Ohua: Implicit Dataflow Programming for Concurrent Systems

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

Christof Fetzer
Systems Engineering Group
Technische Universität Dresden
Dresden, Germany
christof.fetzer@tu-dresden.de

Pascal Felber
Complex Systems Group
Université de Neuchâtel
Neuchâtel, Switzerland
pascal.felber@unine.ch

ABSTRACT

Concurrent programming has always been a challenging task best left to expert developers. Yet, with the advent of multi-core systems, programs have to explicitly deal with multi-threading to fully exploit the parallel processing capabilities of the underlying hardware. There has been much research over the last decade on abstractions to develop concurrent code that is both safe and efficient, e.g., using message passing, transactional memory, or event-based programming. In this paper, we focus on the dataflow approach as a way to design scalable concurrent applications. We propose a new dataflow engine and programming framework, named Ohua, that supports *implicit* parallelism. Ohua marries object-oriented and functional languages: functionality developed in Java can be composed with algorithms in Clojure. This allows us to use different programming styles for the tasks each language is best adapted for. The actual dataflow graphs are automatically derived from the Clojure code. We show that Ohua is not only powerful and easy to use for the programmer, but also produces applications that scale remarkably well: comparative evaluation indicates that a simple web server developed with Ohua outperforms the highly-optimized Jetty server in terms of throughput while being competitive in terms of latency. We also evaluate the impact on energy consumption to validate previous studies indicating that dataflow and message passing can be more energy-efficient than concurrency control based on shared-memory synchronization.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages, Dataflow languages*

1. INTRODUCTION

PPPJ '15, September 08 - 11, 2015, Melbourne, FL, USA
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-3712-0/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2807426.2807431

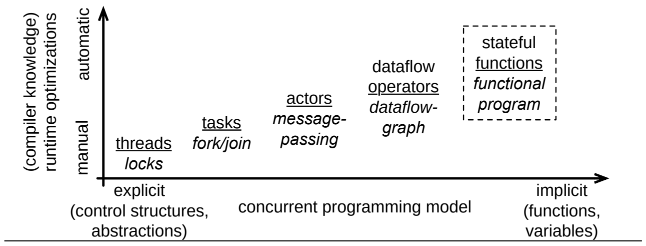


Figure 1: A taxonomy of concurrent programming approaches.

Context and motivations.

The radical switch from ever-faster processors to multi-core architectures one decade ago has had a major impact on developers, as they suddenly had to learn developing *concurrent* applications that are able to harness these new parallel processing capabilities. Doing so in a way that is both safe and efficient is, however, a challenging task. While pessimistic concurrency control strategies like coarse-grained locks are simple to use and easy to prove correct, they severely limit scalability as large portions of the code are serialized. In contrast, fine-grained locking or lock-free algorithms provide better potential for parallelism but are complex to master as one has to deal with concurrency hazards like race conditions, deadlocks, livelocks, or priority inversion. It is therefore desirable to shield as much as possible the developer from the intrinsic challenges of concurrency by providing adequate abstractions in programming languages.

We can distinguish two types of programming language support to create parallelism: *control structures* and *abstractions*. Imperative programming languages such as C, C++, Java, and even newer functional languages like Manticore [18], expose control structures, e.g., threads and processes, to allow the programmer to explicitly schedule and execute code in parallel. In addition, the concepts of locks and barriers provide mechanisms to orchestrate the threaded execution: the programmer explicitly forks threads and controls their interactions with shared data. On the other hand, programming languages such as Erlang [4], Scala [21], MultiLisp [22] or Cilk [8] do not expose full control of parallel execution to the developer but rather provide abstractions such as actors, message passing, futures, tasks, parallel loops or even dataflow graphs to mark code regions that can potentially execute in parallel. Note that several popular programming languages like C, C++, Java can also support such abstractions thanks to extensions such as OpenMP [11] or Akka [24]. Especially on different hardware architectures such as special processors designed for massive parallelism [44], GPUs and FPGAs, the dataflow programming and execution model has become state-of-the-art. In languages such as StreamIt [45] and Intel's Concurrent Collections (CnC) [9], the program is constructed out of a graph of operators. It is left to the compiler and runtime system to exploit these hints in a parallel execution context.

Both types provide appropriate support for developing scalable concurrent programs leveraging the ever increasing multi-/many-core nature of processor architectures, and there are obviously arguments for using either: better control of parallelism vs. safety and simplicity of use. While abstractions avoid the root cause of most concurrency problems, namely shared memory accesses, most of them still expose the explicit constructs of threads, processes, and locks. Additionally, abstractions require developers to structure their programs in a specific, often unnatural, way to enable parallel execution, e.g., Cilk tasks or Scala/Akka actors. Developers are now left with the decision of which approach to use, which often brings more confusion than benefits [36, 43]. Dataflow graphs and operators similarly diverge drastically from the normal programming paradigm. The explicit declarative nature of the dataflow graph construction does not scale to large and complex programs and has therefore prevented this paradigm to be used in general purpose programming. Figure 1 provides an overview of the current approaches to concurrent programming along the ultimate goal: scalability (of the programming model to construct complex systems). A solution to concurrent and parallel programming must fulfil two criteria. It is free of additions and uses solely functions and variables as in a sequential program to release the programmer from the burden to worry about concurrency and even parallelism and therewith enable faster construction of larger systems. On the other hand, it exposes the concurrent nature inside the program to the compiler to enable automatic parallel execution, (scheduling) optimizations and adaptations to newly evolving hardware without changing the program.

Our approach.

In this paper, we argue for an approach based on pure implicitness requiring neither control structures nor abstractions. We propose a new programming model based on *stateful functions* and the implicit construction of a dataflow graph that does not only abstract parallelism, but also promotes reuse by structural decomposition and readability by the differentiation of *algorithm* and *functionality*. Unlike with message passing and threads, this decomposition better matches the natural flow of the program. As requesting the programmer to explicitly construct a dataflow graph would go against our implicitness objectives and limit scalability of the programming model, we instead derive the graph implicitly from a regular functional program. The key idea is that, in functional programming, global state is omitted and the structure of a program closely resembles a dataflow graph, with a set of functions that may maintain local state.

In Ohua, our implementation of this programming model, the actual functionality of the program can be implemented in an imperative form in Java to tie state to functions within objects, while Clojure's functional programming paradigm is used to express the algorithm, and hence the dataflow graph, in a natural way. This combination of object-oriented and functional programming enables us to get the best of both worlds: Java for developing sophisticated functionality using a comprehensive set of libraries and legacy code, and Clojure for the specification of algorithms from which dataflow graphs are derived.

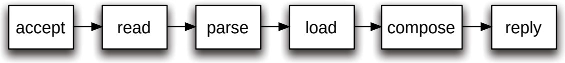


Figure 2: A simple web server.

As an example, consider the pipeline for a simple web server depicted in Figure 2. With our Ohua dataflow programming model and execution engine, the algorithm is composed in a functional programming style in Listing 1.

Listing 1: Ohua-style HTTP server algorithm in Clojure

```
(ohua :import [com.web.server])

; classic Lisp style
(ohua
 (reply (compose (load (parse (read (accept 80))))))

; or using Clojure's threading macro to improve
        readability (which is transformed at compile-time
         into the above)
(ohua
 (-> 80 accept read parse load compose reply))
```

The actual functionality for accepting incoming connections, reading the data from the socket, parsing the request (see Listing 2), loading the requested file from disk, and composing and sending the response is implemented as methods in Java classes.

Listing 2: Ohua-style parse function implementation in Java

```
public class HTTPRequestParser {
  // a pre-compiled regex as the associated state
 private Pattern _p = Pattern.compile(
   "(GET|PUT)\\s([\\.\\/\w]+)\\s(HTTP/1.[0|1])");

 @Function public String[] parse(String header) {
   Matcher matcher = _p.matcher(header).find();
   return new String[] {
     matcher.group(1), // request type
     matcher.group(2), // resource reference
     matcher.group(3) }; // HTTP version
}
```

As a final step, Ohua links both specifications and implicitly executes the program in a pipeline parallel fashion.

As also argued in [37], there are benefits in mixing functional and imperative programming for concurrent programs. Languages like Scala supporting both paradigms do provide neither safety nor guidance on when to use imperative or functional programming. We therefore strongly believe that a clear separation between the two languages also helps the developer to differentiate between the algorithm and its functionality. The conciseness of the functional language is instrumental in presenting a clear perspective of the program flow and making it easier to understand, reason about, and extend. On the other hand, the object-oriented paradigm allows the developer to share and refine functionality easily across different implementations. More importantly it provides the proper programming model to encapsulate state.

Contributions and roadmap.

After an introduction into dataflow programming in Section 2, we present the core concepts of our *stateful functional programming (SFP)* model in Section 3 and its implementation in Ohua. It encompasses a compiler that transparently derives the dataflow graph from a declarative program written in a functional language (Clojure), while the actual operations performed by the application are implemented in an object-oriented model (Java); and a runtime engine and programming framework that builds upon this implicit approach to provide seamless support for concurrent and parallel execution (Section 4). We use the semi-complex real-world example of a concurrent web server to exemplify our concepts clearly. The web server building blocks can be found at the heart of most distributed systems. Apart from concurrency, I/O is at the core of every server design. However, the cost is yet another programming model that is asynchronous and breaks control flow of the server into many cluttered event dispatches [17]. Combining this I/O programming model with any of the concurrent programming abstractions described above is even more challenging. Therefore, we provide a comparative evaluation with the Jetty web server that ships with two implementations, for synchronous (blocking - BIO) and asynchronous (non-blocking - NIO) I/O models respectively, and uses explicit concurrency control managed by the programmer using threads. We study the scalability of our simple Ohua-based web server and Jetty in terms of performance and energy-efficiency in Section 5. We show that the flexibility of our runtime framework, which allows to control concurrency as well as the I/O model without changing the code, achieves better hardware adaptation than barely increasing a thread count. We found that the Ohua-based web server outperforms Jetty in throughput by as much as 50 % for BIO and 25 % for NIO. This gives reason to believe that NIO-based web servers on the JVM do not scale better (at least for web server design) than implementations where asynchronous I/O is realized with threads that block on I/O calls. We discuss related work in Section 6 and give an outlook on future work (Section 7) before we conclude in Section 8.

2. DATAFLOW PROGRAMMING

In flow-based programming (FBP) [34, 14], an algorithm is described in a directed *dataflow graph* where the edges are referred to as *arcs* and vertices as *operators*. Data travels in small *packets* in FIFO order through the arcs. An operator defines one or more input and output ports. Each input port is the target of an arc while each output port is the source of an arc. An operator continuously retrieves data one packet at a time from its input ports and emits (intermediate) results to its output ports. Dataflow programming therefore differs significantly from imperative programming, which relies on control flow. While an imperative program can be translated into a dataflow, as explained in [6], dataflow execution is naturally tied to functional programming.

In classical dataflow [5, 15], operators are fine-grained stateless instructions. In contrast, FBP operators are small blocks of functional sequential code that are allowed to keep state. This programming model is similar to message passing with actors, which recently gained momentum with languages such as Scala [21]. Unlike these, however, FBP cleanly differentiates between functionality, such as parsing a request, and the algorithm, such as serving web pages. An operator makes no assumptions nor possesses any knowledge about its upstream (preceding) or downstream (succeeding) neighbors. Therewith, operators are context-free and highly reusable. Finally, an FBP program defines the communication between operators, which translates into data dependencies, via arcs at compile-time rather than at runtime. This approach avoids concurrent state access problems and allows for an implicit parallel race-free program execution.

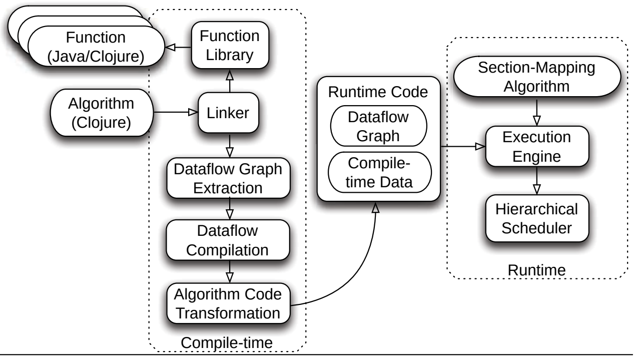


Figure 3: Overview of Ohua.

A strong benefit of this program representation is the simple graph structure, which enables reasoning about parallelism, concurrency, and synchronization without the need to analyze the entire source code. Ohua exploits this property in order to move decisions about parallel execution from compile-time to deployment and even runtime. Although the programming model allows for coarse-grained implicit parallelism with a minimum granularity of an operator, it does not impose any constraints on the mapping of operators to execution units.

(Data)flow-based programming can be found at the core of most advanced data processing systems. In such systems, concurrent processing is key for scalability, and the dataflow approach provides seamless support for exploiting multi-core and parallel architectures. For example, IBM's DataStage [28] products are state-of-the-art systems for data integration that are purely based on the FBP concepts. Many algorithms for database systems [16] and data stream processing [10] are also expressed as directed graph structures. The declarative design of network protocols, as for instance in Overlog [32], are dataflow graphs by definition and even the construction of some highly scalable web servers [48] finds its roots in FBP principles. The data dependency graphs underlying the dataflow programming model can be found as compile-time program representation in most parallel languages, such as Manticore [2] and Cilk [8]. These graphs contain the necessary information to derive runtime task execution constraints and avoid data races; they are therefore instrumental in designing scalable concurrent programs for multi-/many-core architectures.

3. STATEFUL FUNCTIONAL PROGRAMMING IN Ohua

The approach of Ohua is to enable parallel execution while retaining the structure and style of a sequential program. In order to achieve this goal, Ohua marries functional and object-oriented programming under the clear structure of FBP. While functionality is provided in an object-oriented

Listing 3: StreamFlex filter

```
class FileLoad extends Filter {
  Channel<String> in, out;
  void work() {
    // explicit channel control
    String resource = in.take();
    String contents = null;
    // load file data from disk (
        omitted)
    out.put(contents);}}
```

Listing 4: SF implemented in Java

```
class FileLoad {
  @Function
  String load(String resource) {
    String content = null;
    // load file data from disk (
        omitted)
    return content;}}
```

Listing 5: SF implemented in Clojure.

```
(ns com.server.FileLoad
  (:gen−class :methods
    [[^{Function} load [String]
       String]]))

(defn −load [this resource]
  (slurp resource))
```

Figure 4: StreamFlex dataflow operators/filters vs. Ohua functions implemented in Java and Clojure for file loading.

form in Java, the algorithm is defined in the Clojure functional language. Clojure is a natural fit in Ohua because it is JVM-based and it exposes a powerful macro system to extend the language [25]. Figure 3 depicts the internals of Ohua. Functions and algorithms are provided by the developer. In the first step, functions are linked to the algorithm. Thereafter, Ohua performs dataflow detection on the functional algorithm. The extracted dataflow graph is compiled and finally transformed into its runtime representation.

All of these steps are executed in the `ohua` macro during Clojure compilation of the algorithm. The macro generates code that preserves compile-time insights such as restrictions on parallelism (see Section 4.2) and executes the dataflow graph in the execution engine. The *section-mapping* algorithm is the extension point of our runtime framework related to parallelism. It is responsible for breaking the dataflow graph into execution units. This shifts parallelism concerns from compile-time to runtime and thereby provides the opportunity to adapt programs to different deployment contexts. We present more details on the execution model in Section 4, after first focusing on the programming model and compilation aspects.

3.1 From operators to stateful functions

In SFP each operator encapsulates a single function tagged with the `@Function` annotation.[1] The defining class can be thought of as a tuple consisting of the function and the associated operator state as such we refer to them as *stateful functions*[2]. In the following, we highlight the difference in the programming model for typical FBP operators of streaming systems like StreamIt [45]. We use StreamFlex [41] for comparison as it is also written in Java. As depicted in Listing 3 of Figure 4, a typical operator design encompasses explicit control on the abstraction of channels (or arcs) for receiving input and emitting results. The developer has to deal with problems such as unavailability of input data, buffering of partial input and fully loaded outgoing channels. For more advanced operators this requires the implementation of a state machine. Instead SFP resembles the natural implementation of a function. This code may either be implemented in Java or in Clojure (Listings 4 and 5 of Figure 4). The latter uses the `:gen-class` library to generate a Java class from the functional code. It not only allows to define stateless functions in a functional manner but also enables to leverage Clojure libraries naturally.

[1]Note that there may be many other functions in the class, but for reasons of clarity in this paper only one must be tagged with this annotation.

[2]For the remainder of the paper, we use the term operator for nodes in the dataflow graph for historical reasons and mean stateful functions in terms of the programming model.

In a strict sense, functions in SFP resemble the idea of lazy functional state threads [29] based on the concept of monads. Such threads represent state transformers that are never executed concurrently (on the same state). As such they can be described as a pure function on the state itself. State transformers allow programmers to describe a stateful computation in the context of a functional program, which is normally stateless. The rationale behind the concept of a state transformer is to encapsulate the state in such a way that the computation appears stateless to all other components/functions in the system. This closely maps to FBP's notion of context-insensitivity for operators that we mentioned above. Although the Java type system does not support strong encapsulation of state, i.e., state can potentially leak to the "outside world", Ohua ensures that functions never return any reference to their state. This is achieved by static byte code analysis of the function implementation at compile-time to detect escaping references. A detailed introduction of the related concepts and algorithms is outside the scope of this paper. Therewith, stateful functions provide strong encapsulation and allow reasoning about an algorithm in a purely functional manner, with all its benefits such as referential transparency and the potential to execute functions in parallel.

3.2 Algorithms

The role of algorithms in SFP is to provide an untangled view on the system, free of implementation details. Listings 6 and 7 in Figure 5 compare the explicit construction of the web server flow graph in StreamFlex with the algorithm declaration in Ohua. This explicit construction clearly limits the scalability of the programming model while the construction of an algorithm is a natural fit for any developer. The resulting dataflow graphs as presented in Figure 5 show that StreamFlex solely knows arcs, i.e. compound dependencies among operators, while the SFP algorithm defines fine-grained data dependencies. These enable the compiler to clearly understand the dataflow on the algorithm level to increase safety and optimize the parallel execution. Instead the compiler in Ohua derives this knowledge automatically. Note that algorithms in turn can be composed out of other algorithms and therefore Ohua also supports defining algorithms as functions. This allows for a clear structure as known from any other programming language.

Algorithms are also type-agnostic. While types are instrumental for program verification at compile-time, they also add to code verbosity and complexity. SFP algorithms are intentionally written in a non-typed manner, yet without sacrificing the benefits of a statically typed compilation. All

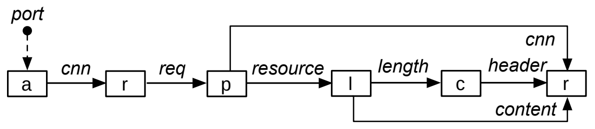
Listing 6: StreamFlex graphs

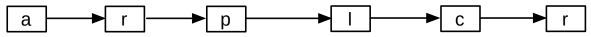
```
public class WebServer extends StreamFlexGraph {
  Filter a, r, p, l, c, rep;
  public WebServer() {
    // explicit dataflow graph construction
    a = makeFilter(Accept.class);
    r = makeFilter(Read.class);
    p = makeFilter(Parse.class);
    l = makeFilter(Load.class);
    c = makeFilter(Compose.class);
    rep = makeFilter(Reply.class);
    connect(a, r);
    connect(r, p);
    connect(p, l);
    connect(l, c);
    connect(c, rep);
    validate();}

  public void start() {
    new Synthetizer(accept).start();
    super.start();}}
```

Listing 7: Ohua algorithms

```
(defn start [port]
 (ohua
  ; most "explicit"/fine-grained data dependency
       matching
  (let [[cnn req] (read (accept port))]
   (let [[_ resource _] (parse req)]
    (let [[content length] (load resource)]
     (reply cnn (compose length) content))))))
```

Derived dataflow graph from the Ohua algorithm in Listing 7.





The StreamFlex dataflow graph from Listing 6.

Figure 5: StreamFlex dataflow graph construction vs. Ohua algorithm design.

type information is kept with the functionality, i.e., the Java code implementing the stateful functions. Hence, although type information is not present in the Clojure algorithms, it can be exploited on the Java side to support type-based compilation of the dataflow graph in future work. Algorithms are constructed from the core Lisp concepts, excluding mechanisms for concurrency control and parallelism such as agents, software transactional memory, or parallel maps. We see SFP as an addition to these approaches and provide more details for their integration in Section 4. The graph construction happens solely on the basis of the Clojure algorithm while its operational semantics strictly adhere to those of normal Clojure functions (for any single data packet).

3.3 Dataflow graph extraction

The Ohua compiler follows the return values and references to locals, and respectively translates them into data dependencies and arcs of the dataflow graph. Algorithm 1 gives a high-level overview of our dataflow graph extraction algorithm. Input to the algorithm is code written in Clojure, which consists of stateful function invocations, control constructs such as conditionals or loops, environment variables such as the server port 80 in Listing 1, and finally a set of explicit data dependencies represented by locals. The result(s) of a function can either be direct (implicit) input to another function or assigned to a so-called *local* in Clojure. Locals differ from variables in that they are immutable. This is also true for locals in a Ohua algorithm for any single data packet. Locals allow us to decompose the result of a function and input portions of it to different functions. This process, called *destructuring*, provides explicit fine-grained control over the data dependencies. For example, Listing 7 provides the most explicit implementation of the server algorithm, which translates into the dataflow graph of Figure 5.

Clojure is a homoiconic language and therewith represents code just as its internal data structures, e.g., lists, vectors, symbols. As such a Clojure program resembles a tree structure. In Figure 6 we depict a variant of our simple web server algorithm that destructures the results of `parse` to

Algorithm 1: Dataflow Graph Extraction Algorithm

```
Data: $\mathbb{A} := (\mathbb{F}, C := \{if, loop, recur\}, V, D)$ a Clojure
      algorithm represented as a tree composed of
      functions that are either linked stateful functions $\mathbb{F}$ or
      represent control structures $\mathbb{C}$, a set of environment
      variables $V$ and a set of explicit dependencies $D$.
$z$ := zipper($\mathbb{A}$) ;                         // functional zipper [26]
while $z$ := $z$.next() do                             // depth first traversal
    $f$ := $z$.node();
    if $z$.up() $\neq \emptyset$ then
        // For simplicity we assume the simple case where
           no control structure is the root.
        create-operator($f$);
    else
        $t$ := $z$.up().leftmost().node();
        if $f \in \mathbb{F}$ then
            create-operator($f$);
            create-dependency($f$.id, $t$.id);
        else if $f$ = if then
            $s$ := create-operator('cond');
            $f$.id := $s$.id;
            $m$ := create-operator('merge');
            Turn condition into executable and attach to $s$;
            create-dependency($m$.id, $t$.id);
        else if $f$ = loop then
            create-operator('merge');
            create-dependency($f$.id, $t$.id);
        else if $f$ = recur then
            $l$ := Find the next loop statement in $z$.path;
            create-dependency($f$.id, $l$.id);
        else if $f \in \mathbb{V}$ then
            register-env-var($f$.node());        // see Section 4.2
        else if $f \in \mathbb{D}$ then
            create-dependency($f$.origin.id, $t$.id);
        end
    end
end
```

assign them to locals `socket` and `res-ref`. The former is passed to `send` while the reference to the requested resource is input to `load`. Before the graph extraction algorithm is executed, our compiler does a first pass over the code and

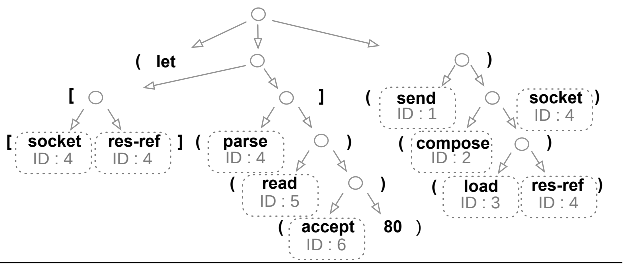


Figure 6: Tree representation of the server algorithm in Clojure.

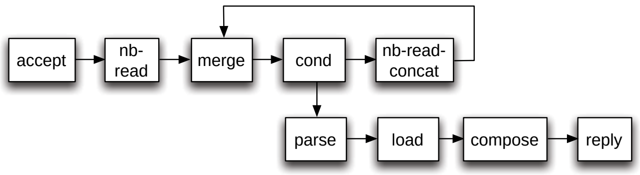


Figure 7: `recur` statements translate into feedback-edges.

assigns each function a unique identifier. Locals are assigned the same identifier as the function whose results they bind.

For simplicity Algorithm 1 omits advanced concepts of formal and actual schema matching, but focuses on the central idea of the tree traversal and creation of dependencies from lower-level nodes to their direct parent (Lines 9–11). If the algorithm encounters a local, it creates a data dependency from the local's origin to the function it is input to (Lines 26–27). Environment variables are registered as static input of a function invocation and are important when the argument list for the call is created (Lines 24–25).

Conditionals are special cases of functions that result in the creation of two functions (Lines 12–17). The `cond` function evaluates the condition and dispatches the current packet to either one of its two outputs representing the `if` and `else` branches. Note the translation of control flow into dataflow at this point. Both branches are represented in the code as input to the `if`, as seen in Listing 8, but we insert here an additional `merge` operator as target of the results from both branches. The algorithm then attaches the identifier of the `merge` to the `if` node in the code tree (Line 17).

Finally, loops in Clojure are composed by combining two special forms: `loop` and `recur`. The `loop` function takes a binding vector with input to the loop and the code of the loop body to be executed. Inside this loop body, a new iteration of the loop is initiated once a `recur` invocation was reached. As a result, loops always incorporate an additional `if` invocation to implement the loop condition with the recursion and exit point. Our algorithm already supports conditions and therefore all that is required is another `merge` to handle newly entering packets (Lines 18–20) and iterations whenever a `recur` node is encountered (Lines 21–23). Listing 8 declares a version of the web server that implements a non-blocking read on the accepted connections via a loop.

Listing 8: Loop statement to implement non-blocking reads.

```
(ohua
 (reply (compose (load (parse
  (loop [[read-data cnn] (nb-read (accept 80))]
   ; stop on blank line
   (if (.endsWith read-data "\n\n")
      [read-data cnn]
      (recur (nb-read-concat cnn read-data))))))))
```

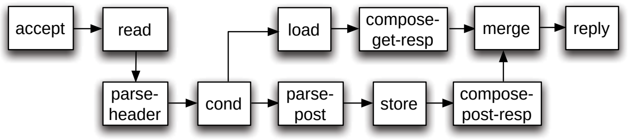


Figure 8: Data parallelism opportunity via conditionals.

Note the implementation of the non-blocking read functions in Listing 9. It uses inheritance to share the code for reading data and extending it by concatenating the already read data to the current data read.

Listing 9: Non-blocking read with two functions.

```
class NonBlockingRead {
  private ByteBuffer b = ByteBuffer.allocate(90);

  @Function
  Object[] nbRead(SocketChannel cnn) {
    cnn.configureBlocking(false);
    int count = cnn.read(b);
    b.flip();
    String data =
             Charset.defaultCharset().decode(b);
    return new Object[]{data, cnn};}}

class NBReadConcat extends NonBlockingRead {
  @Function
  Object[] nbReadConcat(SocketChannel cnn,
                        String read){
    return
       new Object[]{read + super.nbRead(cnn)[0],
                    cnn};}}
```

3.4 Linking

After the graph has been created, the function calls are linked to their respective targets across operators. Linking is implemented by importing the namespace where the functions to be used are defined (see the first line in Listing 1). Internally, the linker loads all Java classes and inspects them to look for functions with the `@Function` annotation. If a class defines such a function, a reference is stored under the signature of the tagged function in the *function library*. The compiler subsequently looks up functions in the function library when performing the dataflow compilation.

Listing 10: Conditional statement on the resource location

```
(ohua
 (let [[type req] (-> 80 accept read parse-header)]
  (if (= type "GET")
   (-> data load compose-get-re)
   (-> data parse-post store compose-post-re))
  reply))
```

3.5 Opportunities for data parallelism

While the dataflow programming model naturally maps to pipeline parallelism, the Clojure language also introduces data parallelism into the dataflow. There are two special forms in the Clojure language that implicitly introduce parallel branches into the graph: `if` and `let`. Listing 10 uses a condition on the request type to implement file storage as in restful services, i.e., using the HTTP POST and GET operations. The resulting dataflow graph is shown in Figure 8. The condition splits the dataflow into two branches,

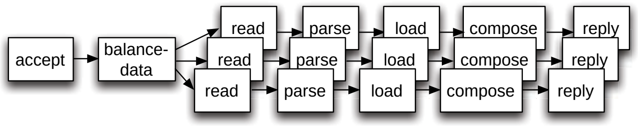


Figure 9: Opportunity for 3-way parallel request processing.

which can be executed in parallel for different requests. The `cond` operator evaluates the condition and turns the control flow decision into a dataflow decision. In the case of a `let`, forms inside the lexical context may not depend on data from each other and therefore resolve to parallel branches by the definition of Algorithm 1.

Data parallelism can also be created explicitly by dedicated operators that produce data parallelism without breaking the semantics of the algorithm. Listing 11 uses a macro (`||`) to create the opportunity to handle requests in a 3-way parallel fashion on the foundation of the `let` special form.

Listing 11: Load balancing via a simple macro

```
(ohua
 (let [cnn (accept 80)]
   (|| 3 cnn (-> read parse load compose reply)))

; macro expanded
(ohua
 (let [cnn (accept 80)]
   (let [[one two three] (balance-data cnn)]
     (-> one read parse load compose reply)
     (-> two read parse load compose reply)
     (-> three read parse load compose reply)))
```

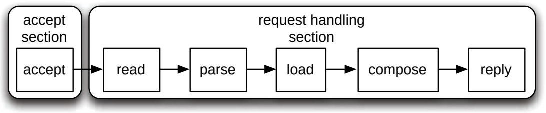
Supporting such parallelism implicitly is not trivial and is left as future work, as it requires a detailed understanding of the semantics of the pipeline. The resulting dataflow graph is presented in Figure 9. It uses a `balance-data` operator that dispatches the incoming independent requests among its outputs according to some predefined decision algorithm, e.g., round robin.

Note that parallelism is an orthogonal concern that is enabled by the algorithm but unleashed via the sections-mapping at runtime. For example, an `if` condition always creates an opportunity for data parallelism. It is, however, up to the section mapping to place the two branches onto different sections (see Section 4).

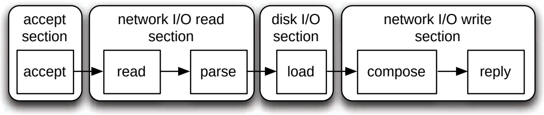
4. FLEXIBLE PARALLEL EXECUTION

The dataflow graphs considered in Ohua are static as they represent the implementation of an algorithm (coarse-grained parallelism) rather than the dynamic evolution of a data structure (fine-grained parallelism). This point is important especially when reasoning about the graph in terms of task granularity and scheduling. Our dataflow graphs are, however, not synchronous as I/O operations prevent us from predicting the runtime schedule [31].

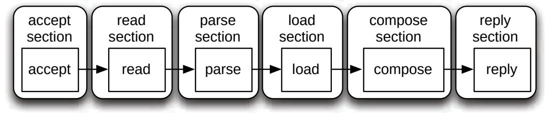
A dataflow graph is a representation that is easy to reason about even without detailed operator knowledge. The execution engine in Ohua bases its decisions solely on operator meta data such as I/O interactions or the number of outgoing and incoming arcs. The execution of the operator function is strict, i.e., the function is executed only if all slots in its formal schema are filled. Note that the `merge` operator is again an exception because it is executed as soon as some input is available among any of its incoming arcs.



(a) Decoupled accept handling and request processing via sections



(b) One I/O-operator per section mapping to decouple I/O



(c) Most fine granular one operator per section mapping

Figure 10: Three section mappings for the basic web server.

It would appear natural to define the granularity of parallelism on the basis of an operator. However, the goal of Ohua is to run computations over dataflow graphs that are potentially very large with hundreds of operators. Under these conditions the granularity of parallelism has been identified as a vital parameter as it must amortize the associated management costs [23]. The same work points out another often neglected aspect when addressing parallel execution: I/O and synchronized memory updates. Therefore, we require a flexible execution model that allows us to adjust the granularity based on deployment-specific properties such as the number of cores, the available RAM, or the attached storage. More specifically, current research on flexible runtime mappings for parallel execution focuses on extracting as much parallelism as possible out of pipelines in a loop body [42, 38]. In these approaches, the lowest degree of parallelism is given by the number of operators. Ohua targets larger graphs, potentially much larger then the number of cores available, and therefore requires the flexibility to reduce the degree of parallelism even below the number of operators in the graph. As such the model should separate this concern to adapt to different deployments without program change.

4.1 Parallelism granularity control

In order to provide this degree of flexibility, Ohua defines the concept of a *section* that spans one or multiple operators in the dataflow graph. At runtime a section maps to a task that gets executed on a thread by a scheduler. A section executes its operators cooperatively on that thread. Therefore, sections execute in parallel and operators concurrently.

Listing 12: 'Acpt-req' and '1 I/O op/sec' section mappings

```
'(["acc.*"] ["read.*" "par.*" "load.*" "com.*" "rep.*"])
'(["acc.*"] ["read.*" "par.*"] ["load.*"] ["com.*" "rep.*"])
```

As shown in Listing 12, developers can specify different section mappings via regular expressions at deployment, thereby changing the degree of parallelism in the execution of the dataflow graph. For example, the section mapping in Figure 10a decouples the tasks of accepting new connections from the processing of requests, while the mapping in Figure 10b provides a finer code granularity per section by decoupling all I/O operations of the dataflow graph. The most fine-granular mapping, depicted in Figure 10c, assigns each operator to its own section. Future work extends this

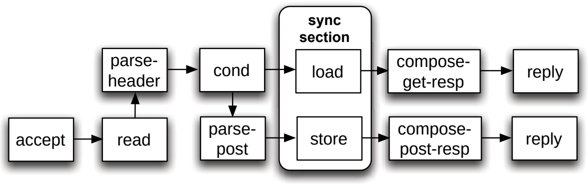


Figure 11: Resource access synchronization via sections.

concept to adapt the section mapping dynamically at runtime, for example to varying load situations or other runtime statistics. Ohua requires every operator of the graph to be mapped to exactly one section. That is, we exclude the parallel execution of an operator instance and the resulting concurrency on its state because it would violate the FBP model.

4.2 Integration into Clojure programs

Ohua is particularly adapted for pipeline parallel execution in the context of large complex systems. Yes, not every aspect of such a system needs to be executed in a parallel fashion. Ohua also integrates easily with normal Clojure programs, taking locals or variables as input and producing output like any function call.

Listing 13 is a variation of our extended web server that handles HTTP POST and GET requests.

Listing 13: A variable map as a cache in Ohua

```
1 (def registry (new java.util.HashMap))
2
3 (ohua
4  (let [[type req] (-> 80 accept read parse-header)]
5   (if (= type "GET")
6    (reply (compose-get-re (load req registry)))
7    (reply (compose-post-re (store (parse-post req)
8           registry)))))
```

Instead of storing and fetching the data to and from disk, it uses an in-memory `registry`. It takes a variable referencing a map as an input to two Ohua functions. The algorithm designer is in charge of assigning actuals in the surrounding closure to formals in the functions of the Ohua computation. We do not place any restrictions on this assignment other than defined by Clojure itself. The Ohua compiler detects that the `registry` variable is shared among the `load` (Line 7) and the `store` (Line 9) operators. While the mapping of operators to different sections provides a way to process requests in parallel, the converse is also true. Hence, the compiler can define the restriction for the section-mapping algorithm to place these operators onto the same section as in Figure 11. Thereby, it implicitly synchronizes access to the `registry`. Furthermore, we used a Java typed variable for the following reason. The compiler can inspect the shared data structure and place this restriction only if the type does not implement an interface from the `java.util.concurrent` package. This means that the compiler can adapt the synchronization mechanism automatically depending on the type. None of the implementation code would have to be changed. In a sense, Ohua solves the problem of providing synchronization implicitly rather than putting the burden onto the developer by introducing locks into the language. On the other hand, not all shared resources are visible to the compiler. Consider the same example but instead of sharing a memory reference both functions access the same external resource, e.g., a file. This may not be known to the programmer as it is a configuration parameter and hence highly deployment specific. Therefore, section mappings are only defined at deployment to adapt not only to hardware but to additional requirements; in this case to enforce synchronization even for resource accesses that are external to Ohua.

5. EVALUATION

Ohua targets coarse-grained parallelism but the granularity of tasks in typical benchmarks even for pipeline parallelism is fine-grained [7]. Therefore, to evaluate the efficiency and overhead of Ohua, we use our example of a simple web server, which represents an ubiquitous component in the days of web services and cloud computing, and we compare our implementation against the state-of-practice Jetty web server. We focus on scalability and observe both latency and throughput, which are the most important characteristics of a web server. Our evaluation encompasses different section mappings, different degrees on (data) parallel transformations, application-specific optimizations and a comparison of BIO and NIO-based web server design.

Our experiments simulate a high load situation where clients concurrently request a small file from the web server. Initially, 10,000 files of 512 bytes each are created. We chose a rather small file size to simulate a scenario that imposes a high load on the web server, similar to a news feed or aggregator. In our experiments, clients continuously request one of these files uniformly at random. This greatly eliminates the disk caching effect of the operating system. The delay between requests of a single client is 20 ms and, if not stated otherwise, we report the mean over an execution period of 100 s. Our test environment is a cluster of 20 nodes with 2 quad-core Intel Xeon E5405 CPUs (i.e., 8 cores) and 8 GB of RAM. The machines are interconnected via Gigabit Ethernet and their hard disks are attached via SATA-2. The operating system is Debian Linux (kernel 3.2.0) with Java 1.7 installed. One node in the cluster is used for the web server while the clients are spread evenly across the remaining 19 machines. We conduct our experiments until 1,200 concurrent clients, at which point the Java garbage collector (GC) starts becoming a bottleneck. Although we study the impact of the GC type on Jetty as well as Ohua to find the optimal configuration, a thorough investigation of GC bottlenecks for web server programs in Java is outside the scope of this evaluation. For our energy studies we had to use a different server machine in a different data center to get access to the energy counters of the processor. The machine composed of an Intel Core SandyBridge processor with a single socket, 2 cores per socket and 2 hardware threads per core, and 4 GB of RAM. The source code of Ohua[3] as well as the web server[4] along with instructions for running the benchmarks are freely available.

5.1 Pipeline parallelism

In order to find the optimal configuration for our Ohua-based web server (see Figure 2) with respect to the exploited parallelism and scheduling overhead, we proceed from a concurrent to a parallel execution. We start by investigating the impact of operator scheduling on the web server perfor-

[3]https://bitbucket.org/sertel/ohua
[4]https://bitbucket.org/sertel/ohua-server

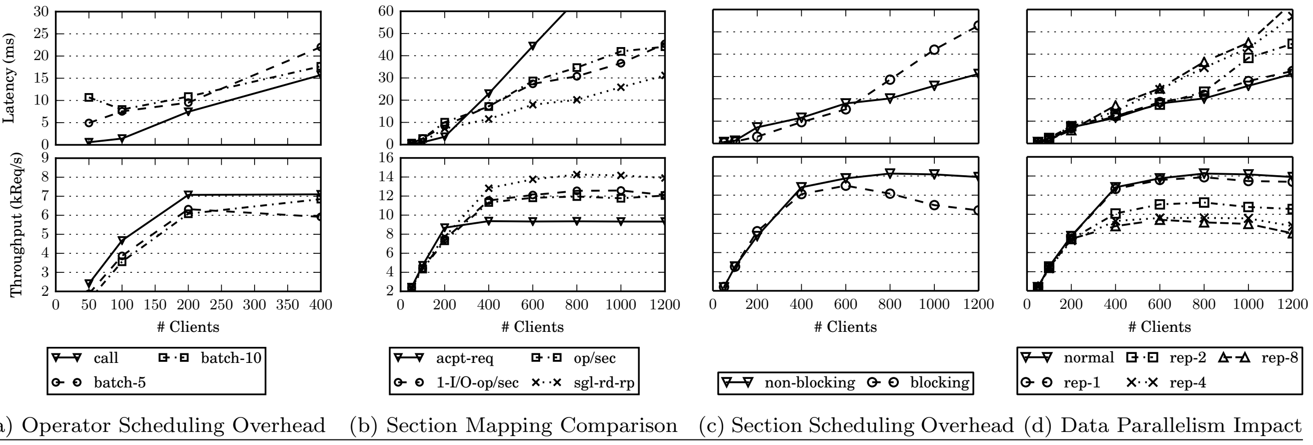


Figure 12: Latency and throughput measurements to find the optimal scheduling, section and parallelism configuration for our HTTP server.

mance. All functions of the algorithm are mapped to the same section to prevent any parallelism. We evaluate the performance by changing the size of the arcs in between the operators. Figure 12a shows the resulting latency and throughput graphs for sizes 1, 5, and 10. Ohua translates an arc size of 1 into a direct function call and therewith avoids operator scheduling. As appears clearly in the graphs, this results in improved latency and even better throughput. When increasing the load, however, the batch-oriented processing model catches up and exhibits similar performance characteristics. We report measurements only up the 400 concurrent clients because a higher load already forced the server to drop requests. Based on these results, we select the direct function call as the optimal configuration for operator execution among the sections in the rest of our evaluation.

Next, we investigate the impact of different section mappings. In addition to the three mappings defined in Figure 10c, we add the "sgl-rd-rp" (single-read-and-reply) mapping, which assigns each network I/O function its own section and puts the functions `parse`, `load` and `compose` together in a different section. Results in Figure 12b show that this mapping is by far the best choice in terms of both metrics, even though it defines the same degree of parallelism as the "1-I/O-op/sec" mapping and even less than "op/sec".

Figure 14 presents detailed breakdowns to understand the reason for this odd behavior. The graph depicts the average, minimum, and maximum processing times of each of the five web server functions for a run of our first experiment. The log-scaled y-axis shows that outliers are the predominant bottleneck, especially in the case of the `read`. While the average processing times clearly converge to the minima, the average processing time of the `read` is still almost twice that of `load` and `compose`. The parsing of the request exhibits the lowest average processing time. In our implementation, we use the `sendfile` option to enables a zero copy operation for the requested feed from disk straight to the network interface. Similarly, the processing time of the `load` is comparatively low because it consists of a single disk seek operation. In contrast, interfacing the network for sending the response exhibits the highest average processing time. We point out again that our Ohua-based server is implemented using blocking I/O only. It represents the most straightforward approach for the programmer and does not impose a complex or unfamiliar programming model on the whole

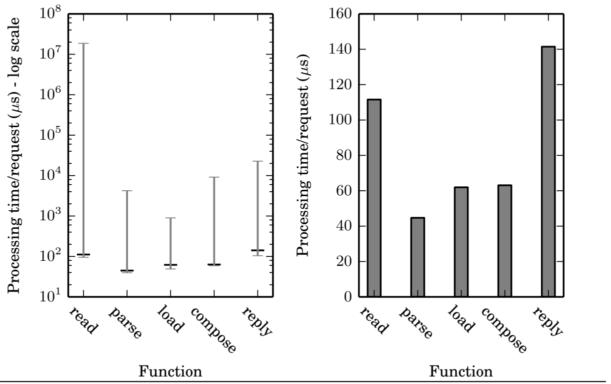
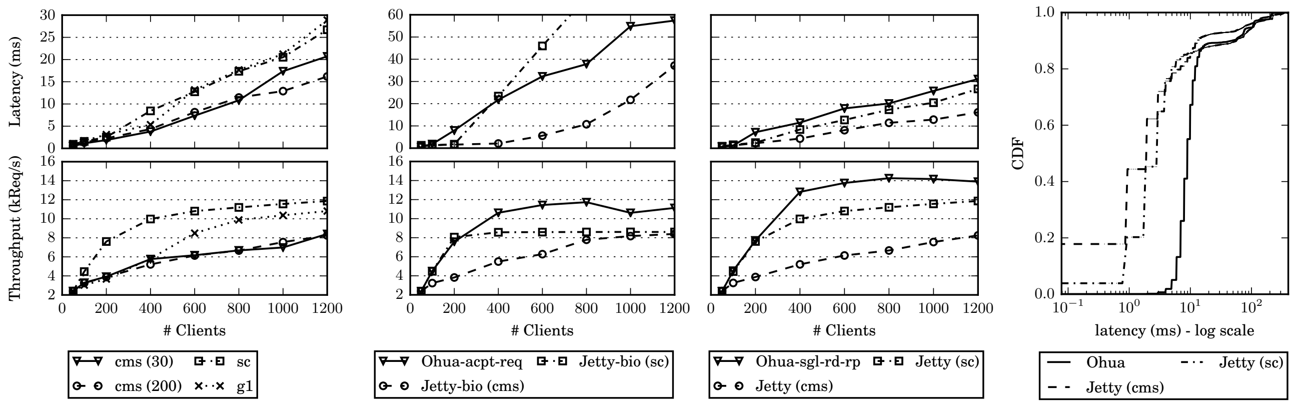


Figure 14: Average, minimum, and maximum processing times per request for each of the web server functions.

program as non-blocking I/O frameworks do, with one central event dispatcher that clutters the code and makes it hard to reason about.

Finally, we study the impact of section scheduling in Figure 12c. Note that we define a safety limit in the size of the inter-section arcs for the unlikely case that one of the I/O functions blocks for a unusually long time, but this limit is not of interest with respect to our section scheduling. Therefore, we compare two different implementations. The first one uses a concurrent non-blocking queue while the second one uses a blocking version. The difference with respect to scheduling is that an unsuccessful dequeue operation in the first case results in a section scheduling cycle, while in the latter case the operation just blocks until data becomes available, thereby avoiding scheduling overhead. The results verify the expected superior performance of the blocking version in terms of latency. These benefits however vanish when the server is highly loaded (beyond 600 concurrent clients) and the amount of unsuccessful dequeue operations becomes negligible. Furthermore, at that point the lock-based nature of the queues does not scale with the number of items and starts to yield substantial overhead in terms latency and a decrease in throughput. This illustrates the importance of a flexible runtime system that adapts automatically to the system load in order to improve performance. Ohua provides the functionality necessary to achieve this goal.



(a) Jetty GC (b) BIO (c) Async NIO (d) Req. latency

Figure 13: Comparison of Jetty against the simple web server implemented in Ohua.

5.2 Data parallelism

We next investigate the potential for processing requests in a data parallel fashion using the `balance` macro from Listing 11. On the basis of our previous results, we use the "sgl-rd-rp" section mapping for each of the parallel branches, in combination with the concurrent arc implementations. Figure 12d shows the results for different replication factors (as the first argument to `balance`). The graphs indicate that additional parallelism does not improve performance, because it translates into more parallel I/O operations that in turn add overheads, likely due to increased contention on internal lock structures. Our Ohua-based web server implementation reaches the I/O boundary here. At this point the only possible way to further improve performance would be to change from the simple blocking I/O model to a more advanced version that would scatter read and write operations to better interface with the network. Although Jetty makes use of all these features, they are left to future work in Ohua.

5.3 Comparison with Jetty

To put our results in context with prevalent I/O programming models in current state-of-practice Java-based web servers, we compare the best configuration of our Ohua web server against Jetty (version 8.1.9) in Figure 13. In all our experiments, we ran Ohua with the concurrent-mark-and-sweep (CMS) garbage collection (GC) algorithm to minimize the overhead on request latency. Figure 13a shows that Jetty's performance actually varies across the different garbage collectors. For CMS, we ran the experiment twice: once configured with 30 worker threads and once with 200 (default). The results show that CMS performance does not depend on the number of threads to be inspected.[5] The lowest latency is achieved by CMS while the parallel scavenge (sc), a stop-the-world GC implementation, provides the highest throughput. The new G1 (garbage first) collector is not yet located in between the scavenge and the cms performance-wise, as one would actually expect. It is on par in terms of latency with scavenge but does not achieve the same throughput especially for lower load situations. It appears that Jetty users have to decide which property is most important to their web sites, latency or throughput. We therefore take both GC flavors into account in our comparison.

We ran Jetty with a thread pool size of 200 in both available modes: blocking (BIO) and non-blocking/asynchronous (NIO). In Ohua, we selected the corresponding section mapping for our pipeline: "acpt-req" (accept-request) and "sgl-rd-rp". Results from BIO in Figure 13b show that Jetty with CMS performs better than Ohua latency-wise. As for throughput, Ohua is the better choice even compared to Jetty with parallel scavenge.

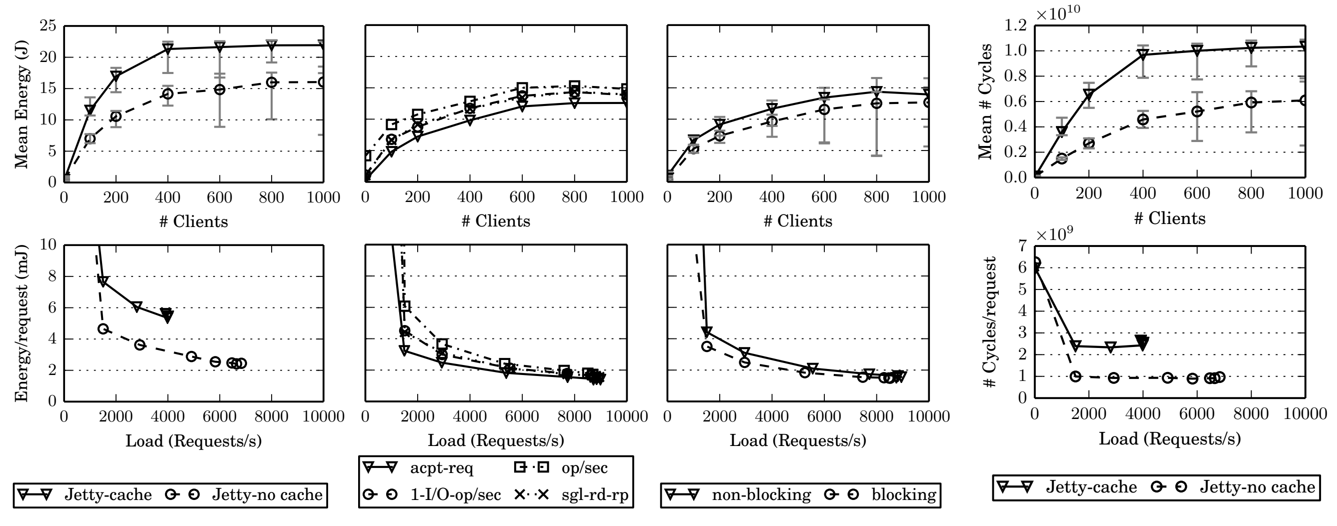
Results for NIO, shown in Figure 13c, indicate that Ohua remains superior to Jetty in terms of throughput. The average latency overhead for Ohua is about 3 ms as compared to Jetty with parallel scavenge, which has in turn higher latency than Jetty with CMS. The latter does, however, exhibit the worst performance in terms of throughput.

We take a closer look at these values for the run with 800 concurrent clients in Figure 13d, which presents the cumulative distribution of request latencies. The constant overhead for Ohua most likely results from the blocking nature of the I/O operations. The GC impact, as experienced by roughly 10% of the requests, is similar to the Jetty executions.

5.4 Energy comparison

Next, we want to validate claims by some researchers [40, 46, 27] that dataflow, message passing, or lock-free algorithms are more energy-efficient than traditional multi-threading with shared-memory synchronization and locks. We are also interested in understanding the performance/energy trade-offs of Ohua's runtime parameters. To that end, we ran experiments on the Intel machine. Each experiment executed for 80 s. After 20 s into the computation we gathered the energy consumption of the server machine for 30 s using `perf state` on the `power/energy-cores` event. In Figure 15 we investigate the energy consumption for different configurations of Jetty and Ohua in NIO mode using the CMS GC configured with 3 GB of memory. In the first row, we report the average, minimum, and maximum energy consumption during the probing period in the different client configurations while the second row depicts the energy consumption per request depending on the load situation on the server. In our first experiment, we deployed Jetty with and without caching enabled. The results of our energy analysis in Figure 15a show that executing Jetty without a cache is far more energy efficient and scales better to more intensive load situations. This is due to the fact that Jetty uses the

[5]Ohua adapts the thread count to the number of sections automatically.



(a) Jetty Cache Comparison (b) Section Mapping (c) Arc Comparison (d) Jetty CPU Cycle Comparison

Figure 15: Energy consumption for Jetty and our HTTP server implemented in Ohua.

`sendfile` primitive to serve static file content without copy operations. The cached version needs to first copy the file to memory and then back to the socket for sending. As shown in Figure 15d, this translates into more cycles and a higher energy demand respectively.

In our second experiment we compare the same 4 section strategies as in Figure 12b. The mean energy consumption in Figure 15b shows that the more sections we have, the more energy is consumed: the "op/sec" strategy with 6 sections consumes the most while the "acpt-req" strategy with only 2 sections consumes the least. As the load increases, however, the mappings with more sections become more efficient. This suggests that section strategies should not only adapt to the execution infrastructure, but also to the load of the system. Note that the load situation is not the same as in the cluster experiment: a similar load of around 13000–14000 requests/s is likely to show again that the "sgl-rd-rp" strategy outperforms the others. Finally, a comparison of the graphs in Figure 15a with those in Figure 15b indicates that Ohua is more energy efficient and scalable than Jetty.

Our last experiment replaces once more the concurrent non-blocking arcs of the inter-section arcs in the "sgl-rd-rp" strategy with blocking implementations. For the load situations depicted in Figure 15c, the blocking variant outperforms the non-blocking one in terms of energy consumption but also exhibits lower throughput. Indeed, with low load, the blocking variant provides the opportunity for the system to put blocked cores into halted state where energy consumption is low. One can observe, however, energy-efficiency per request of the non-blocking variant improves with the load because of the lower scheduling costs.

5.5 Data sensitive request handling

Finally, we consider the impact of optimizations that use application-specific knowledge. Assume that our small web documents are news snippets that catch the interest of clients and bring them to load the full article. The size of a snippet is only around 512 bytes while the article (in a compressed form) is around 20 kB. The goal is for the snippet requests to scale independent of the requests for the full articles. We therefore store the snippets separately in an in-memory file system, essentially removing disk I/O bottleneck. Finally, to ensure that no snippet request is blocked by the disk I/O of an article being loaded, Listing 14 defines a condition that dispatches requests according to the location of the requested resource.

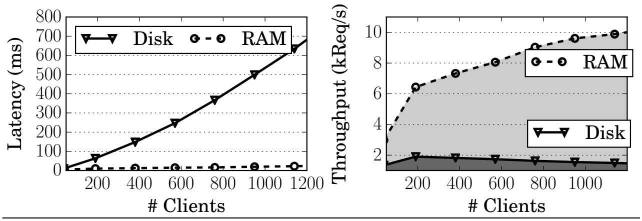


Figure 16: Latency and throughput for concurrent requests on feeds (512 Bytes) residing in RAM and articles (20 kB) located on disk.

Listing 14: Conditional Statement on the Resource Location

```
(ohua
  (let [[_ resource-ref] (-> 80 accept read parse)]
   (if (.startsWith resource-ref "news/")
    (-> resource-ref load-ram write reply)
    (-> resource-ref load-hd write reply))))
```

Each branch is mapped onto a different section. We run this experiment with an equal number of clients requesting snippets and articles. The graphs in Figure 16 strongly support one of Ohua's main arguments: parallelism and concurrency are highly sensitive to the deployment context. Separating the aspect of parallelism from the program fosters clean algorithm design and allows for a highly flexible runtime system with powerful optimizations.

6. RELATED WORK

Purely functional languages are best described by their absence of global state and side-effects (state mutations): data flows through the functions of an algorithm. In this respect, Ohua—and more generally FBP and dataflow—are by nature tied to a functional programming style [37]. It does, however, use the object-oriented paradigm to encapsulate state inside operators with a set of functions that are never executed concurrently. As a result, state access is guaranteed to be sequential and limited to the scope of the functional set of the operator class. Scala unifies functional and imperative programming but does so to rather shorten the code to be written rather than to address the multi-core challenge. Actors are Scala's main concept to address coarse-grained parallelism but it lacks a concept to map it onto the different programming styles and leads to heavily cluttered code [21]. Dataflow languages such as Lustre [20] and Lucid [47] derive the dataflow graph from the program but do not unify functional and imperative programming. To the best of our knowledge, no approach exists that implicitly derives a dataflow graph from a functional program where each of the functions is programmed imperatively with attached state.

Functional languages like Haskell can support implicit parallelism, but the degree at which it operates is typically too fine-grained for our needs [23]. Haskell falls back to explicit language constructs to enable coarse-grained parallelism [33]. Manticore relies on message-passing to achieve similar functionality [18]. Dataflow languages, just as the dataflow monad of Haskell, target fine-grained data parallelism. This is also the case for StreamIt [45], an explicit data streaming framework that targets special hardware such as GPUs to perform computations in a massively parallel fashion, and pH [1], a completely implicit parallel language that uses the execution model of Id [35] and the syntax of Haskell to address special non-von-Neumann architectures. In contrast, Cilk [8] defines again abstractions to separate the data dependencies of imperative C/C++ programs into tasks. Ohua notably differs from other programming models by separating the concern of parallelism from the program. The programming model allows to derive parallelism implicitly from the program.

Recent work on regulating the degree of parallelism focuses especially on loops. DoPE also build explicit dataflow graphs and allows to define the degree of parallelism independent from the program to adapt to different architectures [38]. However, DoPE does not define a concept to essentially decrease the degree of parallelism below the number of tasks. Similarly, feedback-directed pipeline parallelism (FDP) builds explicit dataflow graphs (pipelines) but introduces no concept to reduce the degree of parallelism below the number of operators [42]. FDP concludes that the operator with the highest processing time should be assigned the most resources. Ohua makes the opposite observation especially for I/O blocked operators. Manticore [19] uses the concept of continuations to allow multiple fibers to be executed on the same thread. Yet, the assignment of fibers to threads is specified explicitly in the source code at compile-time. Scala provides support for continuation via a `react` operation. The task of the reacting actor is executed on the sending thread and the developer must decide at compile-time whether a thread-based `receive` or an event-based `react` is appropriate. Ohua's sections concept provides maximum flexibility to adjust the degree of parallelism to different architectures separated from the program.

7. FUTURE WORK

In this paper we introduced the main concepts and ideas of the stateful functional programming model and its implementation in Ohua. In the following, we highlight three extensions that we intend to target as future work to emphasize the potential of SFP to exploit multi- and many-core architectures.

SFP for scalable systems design.

In order to investigate the scalability of our programming model beyond a web server, we will turn towards more complex systems. Interestingly, the popular map/reduce (MR) [13] programming model, which enables implicit data-parallel programming on a cluster of commodity machines, also shares many commonalities with FBP. Processing happens in *map* and *reduce* phases, which are executed as tasks on small chunks of the data. The main structure of MR task processing, expressed as a dataflow in Ohua, is shown in Listing 15.

Listing 15: Dataflow in the Map-Reduce programming model

```
; map task
(ohua (-> read-dfs parse-chunk map sort
    serialize compress store-disk))

; reduce task
(ohua (-> fetch-map-results deserialize merge
    reduce serialize compress store-dfs))
```

In comparison, the implementation of the map and reduce task drivers in Hadoop [3] respectively take ∼1,800 and ∼3,000 lines of Java code with algorithm and functionality mixed within the same classes.

Loops vs. higher-order functions.

Most complex systems contain a substantial amount of loops. We are well aware of the great deal of research that has been conducted for loop parallelism. However, most of these approaches require loop operations to be stateless. Even more so, they often parallelize loops that are only present in the code because the concept of higher-order functions was missing. Higher-order functions are to declarative languages what loops are to imperative languages [39]. Hence, the investigation of loop parallelism in SFP must be accompanied with the introduction of higher-order functions into the declarative functional part. This will increase opportunities for data parallelism via programming level concepts rather than concurrency abstractions.

Automatic scheduling and section configuration.

Finally, a large portion of our future work will focus on runtime optimizations. In current systems, the adaptation to the hardware is rather limited often only to the configuration of a thread count. Our section-mappings allow for a much more powerful and flexible adaptation of the program to the underlying infrastructure. Since manually defining sections for very large programs does not scale, automatic section configuration will be the first goal that we address. In a later step, we will turn towards configuring the sections dynamically at runtime to adapt to changes in the load of the system. Furthermore, due to the explicit nature of threads the presence of a scheduler is often missing. Scheduling often relies either on the JVM or on the operating system that both have very little knowledge on the executing program. The newly introduced fork-join framework and its work-stealing scheduler are first steps into the right direction [30]. However, the success of this model on the JVM may be limited because it breaks computations, and therewith also data, apart into many small pieces. This either has a strong impact on object creation and garbage collection respectively or requires developers to highly optimize the way in which input data is split and results are joined [12]. The presence of a scheduler in Ohua programs allows for far more research on scheduling algorithms in many different application domains.

8. CONCLUSION

The stateful functional programming model is a promising approach to develop concurrent and parallel applications for the new generation of multi-/many-cores architectures. Ohua is a programming framework and runtime system that goes beyond existing dataflow engines by supporting *implicit* parallelism. An application is composed of two parts: the actual functionality developed as a set of functions in an object-oriented language with a rich ecosystem (Java) and the algorithms written in a functional language that naturally fits the dataflow model (Clojure). Ohua derives the dataflow graph from the algorithm at compile time, and schedules the execution of groups of operators (*sections*) on parallel threads at runtime. The partitioning of operators into sections is key to scalability as it allows to control the granularity of concurrency and parallelism.

Ohua does not only provide a safe approach to developing correct concurrent applications, but it also supports highly-efficient implementations. We showed that a web server developed on the basis of simple sequential code performs better than the hand-optimized Jetty implementation in terms of throughput and competitively in terms of latency on a multi-core architecture, while being also energy-efficient. The more general insight gained from our evaluation: Ohua uses blocking calls in combination with threads to achieve asynchronous I/O while Jetty's highest performing implementation uses NIO. Our results give rise to the assumption that event-based programming via Java's NIO framework does not directly translate into more scalable systems than thread-based asynchronous I/O. A focused study on I/O models in Java has to show the universality of this assumption.

Acknowledgements

This work has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under the ParaDIME Project (GA no. 318693).

9. REFERENCES

[1] *Implicit Parallel Programming in pH.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[2] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. OOPSLA '11. ACM.

[3] Apache. Hadoop. http://hadoop.apache.org/.

[4] J. Armstrong. The development of erlang. ICFP, 1997.

[5] Arvind and D. E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow architectures. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[6] M. Beck and K. Pingali. From control flow to dataflow. ICPP '90, 1990.

[7] C. Bienia and K. Li. Characteristics of workloads using the pipeline programming model. ISCA'10, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. PPOPP '95.

[9] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, Aug. 2010.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. SIGMOD, 2003.

[11] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[12] M. De Wael, S. Marr, and T. Van Cutsem. Fork/join parallelism in the wild: Documenting patterns and anti-patterns in java programs using the fork/join framework. PPPJ '14, New York, NY, USA, 2014. ACM.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[14] J. Dennis. A parallel program execution model supporting modular software construction. MPPM '97. IEEE Computer Society.

[15] J. B. Dennis. Data flow supercomputers. *Computer*, 1980.

[16] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. VLDB, 1986.

[17] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous i/o for event-driven servers. ATEC '04, Berkeley, CA, USA, 2004. USENIX Association.

[18] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in manticore. ICFP '08, New York, NY, USA, 2008. ACM.

[19] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. DAMP '07. ACM, 2007.

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. Proceedings of the IEEE, 1991.

[21] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 2009.

[22] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 1985.

[23] T. Harris and S. Singh. Feedback directed implicit parallelism. ICFP '07. ACM.

[24] J. He, P. Wadler, and P. Trinder. Typecasting actors: From akka to takka. SCALA '14, New York, NY, USA, 2014. ACM.

[25] R. Hickey. The clojure programming language. DLS '08. ACM.

[26] G. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, Sept. 1997.

[27] N. Hunt, P. S. Sandhu, and L. Ceze. Characterizing

the performance and energy efficiency of lock-free data structures. INTERACT '11. IEEE.

[28] IBM. Infosphere datastage data flow and job design. http://www.redbooks.ibm.com/, July 2008.

[29] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. PLDI '94.

[30] D. Lea. A java fork/join framework. JAVA '00, New York, NY, USA, 2000. ACM.

[31] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1), Jan. 1987.

[32] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. SOSP, 2005.

[33] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: Better strategies for parallel haskell. Haskell '10. ACM, 2010.

[34] J. P. Morrison. *Flow-Based Programming.* Nostrand Reinhold, 1994.

[35] R. S. Nikhil. *Id language reference manual.* Laboratory for Computer Science, MIT, July 1991.

[36] S. Okur and D. Dig. How do developers use parallel libraries? FSE '12. ACM.

[37] V. Pankratius, F. Schmidt, and G. Garretón. Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java. ICSE '12, Piscataway, NJ, USA, 2012. IEEE Press.

[38] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using dope: The degree of parallelism executive. PLDI '11. ACM.

[39] P. V. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, Cambridge, MA, USA, 2004.

[40] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. LCTES '12. ACM.

[41] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: High-throughput stream programming in java. OOPSLA '07, New York, NY, USA, 2007. ACM.

[42] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. PACT '10. ACM, 2010.

[43] S. Tasharofi, P. Dinges, and R. Johnson. Why do scala developers mix the actor model with other concurrency models? ECOOP '13.

[44] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, Mar. 2002.

[45] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. CC, 2002.

[46] A. Vishnu, S. Song, A. Marquez, K. Barker, D. Kerbyson, K. Cameron, and P. Balaji. Designing energy efficient communication runtime systems for data centric programming models. GreenCom'10, 2010.

[47] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[48] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. SOSP, 2001.