>
<th>Version</th>
<th>seq</th>
<th>base</th>
<th>batch</th>
<th>full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time [ms]</td>
<td>275 ± 25</td>
<td>292 ± 19</td>
<td>79 ± 21</td>
<td>66.5 ± 5.2</td>
</tr>
</tbody>
</table>

Table 1. Execution times of the Blog Example

3https://tud-ccc.github.io/yauhau-doc/
a random level graph has an edge \((v, w)\) with probability \(2^{-(v-1)(w-1)} < 1\).

To generate code, nodes are annotated with a label for the type of code they will represent. These labels are also assigned independently at random. We use four different annotation types: `compute`, `req->io` (to simulate an I/O operation), `map`, and `v.alt`. The first two generate code that simulates external function calls, involving computation or I/O and foreign function calls. Subfunction/map nodes generate a call to a map over a local function, for which a new (small) random graph is generated in turn.

In order to generate code from a code-annotated Level Graph, we only need to traverse the nodes and generate functions with dependencies in accordance to the graph. If there are subfunctions labeled as such in the graph, we need to do this recursively for all corresponding subgraphs as well. The example of Figure 10 can be converted into the following Clojure code, omitting the code generated for subfun-3.

```
(let [x4 (req->io "source" 100) x5 (compute 100)
      x6 (req->io "source" 100) x7 (compute 100)]
  (let [x3 (subfun-3 x4 x5 x7)]
    (let [x1 (compute 100 x4)
          x2 (req->io "source" 100 x3)]
      (req->io "source" 100 x1 x2 x6))))
```

Generating Haskell code for Haxl is in fact very similar.

### 5.2 Experimental Setup

We designed three experiments to measure different aspects of Yauhau and compare to state-of-the-art frameworks. The size of the generated code and the included number of I/O calls is on par with numbers from Facebook published in [20]. We performed every measurement 20 times with different (fixed) random seeds and report the averages.

<table>
<thead>
<tr>
<th>exp.</th>
<th>indep. var.</th>
<th>dep. var.</th>
<th># lvl.</th>
<th>pr. subf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td># I/O calls</td>
<td>1-20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>conc.</td>
<td>latency</td>
<td>1-20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>modular</td>
<td>pr. subf.</td>
<td># I/O calls</td>
<td>20</td>
<td>0-0.4</td>
</tr>
</tbody>
</table>

The first experiment, **baseline**, shows a general comparison of the batching properties of Yauhau, Haxl and Muse, comparing to an unoptimized sequential execution (seq). For this, as the independent variable (indep. var.) we change the number of levels in a graph (# lvl.), between 1 and 20, and for each number of levels we look at average number of I/O calls (# I/O calls) as the dependent variable (dep. var.).

The second experiment, **concurrent I/O**, is similar to the baseline comparison, with a crucial difference in the structure of the graphs. We add an independent I/O operation that directly connects to the root of our level graphs. This is meant to simulate an I/O call that has a significant latency (of 1000 ms), several times slower (at least 3) than the other I/O calls in the program. In a production system, the cause can be the retrieval of a large data volume, a request that gets blocked at an overloaded data source or communication that suffers network delays. To measure the effects of this latency-intensive fetch, instead of the total number of I/O calls, we report the total latency for each execution.

Finally, the third experiment, **modular**, aims to measure the effect of having function calls in the execution. We fix the number of levels (20). Then, we generate the same graph by using the same seed, while increasing the probability for a node to become a function call when being annotated for code generation (pr. subf.). This isolates the effect of subfunction calls on batching efficiency. We do this for 10 different random seeds and report the averages.

Table 2 summarizes these experiments. Our experiments ran on a machine with an Intel i7-3770K quad-core CPU, running Ubuntu 16.04, GHC 8.0.1, Java 1.8 and Clojure 1.8.

### 5.3 Results

Figure 11 shows the baseline comparison, see Table 2. The plot shows that batching in Yauhau is superior to Haxl and Muse because our rewrites manage to batch across levels in the code. In this set of experiments, Yauhau achieves an average performance improvement of 21% over the other frameworks for programs with more than a single level. To maximize batching, Muse and Haxl require the developer to optimally reorder the code, ensuring the I/O calls that can be batched in one round are in the same applicative-functor block. This essentially contradicts the very goal of Haxl and Muse, namely, relieving the developer from having to worry about efficiency and instead focus on functionality.

With Version 8, the Haskell compiler GHC introduced a feature called applicative-do [21], which allows developers...
to write applicative functions in do-notation. This should, in principle, provide an ideal execution order for usage of applicative functor code. We tested different variants of code for Haxl. As expected, applicative-do code produced the exact same results as code written with explicit applicative functors, and better than the variants using a monadic-do. For this reason, we report the results only as "haxl", ignoring the worse results obtained from monadic-do code.

Figure 12 compares Haxl to Ÿauhau with and without support for concurrent I/O, showing how it can be indeed very beneficial. To make the comparison fair, we add asynchronous I/O support to Haxl using the async package. It internally uses forkIO to fork threads and execute the requests in a round concurrently, effectively putting the fork back into Haxl. After retrieving all responses, the program continues its execution. On the other hand, the non-strict unbatch dataflow node from Ÿauhau’s concurrent I/O executes the slow I/O operation in parallel to the computation in the rest of the graph. In particular at level 7, the plot starts to depict the full 1000 ms latency of the slow service as the difference between of Ÿauhau and Ÿauhau (conc I/O). This is the case when there are enough I/O rounds to displace the slow data source as the latency bottleneck.

Finally, the results of the modular experiment can be seen in Figure 13. It clearly shows that subfunctions have an effect on efficiency in the other frameworks, but none in Ÿauhau. The plot shows that the more modular the program, the less efficient the other frameworks become in terms of batching I/O requests. This is unfortunate because introducing a more modular structure into the program is one way to simplify complex applicative expressions. This is not only true for the explicitly applicative program versions but also for the implicit applicative do-desugaring in GHC 8. Since Ÿauhau flattens the graph, including the graphs for subfunctions, it avoids these problems through the dataflow execution model. Thus, it is also efficient with programs written in a modular fashion using subfunctions. Note that we did not investigate the scalability of this flattening for very large programs. With the moderate size of microservices we do not expect this to be a problem. In case it is, devising a partial flattening structure favoring I/O calls would be straightforward.

6 Conclusion and Future Work

Today’s internet infrastructures use a microservice-based design to gain maximum flexibility at the cost of heavy I/O communication. In this paper, we argued that a compiler can provide the necessary I/O optimizations while still allowing the developer to write concise code. To support this claim, we presented our compiler framework Ÿauhau that uses two intermediate representations. The first is an expression IR that allows to define semantic preserving transformations to reduce the number of I/O calls. The second is a dataflow IR which is concurrent by design and allows to unblock computation from I/O. We implemented Ÿauhau on the JVM in Clojure and show that it can speedup the latency of even simple services, e.g., for constructing a web blog, by almost 4x. To compare against state-of-the-art approaches, we used a microservice benchmarking tool to generate more complex code. For the microservices that we generated in the benchmark Ÿauhau performs 21% less I/O than runtime approaches such as Haxl and Muse.

Ÿauhau prioritizes batching over concurrency. This might be the best default solution, but it is not clear it is always ideal. In future work we plan to address this trade-off. Furthermore, our technique for batching I/O is only sound for independent calls to the same data source. We plan to investigate the possibility of batching multiple calls to the same data source, i.e., effectively sending part of the computation to a different service, and not only a simple I/O request.

Acknowledgments

The authors thank the anonymous reviewers and the shepherd Yun Liang for their help to improve the paper. This work was supported in part by the German Research Foundation (DFG) within the Collaborative Research Center HAEC and the Center for Advancing Electronics Dresden (cfaed).

References
