

Dissertation

# TOWARDS IMPLICIT PARALLEL PROGRAMMING FOR SYSTEMS

Sebastian Ertel  
Mat.-Nr.: 2847322

Supervised by:  
Prof. Dr. Jeronimo Castrillon  
and:  
Prof. Dr. Hermann Härtig  
Submitted on April 3rd, 2019

## ACKNOWLEDGEMENTS

This thesis would not have been possible without the loving support of my wife Luisa and my two kids Flint and Maeva. You are my foundation!

When I transitioned from an industry position at IBM to academia to pursue a PhD, I already had a clear goal: making parallel programming easier using the dataflow model. Over the years, I turned this initial thought into Ohua, an implicit parallel language. Along the way to this thesis, there were many people that pushed, questioned and supported me and my ideas. I am indebted to all of them!

I would like to thank Michael J. Beckerle for introducing me to dataflow-based programming in the context of the E2 project at IBM. A big thanks goes to Amir Bar-Or and David Loose for shaping my understanding of dataflow-based systems. I would also like to thank my former office mates Jons-Tobias Wamhoff, Diogo Behrens, Stefan Weigert and Martin Nowack for their help during my time at the chair for Systems Engineering. Thanks Christof Fetzer for introducing me to research and Pascal Felber for helping me publish the first two Ohua papers.

Most importantly, I would like to thank my supervisor Jeronimo Castrillon without whom the Ohua project and my PhD would have died before it had the chance to unfold its potential. He was brave enough to believe in the unshaped ideas, that they were back then, and gave me and the Ohua project the first funding. It was only this lucky meeting that allowed me to work on Ohua full time and to bring more people on board. I thank Justus Adam, my student for the second part of my thesis, for “thinking Ohua” with me now for quite a long time and introducing me to Haskell. I also wish to thank Felix Wittwer for working with me on the current integration of Ohua into Rust. Last but not least, I am in debt to my colleagues Sven Karol, Andrés Goens and Norman Rink for helping me find the formal foundations of Ohua. I dearly enjoyed the work with all of you!

# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Concurrency does not come for free! . . . . .	9
1.2	The Landscape of Concurrent and Parallel Programming . . . . .	11
1.2.1	Concurrency Building Blocks . . . . .	11
1.2.2	Concurrency Abstractions . . . . .	16
1.2.3	Aspects of Parallel Programming Models . . . . .	19
1.3	Contributions and Thesis Structure . . . . .	22
<b>2</b>	<b>Ohua: Implicit Dataflow Programming for Concurrent Systems</b>	<b>23</b>
2.1	Introduction . . . . .	23
2.2	Dataflow Programming . . . . .	26
2.3	Stateful Functional Programming in Ohua . . . . .	27
2.3.1	From Operators to Stateful Functions . . . . .	27
2.3.2	Algorithms . . . . .	28
2.3.3	Dataflow Graph Extraction . . . . .	29
2.3.4	Linking . . . . .	31
2.3.5	Opportunities for Data Parallelism . . . . .	32
2.4	Flexible Parallel Execution . . . . .	33
2.4.1	Parallelism Granularity Control . . . . .	34
2.4.2	Integration into Clojure Programs . . . . .	35
2.5	Evaluation . . . . .	35
2.5.1	Pipeline Parallelism . . . . .	37
2.5.2	Data Parallelism . . . . .	38

2.5.3	Comparison with Jetty . . . . .	38
2.5.4	Energy Comparison . . . . .	39
2.5.5	Data-sensitive Request Handling . . . . .	41
2.6	Related Work . . . . .	41
2.7	Future Work . . . . .	42
2.8	Conclusion . . . . .	43
<b>3</b>	<b>Supporting Fine-grained Dataflow Parallelism in Big Data Systems</b>	<b>45</b>
3.1	Introduction . . . . .	45
3.1.1	Scalability Issues of Big Data Systems . . . . .	45
3.1.2	Contributions . . . . .	46
3.2	The Core of Big Data Processing . . . . .	48
3.2.1	Data Processing: Code Study . . . . .	48
3.2.2	Analysis . . . . .	49
3.3	Implicit Parallel Programming in Ohua . . . . .	50
3.3.1	Algorithms and Stateful Functions . . . . .	50
3.3.2	Dataflow-based Runtime System . . . . .	52
3.4	Rewriting HMR’s Mapper in Ohua . . . . .	53
3.5	Evaluation . . . . .	56
3.5.1	Experimental Setup . . . . .	56
3.5.2	Runtime Analysis . . . . .	57
3.6	Related Work . . . . .	59
3.7	Conclusion and Future Directions . . . . .	59
<b>4</b>	<b>Compiling for Concise Code and Efficient I/O</b>	<b>61</b>
4.1	Introduction . . . . .	61
4.1.1	Code Conciseness versus I/O Efficiency . . . . .	62
4.1.2	Contribution . . . . .	63
4.2	Reducing I/O on an Expression IR . . . . .	63
4.2.1	I/O Reduction . . . . .	65
4.2.2	I/O Lifting . . . . .	66
4.3	A Dataflow IR for Implicit Concurrency . . . . .	69
4.3.1	From Implicit to Explicit Concurrency . . . . .	69
4.3.2	Concurrent I/O . . . . .	71

4.4	Related Work . . . . .	72
4.5	Evaluation . . . . .	72
4.5.1	Microservice-like Benchmarks . . . . .	73
4.5.2	Experimental Setup . . . . .	74
4.5.3	Results . . . . .	75
4.6	Conclusion and Future Work . . . . .	76
<b>5</b>	<b>A Framework for the Dynamic Evolution of Highly-Available Dataflow Programs</b>	<b>79</b>
5.1	Introduction . . . . .	79
5.2	FBP for Live Updates . . . . .	81
5.2.1	Referential Transparency by Design . . . . .	82
5.2.2	Sequential Operator State Access . . . . .	83
5.3	A Time-based Algorithm . . . . .	83
5.3.1	The Right Point in Computation Time . . . . .	84
5.3.2	Marker-based Update Coordination . . . . .	84
5.3.3	Marker Joins . . . . .	85
5.3.4	Deadlocks . . . . .	86
5.3.5	State . . . . .	87
5.3.6	Timeliness . . . . .	87
5.4	FBP Extensions . . . . .	87
5.4.1	Structured Operators and Packet Dispatch . . . . .	87
5.4.2	Dataflow Graph Transformations . . . . .	88
5.5	Structural Updates . . . . .	90
5.5.1	Coordinated Reconstructions . . . . .	90
5.5.2	Distributed Rewriting . . . . .	90
5.6	Programming Model . . . . .	92
5.6.1	Operators and Mutual Dependencies . . . . .	92
5.6.2	Algorithm Functions . . . . .	93
5.6.3	Integration with Clojure Programs . . . . .	94
5.7	Evaluation . . . . .	94
5.7.1	Runtime Overhead Evaluation . . . . .	94
5.7.2	Coordinated NIO Switch . . . . .	95
5.7.3	Dynamic HTTP Server Development . . . . .	95
5.7.4	Cache Loading Strategies . . . . .	96
5.8	Related Work . . . . .	97
5.9	Conclusion and Future Work . . . . .	97

<b>6 Conclusion</b>	<b>99</b>
6.1 Opportunities for SFP in the domain of data management systems . . . . .	100
6.2 Other Directions . . . . .	104

## ABSTRACT

Processor architectures have reached a physical boundary that prevents scaling performance with the number of transistors. Effectively, this means that the sequential programs of the past will not gain performance boosts from new processor designs any more as they used to. Since single-core performance does not scale anymore, new processor architectures now host several cores. In order to speedup execution with these multiple cores, a program needs to run in parallel. Ideally, a compiler could parallelize a program automatically and optimize it for a given multi-core architecture. But most prevalent programming languages encourage the developer to use state in order to avoid recomputations and make programs fast. Keeping track of state references is impossible in general and prevents compilers from parallelizing code automatically. The programmer must manually transform its algorithm into a parallel version by leveraging concurrency constructs, such as threads and locks. Writing correct, i.e., data race free and deadlock free, concurrent code is hard. As such, parallel programming models often discard the state aspect.

In this thesis, we propose the *stateful functional programming (SFP)* model. A program in our model consists of two abstractions an *algorithm* and *functions*. We say that an algorithm defines the composition of functions. A function may operate on its own *private state* where each call-site of the function has a state assigned to it at runtime. This state is required to be private such that the associated compiler can automatically parallelize algorithms without introducing data races.

The programming model is inspired by the well-known dataflow execution model which serves as the state-of-the-art for parallelizing data management systems and signal processing applications on embedded system. As such, our SFP compiler translates algorithms into dataflow graphs to provide dataflow parallelism implicitly to programs in the domain of server-based back-end infrastructures and data processing systems. In support of that, this thesis establishes the idea of stateful functional programming in the context of a server. The dataflow representation of the program allows us to study different well-known server designs and select the appropriate option with respect to a target metric, i.e., throughput, latency or energy-efficiency. To further evolve our SFP model, we study it as a part of a big data execution engine. The results show that the associated compiler and runtime system can introduce parallelism into parts that are current execution bottlenecks.

In the second part of the thesis, we argue that our compiler and dataflow-based runtime can also solve problems that are directly connected to a parallel execution. In the context of large back-end infrastructures with lots of I/O, we show that our compiler can optimize I/O without requiring any code changes. This is possible because SFP algorithms adhere to a functional programming paradigm and as such directly translate into expressions of the lambda calculus. Our compiler uses the lambda calculus as its high-level intermediate representation (IR) and lowers expressions into the dataflow representation. This allows us to define (most of) the I/O optimizations as transformations on lambda expression and prove that they preserve the semantics of the program. In the context of highly-available server programs that need to evolve without down times, we define an algorithm for non-blocking live updates. Our dataflow-based runtime representation makes the communication in the program explicit and identifies side-effects to state. This information is the foundation to streamline change requests into the data stream that not only update a single node but whole parts of the graph. The algorithm thereby makes sure that the system transitions from one consistent state into another without requiring additional synchronization.

To present interesting directions for future research, we analyze the applicability of our SFP model to the domain of data management systems, including database engines, data streaming engines and database applications.



# 1 INTRODUCTION

In 2004, Herb Sutter declared that the “Free lunch is over”: sequential programs will not gain better performance from new processor generations [206]. Before 2004, developers could expect a program performance increase by a factor of two from one processor generation to the next. This expectation was derived straight from Moore’s law that predicts the transistor count on a die to double every generation [162]. Phrased differently, the die size for the same amount of transistors shrinks by 50%. Interestingly, already in 1965 when Moore developed his law, he saw the physical boundary that would end the free lunch: “Will it be possible to remove the heat generated by tens of thousands of components in a single silicon chip?” The heat problem eventually translates into a physical boundary which prevents a processor to scale its performance with the number of transistors [31]. To circumvent this boundary, processor architects designed multi-core processors where a single chip hosts more than one core that can operate in parallel. This is known as the dawn of the *multi/many-core era*. In order to benefit from the multiple cores, the program must provide opportunities to execute parts of it independently at the same time, i.e., *in parallel*. This property of a program, called *concurrency*, is what Sutter proclaims to be the most fundamental revolution in software since object-oriented programming.

Figure 1.1 plots the scaling trends of CPUs up to the year 2014 using data from the CPU database [56]. Moore’s law still holds: the transistor count on a die still doubles every generation. And certainly, processor architects found tricks in order to also improve the performance of sequential programs. But these improvements are not comparable with the ones encountered in previous generations. Indicators for that are the stagnation in clock frequency and power consumption around 2004. The rise of multi-cores and the hyper-threading technology (also referred to as hardware threads) starting around 2006 demand concurrent programs.

## 1.1 CONCURRENCY DOES NOT COME FOR FREE!

Writing concurrent programs is hard and best left to experts but multi-core architectures force developers to become experts in concurrent programming. Concurrency is not an invention that was triggered by multi-cores. The concept is much older and was primarily used to unblock computation from I/O, such as a disk read or a call over the network. It was best left to operating system, database or high-performance server experts. Multi-cores made concurrency a necessary requirement for every program that wishes to improve performance. But a long time ago, the industry decided to write programs relying on the program counter imperative (von-Neumann style architectures) instead of (the mathematical notion of) functions [20]. In the advent of multi-cores, this decision became a boomerang. The main benefit of a functional program is referential transparency which guarantees the same result no matter how often the program runs. The same accounts for its parts and allows a compiler to decompose the program into parts that can execute in parallel, i.e., turning a sequential into a concurrent program. In general, this is not possible for imperative programs because their very essence is based on the modification of variables that reference memory cells. An imperative program may declare access to the same variable in different parts of the code. A runtime system that executes these parts concurrently

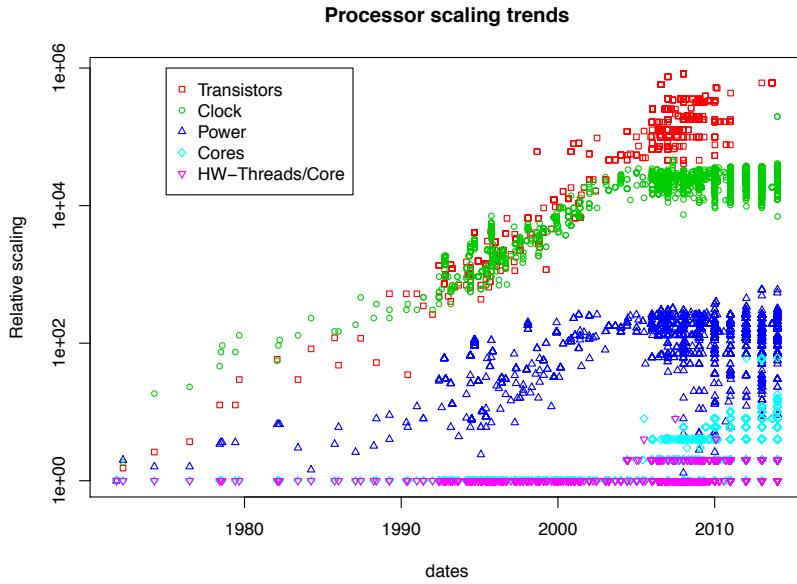


Figure 1.1: Scaling trends of CPUs

introduces non-deterministic results that violate the correctness of the program [138]. Concurrent code parts that access the same variable are said to race for the data it references [167]. A program with such code parts is said to contain *data races*. In general, detecting data races is known to be an NP-hard problem which prevents a compiler to transform an imperative sequential program into a parallel one. The programmer needs to perform this transformation manually in the code.

The plots in Figure 1.2 give an idea of how challenging this transformation is for the programmer. Both plots relate the concurrency in a program to the speedup it gets from multi-cores and show the speedup curves for programs with a different portion of concurrent code. The left plot shows the theoretical speedups with respect to Amdahl's law [8]. The curves in the right plot follow Gustafson's law [97]. The main take-away in both of these plots is that the code needs to be highly concurrent in order to fully benefit from the available cores. Phrased differently, the goal for the developer must be to minimize the sequential portion of the code. For example, in the plot following Amdahl's law the code with a concurrent portion of 95% is twice as fast as one with only 90%. Finding these 5% is often extremely challenging, especially in a program that is already concurrent. The plot following Gustaffson's law similarly emphasizes this fact. The more cores are available the greater is the speedup difference between the curves. For example, if the parallel-executing portion of the code remains only 50% then the program will not scale larger workloads. It will be bottlenecked by the sequential half of the code. As a result, the programmer has to squeeze as much concurrency(/parallelism) out of its algorithms as possible without breaking their semantics. This may even require to replace an algorithm with another one that may be less efficient when executed sequentially but exposes more concurrency.

Crafting concurrent algorithms is tough and even more so implementing them without concurrency-related bugs [180]. Contrary to this belief, a recent study of code quality in open source projects on github relates only 1.99% of all bugs to concurrency [190]. But previous studies directly targeted concurrency-related issues to give more insight into the characteristics of concurrency bugs [152]. They report that concurrency issues are extremely complicated and often require a substantial amount of time to get fixed. Often the associated patches are either not correct or introduce new concurrency bugs. A study of concurrency bugs in the MySQL database server application found that some issues required more than a year to fix [82]. The mean time to a fix was on average more than 3.5 months. This is presumably much more time than what is spent on fixing other types of bugs. In fact, all these studies acknowledge that concurrency bugs are underrepresented in the bug databases. This is due to the following facts: Concurrency bugs are often classified as Heisenbugs due to their non-deterministic behavior [95, 165]. This makes them hard to reproduce and as such developers often do not report them if they do not manifest during a rerun of the regression tests. Furthermore, concurrency bugs are often architecture dependent. As such, a test case for a concurrency issue may fail on one machine but not on the other. Finally, if a developer gets lucky

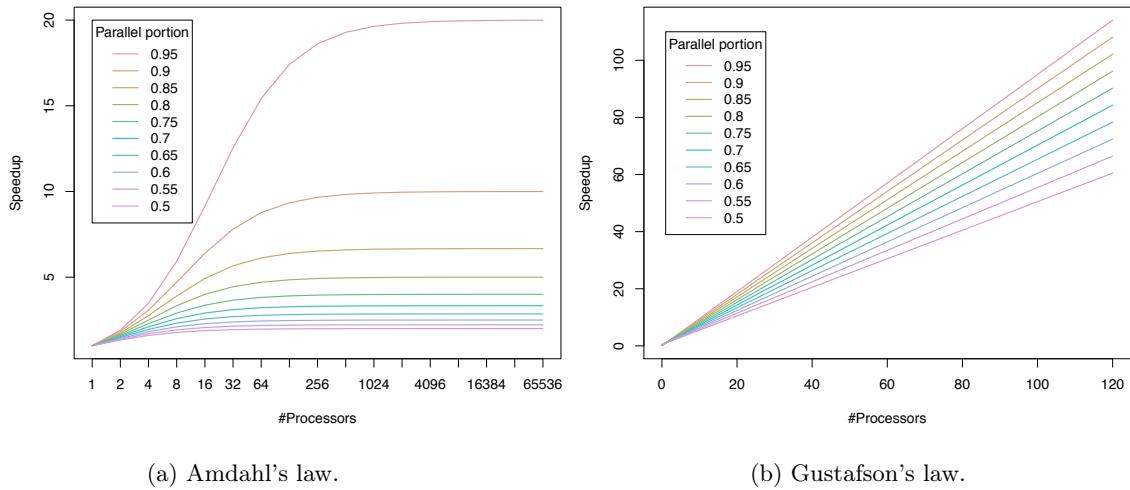


Figure 1.2: The need for concurrency.

enough to have a test case that reproduces the bug then debugging tools and instrumentation often make the bug disappear again.

As a result, in order to make programs scale with the new multi-core hardware, the programming model and the associated compiler needs to assist the programmer in preventing or detecting these bugs [100]. Such a concurrent programming model must enforce a deterministic computation unless the programmer explicitly requests non-determinism [29, 143]. The problem with most prevalent concurrent programming models is that they work the other way around.

## 1.2 THE LANDSCAPE OF CONCURRENT AND PARALLEL PROGRAMMING

A concurrent programming model that prevents concurrency bugs (such as data races and deadlocks) ideally accompanies a compiler that optimizes the program’s parallel execution. Sadly, many concurrent programming models offer only limited safety guarantees and very restricted automatic optimizations. I use this dissertation to review at least the most important current state-of-the-art in concurrent (and often also referred to as parallel) programming visualized in Figure 1.3. I start with what can be considered the fundamental abstractions that a developer can use to introduce concurrency into its program. Afterwards, I study higher-level programming models that are based on these abstractions and offer some safety guarantees as well as optimizations. I conclude my review with an evaluation of parallel programming aspects and a set of programming models that successfully applied them to hide all concurrency aspects from the programmer. This review should help the reader to get the big picture of parallel programming challenges and opportunities, classify new innovations in this field and understand the contributions made in this thesis.

### 1.2.1 Concurrency Building Blocks

We start off with the most basic abstractions that are exposed by the operating system to user-level programs. As an application example, we use the server program of a simple key-value store that provides a cache to increase the scalability of large backend infrastructures[170]. Requests either read content from, load content into or update already present content in the key-value store. Internet companies such as Google, Facebook and Amazon, apply key-value stores to hot spots in their systems with a very high load of requests [19]. To scale these bottlenecks, they have to take advantage of multi-cores and process requests in parallel.

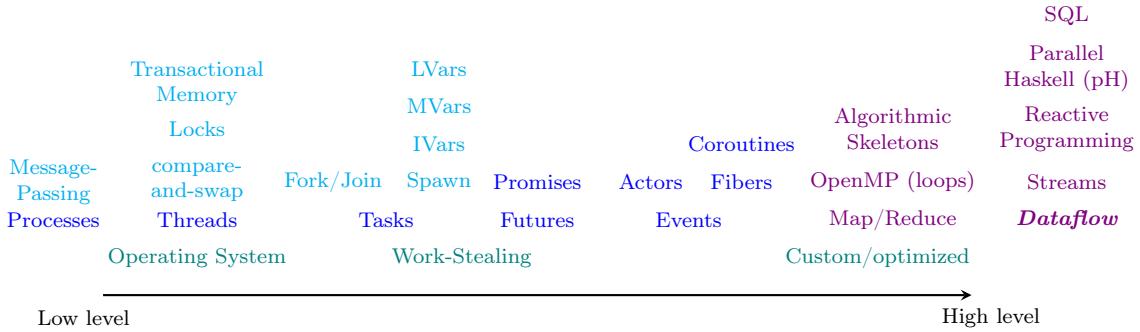


Figure 1.3: **Concurrency abstractions, synchronization mechanisms, parallel programming models/languages** and their underlying **scheduling approaches**.

**Processes and Message-Passing** The most basic abstraction to execute a piece of code is a *process*. Two processes can communicate with each other via sockets, i.e., the programming model assumes a distributed shared nothing memory model. The corresponding *message-passing* interface (MPI) seeks to standardize the ways in which processes may communicate with each other. MPI was primarily used for high-performance computing across the nodes in a cluster.<sup>1</sup> But at the beginning of this century, big data companies joint the field of cluster computing and developed more efficient programming models. The most prominent one is the Map-Reduce programming model [59]. Map-Reduce is simple enough to allow even non-expert programmers to deploy and execute their algorithms across the nodes in a cluster. As a result, more and more algorithms moved out of the HPC community and became mainstream. Current frameworks such as Spark [231] and Flink [35] execute complex analytical tasks on data. Frameworks such as TensorFlow provide efficient neural network implementations for machine learning applications [2]. The HPC community is currently shifting to these frameworks or builds similar frameworks on their own [9, 150]. And although MPI is becoming obsolete, the idea of communicating via messages is fundamental to high-level approaches such as actors that we review below. The main disadvantage of message-passing is the cost for inter-process communication. Data must be serialized before it can be sent via a socket and the receiver has to deserialize before processing it. Serialization and its inverse operation are costly.

**Threads and Locks** The costly data transfer between processes might be the reason why the most prominent abstraction for concurrent programming on multi-cores is a *thread*. On the operating system level, a thread shares its address space with the other threads of a program. The threads can use this shared address space to communicate directly without serialization overhead. Multi-cores further optimize the communication between two threads on the same processor. Instead of paying the cost for a memory access, the receiving thread is likely to find the sent data in one of the on-chip caches. But this optimization comes at a high cost. Two threads accessing the same memory address may create a data race. The developer has to be aware of such situations and prevent concurrent (i.e., racy) data access via *locks*. However, locks introduce two new challenges for the programmer. In its simplest form, a lock can be considered a mutually exclusive resource, i.e., it can only be used by a single thread at a time. As such, a lock-based algorithm must solve the problem of the dining philosophers posed by Dijkstra to his students in the mid 70s and later formulated by Hoare [66, 118]. When two threads try to acquire the same lock then one of them has to wait. If the other thread never returns the lock because of a bug in the code or because the thread has encountered an error, then the other threads waits forever, i.e., the program deadlocks. For example, consider this simple code of a thread that updates the value stored in the internal hash map of our key-value store that is shared with other threads in the program:<sup>2</sup>

```

1 value = sharedHashMap.get(key);
2 value.lock();
3 updatedValue = update(value);
4 sharedHashMap.put(key, updatedValue);
5 value.unlock();

```

<sup>1</sup><https://www.mpi-forum.org/>

<sup>2</sup>The code snippets are Java-based and refer to functions of the `java.util.concurrent` library. But the discussed aspects can be found in most of the major programming languages such as C/C++ etc.

If the `update` function fails then the lock is never released. The programmer has to be aware of these aspects and write the code accordingly:

```

1 value = sharedHashMap.get(key);
2 value.lock();
3 try {
4     updatedValue = update(value);
5     sharedHashMap.put(key, updatedValue);
6 } finally {
7     value.unlock();
8 }
```

If the algorithm for acquiring the lock is biased for some reason and always favors one of the threads which continuously asks for the lock, then the other threads starve. This is also called a life-lock. Luckily, life-locks are (for most of the cases) taken care of by the experts implementing the locks [113]. The developer has to “only” find the data races in the algorithm, remove them using locks and make sure that the resulting algorithm can not deadlock. This is already hard for very simple code with a single lock [144]. It becomes even harder when the program contains more than one lock because locks do not compose. That means, locks prevent a major aspect of software design: code modularity. For example, assume the implementation of the shared hash map would implement the locking:

```

1 value = sharedHashMap.get(key); // acquires a lock on the value
2 if (check(value)) {
3     updatedValue = update(value);
4     sharedHashMap.put(key, updatedValue); // releases the lock on the value
5 } else {
6     // deadlock
7 }
```

If two threads execute this code and request the same key then the system deadlocks when the `check` fails because the acquired lock is not returned.

**Transactional Memory** One way to solve this composability problem is to rely on atomic operations. *Atomic operations* are supposed to reduce the number of locks in a program. Among these, *compare-and-swap* instructions allow to implement lock-free and wait-free data structures and algorithms [111]. They were exposed as library calls to the programmer:

```
1 currentValue = sharedHashMap.putIfAbsent(key, newValue);
```

But they are not powerful enough to allow our simple hash map example to update an existing value. For more coarse-grained atomic operations, researchers brought the concept of a transaction from databases to concurrent programs [94]:

```

1 atomic { // transactional code block
2     value = sharedHashMap.remove(key);
3     if (check(value)) {
4         updatedValue = update(value);
5         // transferring a shared value into another shared data structure
6         anotherSharedHashMap.put(key, updatedValue)
7     } else {
8         // no deadlock
9     }
10 }
```

The `atomic` block declares a memory transaction across all the variables that the code reads from or writes to. The semantics follow only three of the four transactional (ACID) properties defined for database systems: atomicity, consistency and isolation [98]. Durability is not required for data residing in memory. Since the announcement of Intel in 2004 that the next generation of processors are going to be multi-cores, transactional memory (TM) was probably one of the most thought after research fields in concurrent programming. TM comes in two flavor: as transactions in hardware (HTM) [112] and in software (STM) [196]. Hardware transactions can only be very small because the implementation uses the on-chip memory (L1–L3 caches) instead of main memory [65, 104]. HTM support was not available in commodity processors up until Intel released its TSX extension to the x86 instruction set in 2012/13<sup>3</sup>.

Therefore, research focused primarily on STM. Software transactional memory was originally proposed to the community of functional programming. Functional languages do not mutate data in place because this would violate referential transparency. Instead data is immutable and every alteration creates a new copy. Immutable data makes it easy to implement isolation, i.e., each transaction operates on its own snapshot of the data [133]. Therefore, it comes as no surprise that the first STM was implemented in Haskell [67]. Since functional languages were abandoned by the industry in the middle of the 20th century, a lot of research went into the idea of bringing this concept into imperative languages. But STM incurs a quite substantial runtime overhead [36, 181]. And although there is evidence that STM can be efficient and scalable with respect to the number of available cores, it is no silver bullet for concurrent programming [69]. A study of concurrency bugs revealed that only about one third of the bugs would benefit from transactional memory [152]. Furthermore, Transactions are by their very nature speculative, i.e., the programming model hopes that conflicts are rare. The larger a transaction the more likely it is to have conflicts. As a result, one of the conflicting transactions needs to be aborted and retried in which case parallelism turns into wasted cycles, i.e., inefficiency [64]. An efficient STM implementation (for some workloads) is complex. This might be the reason why almost 14 years after the first STM appeared, it is hard to find a stable and efficient implementation.<sup>4</sup> Two other conceptual limitations might have hindered the adoption of (S)TM too. A transaction needs to be able to roll back its changes and as such it can not perform any irreversible I/O actions (side-effects). More fundamentally, TM does not remove the non-determinism created by the data races from the algorithm (and neither do locks) [144]. The developer has to be aware of that.

**Tasks and Work-Stealing** With threads and locks (or TM) at hand, we can now create concurrent algorithms. As an example, we now integrate the above code into our key-value store. To execute requests in parallel, the code for (update) requests must be assigned to different threads. The developer may therefore write the following code:

```

1 kvUpdater = () -> { // a (lambda) function that processes a request
2     var key = keyQueue.poll();
3     // use "key" to update "sharedHashMap"
4 }
5
6 for(int i=0;i<numWorkers; i++) // starting the request handlers
7     new Thread(kvUpdater).start();
8
9 while(true){
10     var key = acceptAndParse();
11     keyQueue.add(key);
12 }
```

This version starts a configurable number of workers (`numWorkers`) that perform the updates. The threads are preallocated and do not change throughout the computation. The implementation needs another concurrent data structure (`keyQueue`) in order to communicate the keys to be updated to worker threads. If, at any point in execution, the number of incoming requests exceeds the number of workers then latencies increase and throughput drops. In order to scale the number of workers with the number of incoming requests, the developer has to spawn a new thread per request:

---

<sup>3</sup><https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell>

<sup>4</sup>Haskell and Clojure provide stable STM implementations due to the above mentioned benefits of functional languages. But there also exist STM implementations for imperative languages such C/C++ and Rust.

```

1 while(true) {
2     var key = acceptAndParse();
3     var kvUpdater = () -> {
4         // use "key" to update "sharedHashMap"
5     }
6     new Thread(kvUpdater).start();
7 }

```

In this version, each request has to pay the additional cost of spawning a thread. In the other version, each request had to pay the additional cost for accessing the concurrent `keyQueue` data structure. To make an appropriate decision, the developer has to know the load of the server before-hand and understand the efficiency of the concurrent queue implementation.

In order to release the developer from these decisions, Cilk was the first approach that lifted the scheduling abstractions from the operating system right into the programming model [28]. Cilk achieved that in two ways: First, the *task* abstraction allows the programmer to express its algorithm *implicitly* as a directed acyclic graph. The original Cilk pre-processor turned these tasks into closures such that the runtime system could understand the data dependencies that had to be satisfied before a task could execute. Second, it provided a *work-stealing scheduler* which is up until today considered state-of-the-art for workloads with limited compile-time knowledge. The key-value store implementation would then *spawn* tasks instead of threads:

```

1 var acceptAndParseFn = () -> {
2     // accept next connection and parse request
3 };
4
5 while(true){
6     // caveat: this task is only here to show how "spawn" works
7     cont String key; // continuation
8     spawn(acceptAndParse, key);
9
10    var updateFn = () -> {
11        // use "key" to update "sharedHashMap"
12    };
13    new Task(updateFn).fork();
14 }

```

The original Cilk programming model was build on top of the concept of continuations and used continuation-passing style. Since continuations are not available in many languages, such as for example Java, this aspect of the programming model did not become mainstream. To nevertheless provide tasks and work-stealing, the Java implementation makes synchronization explicit again:

```

1 var key = spawn(acceptAndParse);
2
3 var spawn = (func) -> {
4     var task = new Task(func);
5     task.fork();
6     var result = task.join();
7     return result;
8 };

```

A task is submitted to the scheduler via `fork` and synchronization on the result is done by calling `join`. This call blocks until the task is done and the result is available. For this reason, it is known as the *fork/join programming model* [142]. The work-stealing scheduler now efficiently adapts the program to the existing workload [84]. When a task is submitted to the scheduler, it gets enqueue into a thread-local work queue. As such, if there is no heavy load then the task may be executed by the same thread that enqueued the task without additional costs for thread creation and inter-thread communication. When tasks queue up in the local work queue then other idle worker threads start to steal tasks from the back of the queue. This programming and execution model underpins many higher-level concepts.

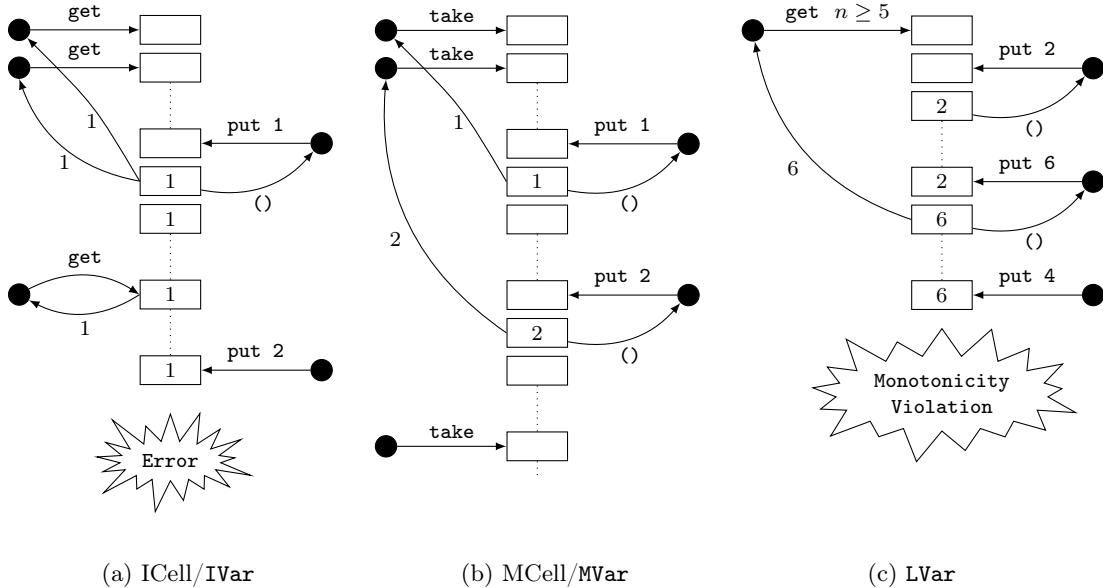


Figure 1.4: Concurrent variables and their semantics. The semantics for LVars are inarguably more sophisticated than the diagram suggests. It is only meant to give the reader an idea of the intuition behind them.

### 1.2.2 Concurrency Abstractions

The abstractions that we study next are meant to further remove scheduling and concurrency aspects from the language to enable an easy composition of concurrent programs.

**Concurrent Variables** It is challenging to compose concurrent data structures that are race-free and do not jeopardize the determinism of the program that uses them. An approach to simplify this task is to construct the data structure from variables that guarantee these properties. Figure 1.4 depicts the semantics for the concurrent variables that we review next.

Consider again the task-based programming model. Cilk used a pre-processor to integrate it into the programming language C. The functions in Cilk are essentially C functions and can do whatever C allows them to do. As such, the `updateFn` can access the `sharedHashMap` which is shared among all the tasks executing this function. That is, the `updateFn` code still needs to be protected with locks or transactional memory. The main take-away is that Cilk’s programming model in combination with C does not provide any deterministic guarantees. Neither does Java’s fork/join framework. A deterministic execution can only be guaranteed if the programming model prevents mutable data.

*I*Structures are variables that can only be written to once. Concurrent attempts to write a value to the variable trigger an error at runtime. Once set, the value never changes again. This prevents non-determinism from this basic operation because the value is defined by exactly one writer and every reader witnesses the same value. IStructures were introduced in the parallel language Id [18]. Haskell implements *IVars* and the task-based programming model including the work-stealing scheduler in its `Par` monad [157]. The most important part in this implementation is the `spawn` function because it communicates the result of a task to another one. The original Cilk programming model used a continuation-passing style. Monad `Par` uses an *IVar* to reference the result of a concurrent computation. Semantically, retrieving the value from an *IVar* blocks until the value was set. The continuation of the computation is then defined by the first function that requests this value in order to make progress.

*I*Structures also got introduced into mainstream imperative languages such Java. In Java, they are called *futures* [227]. In monad `Par`, the *IVars* are tightly integrated into the work-stealing scheduler. A call to `get` either returns the result or traps into the scheduler to schedule the task that computes the result. This is possible because Haskell and especially the implementation of

monad `Par` can use the concept of continuations. A continuation can be considered the rest of the code that executes once the `get` call returns. The initial design of futures introduced continuations into the Jikes Research VM from IBM. At the time of this writing, continuations are not yet part of any production-level JVM. So the current implementation of futures in the JVM had to fall back on lock-based synchronization.

*MStructures*, in Haskell implemented as `MVars`, have slightly different semantics that guarantee race-freedom but sacrifice determinism [23, 182]. Instead of the `get` method, the `MVar` provides a `take/fetch` function. If the variable is “empty”, i.e., nothing was stored yet, the computation blocks (semantically). Assigning a value to an `MVar` that hosts already a value will fail the program. When a value is available, it is returned and the variable is set back to an “empty” state such that a new value can be stored. Hence, only a single task can acquire the result. Which of the tasks is not defined, i.e., non-deterministic. This increases the efficiency of algorithms because it reuses already allocated memory but trades determinism.

*LVars* preserve determinism for concurrent data structures that are monotonically increasing with respect to a lattice [135]. The `get` call for `LVars` requires the developer to specify a threshold set which defines when the call is allowed return the value. Until this threshold set is satisfied, i.e., present in the current data structure, the call blocks. Finding these threshold sets for data structures is hard and may not even be possible. So, only a limited number of data structures fulfills can be based on `LVars` and implementing them correctly is non-trivial.

The work on `MVars` and `LVars` was primarily conducted in the functional programming language community to provide low-level building blocks for concurrent data structures. Researchers in the imperative programming language community instead crafted highly optimized versions of the most common data structures using lock-free and wait-free synchronization [113]. For the composition of multiple data structures, transactional memory is their favored solution.

**Concurrent I/O** *Promises* are futures that perform (network) I/O and as such have to deal with the associated class of errors, ultimately reporting them back to the caller [149]. I/O is particularly important in server-based systems because their very essence is to receive client requests over the network, store data to or retrieve it from disk and send a response. Since I/O blocks computation, a programming model for such applications must enable efficient I/O. A future is spawned only when all its input variables are available but it can not wait for the completion of an I/O call. The same accounts for tasks even in the original design of Cilk. In order to do so, the programming model needs support from the operating system (OS). The OS needs to signal an (I/O) *event* to the application when an I/O operation completes and the result is available. The programming model that exhibits this OS callback is then called *event-driven*.<sup>5</sup> Event-driven programs associate events with so called *event-handlers*, i.e., tasks that depend on I/O input. Normally, event handlers execute sequentially but library approaches exist that allow them to run concurrently [232]. This spawned a long debate on whether thread-based or event-driven programming is better [174, 219], although both are dual to each other semantically [140].

On a more fundamental level, event-driven programming has severe implications on the structure of the program. Without the OS support for events, an I/O operation blocks the execution. The thread gets put into a wait queue of the according I/O device driver. On the other hand, event-driven programming is *non-blocking*. All I/O calls return immediately and the programmer has to register an event handler that performs the rest of the computation when the I/O operation completed. Essentially, an event-driven program is a (big) event dispatch of I/O events to handlers. The fundamental problem that arises is coined *stack ripping* [5]. Consider for example a straight-forward blocking version of our server application:

```

1 handleRequest = (cnn) -> {
2     var req = read(cnn);
3     var key = parse(req);
4     var newVal = update(key, sharedHashMap);
5     respond(newVal, cnn);
6 }
```

All variables in the code remain on the stack. But in the event-driven version of the code, this is not possible anymore.<sup>6</sup>

---

<sup>5</sup>One often finds the similar term *event-based* programming.

<sup>6</sup>I only sketch Java NIO here to keep the listings concise.

```

1 // manual stack management
2 Map<Socket, Object> newVals = new HashMap<>();
3
4 handleEvent = (event) -> {
5     switch(event) {
6         case ACCEPT:
7             var serverSocket = event.channel();
8             var cnn = accept(socketSocket);
9             register(READ, cnn);
10            break;
11        case READ:
12            var cnn = event.channel();
13            var req = read(cnn);
14            var key = parse(req);
15            newVal = update(key);
16            newVals.put(cnn, newVal);
17            register(WRITE, cnn);
18            break;
19        case WRITE:
20            var cnn = event.channel();
21            var newVal = newVals.remove(cnn);
22            respond(newVal, cnn);
23            break;
24    }
25 }

```

The developer has to push the variables from the stack to the heap in order to make them available to the event handler executing the rest of the computation. As such, the developer manually manages the stack. In our example, this accounts only for the variable `newVal` but gets worse when we need to perform even more I/O to store our key-value map on disk. To solve this problem, a *fiber* is a lightweight thread that gets scheduled cooperatively with other fibers on a single OS thread. When the executed code performs an I/O call then the fiber creates a continuation for the rest of the computation that captures all the variables, i.e., it creates a closure. It then automatically performs the non-blocking I/O call and returns control to the scheduler. When the I/O event occurs it executes the associated closure. To the developer, this looks like a blocking call and preserves the sequential structure of the code that is easy to understand. However, as mentioned before, continuations are not yet supported in mainstream languages such as Java [200].<sup>7</sup> C++ only recently introduced coroutines that are best known from the functional language Scheme and also build on continuations [108].<sup>8</sup> The programmer may use coroutines to implement fibers (and generators).

On the server-side, I/O programming models are classified either as *blocking* or *non-blocking*. In turn, client-side programs for graphical user interfaces refer to these programming models as *synchronous* and *asynchronous*. Most prominently, C# and F# have support for asynchronous programming built into their language [208, 172]. These are Microsoft's very own languages to create desktop applications. The essential goal is the same as on the server-side: unblock computation from I/O. JavaScript's most prominent event-driven framework, `node.js` has realized this and can be used for both, client- and server-side programming.<sup>9</sup>

**Composition** These concurrency abstractions must be easily composable to write concise concurrent programs, but this is especially problematic in imperative languages. A task (to compute the value of a future) gets triggered when all its input arguments are available. As such, one could see the activation of a task as a single event triggered by the arrival of its arguments. The event-driven programming model essentially applies functions (tasks) to more than one event. This (potentially infinite) sequence of events is referred to as a *stream*. So, in both programming models, composition must be defined on these abstractions, futures and streams. This is challenging for most imperative languages. For example, Twitter decided to base its backend infrastructure on the concept of Scala's futures [71]. And while this was very successful in terms of concurrency, the code still became hard to follow. Consider the following code taken from the online tutorial of Scala's futures:<sup>10</sup>

---

<sup>7</sup>Java has Project Loom that targets to bring fibers and continuations to the JVM:  
<https://openjdk.java.net/projects/loom/>.

<sup>8</sup>At the time of this writing, documentation on the coroutines that C++ provides is very rare.

<sup>9</sup><https://nodejs.org>

<sup>10</sup><https://docs.scala-lang.org/overviews/core/futures.html>

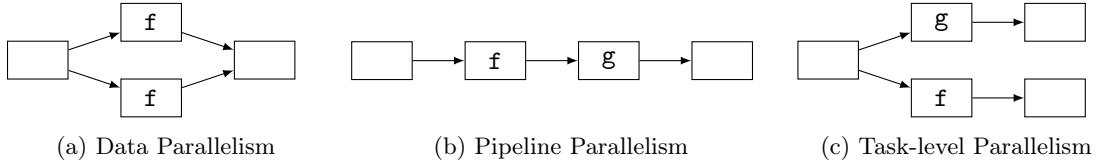


Figure 1.5: Types of parallelism in a dataflow graph.

```

1 Future {
2   session.getRecentPosts
3 } andThen {
4   case Success(posts)    => alreadyRetrievedPosts ++ posts
5 } andThen {
6   case Success(allPosts) => for (post <- allposts) render(post)
7 }
```

The code first retrieves the recent posts (from a social network) and then adds them to the already retrieved ones. Afterwards, it renders all the posts and returns them. The `andThen` function composes two futures to execute in order. On a more formal note, a `Future<T>` is essentially a type that wraps another type `T`. Without going into further detail, this gives rise to a category theoretical treat where the type `Future<T>` would define a functor and a monad respectively [224]. The `andThen` operation is the monadic `bind (>=)` that expresses composition. As such, Haskell's `Par` monad does not suffer from this overly cluttered code structure:

```

1 do
2   posts      <- spawn $ getRecentPosts session
3   allposts   <- spawn $ alreadyRetrievedPosts ++ posts
4   renderedPosts <- forM allposts render
5   return renderedPosts
```

This `do` notation is common to Haskell developers working with monads and nothing specific to the monad `Par` implementation. The Haskell compiler “desugars” the expression in `do` notation into Haskell’s version of the above code.<sup>11</sup> Programming with event streams and signals, that is known as *Reactive (functional) programming*, faces the very same challenge. It allows a program to interface with many I/O sources and often lifts normal functions on values into the realm of streams [21]. However, monads allow composing functions for single events, i.e., normal one-shot calls, but not for continuous invocations over streams. To provide composition, reactive programming relies on a more general concept known as arrows [121] and leverages a similar `do`-notation.

### 1.2.3 Aspects of Parallel Programming Models

In all of these approaches, tasks, futures, events, streams, etc. the program structure that arises is a directed acyclic graph (DAG), called a *dataflow graph*.<sup>12</sup> The vertices of the graph are functions and the edges in the graph define the exchange of data in between them. That is, the inverse of an edge in the graph defines a data dependency. A dataflow graph has two important properties for parallel programming. At first, non-deterministic behavior is explicit in the graph as a so called `non-deterministic merge` node [17]. All other nodes in the graph behave deterministically. Second, concurrency is explicit in the graph. Taking the data dependencies into account, we distinguish the three types of parallelism(/concurrency) depicted in Figure 1.5. When two nodes that execute the same function `f` on different data items then we classify them as *data parallel* (Figure 1.5a). For the case where the dataflow graph processes more than one data item (i.e., streams of events), two data-dependent nodes `f` and `g` can execute in *pipeline-parallel* (Figure 1.5b). Function `g` is applied to the result of `f` at the same time when `f` computes the result using the next item. Two nodes that apply different functions `f` and `g` independently expose *task-level parallelism* (Figure 1.5c). A dataflow graph is often derived from a higher-level specification or programming model.

<sup>11</sup>In order to define a monad, one must first define a functor. In order to define a functor one needs higher-kinded types. Most imperative and even functional languages miss this feature.

<sup>12</sup>A dataflow graph may very well accommodate cycles but I will concentrate on acyclic graphs for the purpose of this overview.

**Data Parallelism** The most prominent model for data parallel programming is *Map/Reduce*. The functions `map` and `reduce` are two well-known higher-order functions from the domain of functional languages [120]. Researchers at Google understood their benefits and exposed them in a programming model for big data that made writing data parallel programs for a cluster easy [59]. In the meantime, both functions are included in the standard libraries of most mainstream programming languages such as Java:

Haskell	Java (Declarative)	Java (Imperative)
<code>map f list</code>	<code>list.stream().map(f).collect(Collectors.asList());</code>	<code>var results = new ArrayList(); for(var elem : list)     results.add(f(elem));</code>

The `map` function applies  $f :: a \rightarrow b$  to each element of a `list` and stores the results in the same order in a new list. Since `f` must be free of side-effects, the computations are guaranteed to be independent of each other and may be executed in (data-)parallel. Instead of a result list, the `reduce` function folds the list to a single value:

Haskell	Java (Declarative)	Java (Imperative)
<code>foldl g initialValue list</code>	<code>list.stream().reduce(initialValue, g);</code>	<code>var result = initialValue; for(var elem : list)     result = g(result, elem);</code>

In general, a folding operation is strictly sequential because the result of the previous computation is input to the next [201]. But when `g` has type  $g :: a \rightarrow a \rightarrow a$  and is additionally associative and commutative then it is possible to partition the list into non-overlapping pairs and apply `g` to each of them in parallel. The resulting list of values can again be folded the same way until only a single value remains. Mainstream programming languages such as Java and C++ do not offer any help at compile-time to detect whether `g` really is commutative. As a result, several Map/Reduce algorithms deployed in production suffer from non-deterministic behavior [230]. Parallel implementations of `map` and `reduce` exist in Java (based on futures) and C/C++ to speed up programs on multi-cores [203, 188] and even GPUs [109]. OpenMP’s parallel loops mainly concern the imperative implementations of the two functions [54, 147]. Polyhedral compilation can even optimize loop nests towards an efficient parallel execution [30].

**Pipeline Parallelism** Data parallelism is relatively easy to extract because it appears naturally in a program in the form of a special type of loop. In order to extract pipeline parallelism, the structure of the loop body, i.e., the function `f`, must expose a (dataflow graph) pipeline. Ideally, the compiler analyzes `f` and extracts the pipeline but adding this to the compilers of the mainstream languages is challenging. Domain-specific programming models and compilers for signal processing, such as StreamIt, Gramps and Halide, support this analysis [211, 204, 185]. It is not surprising that pipeline parallelism is not yet integrated into mainstream languages due to the fact that a pipeline may not always provide a speedup as is the case for data parallelism. The maximum speedup of a pipeline does not depend exclusively on the number of the pipeline stages. For that the pipeline must be balanced, i.e., all stages require (nearly) the same amount of time to process a signal. Furthermore, the more signals flow through the pipeline, the closer is the actual speedup to the maximum, i.e., the number of stages. Approaches in the domain of high-performance servers exist that were able to exploit the inherent pipeline in HTTP request processing to increase throughput performance [228]. Big data systems such as the key-value store Cassandra use this pipelined server architecture to manage high load situations [136].

**Task-level Parallelism** Programs using the task-based programming model and futures build dynamic dataflow graphs that expose task-level parallelism naturally. They often underpin higher-level parallel abstractions such as the `map` function in monad `Par`. But it is not only the work-stealing scheduler that appeals to task-based dataflow programming. Similar approaches show that a dataflow-based runtime abstraction for a stateful program can remove the bottleneck of stop-the-world garbage collection on multi-cores [87]. Actor-based programming is similar but assumes a distributed system as its execution model where actors exchange data via messages [114, 7]. Similar to a task, an actor needs input data to perform its computation. A task has a dedicated successor task that eventually provides this input data. Instead, an actor has a mailbox where it may receive messages (, i.e., input data,) from arbitrary actors in the system. This is why an actor does not only perform a single computation but it essentially loops over the messages in its mailbox forever. If there is no message available it blocks until the next one arrives. Actors are primarily geared

towards distributed system where actors frequently join and leave the system due to machine or transmission failures. To support this flexibility, each mailbox has an address and actors can lookup other actors via this address. The dataflow graph of the actors in the system dynamically changes at runtime and is therefore hard to optimize. For this reason, Erlang, the most prominent implementation of fault-tolerant actors, targeted highly distributed applications for Ericsson, a telecommunications provider [15]. Akka is a famous actor library for the Scala language that runs on the JVM<sup>13</sup> and aspires to be a true alternative for highly concurrent systems. For that it needs to remove the message-passing overhead that is inherent in the underlying fully distributed execution model in favor of a shared memory communication. This has a severe implication: Actors can be *stateful*, i.e., have side-effects to a specific memory region, over the course of processing messages. In order to preserve the strong encapsulation of state of the actor model, an implementation needs to make sure that an actor can not leak state via its messages. This is again challenging because compile-time-enforced state encapsulation requires a more evolved type system than mainstream languages provide [199, 141]. Once the actor implementation is based on such a rigid type system, each actor may have its own heap which in turn makes garbage collection concurrent and an actor program fast [49, 50]. Note that this insight drawn from the work on actors, applies on a more abstract level to all programming models that translate a program into a dataflow graph (with stateful nodes).

**Safe Concurrency** To guarantee a data race free execution of a dataflow graph, there are essentially two options. Either the runtime system copies all the data that flows along the arcs in the graph or the type system detects shared data at compile-time. More generally, parallel programming is safe if either runtime data is immutable or the type system solves the alias problem which is known to be undecidable via static analysis (for languages supporting conditionals, loops, etc.) [139, 186]. Pure functional languages such as Haskell fall into the first category and pay the price of copying data. For a long time, there were no serious candidates in the second category up until the programming language Rust appeared. Rust incorporates an ownership type system which prevents the developer from sharing mutable data [128]. This type system is strong enough to provide the compiler with the information it needs to compute the memory layout at compile-time removing the need for garbage collection at runtime. A related field is linear types which was discovered a long time ago but only recently was successfully implemented into a version of the Haskell compiler GHC [222, 25]. Linear types also prevent the sharing of data and as such allow computations on mutable data (outside of Haskell's I/O monad). Connecting linear types to a concrete parallel programming model is a very interesting ongoing research topic.

**Implicit Parallel Programming Languages** Languages that are safe for parallel programming and provide the according abstractions to the programmer that a compiler can harvest all the above types of parallelism are rare. Algorithmic skeletons are a library of patterns and abstractions (among them `map` and `reduce`) for side-effect-free parallel programs but do not provide any safety guarantees [51]. Rust provides ownership types but requires the developer to spawn threads that communicate via typed channels. Haskell is a safe language and used to execute programs in parallel by default. But research has shown that blindly parallelizing all concurrency in the program does more harm than good [103]. As a consequence of this observation, the GHC(/RTS) executes programs now sequentially and provides various higher-level concepts to introduce parallelism [156, 157]. *pH* was a parallel dialect of Haskell that was directly targetting dataflow machines. Sadly, it vanished when the interest in these machines disappeared but many of its concepts, such as for example IStructures and MStructures, finally found their way into mainstream parallel programming.

When it comes to successfully exploiting the inherent concurrency of a program, the most successful languages are the ones that extract a dataflow for execution. There are mainly two types of languages that achieve this: visual languages and declarative languages [126]. Visual languages allow the developer to draw the dataflow graph in a visual user interface. The nodes are either predefined or can be implemented in other mainstream languages such as Java or C/C++ (as such safety suffers again). This approach was very successful in the domain of signal processing and in tools for integrating data into a data warehouse [32, 57]. When the data is in the data warehouse then the database engine translates SQL queries eventually into dataflow graphs. This is the pre-dominant standard for a database system implementation that exploits all above types of data parallelism [93]. The completely declarative nature of the standard query language (SQL) removes all of these aspects such that the programmer only needs to be a business analysts. The programming and execution model was extended in order to accommodate queries over continuously arriving data, i.e., data streams [40]. At the time of this writing, the state-of-the-art systems

---

<sup>13</sup> Java Virtual Machine

for data processing of continuously-arriving (online) and resident (offline) data compile their query down into a dataflow graph and deploy them across clusters of machines. But the programming models are still very restricted to common SQL constructs that have been studied excessively for parallelism. User-defined functions (/nodes) are generally a bottleneck because the database engines do not have the additional knowledge they need to optimize their execution. But these functions become more and more important because database engines need to crunch more and more semi- or even un-structured data coming from the web or the internet of things (IoT). This seems to be the language challenge for future database systems.

## 1.3 CONTRIBUTIONS AND THESIS STRUCTURE

The list of concurrent and parallel programming abstractions is large and it is hard for a programmer to pick the right ones for a given algorithm. The work in this thesis has the goal to remove some parts of this decision from the developer. We started from the observation that dataflow is the abstraction of choice for a highly parallel execution and addressed problems that are hard to solve with existing parallel programming models. In particular, we contribute to the following fields of (implicit) parallel programming:

**Programming highly-concurrent servers [76] (Chapter 2)** We remove the need of the programmer to manually construct the dataflow graph and suggest to combine functional and imperative programming. This programming model, which we termed *stateful functional programming (SFP)*, allows the compiler to extract the dataflow graph from the server algorithm(/code). The benefits are a concise algorithm definition without concurrency abstractions and a compiler and runtime system that can be exploited to find the most efficient mapping of operators to threads with respect to I/O.

**Fine-grained pipeline parallelism for big data engines [72] (Chapter 3)** We then extend our SFP model with a higher-order function to implicitly exploit pipeline parallelism and show the benefits in the design of big data processing engines: The complex code inside these engines gets more structured and therewith better maintainable. Our evaluation in the Hadoop Map/Reduce system supports our claim that the extracted pipeline parallelism can speed up parts that previously only executed sequentially.

**I/O optimizations for micro-services [77] (Chapter 4)** We present a compiler framework for SFP that implements semantic preserving transformations to minimize the I/O operations of a program without any addition to the SFP model. The programmer writes simple code with blocking I/O calls and leaves the batching, deduplication and parallelism entirely to the compiler. We demonstrate the effectiveness of this approach in the domain of micro-services where network I/O is the dominating source for latency bottlenecks.

**Non-blocking live updates for highly concurrent servers [75] (Chapter 5)** One major benefit of decomposing a large server application into small micro-services is the possibility to efficiently update software while it is running. Live updating a program is generally hard, especially when it executes in parallel. We contribute an algorithm that is based on the abstraction of a dataflow graph to enable live updates even for highly concurrent and heavily loaded server applications. We make the point that a parallel programming language that compiles down to a dataflow execution model can provide efficient live updates with very little effort.

We conclude in Chapter 6 with an outlook on future research directions for our stateful functional programming model.

This is a publication-oriented dissertation and as such each chapter was previously published in a peer-reviewed workshop or conference. At the beginning of each chapter I provide the reference to the corresponding publication. Small paragraphs at the beginning and the end of each chapter position the addressed challenges and the contributions in the context of implicit parallel programming for server-based systems.

# 2 OHUA: IMPLICIT DATAFLOW PROGRAMMING FOR CONCURRENT SYSTEMS

**Prelude** This section covers the paper with the same title co-authored by Christof Fetzer and Pascal Felber that I presented at the International Conference for Principles and Practices of Programming on the Java Platform in 2015 [76]. Our work presents the core ideas of the stateful functional programming model, i.e., the construction of a dataflow graph from an algorithm that composes (stateful) functions.

## 2.1 INTRODUCTION

**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 [80], 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 [14], Scala [101], MultiLisp [102] or Cilk [28] 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 [55] or Akka [110]. Especially on different hardware architectures such as special processors designed for massive parallelism [210], GPUs and FPGAs, the dataflow programming and execution model has

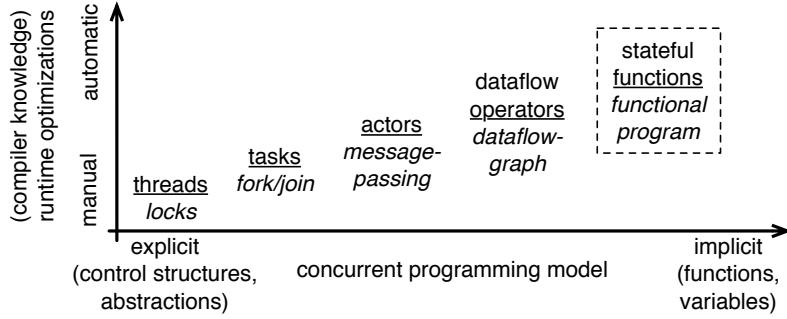


Figure 2.1: A taxonomy of concurrent programming approaches.

become state-of-the-art. In languages such as StreamIt [211] and Intel’s Concurrent Collections (CnC) [33], 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 [171, 209]. 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 2.1 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.

As an example, consider the pipeline for a simple web server depicted in Figure 2.2. With our Ohua dataflow programming model and execution engine, the algorithm is composed in a functional programming style in Listing 5.3.

The actual functionality for accepting incoming connections, reading the data from the socket, parsing the request (see Listing 2.4), loading the requested file from disk, and composing and sending the response is implemented as methods in Java classes.

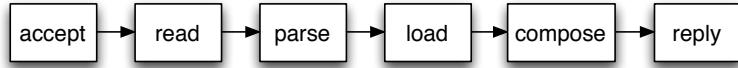


Figure 2.2: A simple web server.

---

```

1 (ohua :import [com.web.server])
2
3 ; classic Lisp style
4 (ohua (reply (compose (load (parse (read (accept 80))))))
5
6 ; or using Clojure's threading macro to improve readability
7 ; (which is transformed at compile-time into the above)
8 (ohua (-> 80 accept read parse load compose reply))

```

Figure 2.3: Ohua-style HTTP server algorithm in Clojure.

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

As also argued in [178], 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 2.2, we present the core concepts of our *stateful functional programming (SFP)* model in Section 2.3 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 2.4). 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 [70].

---

```

1 public class HTTPRequestParser {
2     // a pre-compiled regex as the associated state
3     private Pattern _p = Pattern.compile(
4         "(GET|PUT)\\s([\\w\\.\\/\\w]+)\\s(HTTP/1.[0|1])");
5
6     @Function public String[] parse(String header) {
7         Matcher matcher = _p.matcher(header).find();
8         return new String[] {
9             matcher.group(1), // request type
10            matcher.group(2), // resource reference
11            matcher.group(3)}; // HTTP version
12     }

```

Figure 2.4: Ohua-style parse function implementation in Java.

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 2.5. 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 5.8 and give an outlook on future work (Section 2.7) before we conclude in Section 5.9.

## 2.2 DATAFLOW PROGRAMMING

In flow-based programming (FBP) [163, 61], 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 [24], dataflow execution is naturally tied to functional programming.

In classical dataflow [16, 62], 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 [101]. 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.

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 [123] products are state-of-the-art systems for data integration that are purely based on the FBP concepts. Many algorithms for database systems [63] and data stream processing [40] are also expressed as directed graph structures. The declarative design of network protocols, as for instance in Overlog [151], are dataflow graphs by definition and even the construction of some highly scalable web servers [229] 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 [4] and Cilk [28]. 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.

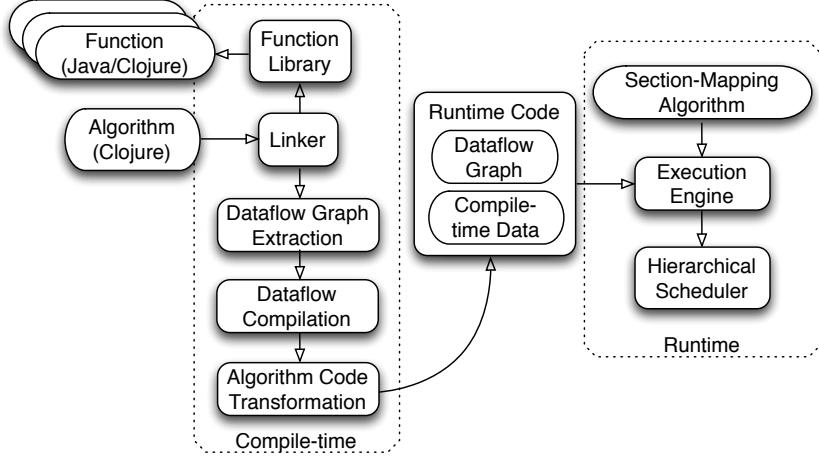


Figure 2.5: Overview of Ohua.

## 2.3 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 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 [115]. Figure 2.5 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 2.4.2) 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 2.4, after first focusing on the programming model and compilation aspects.

### 2.3.1 From Operators to Stateful Functions

In SFP each operator encapsulates a single function tagged with the `@Function` annotation.<sup>1</sup> 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*<sup>2</sup>. In the following, we highlight the difference in the programming model for typical FBP operators of streaming systems like StreamIt [211]. We use StreamFlex [198] for comparison as it is also written in Java. As depicted in Listing 2.6a of Figure 2.6, 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 2.6b and 2.6c of Figure 2.6). 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.

<sup>1</sup>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.

<sup>2</sup>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.

```

1  class FileLoad extends Filter {
2
3      Channel<String> in, out;
4
5      void work() {
6          // explicit channel control
7          String resource = in.take();
8          String contents = null;
9          // load file data from disk (omitted)
10         out.put(contents); }}
```

(a) StreamFlex filter

```

1  class FileLoad {                                1  (ns com.server.FileLoad
2      @Function                               2  (:gen-class :methods
3      String load(String resource) {           3  [[^{:Function} load [String] String]]))
4          String content = null;               4
5          // load data from disk (omitted)     5  (defn -load [this resource]
6          return content; }}                   6  (slurp resource))
```

(b) SF implemented in Java

(c) SF implemented in Clojure.

---

Figure 2.6: StreamFlex dataflow operators/filters vs. Ohua functions implemented in Java and Clojure for file loading.

In a strict sense, functions in SFP resemble the idea of lazy functional state threads [141] 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.

### 2.3.2 Algorithms

The role of algorithms in SFP is to provide an untangled view on the system, free of implementation details. Listings 2.7a and 2.7b in Figure 2.7 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 2.7 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.

```

1  public class WebServer extends StreamFlexGraph {
2
3      Filter a, r, p, l, c, rep;
4
5  public WebServer() {
6      // explicit dataflow graph
7      // construction
8      a = makeFilter(Accept.class);
9      r = makeFilter(Read.class);
10     p = makeFilter(Parse.class);
11     l = makeFilter_Load.class);
12     c = makeFilter(Compose.class);
13     rep = makeFilter(Reply.class);
14     connect(a, r);
15     connect(r, p);
16     connect(p, l);
17     connect(l, c);
18     connect(c, rep);
19     validate(); }
20
21     public void start() {
22         new Synthesizer(accept).start();
23         super.start(); }}
```

(a) StreamFlex graphs

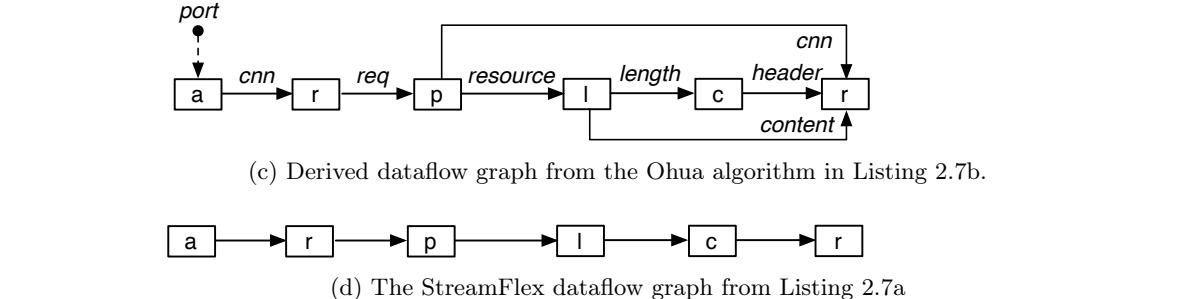


Figure 2.7: StreamFlex dataflow graph construction vs. Ohua algorithm design.

All 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 2.4. 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).

### 2.3.3 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 5.3, and finally a set of explicit data dependencies represented by locals. The result(s) of a function can either be direct (implicit)

---

**Algorithm 1:** Dataflow Graph Extraction Algorithm

---

**Data:**  $\mathbb{A} := (\mathbb{F}, \mathbb{C} := \{if, loop, recur\}, V, D)$  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  $D$ .

```
1 z := zipper( $\mathbb{A}$ ) ;                                     // functional zipper [119]
2 while z := z.next() do                                // depth first traversal
3   f := z.node();
4   if z.up() ≠ ∅ then
5     // For simplicity we assume the simple case where no control structure is the root.
6     create-operator(f);
7   else
8     t := z.up().leftmost().node();
9     if f ∈  $\mathbb{F}$  then
10       create-operator(f);
11       create-dependency(f.id, t.id);
12     else if f = if then
13       s := create-operator('cond');
14       f.id := s.id;
15       m := create-operator('merge');
16       Turn condition into executable and attach to s;
17       create-dependency(m.id, t.id);
18     else if f = loop then
19       create-operator('merge');
20       create-dependency(f.id, t.id);
21     else if f = recur then
22       l := Find the next loop statement in z.path;
23       create-dependency(f.id, l.id);
24     else if f ∈  $\mathbb{V}$  then                                // see Section 2.4.2
25       register-env-var(f.node());
26     else if f ∈  $\mathbb{D}$  then
27       create-dependency(f.origin.id, t.id);
28     end
29   end
30 end
```

---

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 2.7b provides the most explicit implementation of the server algorithm, which translates into the dataflow graph of Figure 2.7.

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 2.8 we depict a variant of our simple web server algorithm that destructures the results of `parse` to 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 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

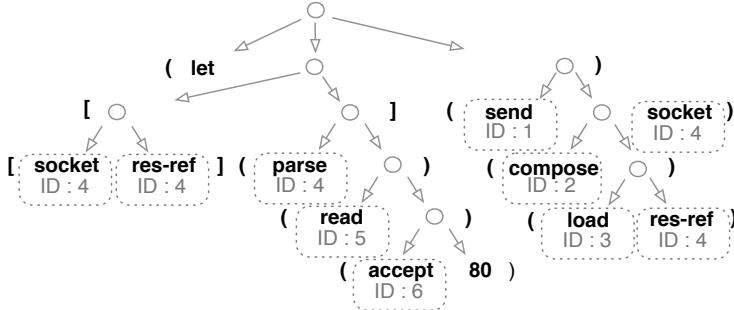


Figure 2.8: Tree representation of the server algorithm in Clojure.

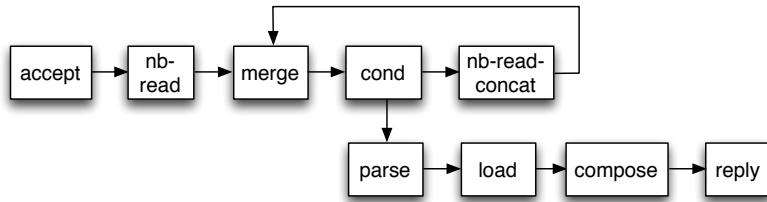


Figure 2.9: `recur` statements translate into feedback-edges.

Listing 2.12, 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 2.10 declares a version of the web server that implements a non-blocking read on the accepted connections via a loop.

Note the implementation of the non-blocking read functions in Listing 2.11. It uses inheritance to share the code for reading data and extending it by concatenating the already read data to the current data read.

### 2.3.4 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 5.3). 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

```

1 (ohua
2   (reply (compose (load (parse
3     (loop [[read-data cnn] (nb-read (accept 80))]
4       ; stop on blank line
5       (if (.endsWith read-data "\n\n")
6         [read-data cnn]
7         (recur (nb-read-concat cnn read-data)))))))

```

Figure 2.10: Loop statement to implement non-blocking reads.

```

1  class NonBlockingRead {
2      private ByteBuffer b = ByteBuffer.allocate(90);
3
4      @Function
5      Object[] nbRead(SocketChannel cnn) {
6          cnn.configureBlocking(false);
7          int count = cnn.read(b);
8          b.flip();
9          String data = Charset.defaultCharset().decode(b);
10         return new Object[]{ data, cnn };
11     }
12 }
13
14 class NBReadConcat extends NonBlockingRead {
15
16     @Function
17     Object[] nbReadConcat(SocketChannel cnn, String read) {
18         return new Object[]{ read + super.nbRead(cnn)[0], cnn };
19     }
20 }
```

Figure 2.11: Non-blocking read with two functions.

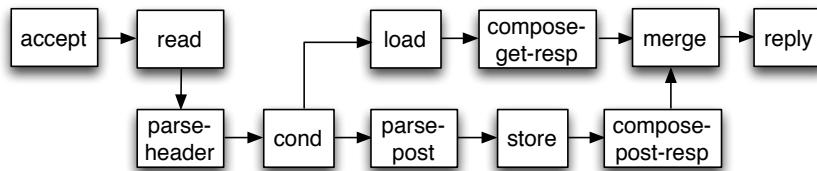


Figure 2.12: Data parallelism opportunity via conditionals.

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.

### 2.3.5 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 2.13 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 2.12. The condition splits the dataflow into two branches, 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.

```

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

Figure 2.13: Conditional statement on the resource location.

```

1  (ohua
2    (let [cnn (accept 80)]
3      (|| 3 cnn (-> read parse load compose reply)))
4
5    ; macro expanded
6    (ohua
7      (let [cnn (accept 80)]
8        (let [[one two three] (balance-data cnn)]
9          (-> one read parse load compose reply)
10         (-> two read parse load compose reply)
11         (-> three read parse load compose reply)))

```

Figure 2.14: Load balancing via a simple macro.

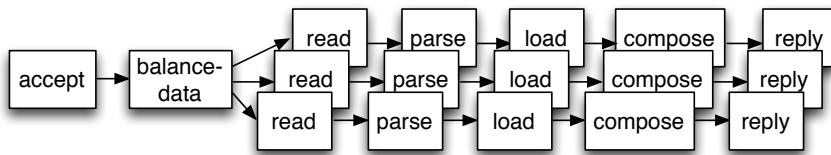


Figure 2.15: Opportunity for 3-way parallel request processing.

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 2.14 uses a macro (`||`) to create the opportunity to handle requests in a 3-way parallel fashion on the foundation of the `let` special form.

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 2.15. 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 2.4).

## 2.4 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 [145].

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.

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

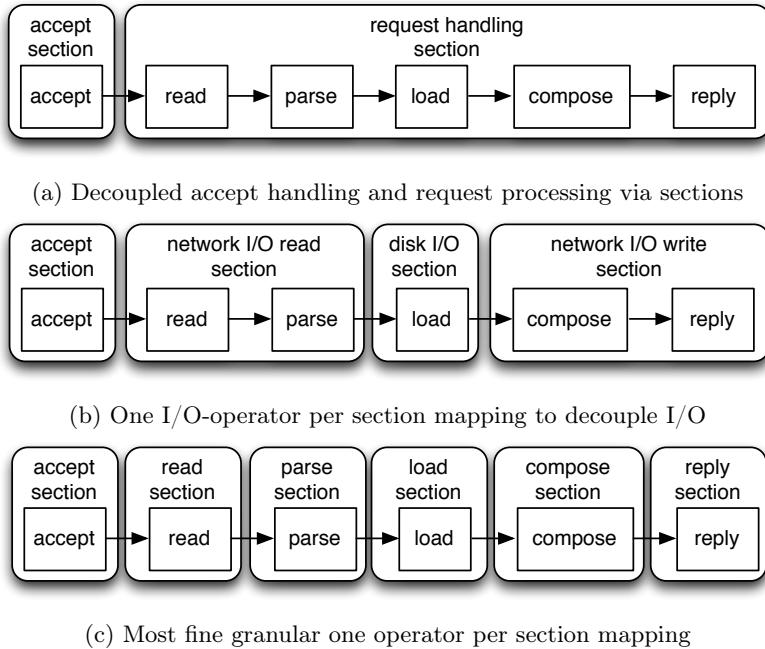


Figure 2.16: Three section mappings for the basic web server.

```

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

```

Figure 2.17: 'Acpt-req' and '1 I/O op/sec' section mappings.

identified as a vital parameter as it must amortize the associated management costs [103]. 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 [205, 187]. In these approaches, the lowest degree of parallelism is given by the number of operators. Ohua targets larger graphs, potentially much larger than 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.

#### 2.4.1 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.

As shown in Listing 2.17, 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 2.16a decouples the tasks of accepting new connections from the processing of requests, while the mapping in Figure 2.16b 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 2.16c, assigns each operator to its own section. Future work extends this 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.

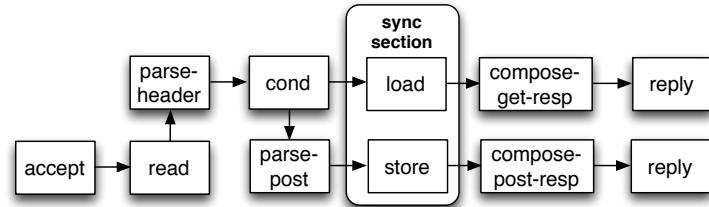


Figure 2.18: Resource access synchronization via sections.

```

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) registry))))))

```

Figure 2.19: A variable map as a cache in Ohua.

#### 2.4.2 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 2.19 is a variation of our extended web server that handles HTTP POST and GET requests.

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 2.18. 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.

## 2.5 EVALUATION

Ohua targets coarse-grained parallelism but the granularity of tasks in typical benchmarks even for pipeline parallelism is fine-grained [27]. 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

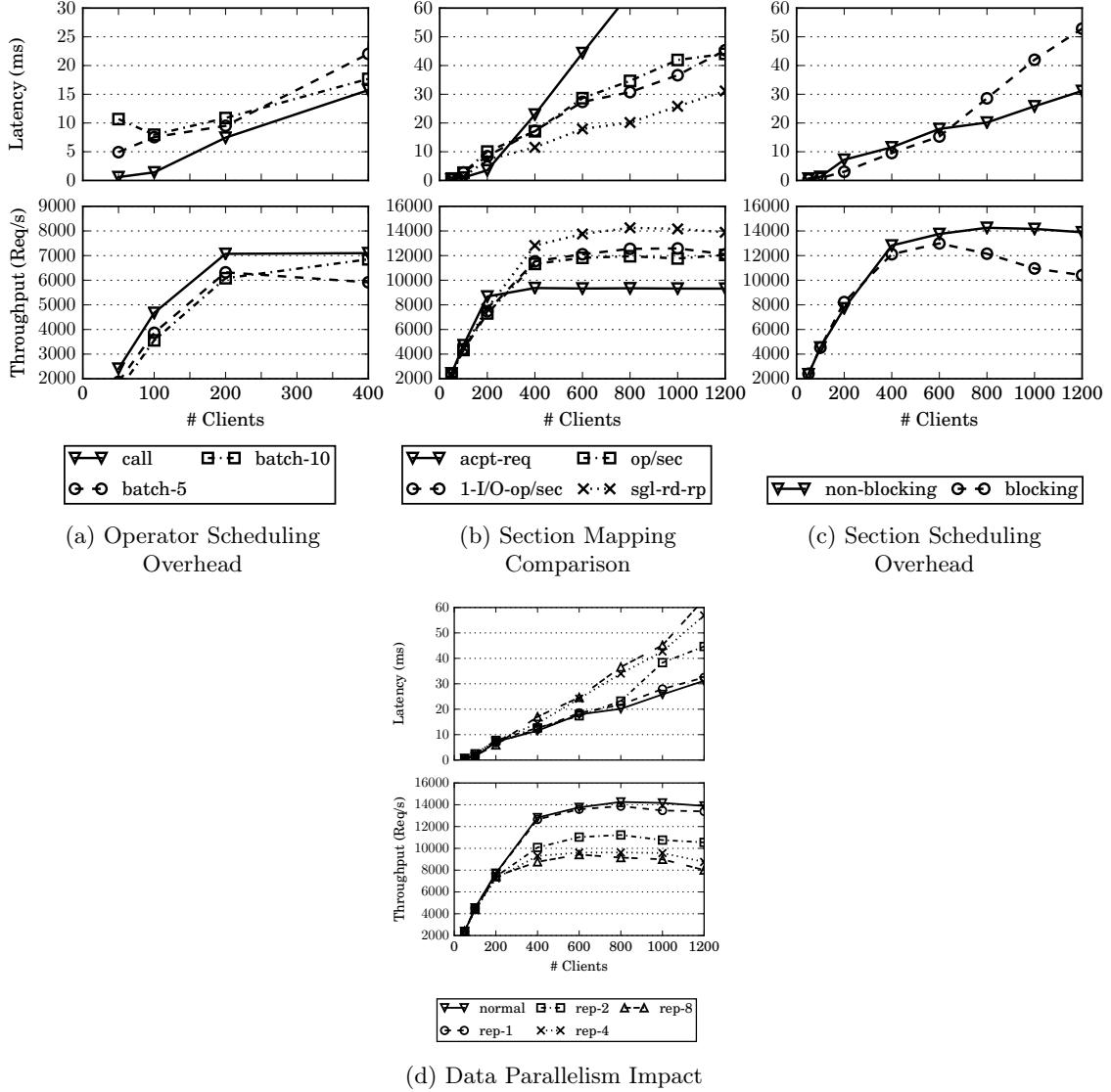


Figure 2.20: Latency and throughput measurements to find the optimal scheduling, section and parallelism configuration for our HTTP server.

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<sup>3</sup> as well as the web server<sup>4</sup> along with instructions for running the benchmarks are freely available.

### 2.5.1 Pipeline Parallelism

In order to find the optimal configuration for our Ohua-based web server (see Figure 2.2) 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 performance. 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 2.20a 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 2.16c, 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 2.20b 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 2.22 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 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 2.20c. 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.

---

<sup>3</sup><https://bitbucket.org/sertel/ohua>

<sup>4</sup><https://bitbucket.org/sertel/ohua-server>

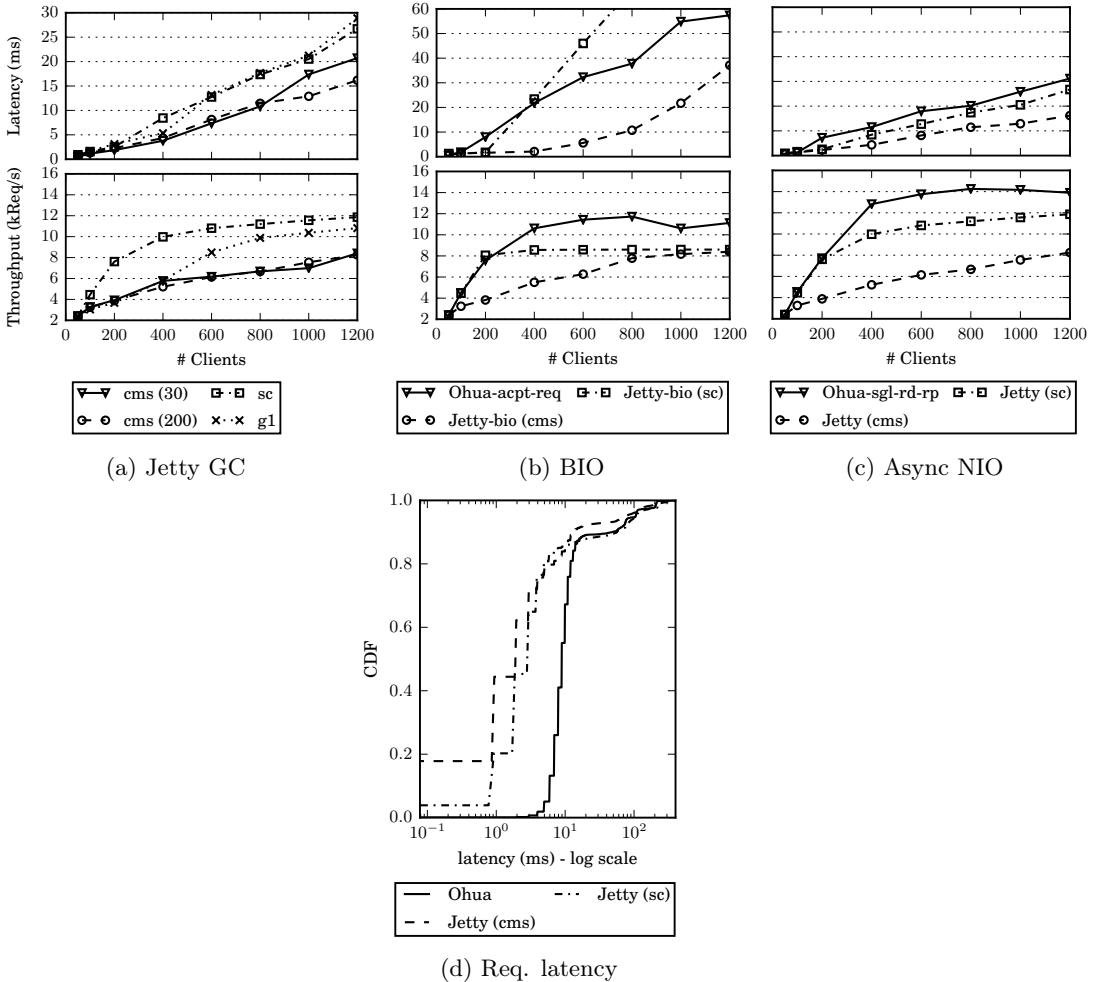


Figure 2.21: Comparison of Jetty against the simple web server implemented in Ohua.

### 2.5.2 Data Parallelism

We next investigate the potential for processing requests in a data parallel fashion using the `balance` macro from Listing 2.14. 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 2.20d 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.

### 2.5.3 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 2.21. 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 2.21a 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

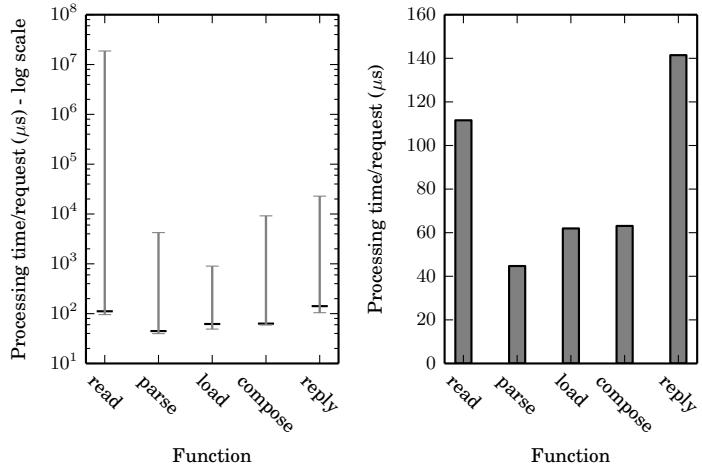


Figure 2.22: Average, minimum, and maximum processing times per request for each of the web server functions.

(default). The results show that CMS performance does not depend on the number of threads to be inspected.<sup>5</sup> The lowest latency is achieved by CMS while the parallel scavenger (sc), a stop-the-world GC implementation, provides the highest throughput. The new G1 (garbage first) collector is not yet located in between the scavenger and the cms performance-wise, as one would actually expect. It is on par in terms of latency with scavenger 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 2.21b 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 scavenger.

Results for NIO, shown in Figure 2.21c, 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 scavenger, 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 2.21d, 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.

#### 2.5.4 Energy Comparison

Next, we want to validate claims by some researchers [193, 218, 122] 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 2.23 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 2.23a show that executing Jetty without a cache is far more energy efficient and scales better to more intensive load situations.

<sup>5</sup>Ohua adapts the thread count to the number of sections automatically.

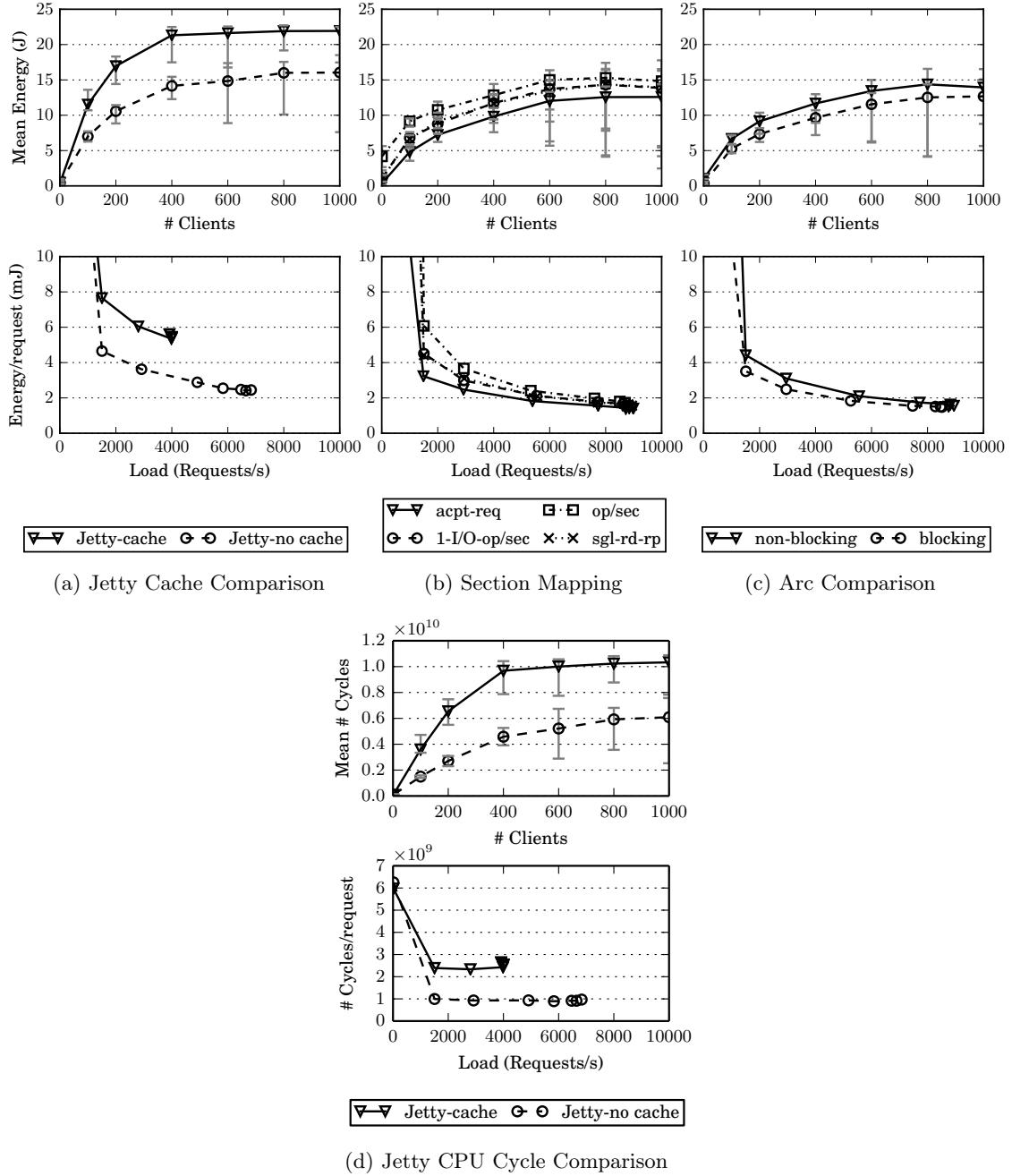


Figure 2.23: Energy consumption for Jetty and our HTTP server implemented in Ohua.

This is due to the fact that Jetty uses the `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 2.23d, 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 2.20b. The mean energy consumption in Figure 2.23b 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

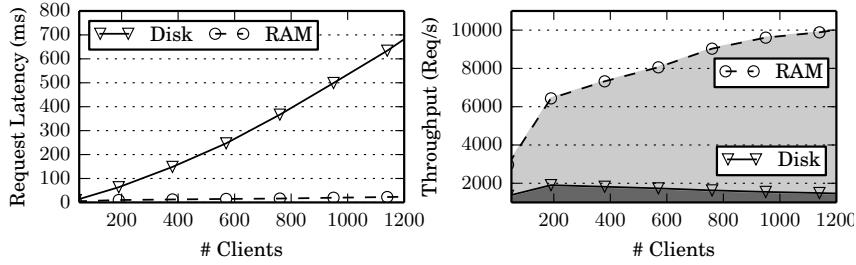


Figure 2.24: Latency and throughput for concurrent requests on feeds (512 Bytes) residing in RAM and articles (20 kB) located on disk.

```

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

```

Figure 2.25: Conditional Statement on the Resource Location.

graphs in Figure 2.23a with those in Figure 2.23b 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 2.23c, 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.

### 2.5.5 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 2.25 defines a condition that dispatches requests according to the location of the requested resource.

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 2.24 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.

## 2.6 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 [178]. 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 [101]. Dataflow languages such as Lustre [99] and Lucid [221] 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 [103]. Haskell falls back to explicit language constructs to enable coarse-grained parallelism [156]. Manticore relies on message-passing to achieve similar functionality [80]. Dataflow languages, just as the dataflow monad of Haskell, target fine-grained data parallelism. This is also the case for StreamIt [211], 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 [169] and the syntax of Haskell to address special non-von-Neumann architectures. In contrast, Cilk [28] 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 [187]. 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 [205]. 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 [81] 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.

## 2.7 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) [59] 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 2.26.

In comparison, the implementation of the map and reduce task drivers in Hadoop [10] respectively take  $\sim 1,800$  and  $\sim 3,000$  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 [192]. 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.

```

1 ; map task
2 (ohua
3   (-> read-dfs parse-chunk map sort serialize compress store-disk))
4
5 ; reduce task
6 (ohua
7   (-> fetch-map-results deserialize merge reduce serialize compress store-dfs))

```

---

Figure 2.26: Dataflow in the Map-Reduce programming model.

**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 [142]. 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 [58]. The presence of a scheduler in Ohua programs allows for far more research on scheduling algorithms in many different application domains.

## 2.8 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.

**Postscript** The presented programming model removes the burden from the programmer to construct the dataflow graph manually. It involves the design of a compiler and a runtime system that can perform optimizations. In this paper, we show how our programming model helps to make the code concise. Our runtime system allows to create classical server designs easily without the need to change a single line of code.

However, the programming model does not yet fully abstract from the notion of a dataflow graph. The developer has to understand that nodes without incoming edges are re-executed such as the `accept` function that accepts client connections. This

diverges from the normal programming model because the algorithm does not express this aspect explicitly and makes it harder to understand for the developer. Furthermore, it would require a special type of stateful function to notify the runtime system when no new input is being retrieved anymore. A web server is meant to run forever but for example the `map` and `reduce` tasks listed in Figure 2.26 should stop after the last line was read from the file. The next paper addresses these missing control constructs in the algorithm language to re-implement Hadoop's Map task and exploit pipeline parallelism.

# 3 SUPPORTING FINE-GRAINED DATAFLOW PARALLELISM IN BIG DATA SYSTEMS

**Prelude** This section covers the paper with the same title co-authored by Justus Adam and Jeronimo Castrillon that I presented at the International Workshop on Programming Models and Applications for Multicores and Manycores in 2018 [72]. This work extends the stateful functional programming model with its first higher-order function: `smap`. It is a version of the well-known higher-order `map` function. In combination with the concept of the stateful functions, `smap` enables implicit pipeline and task-level parallelism. More importantly, it removes the remaining dataflow aspect from the stateful functional programming model. Using `smap`, the developer has clear semantics of the algorithm’s execution and does not need to be aware of special functions. All stateful functions called in an algorithm have the same well-known application semantics. Towards a more formal description of these semantics, the paper defines the syntax of the language and a translation of the language constructs into dataflow.

## 3.1 INTRODUCTION

Over the last decade big data analytics became the major source of new insights in science and industry. Applications include the identification of mutations in cancer genome [124] and the tracking of other vehicles around an autonomously driving car. The big data systems (BDS) that enable such analyses have to be able to process massive amounts of data as fast as possible. In order to do so, current BDS apply coarse-grained data parallelism, i.e., they execute the same code on each core of the nodes in a cluster on a different chunk of the data. As such, the application is said to scale with the number of cores in the cluster. However, not every aspect of a big data application exposes data parallelism. Whenever this is the case, current big data systems fail to scale.

### 3.1.1 Scalability Issues of Big Data Systems

A typical big data analysis program assembles a set of predefined operations and applies them to the data. Execution proceeds in multiple phases where each phase applies a part of the program to the data that was output by a previous one. Afterwards, the results are redistributed among the

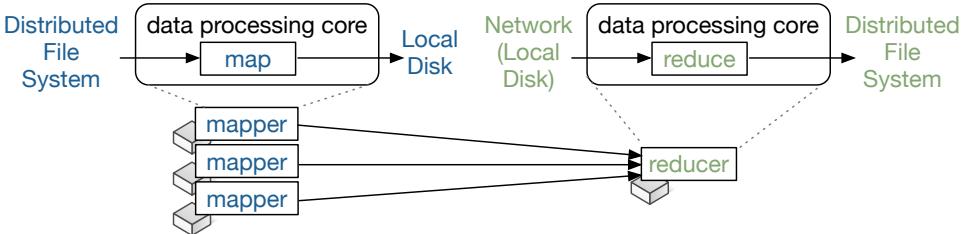


Figure 3.1: Data processing cores in Hadoop’s MapReduce.

nodes in the cluster to compute the next phase. For example, the famous MapReduce programming model defines exactly two phases as shown in Figure 3.1: a map and a reduce phase [59]. The map phase is data parallel by definition but the data parallelism of the reduce phase depends on the application. For example, in data analytics queries, the join operation for two tables can not be performed in a data parallel way (when the input data is not partitioned). In such a case, a single node receives all results from the map phase and becomes the throughput bottleneck. Even if the reduce phase runs in a data parallel fashion, the load across the reducers can only be balanced evenly by making assumptions on the map output. These assumptions are hard to estimate correctly such that the load is often spread unevenly decreasing data parallelism.

BDS have been traditionally designed to execute applications in a massive coarse-grained data parallel way across a cluster of machines. The assumption was that applications would process large amounts of simply structured data, e.g., text, that is fast to serialize and deserialize, i.e., transforming them to bytes (and its inverse operation on the receiver side). This setup led to the common belief that network I/O, instead of computation, is the performance bottleneck in these systems.

Only recently, researchers have shown that I/O is not always the limiting factor for performance [176]. Over the last years, big data applications have grown in complexity and use much more complex data structures especially in the area of data analytics. At the same time, network hardware for big data appliances improved, reaching transfer rates above 40 GB/s [127]. Authors in [216] benchmarked the current state-of-the-art BDS Apache Spark and Apache Flink in a high bandwidth cluster setup. The results show that reduce operations do not profit from modern multi-core architectures since their cores do not take advantage of fine-grained parallelism. Consequently, the data throughput does not increase for faster network devices, i.e., it does not scale with the network.

To better exploit new hardware, the design of BDS must be revisited. This cannot be done with local optimizations for individual steps in the data processing core, but requires fundamental design changes [216]. Redesign is non trivial due to the complexity of the code bases of state-of-the-art BDS, e.g., with over 1.4 million lines of code in Hadoop MapReduce (HMR). Approaching this task with common parallel programming means, like threads, tasks or actors and their respective synchronization via locks, futures or mailboxes, inevitably increases code complexity even further. As a result, these systems become even harder to reason about, maintain and extend. We believe that a much more concise and scalable redesign can be achieved with new programming abstractions that enable a compiler to prepare the code for an efficient parallel execution at runtime. This paper represents first steps in this direction.

### 3.1.2 Contributions

In this paper, we present a rewrite for the processing core of current big data systems to increase data throughput, effectively improving scalability with new hardware. Our rewrite uses an implicit parallel programming language to provide concise code that is easy to maintain. The corresponding compiler transforms the program into a dataflow graph that the runtime system executes in a pipeline and task-level parallel fashion across the cores of a single cluster node. Hence, our approach extends the coarse-grained data parallelism in BDS with fine-grained pipeline and task-level parallelism inside the data processing cores to resolve the throughput bottlenecks.

The contributions of this paper are as follows:

```

1  public class Mapper<KEYIN, VALUEIN,
2      KEYOUT, VALUEOUT> {
3      /* The default implementation
4          is the identity function. */
5      protected
6      void map(KEYIN key, VALUEIN value,
7          Context ctxt) {
8          ctxt.write((KEYOUT) key,
9                  (VALUEOUT) value);
10     }
11
12     public
13     void run(Context ctxt) {
14         while (ctxt.nextKeyValue())
15             map(ctxt.getCurrentKey(),
16                 ctxt.getCurrentValue(),
17                 ctxt);
18     }
19
20     private[spark] class
21     ShuffleMapTask(partitionId: Int,
22                     partition: Partition)
23     extends Task[MapStatus] {
24
25         override
26         def runTask(ctxt: TaskContext)
27             : MapStatus = {
28             /* Deserialization and init of
29                 variables omitted for brevity. */
30             var writer: ShuffleWriter[Any, Any] =
31                 manager.getWriter[Any, Any](
32                     dep.shuffleHandle,
33                     partitionId, ctxt)
34             writer.write(rdd.iterator(partition, ctxt)
35                         .asInstanceOf[
36                             Iterator[_<:>Product2[Any, Any]]])
37             writer.stop(success = true).get}
38

```

(a) Hadoop

(b) Spark

```

1  public class DataSourceTask<OT> extends AbstractInvokable {
2      private InputFormat<OT, InputSplit> format;
3      private Collector<OT> output;
4
5      @Override public void invoke() throws Exception {
6          OT reuse = serializer.createInstance();
7          while (!this.taskCanceled && !format.reachedEnd()) {
8              OT returned;
9              if ((returned = format.nextRecord(reuse)) != null)
10                  output.collect(returned);
11          }
12      }
13

```

(c) Flink

---

Figure 3.2: The data processing cores of Hadoop MR, Spark and Flink are all based on the abstraction of a context.

1. We contribute an analysis of the code base of HMR, Spark and Flink. This analysis shows that the different data processing cores are structurally equivalent and thus suffer the same scalability issues. Further, our study reveals design patterns that indicate pipeline-parallelizable code.
2. As a case study, we change the HMR data processing core and present four different rewrites using an implicitly parallel language. Our rewrites are minimally invasive and reuse existing code. The rewritten code is concise and free of concurrency abstractions.
3. We compare our data processing core to the original HMR implementation using parts of the TPC-H database benchmark. Experimental results report throughput speedups of up to 3.5x for compute-intensive configurations.

The rest of the paper starts with the analysis of state-of-the-art big data systems in Section 3.2. Afterwards, we give a brief introduction into implicitly parallel programming with Ohua (Section 3.3) and then present the HMR rewrites in Section 3.4. Section 3.5 compares our rewritten data processing core to the original one. Finally, we review related work in Section 4.4 and conclude in Section 3.7.

## 3.2 THE CORE OF BIG DATA PROCESSING

In this section, we study the three big data systems with a focus on the programming style of their data processing cores. This analysis serves to identify common code patterns across the different BDS that lend themselves well for a pipeline and task-level parallel execution. With this analysis we also provide concrete reasons for the scalability issues discussed above (cf. Section 3.1.1) and reported in [216]. This has only been supported by experimental observations but not by an in-depth analysis of the code structure. The first part of this section lists and explains the code of the data processing cores in these BDS. Then, Section 3.2.2 analyzes promising code patterns to exploit other forms of parallelism.

### 3.2.1 Data Processing: Code Study

We investigate the three most-widely-used systems in todays big data domain, Hadoop MapReduce (HMR), Spark and Flink.

#### Hadoop MapReduce

The MapReduce programming model consists of two functions: a `map` function that pre-processes the data and a `reduce` function that performs an aggregation [59]. The `map` function is executed in a data parallel fashion on every chunk of the input data. The results are partitioned and dispatched to nodes that execute the `reduce` code. The number of `reduce` tasks depends on the number of partitions defined by the application.

Figure 3.2a lists the data processing core of the framework in the `run` method. The main concept is the abstraction of a context through which data is retrieved and emitted. The `Context` class implements an iterator-like interface that is used in the `run` method to drive the input side, i.e., retrieve the data. In case of the `Mapper`, it retrieves one key-value pair at a time and passes it to the `map` function (Lines 14–17). The implementation of the `Reducer` looks almost identical, except for the fact that it receives a list of values with each key. We omit it here for brevity and concentrate in the rest of the paper on the `Mapper` keeping in mind that our rewrites apply in the same way to the `Reducer`. The `map` function emits data via the `context`, driving the output side of the processing in the task (Lines 8–9). The `context` implementation uses the abstractions of the `InputFormat` and `OutputFormat` to interface with the various big data storage systems such as HDFS [197] and HBase [42, 83].

#### Spark

Spark [231] provides a new API which is heavily based on Scala and much closer to the original higher-order functions `map`, `reduce` and `filter`. In Spark, these functions, also called transformations, are applied to Resilient Distributed Datasets (RDD), the key abstraction in Spark. RDDs are immutable and therefore transformations create new RDDs. Like HMR, Spark partitions the input data and the intermediate results (if possible) to apply coarse-grained data parallelism.

Internally, Spark distinguishes among two types of tasks: the `ShuffleMapTask` and the `ResultTask`. All transformations are executed on a `ShuffleMapTask`, except the one performed on the final RDD in the program. A `ShuffleMapTask` first performs transformations and finally shuffles the result data over the network. Lines 15–18 in Figure 3.2b list the corresponding implementation code. Spark groups transformations to compose a pipeline that executes in a single task. Each transformation applies a function to all key-value pairs of the partition/RDD. While doing this, it uses the concept of an iterator to navigate over the whole set of key-value pairs in a partition. Spark chains iterators where each iterator performs one transformation and passes the results on to the next. The code that uses the outermost iterator instance moves one key-value pair through the whole transformation pipeline before acquiring the next. In the `ShuffleMapTask`, the `writer` shuffles the results over the network. This on-demand processing of data actually implements a lazy evaluation strategy such that actions like `take(n)` do not process the whole data but only the first  $n$  records requested. The `ResultTask` either sends the results back to the program executing the Spark query or persists them in the underlying storage system.

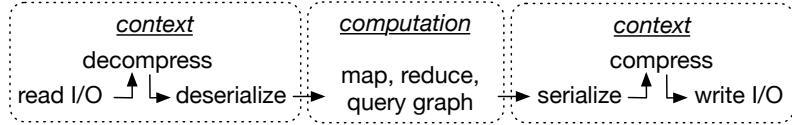


Figure 3.3: The data processing core of all three BDSS.

## Flink

HMR and Spark were designed to perform computations over data stored in a distributed storage system, called batch processing. However, big data applications often gather and process data continuously which is referred to as data streaming. Therefore, Flink [35] extends Spark’s programming model with a notion of time which in turn requires to incorporate stateful transformations into the runtime system. In order to do so, Flink builds on top of the dataflow execution model that is popular for massively parallel processing (MPP) in distributed database systems [93]. It derives a dataflow graph from a program where each function call is represented as a node, referred to as *operator*. An operator can be replicated for a data parallel execution if its private state and its incoming data streams can be partitioned.

Similar to Spark, Flink groups pipelines of operators working on the same data partition together into a chain of operators and executes them in a single task. Flink uses a push-based model to drive the pipeline while Spark builds on a pull-based approach. Therefore, the first operator in the pipeline retrieves the data and drives the rest of the pipeline. We list the code for the `DataSourceTask` in Figure 3.2c. Like HMR, it uses an `InputFormat` abstraction to load data from distributed storage and pushes every element into a chain of collectors, i.e., operators (Lines 11–17). Each collector applies a reduction function to the pushed element and stores the result into its private state. A collector emits its private state to the next collector in the chain only when all values were processed and `close` was called by the previous collector.

### 3.2.2 Analysis

We now analyze these three implementations in terms of execution models, design patterns and their applicability to pipeline and task-level parallelism.

#### Dataflow Inside

All three systems are dataflow systems. They all build a directed acyclic graph (DAG) to represent the computation. The state of the nodes in the DAG is partitioned and replicas are created for each partition to introduce data parallelism. HMR implements a very coarse-grained system that allows to write a DAG with only the two nodes, `map` and `reduce`, that are executed in a massively parallel way. Higher-level systems and languages such as Hive (SQL) [214], Pig (Pig Latin) [173, 85] and IBM’s SystemML (R) [86] derive more fine-granular DAGs and use HMR as a foundation to execute them. Spark has such a program analyzer that automatically converts the program into a DAG built-in. This allows Spark to inspect the dependencies and derive data pipelines for RDDs which are then run in a data parallel fashion across the RDD partitions. HMR with Hive or Pig, Spark and Flink analyze the DAG, use certain metrics to fuse nodes and map them onto tasks which are then executed in a single JVM process [117]. Hence, the dataflow execution model can be seen as the de-facto standard for big data processing engines.

However, the dataflow processing model has not been used to execute the data processing pipeline inside a task. Instead, parallelism is exploited very sparsely. HMR uses a dedicated thread on the output-side of the mapper in order to buffer and combine records before writing them to local disk. Furthermore, the reducer uses multiple threads to retrieve the map outputs for its partition over the network in parallel. The rest of the processing is performed sequentially. While HMR executes each task on a new JVM process, Spark and Flink use a single JVM per cluster node to run tasks in parallel which speeds up task startup times significantly. However, for reduce tasks which are not executable in a data parallel fashion, even Spark and Flink do not benefit from a multi-core machine.

## Design Patterns and Parallelism

Interestingly, although only reductions are purely sequential, a lot more code is executed sequentially. This is due to the composition of the data processing pipeline shown in Figure 3.3 which always consists of at least the following steps: retrieval of the data from I/O (disk or network), deserialization, computation, serialization and writing the data to I/O (disk or network). Decompression and compression can be dynamically added via configuration parameters. Furthermore, in HMR (Hive, SystemML), Spark and Flink, the computational part, i.e., the actual operations applied to the data, often encompasses more than a single function. To enable this dynamic composition of the processing core, the code uses the iterator and observer design patterns. The iterator allows chaining transformations in Spark while the observer pipelines operators in Flink. Furthermore, all three BDS integrate the various file formats to access the vast landscape of big data storage systems. For this, they all rely on the `InputFormat` and `OutputFormat` abstractions that were defined in earlier HMR versions. The `InputFormat` defines an iterator while the `OutputFormat` uses an observer, i.e., data is pulled from and pushed to I/O. The iterator and observer design patterns are duals [160] and we use them as an indicator in the code to derive parallelism. Both enforce encapsulation of state inside the concrete class. Hence, in between these two sequentially executing duals there exists a pipeline parallel execution which is semantically equivalent. Furthermore, we found that the (de-)compression and (de-)serialization steps can be executed in a task-level parallel fashion for a key and its value(s).

In the rest of the paper, we investigate how an implicitly parallel language would influence the code structure and scalability of such a data processing core. Although, we port only HMR to Ohua, this section made the case that Spark and Flink would benefit from the same rewrites.

## 3.3 IMPLICIT PARALLEL PROGRAMMING IN OHUA

This section introduces the main concepts of Ohua’s language and runtime system using the simplest HMR rewrite as an example.

### 3.3.1 Algorithms and Stateful Functions

The Ohua programming model makes the following fundamental observation: An application is defined in terms of an *algorithm* that comprises smaller independent and self-contained functional building blocks. For example, in the HMR system, one algorithm defines the data processing for a mapper and another for the reducer. The associated functional building blocks include the functions to deserialize and decompress a single record in the data set. In Ohua, these functions may access their private state throughout the whole computation and are therefore called *stateful functions*.

We list the most coarse-grained algorithm implementation for the mapper in Figure 3.4a. We define the algorithm inside a Clojure function called `coarse`, because Ohua is an embedded domain specific language in Clojure<sup>1</sup>. We invoke this algorithm inside the `run` method of our new `Mapper` (see Figure 3.2a). The algorithm uses a higher-order map function (`smap`) to define a computation (Lines 8–12) that is applied to each of the records (Lines 5 and 13) in the data chunk to be processed by this `Mapper`. Note that we define this computation in terms of another algorithm named `compute-and-output`, i.e., algorithms may call other algorithms. In the following, we refer to this language as the *algorithm language*.

The overall algorithm uses the application-defined `Mapper` and HMR’s `Context` abstraction. The iterator that is defined at Line 5 uses the `Mapper$Context` to retrieve one record at a time. The `hmr-map` function (Line 8) emits the list of key-value pairs produced for a single `map` invocation on the user-supplied `Mapper`. The `output` function (Line 10) passes one key-value pair at a time to the `Mapper$Context` and executes the output side of the pipeline. As an example of a functional building block for this algorithm, we list the implementation of the `output` function in Figure 3.4b.

---

<sup>1</sup> Clojure primer: Function abstraction such as `(defn fun [arg1 arg2] code)` is equivalent to `public Object fun(Object arg1, Object arg2){ /*code*/ }` in Java. Function application such as `(fun 4 5)` is equivalent to `fun(4, 5)`.

```

1 (defn coarse
2   [^org.apache.hadoop.mapreduce.Mapper$Context reader
3    ^org.apache.hadoop.mapreduce.Mapper mapper
4    ^org.apache.hadoop.mapreduce.Mapper$Context writer]
5   (let [records-on-disk (new InputIterator reader)]
6     (ohua
7      (smap
8        (algo compute-and-output [ [line content] ]
9          (let [kv-pairs (hmr-map line content mapper)])
10         (smap
11           (algo output-side [ [k v] ] (output k v writer))
12             kv-pairs)))
13       records-on-disk)))

```

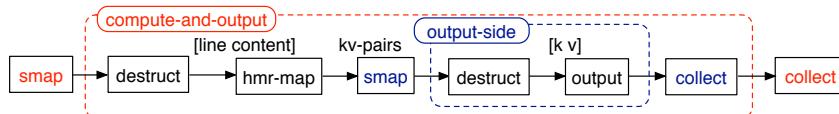
(a) Coarse-grained algorithm for the data processing in HMR’s mapper.

```

1 public class Output {
2   @defsfn public
3   void output(Object key, Object value, Context ctxt) {
4     ctxt.write(key, value); }

```

(b) Stateful function `output`.



(c) The dataflow graph that Ohua derives.

Figure 3.4: The data processing core of HMR implemented in Ohua. Figure 3.4a lists the code of the algorithm defined in the `Mapper` implementation of Figure 3.2a. The algorithm code is concise, i.e., free of clutter and concurrency constructs, and reuses the original implementation. For example, the stateful function `output` is written in Java (see Figure 3.4b) emits the key and its value via HMR’s context abstraction. From the algorithm code, Ohua derives the dataflow graph in Figure 3.4c.

It is implemented as a Java class and identified as stateful function via the `@defsfn` annotation<sup>2</sup>. A call to `output` in a Java program differs from one in an Ohua algorithm:

Java	Ohua
<code>o.output(k, v, writer);</code>	<code>(output k v writer)</code>

The Java call needs an instance `o` of class `Output` which defines the state that is accessible inside the function. At runtime, the instance encapsulates this (private) state while the surrounding Java program needs to create and maintain it. In Ohua this instance is implicit, i.e., it is created and managed by the runtime system. For each function call Ohua creates such an instance once and reuses it throughout the entire computation. When the function performs side-effects against a field of the class in one call then these changes are visible to the succeeding calls. An example for such a state is the dictionary used in many compression algorithms. Note that the tried to keep our rewrite as minimal as possible without requiring edits to the HMR code base. As such, the class `Output` does not extend from `Context` but delegates to a context instance it receives from the surrounding program. This instance counts towards the state of `output`, including all transitive references such as for instance to the compression dictionary.

We define the Ohua language (constructs relevant to this paper) in Figure 3.5. The syntax is in line with that of Clojure. The language features variables, abstractions and applications for algorithms and lexical scoping of variables. The central contribution is the application of

<sup>2</sup>Ohua additionally supports stateful function implementations in Scala and Clojure.

*Terms:*

$t ::=$	$x$	variable
	$(\text{algo} [x] t)$	abstraction
	$(t t)$	application
	$(\text{let} [x t] t)$	lexical scope (variable binding)
	$(f x_1 \dots x_n)$	apply JVM function $f$ to $x_1 \dots x_n$ with $n \geq 0$
	$(\text{if} t t t)$	conditionals
	$(\text{seq} t t)$	sequential evaluation order (side-effect dependency)

*Values:*

$v ::=$	$o \in V_{\text{JVM}}$	JVM value
	$(\text{algo} [x] t)$	abstraction
	$[v_1 \dots v_n]$	list of $n$ values

*Predefined Functions:*

$(\text{smap} (\text{algo} [x] t) [v_1 \dots v_n])$  apply abstraction to list

Figure 3.5: Definition of Ohua’s algorithm language.

potentially stateful functions defined on the JVM in either Java, Scala or Clojure to variables. Program evaluation is entirely data driven just as known from any other functional language. To control the evaluation otherwise, the developer can either use conditionals or the higher-order function `seq`. With `seq`, the developer can express dependencies on I/O side-effects such as for example writing to disk or network communication. In the next section, we show that HMR relies a lot on such dependencies when reporting the mapper’s progress, e.g., the bytes written to HDFS. A value is either an object  $o$  from the domain  $V_{\text{JVM}}$ , i.e., JVM objects emitted by stateful functions, an algorithm abstraction or a list of values. We also redefine the function `smap`, Ohua’s variant of the well-known higher-order function `map`.

### 3.3.2 Dataflow-based Runtime System

Despite incorporating state, Ohua can derive a parallel execution. To this end, the compiler first transforms the program expression into a form that binds each result of a stateful function call to a variable and then lowers this expression into a dataflow graph. In Figure 3.6a, we define the terms of our dataflow representation. The basic constructs of our dataflow representation are nodes, edges and ports. A node retrieves data via its input ports, performs a computation (step) and emit the result via its output ports. The data travels through the graph via edges where an edge originates at one output port and terminates at an input port. An output port may have multiple outgoing edges to pass the value to more than a single node while an input port always only connects a single incoming edge. We classify nodes according to their correspondence of retrieved and emitted data. A  $1-1$  node retrieves at most one data item from any of its input ports to produce one data item to be emitted via its output ports. A  $1-N$  node emits  $n$  data items to its output port before it retrieves the next data item among its input ports. A  $N-1$  node retrieves  $n$  data items from any of its input ports before it produces a single output data item. Predefined  $1-1$  dataflow nodes include `not`, the negation of a boolean value, `true`, the mapping of any value to a boolean true value, `ctrl`, the conversion of a boolean value into a control signal and `sel`, a selection. A selection node receives a choice on its lower input port that specifies from which of its input ports to forward the next data item. We also define nodes to work with list values. The `len-[]` node has a  $1-1$  correspondence. The  $1-N$  node `[]~` takes a list and emits the values one at a time, i.e., it streams the values. In order to perform the inverse  $N-1$  operation, the `~[]` requires  $n$ , the number of elements in the result list, to be available on its upper input port. Finally, the node `[~]` streams the values from a list that is unbounded. An unbounded list resembles the idea of an iterator, i.e., a list where it is possible to walk over the values but where the actual size is only known once the last value was accessed. This is especially important when we retrieve data via I/O such as the records from the chunk file. As such, `[~]` additionally outputs the length of the list.

With these constructs defined, we translate the terms of our language from Figure 3.5 into dataflow in Figure 3.6b. Note that a stateful function call maps to a node with an edge to itself that transfers the state from one invocation to the other. This is the common representation for state in dataflow models. We translate control flow into dataflow using `ctrl` node. In case of `seq`, the `ctrl` node always receive a positive input and sends a control signal that enables the downstream subgraph to proceed with the computation. In case of conditionals, the downstream subgraph to enable depends on the result of the subgraph for the condition. The translation of `smap` computations over bounded and unbounded lists is a straightforward use of the according list-handling nodes. For brevity reasons, we omit the translation of access to variables in the lexical scope from within an expression passed to `if` or `smap` and refer the interested reader to [215]. To keep the visual presentation of the algorithms in their dataflow representation concise and clear, we omit state edges and translation details such as `len - []` nodes. For clarity reasons, we refer to `[] ~` and `~ []` as `smap` and to `~ []` as `collect`. For example, Figure 3.4c depicts the dataflow graph for the algorithm listed in Figure 3.4a. It contains the three coarse steps from Figure 3.3 that our analysis found: the first `smap` node retrieves the data via the context, the `hmr-map` function performs the computation and `output` uses again the context to emit the results. Note that streams inside the dataflow graph represent an opportunity for pipeline parallelism and independent nodes for task-level parallelism. A scheduler finally uses a thread pool to execute the graph.

## 3.4 REWRITING HMR’S MAPPER IN OHUA

This section presents the algorithms that we extracted from the original Hadoop MapReduce code and re-implemented in Ohua. We already listed a coarse-grained version for the map task in Figure 3.4a. In the following, we rewrite the steps before and after the `hmr-map` function call which include decompression, deserialization and their inverse operations. In HMR these steps are encapsulated in the `InputFormat` and `OutputFormat` classes that are instantiated in the `Context` of the map task. We focused our rewrite on the implementation for the `SequenceFile`<sup>3</sup> format which ships with HMR and is widely used. Figure 3.7a lists the algorithm for the map task. It applies the `data-ingress` algorithm to the raw bytes of a single key and its associated value as retrieved from disk (Lines 14–15). Note that this actually defines a computation for all key-value pairs retrieved via the network, i.e., data residing on remote disks, instead of from an in-memory data structure. The retrieved `line` and `content` objects are then input to the function call that runs the application-defined mapper (Line 16). The produced key-value pairs are finally input to the `data-egress` algorithm that handles one pair at a time (Lines 17–20). We define the `data-ingress` algorithm in Figure 3.7b. It decompresses the value bytes and returns the serialized key and value. The `data-egress` algorithm in Figure 3.7c applies the inverse functions. It first serializes the objects in Lines 5 and 6 and then compresses the bytes at Line 7 before they are written to disk (Line 8). The rest of the stateful function calls concern HMR’s status reporting and statistics, i.e., their execution order depends on their side-effects to the distributed file system. We use `seq` to encode these dependencies into the program.

We believe the Ohua code for the map task is concise. It only requires a few lines and omits implementation details. Instead, it allows the BDS developer to immediately understand the algorithms of the data processing core. Additionally, our rewrites use almost all of the existing code, i.e., we solely changed the composition of the data processing steps. Ohua’s algorithm language allows to bind abstractions to variables. This establishes similar compositional flexibility that the iterator and the observer design patterns provide. We use it to define 4 programs with the following composition:

**Coarse (C)** uses the coarse-grained `Context` abstraction for the input and output side as shown in the algorithm of Figure 3.4a.

**Coarse-Input-Fine-Output (CIFO)** uses the `Context` for the input side and the fine-grained algorithm of Figure 3.7c for the output side.

**Fine-Input-Coarse-Output (FICO)** uses the fine-grained algorithm of Figure 3.7b for the input side and the `Context` for the output side.

**Fine (F)** uses the fine-grained algorithms for the input and output side.

---

<sup>3</sup><https://wiki.apache.org/hadoop/SequenceFile>

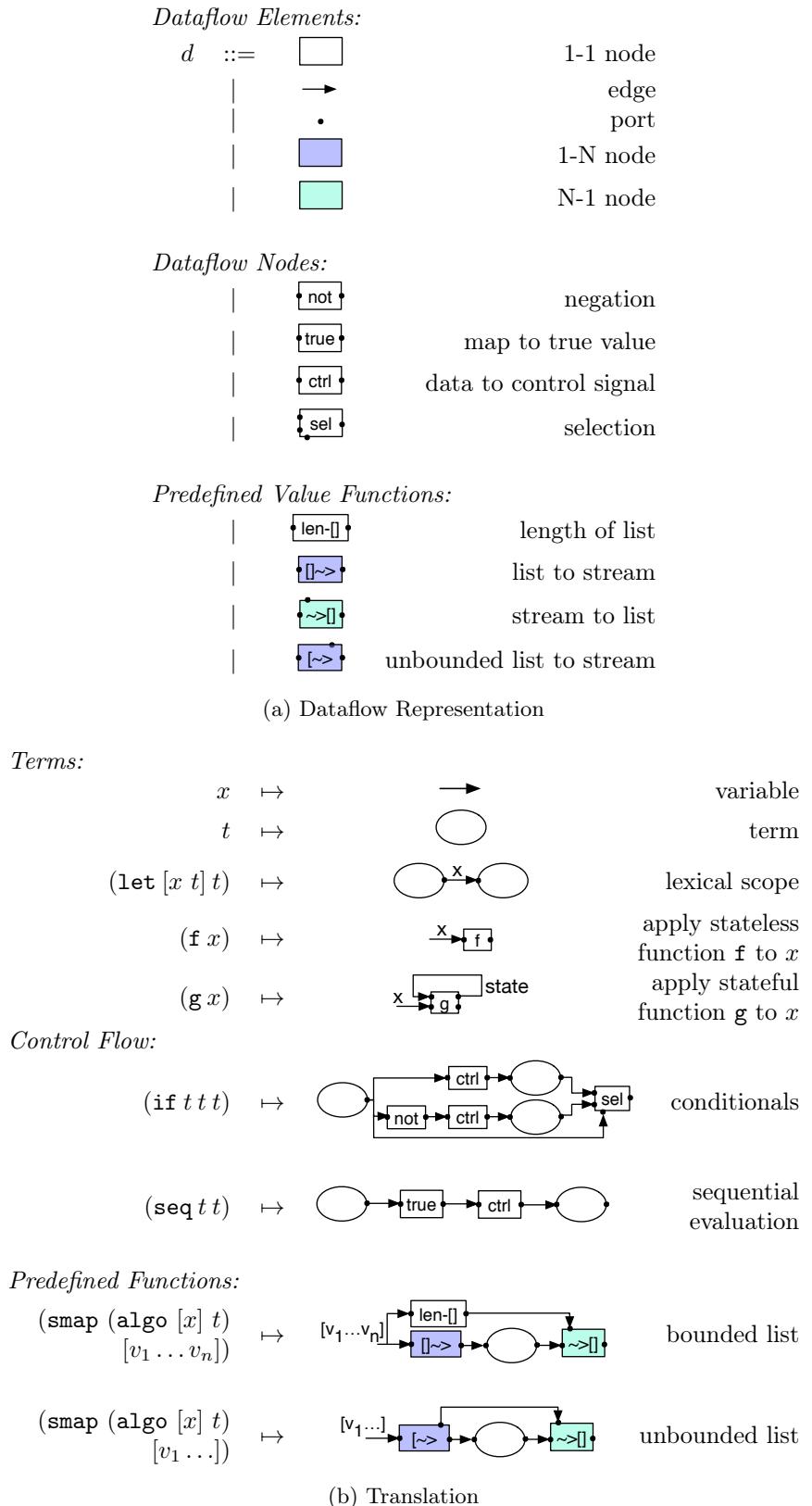


Figure 3.6: Definition of the dataflow representation and the translation from the language terms to the dataflow representation.

```

1  (defn map-task
2    [; decomposed input context
3     ^SequenceFile$Reader reader ^Deserializer key-deserializer
4     ^Deserializer val-deserializer
5     ^Mapper mapper ; app-supplied map implementation
6     ; decomposed output context
7     ^Serializer key-serializer ^Serializer val-serializer
8     ^Progressable reporter ^List stats
9     ^FSDatOutputStream out ^Counter map-out-records
10    ^Counter file-out-bytes ^CompressionCodec codec]
11  (let [records (new NetworkDataIterator reader)])
12    (ohua
13      (smap
14        (algo map-task-algo [[key-buf value-buf]])
15        (let [[line content] (data-ingress key-buf value-buf
16              key-deserializer val-deserializer)
17          kv-pairs (hmr-map line content mapper)])
18        (smap
19          (algo kv-algo [kv-pair]
20            (data-egress kv-pair reporter stats
21              key-serializer val-serializer codec
22              out map-out-records file-out-bytes))
23            kv-pairs)))
24        records)))

```

(a) Algorithm for the map task.

```

1  (defalgo data-ingress [serialized-key serialized-val
2                           key-deserializer val-deserializer]
3    (let [key-in (deserialize serialized-key key-deserializer)
4          val-in (deserialize (decompress serialized-val) val-deserializer)]
5      (array key-in val-in)))

```

(b) Algorithm for the data ingestion.

```

1  (defalgo data-egress [kv-pair reporter stats key-serializer
2                        val-serializer codec out
3                        map-out-records file-out-bytes]
4    (let [[k v] kv-pair
5          bytes-before (seq (report-progress reporter) (fs-stats-out stats))
6          key-ser (seq bytes-before (serialize k key-serializer))
7          val-ser (seq bytes-before (serialize v val-serializer))
8          val-compressed (compress val-ser codec)
9          bytes-after (seq (write key-ser val-compressed out)
10                  (fs-stats-out stats))]
11        (update-counters bytes-after bytes-before
12                      map-out-records file-out-bytes)))

```

(c) Algorithm for the data emission.

---

Figure 3.7: The modular rewrite for the HMR data processing core of the map task allows to construct four different scenarios. It facilitates to exchange the coarse-grained implementations of the `data-ingress` (see Figure 3.4) and `data-egress` algorithms for the fine-grained ones (see Figure 3.7b and 3.7c).

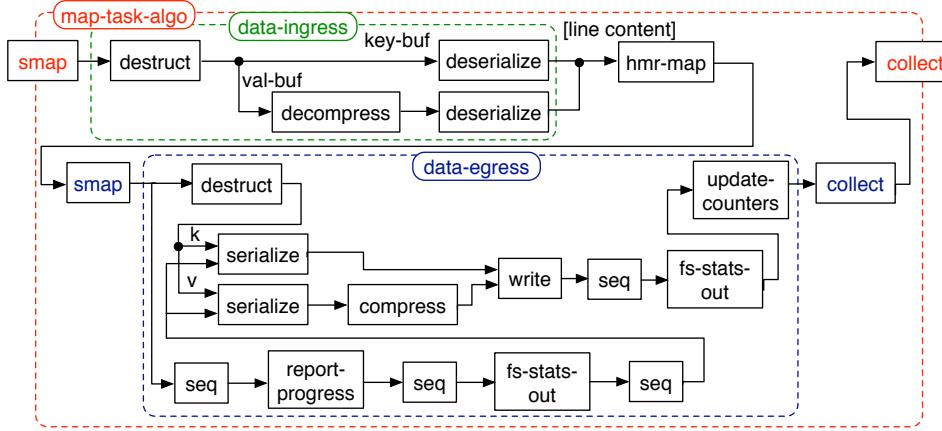


Figure 3.8: The dataflow graph for the Fine rewrite expresses the inherent task-level parallelism in the decompression/deserialization of key and values. The same accounts for the inverse operations in the **data-egress** algorithm. The overall program uses two nested **smap** applications which directly translate into pipeline parallelism at runtime.

Since Ohua inlines algorithm calls, this composition does not sacrifice parallelism. Figure 3.8 shows the dataflow graph for the Fine rewrite. It contains nodes for language constructs such as **seq** and **destruct** for destructuring tuples, arrays and lists<sup>4</sup>. The graph contains nested uses of **smap** and thus enough potential for a pipeline parallel execution. Additionally, deserialization, decompression and its inverse functions on the output side are independent for a key and its value and therefore can benefit from task-level parallelism.

## 3.5 EVALUATION

In this section, we evaluate the increase in throughput that can be achieved with our Ohua-based HMR rewrites. We first describe the setup for our experiments and then give breakdowns of the execution times of the individual stateful function calls of the program. Only these breakdowns allow to fully understand the speedup that can be achieved with Ohua’s dataflow execution. Afterwards, we verify that Ohua achieves the maximal speedup possible for throughput and analyze the overheads of the Ohua-based execution.

### 3.5.1 Experimental Setup

In order to evaluate the data processing core of the HMR map task, we executed a single HMR map task on a 12-core (24 hardware threads) Intel NUMA machine at 2.6 GHz with 2 CPU sockets and 128 GB of RAM. All experiments used a JVM (JDK 1.8) with G1 enabled, 10 GB initial and 30 GB maximal heap size. A study of our new data processing core on the overall throughput of a job executed across multiple machines is future. This is due to that fact that normal benchmarks such as WordCount and Sort do not apply because of their simple data formats and their low-profile reduce phases. We studied the internals of Hive and found that the query execution engine does not use HMR’s **InputFormat** and **OutputFormat** interfaces for data (de-)serialization. Hive developers moved these aspects into the computational part of HMR’s data processing core in order to use the same code base for executing queries on HMR, Spark and Tez<sup>5</sup>. As such, we focus our evaluation on the data processing core directly. Instead of the Hadoop Distributed File System, we implemented our own Network File System (NFS) on Hadoop’s file system abstraction to enforce data retrieval over the network. It retrieves a file chunk from local disk and transfers it over the network. This also happens during normal processing when the map task can not be executed on the node that hosts the input data chunk and hence enables complete code reuse on

<sup>4</sup>The actual graph also contains constructs such as **scope** for scoped variable usage and **size** to get the number of items a **collect** must gather for a single result.

<sup>5</sup><https://tez.apache.org/>

```

1 CREATE TABLE PART (
2     P_PARTKEY      SERIAL PRIMARY KEY,
3     P_NAME          VARCHAR(55),
4     P_MFGR          CHAR(25),
5     P_BRAND         CHAR(10),
6     P_TYPE          VARCHAR(25),
7     P_SIZE          INTEGER,
8     P_CONTAINER     CHAR(10),
9     P_RETAILPRICE   DECIMAL,
10    P_COMMENT        VARCHAR(23));

```

Figure 3.9: TPC-H table for parts.

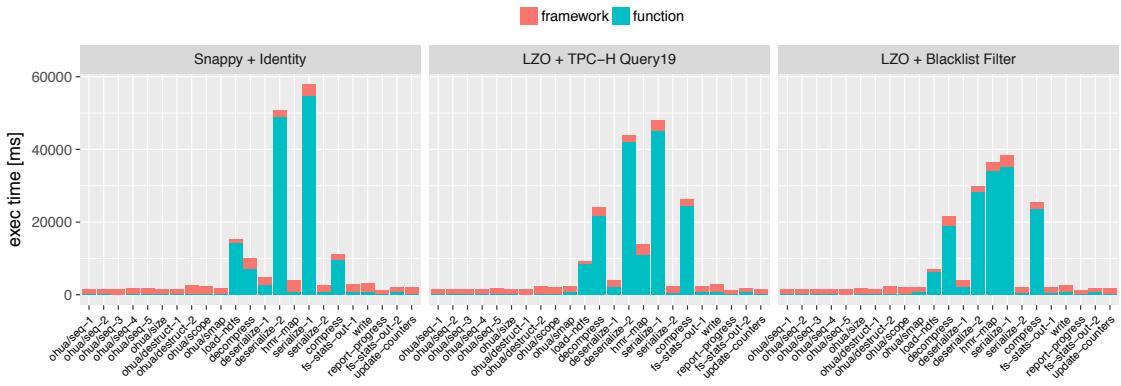


Figure 3.10: Execution breakdowns for executions of the Fine rewrite.

the input side. The requested file chunk for our experiments stores 1.4 GB of data. In order to achieve high bandwidth the NFS data provider transfers the file without copy overhead (`sendfile`) from the disk to the network while the map task executes on the same machine. To nevertheless study a portion of a real data analytics query, we retrieve and process randomly generated data for the PART table as defined in Figure 3.9 from the TPC-H benchmark [52]. Table records are stored in the widely used JSON format. We study the impact of Snappy and LZO compression in combination with either the default identity function in the mapper or a map function that applies a filter in a WHERE clause of an SQL query. For the latter we provide two versions: The first checks the conditions given in TPC-H query 19. The second implements a blacklist filter that tries to locate certain words in the P\_COMMENT column.

### 3.5.2 Runtime Analysis

It has been observed that it is hard to analyze in detail the performance of current BDS, mainly due to a lack of support for instrumentation [176]. This lack of analysis support could explain why, for so many years, researchers stuck to the common believe that big data applications are network-bound. The modular structure of Ohua’s dataflow graph makes such an instrumentation straight forward. We use it in the following to perform an in-depth analysis for the Fine rewrite. It allows us to understand the execution pattern of the data processing core and to analyze Ohua’s runtime overheads. Afterwards, we study the throughput of our rewrites and the impact of maintaining state.

#### Execution Pattern Analysis

Figure 3.10 shows the total execution time of the stateful functions for the Fine rewrite. The stateful functions that facilitate constructs of the algorithm language are depicted on the left of

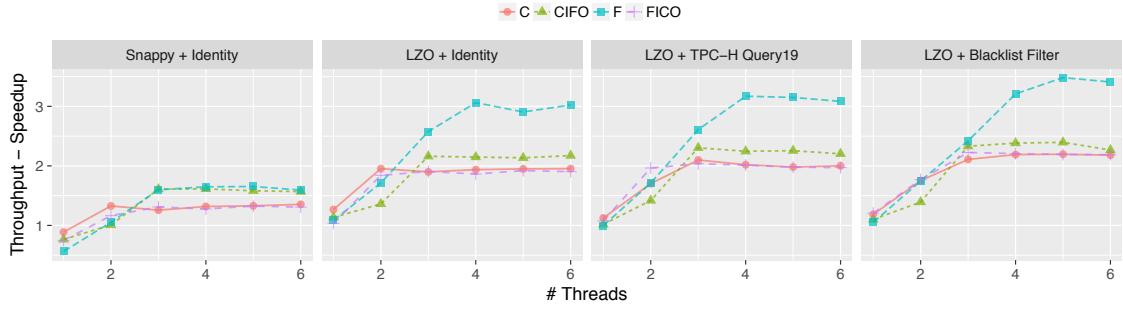


Figure 3.11: Speedups in throughput of the rewrites in the four configurations.

each plot and prefixed with “ohua”<sup>6</sup>. The rest of the functions execute HMR functionality. The execution time of each function is broken down into the time spent in the Ohua operator framework code and the time spent in the actual stateful function. With this analysis, we can differentiate three aspects: 1) the overhead that comes with the algorithm language, 2) the runtime overhead of Ohua’s operator framework and 3) the actual execution of the HMR functionality. The breakdown shows that most time is spent in (de)-serializing the value from and into JSON. Execution times vary slightly across the four configurations due to the fact that these functions create a lot of objects. Note that the execution time includes garbage collection. The more work other functions perform, the more likely it is that they are also interrupted by a garbage collection. Furthermore, the breakdown shows that Snappy requires much less CPU cycles than LZO, but LZO provides better compression rates. As a result, the configuration with LZO provides a higher potential to speedup the pipeline parallel execution than the one with Snappy because more processing can be performed in parallel. This potential further increases when the mapper performs actual work such as evaluating the conditions of TPC-H query 19 or our blacklist filter.

## Throughput

Figure 3.11 depicts the throughput speedup for our four rewrites across each of the configurations. In all four configurations the Fine rewrite achieves the highest speedup with  $3.5\times$  for the LZO + Blacklist Filter configuration. Naturally, no configuration reaches the maximum theoretical speedup due to the overhead of the Ohua framework and the load profile of the different functions (cf. Figure 3.10). This is particularly notable in the case of Snappy + Identity, with a speedup below a maximum of around  $2\times$ <sup>7</sup>. In this configuration the overhead of the Ohua framework has a higher impact on the throughput. The overall function execution is less than in other workloads while the overheads are the same in all configurations. The resulting performance penalty is roughly  $0.4\times$ , leading to maximum achievable speedup of  $1.6\times$ . In the other configurations this penalty is negligible with speedups around  $0.1\times$  below the maximum.

## Cost

The COST metric refers to the number of additional cores that are necessary in order to achieve the same performance as the original implementation [159]. Figure 3.12 shows the COST results for the four configurations and compares a stateless implementation (as suggested by the internal HMR API) to a stateful one. Note that the original (coarse) parts are stateful. The graphs provide an important argument in favor for stateful computations and thus Ohua’s programming model. The two functions with state are `decompress` and `compress` which both use a dictionary to speed up the discovery of words in the bytes. Removing this state can degrade the performance by as much as an order of magnitude, as in the case of our last two compute-heavy configurations.

<sup>6</sup>Ohua implements language constructs as stateful functions as well.

<sup>7</sup>Intuitively, the maximum speedup is calculated by determining the most appropriate pipeline, i.e., number of stages and stage balancing (load in Figure 3.10). For Snappy + Identity, with clearly two dominant functions of comparable load (deserialize-2, serialize-1), a two stage configuration would deliver around  $2\times$  speedup.

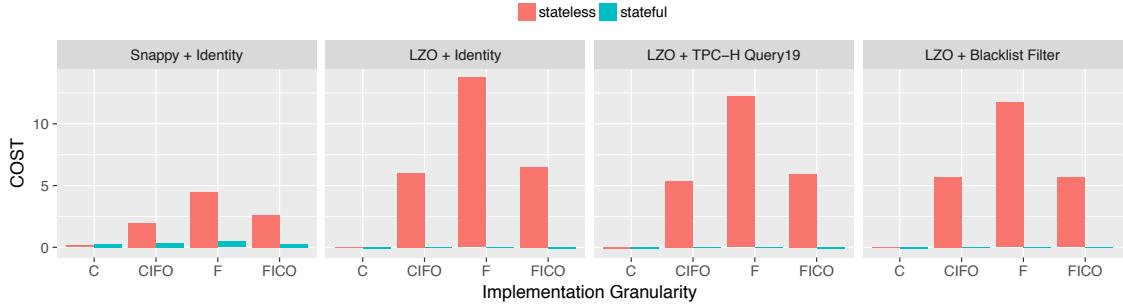


Figure 3.12: COST of the four rewrites when implemented with or without state.

## 3.6 RELATED WORK

We could not find any work that tries to make current big data systems scale with new hardware by introducing parallelism into their data processing cores. To the best of our knowledge there is no existing approach to do so using an implicitly parallel language. For example, Weld [177] applies compiler optimizations such as loop tiling and vectorization to speed up data analytics across different frameworks. It requires to re-implement main functions and operators to make them Weld-aware. Weld works on the application level rather than the data processing core of BDS. Crail [202] leverages RDMA and storage tiering to speed up the I/O functions (`load-ndfs` and `write` in the HMR data processing core). This works for sorting data but fails to scale as soon as additional analytics take place which require complex data types, their associated serialization, compression and perform additional operations on the data. Researchers also made the case of a single global cluster-wide address space to further speed up serialization and deserialization [131]. The Apache Spark project Tuncsten tries to remove garbage collection penalties via their own off-heap memory management similar to Flink [35]. However, all these approaches optimize only isolated parts of the data processing core, i.e., they shrink one bar in the execution breakdown graphs of Figure 3.10 just for another to become the bottleneck. They speed up individual parts instead of making the data processing core scale via an impacting structural change as researchers concluded [216]. In this paper, we do exactly this. We re-implemented the algorithms of the data processing core using an implicitly parallel language that can automatically exploit pipeline and task-level parallelism to scale the computation without increasing code complexity. Note that there exist frameworks such as Phoenix [189] or Metis [154] that scale analytics applications for in-memory data on multi- and many-core architectures. Although Ohua can be used for that as well, this is not the topic of this paper. In this paper, we give a redesign of the data processing cores of the most-widely-used big data systems in the field today.

## 3.7 CONCLUSION AND FUTURE DIRECTIONS

The data processing core of current big data systems do not scale with improved network performance. An analysis of Hadoop MapReduce, Spark and Flink found that the implementations use either the iterator or the observer design pattern. Both provide the opportunity for a semantically equivalent pipeline parallel execution. We replaced these patterns in Hadoop MapReduce using Ohua, an implicitly parallel language. The resulting Ohua program heavily reuses the existing code, while being more concise. Furthermore, our evaluation shows that our rewrites provide speedups of up to 3.5x exploiting pipeline as well as task-level parallelism. In the future, we want to integrate a compiler-based approach that finds stateful functions that are applicable to a data parallel execution. This would enable us to make the data processing cores fully scalable independent of the size of the data processing pipeline.

**Postscript** This paper shows that our programming model is essentially a good candidate to replace the streams, iterator and observer programming model. More use cases need to be studied in order to fully make that conclusion. This paper focused on the runtime system and especially on its pipelined execution. The next chapter presents the details of the compiler for the algorithms.



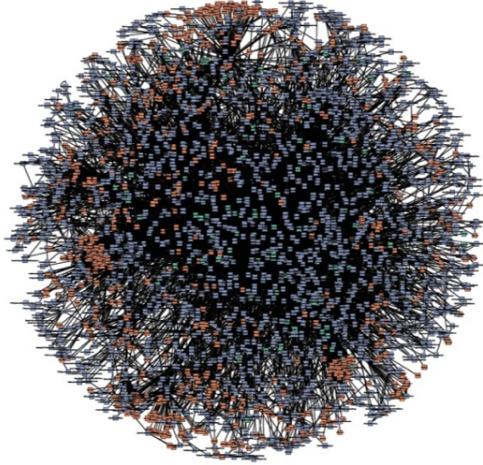
# 4 COMPILING FOR CONCISE CODE AND EFFICIENT I/O

**Prelude** This section covers the paper with the same title co-authored by Andrés Goens, Justus Adam and Jeronimo Castrillon that I presented at the International Conference for Compiler Construction in 2018 [77]. This work focused primarily on the compiler that translates a program that uses our stateful functional programming model into a dataflow graph. In Section 2, the extraction of the dataflow graph from an algorithm was based on the Clojure syntax tree. In this section, we provide a more general approach on the abstraction of an intermediate language that follows the call-by-need lambda calculus. As a result, we can perform semantic-preserving transformations to arrive at, what we called extended data dependency (EDD) form. This form allows us to derive a dataflow graph easily and builds the foundation for an intermediate representation where (semantic-preserving) transformations optimize I/O calls.

## 4.1 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 [3], Twitter [68], and Amazon [60] serve millions of complex web pages. They run hundreds of small programs, called *microservices*, that communicate with each other via the network. Figure 4.1 shows the interactions of the microservices at Amazon. In these infrastructures, reducing network I/O to satisfy sub-millisecond latencies is of paramount importance [22].

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 [179] primarily addresses updates while the extension to microservices deals with failures [34]. The idea is to expose functions as a service that runs as a stand-alone 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.



[Source: I Love APIs 2015 by Chris Munns, licensed under CC BY 4.0, available at:  
<http://bit.ly/2zboHTK>]

---

Figure 4.1: Microservices at Amazon.

#### 4.1.1 Code Conciseness versus I/O Efficiency

Programmers commonly use frameworks, like Thrift [6], 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 [168].

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 code. To this end, it fetches post contents and associated meta data via I/O from other (database) services at various places in the code. We implemented the database service for our blog using Thrift and introduced a single call to retrieve the contents for a list of posts. Figure 4.2 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 $\times$  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

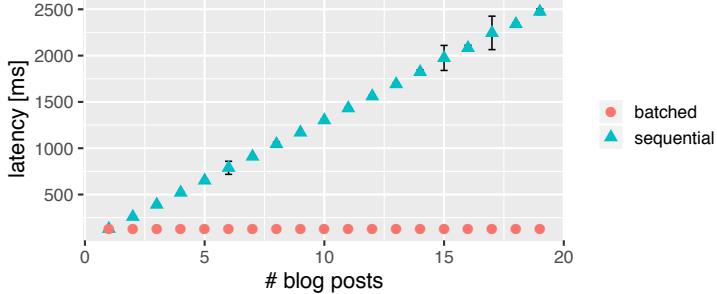


Figure 4.2: Latency reduction via batching in a simple microservice for a blog.

deadlocks [143] and event-based programming suffers from stack ripping [5]. There is a long debate in the systems community which of the two approaches provides the best performance [219, 175]. 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 [71]. 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 [155] and Muse [130] 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.

#### 4.1.2 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 4.3, builds on top of two intermediate representations: an expression language and a dataflow representation [16]. The expression language is based on the lambda calculus which we extend with a special combinator to execute I/O calls. The dataflow representation of an expression is a directed graph where nodes represent operations of the calculus and edges represent the data transfer between them. The compiler translates a program, i.e., an expression, into a dataflow graph. To reduce the number of I/O calls, 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 Ÿauhau, 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 Ÿauhau 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 4.2 introduces the expression IR and the transformations to reduce the number of I/O calls. In Section 4.3, we define the dataflow IR and the transformations to support a fully concurrent execution. We review related work in Section 4.4. Section 5.7 briefly describes our Ÿauhau implementation and evaluates it before we conclude in Section 5.9.

## 4.2 REDUCING I/O ON AN EXPRESSION IR

We now present our expression IR. It is based on the call-by-need lambda calculus [13, 12], which prevents duplicated calculations. We want this for two reasons. First, our expressions can contain



Figure 4.3: Overview of Yauhau.

*Terms:*

$t ::=$	$x$	variable
	$\lambda x.t$	abstraction
	$t t$	application
	$\text{let } x = t \text{ in } t$	lexical scope (variable binding)
	$\text{if}(t t)$	conditionals
	$\text{ff}_f(x_1 \dots x_n)$	apply foreign function $f$ to $x_1 \dots x_n$ with $n \geq 0$
	$\text{io}(x)$	apply I/O call to $x$

*Values:*

$v ::=$	$o \in V_{\text{ff}}$	value in host language
	$\lambda x.t$	abstraction
	$[v_1 \dots v_n]$	list of $n$ values

*Predefined Functions:*

$$\begin{aligned} \text{map}(\lambda x.t [v_1 \dots v_n]) &\equiv [(\lambda x.t) v_1 \dots (\lambda x.t) v_n] \\ \text{nth}(n [v_1 \dots v_n \dots v_p]) &\equiv v_n \end{aligned}$$

Figure 4.4: Language definition of the expression IR.

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 4.4 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 4.2.2, 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:  $\text{io} \equiv \text{ff}_{\text{io}}$ . 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 denote the I/O call to fetch all identifiers of the existing posts, we write  $\text{io}_{\text{postIds}}$ . As an example, Figure 4.5 lists the expression for our blog service. 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 [12]. 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  $\text{map}$  can participate in these transformations.

```

let x_mainPane =
  ff_render(
    map(λx.io(ff_reqPostContent(x)))      5. produce HTML
    ff_last-n(10)                         4. fetch post contents
    map(λx.io(ff_reqPostInfo(x)))         3. filter last 10 posts
    io(ff_reqPostIds(())))) in 2. fetch meta data
                                1. fetch ids of all posts

let x_topics =
  ff_render(
    ff_tag(
      map(λx.io(ff_reqPostInfo(x)))      4. produce HTML
      io(ff_reqPostIds(())))) in 3. tag topics
                                2. fetch meta data
                                1. fetch ids of all posts

let x_popularPosts = t_popularPosts in (omitted for brevity)
let x_leftPane = ff_render(ff_compose(x_topics x_popularPosts)) in
  ff_render(ff_compose(x_mainPane x_leftPane))

```

---

Figure 4.5: Lambda expression for the blog example.

#### 4.2.1 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 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  $x_1 \dots x_n$

$$\text{let } \underbrace{x_1 = t_1}_{\text{binding}} \text{ in let } x_2 = t_2 \text{ in } \dots \text{let } x_n = t_n \text{ in } x_n$$

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

```

let x1 = ff_reqPostIds() in
let x2 = io(x1) in
let f = λx_postId.let y1 = ff_reqPostInfo(x_postId) in
  let y2 = io(y1) in y2 in
let x3 = map(f x2) in
let x4 = ff_last-n(10 x3) in
let g = λx_postInfo.let z1 = ff_reqPostContent(x_postInfo) in
  let z2 = io(z1) in z2 in
let x5 = map(g x4) in
let x6 = ff_render(x5) in x6

```

(4.1)

In order to find independent groups of I/O calls in the EDD expression, we use a well-known technique called *let-floating* that is also used in the Haskell compiler [183]. 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.

$$\text{I/O group } \left\{ \begin{array}{ll} \text{let } x_1 = \text{ff}_{\text{reqPostIds}}() \text{ in} & 1. \text{ mainPane} \\ \text{let } x_2 = \text{ff}_{\text{reqPostIds}}() \text{ in} & 1. \text{ topics} \\ \text{let } x_3 = \text{io}(x_1) \text{ in} & 2. \text{ mainPane} \\ \text{let } x_4 = \text{io}(x_2) \text{ in} & 2. \text{ topics} \\ e \end{array} \right.$$

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 `bio` which performs the batching and executes the I/O. As such the following semantic equivalence holds:

$$\text{bio}(i_1 \dots i_n) \equiv [\text{io}(i_1) \dots \text{io}(i_n)] \quad (4.2)$$

The function is defined using two new (foreign) functions, `batch` and `unbatch`:

$$\begin{aligned} \text{bio}(x_1 \dots x_n) ::= & \text{let } y_{1..m} = \text{batch}(x_1 \dots x_n) \text{ in} \\ & \text{let } z_{1..m} = \text{map}(\lambda x. \text{io}(x)) y_{1..m} \text{ in} \\ & \text{let } y_{1..n} = \text{unbatch}(z_{1..m}) \text{ in } y_{1..n} \end{aligned}$$

We say that variable  $y_{1..m}$  stores the list of values that would otherwise be bound to variables  $y_1, \dots, y_m$ . Since we do not know at compile-time which calls can be batched, `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 `bio`. We assume that this information is preserved across the I/O call such that `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 `unbatch` to preserve the equivalence defined in Equation 4.2. Finally, this list needs to be destructured again to re-establish the bindings:

$$\begin{aligned} & \text{let } y_1 = \text{io}(x_1) \text{ in } \dots \text{ let } y_n = \text{io}(x_n) \text{ in } e \\ \Rightarrow & \text{let } y_{1..n} = \text{bio}(x_1 \dots x_n) \text{ in} \\ & \text{let } y_1 = \text{nth}(1 y_{1..n}) \text{ in } \dots \text{ let } y_n = \text{nth}(n y_{1..n}) \text{ in } e \end{aligned}$$

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-affecting ones.

### 4.2.2 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`<sup>1</sup> 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  $\text{scan} \equiv \text{ff}_{\text{scan}}$ .

The `if` combinator and `map` function are special because they control the evaluation of the expression(s) passed to them as arguments. The `if` combinator may evaluate only one of the two

---

<sup>1</sup>`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.

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 rewrites:

$$\text{if}(c \text{ io}(x) \text{ io}(x)) \equiv \text{io}(x) \quad (4.3)$$

$$\text{map}(\lambda x. \text{io}(x) [v_1 \dots v_n]) \equiv [\text{io}(v_1) \dots \text{io}(v_n)] \quad (4.4)$$

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* [125]. Figure 4.6 visualizes the transformations. Figure 4.6a 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 4.6b, 4.6c and 4.6d, 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 4.6b, 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  $e_{j+1\dots n}$ .<sup>2</sup>

$$\text{let } x_j = g x_{j-1} \text{ in } e_{j+1\dots n}$$

In Figure 4.6c, we lift the computation of the request  $e_{1\dots j-1}$  and in Figure 4.6d the continuation of the computation on the result of the I/O call  $e_{j+1\dots n}$ . To cover the general case, we assume that  $e_{j+1\dots n}$  does not only contain applications to the I/O result  $x_j$  but also to all other variables  $x_1, \dots, x_{j-1}$  in its scope. To provide them in the lexical scope for the application to  $h$ , the lambda lifting of  $e_{1\dots x-1}$  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  $e_{j+1\dots n}$  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  $x_1 \dots x_n$  from  $x_{1\dots j-1}$ . For  $c$ , we define the following semantic equivalence:

$$\begin{array}{ccc} c(\text{let } x_{1\dots j-1} = f x_0 \text{ in} \\ \quad \text{let } x_j = g x_{j-1} \text{ in} \\ \quad \text{let } x_n = h x_1 \dots x_j \\ \quad \text{in } x_n) & \equiv & \text{let } x_{1\dots j-1} = c(f x_0) \text{ in} \\ & & \text{let } x_j = c(g x_{j-1}) \text{ in} \\ & & \text{let } x_n = c(h x_1 \dots x_j) \\ & & \text{in } x_n \end{array}$$

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

---

<sup>2</sup>We use the notation  $e_{1\dots n}$  to refer to the part of the expression defined in Figure 4.6a that binds the results of the terms  $t_1 \dots t_n$  to variables  $x_1 \dots x_n$ .

```

let x1 = t1 in
let x2 = t2 in
⋮
let xj-1 = tj-1 in
let xj = io(xj-1) in
let xj+1 = tj+1 in
⋮
let xn = tn in xn

```

(a) Isolated IO form

```

let g =
λx'j-1.(io(x'j-1)) in
let x1 = t1 in
let x2 = t2 in
⋮
let xj-1 = tj-1 in
let xj = g xj-1 in
let xj+1 = tj+1 in
⋮
let xn = tn in xn

```

(b) Lifting I/O

```

let f = λx'_0.(
let x1 = t1 in
let x2 = t2 in
⋮
let xj-1 = tj-1 in
[x1 … xj-1] in
let g = λx'j-1.(io(x'j-1)) in
let x1..j-1 = f x0 in
let x1..j-1 = nth(1, x1..j-1) in
⋮
let xj-1 = nth(j - 1, x1..j-1) in
let xj = g xj-1 in
let xj+1 = tj+1 in
⋮
let xn = tn in xn

```

(c) Lifting Request Computation

```

let f = λx'_0.(
let x1 = t1 in
let x2 = t2 in
⋮
let xj-1 = tj-1 in
[x1 … xj-1] in
let g = λx'j-1.(io(x'j-1)) in
let h = λx'_1 … x'_j.(
let xj+1 = tj+1 in
⋮
let xn = tn in xn) in
let x1..j-1 = f x0 in
let x1..j-1 = nth(1, x1..j-1) in
⋮
let xj-1 = nth(j - 1, x1..j-1) in
let xj = g xj-1 in
h x1 … xj

```

(d) Lifting Continuation

Figure 4.6: We extract an I/O call out of an expression (Fig. 4.6a) using lambda lifting. In this process, we split the expression into three parts: the I/O call  $g$  (Fig. 4.6b), the computation of the request  $f$  (Fig. 4.6c), and the continuation of the computation with the response  $h$  (Fig. 4.6d). To extract one of the parts, we create an abstraction, apply it individually and recreate (destructure) the original lexical scope.

operational semantics of the single application. When  $c$  is a conditional  $\text{if}(t_{\text{cond}} t_{\text{true}} t_{\text{false}})$ , this requires that the individual  $\text{if}$  applications use the same condition result bound to  $x_{\text{cond}}$ :

```

let x1..j-1 = if(xcond (ftrue x0) (ffalse x0)) in
let xj = if(xcond (gtrue xj-1) (gfalse xj-1)) in
let xn = if(xcond λ.(htrue x1 … xj) λ.(hfalse x1 … xj))
in xn

```

The very same concept applies to the  $\text{map}$  function. Since  $\text{map}$  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  $x_j$ .

```

let x[1..j-1]_m = map(λx0..f [v1 … vm]) in
let x[j]_m = map(λxj-1..g xj-1 x[j-1]) in
let x[n]_m = map(λx1..j..h x1 … xj x[1..j]_m)
in x[n]_m

```

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  $t_{\text{true}}$  and  $t_{\text{false}}$  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.

## 4.3 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 [12]. 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.

### 4.3.1 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 [148] or to processors and interconnects, for example on an FPGA [215]. 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 in Figure 4.7a. 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 or 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-[]` 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.

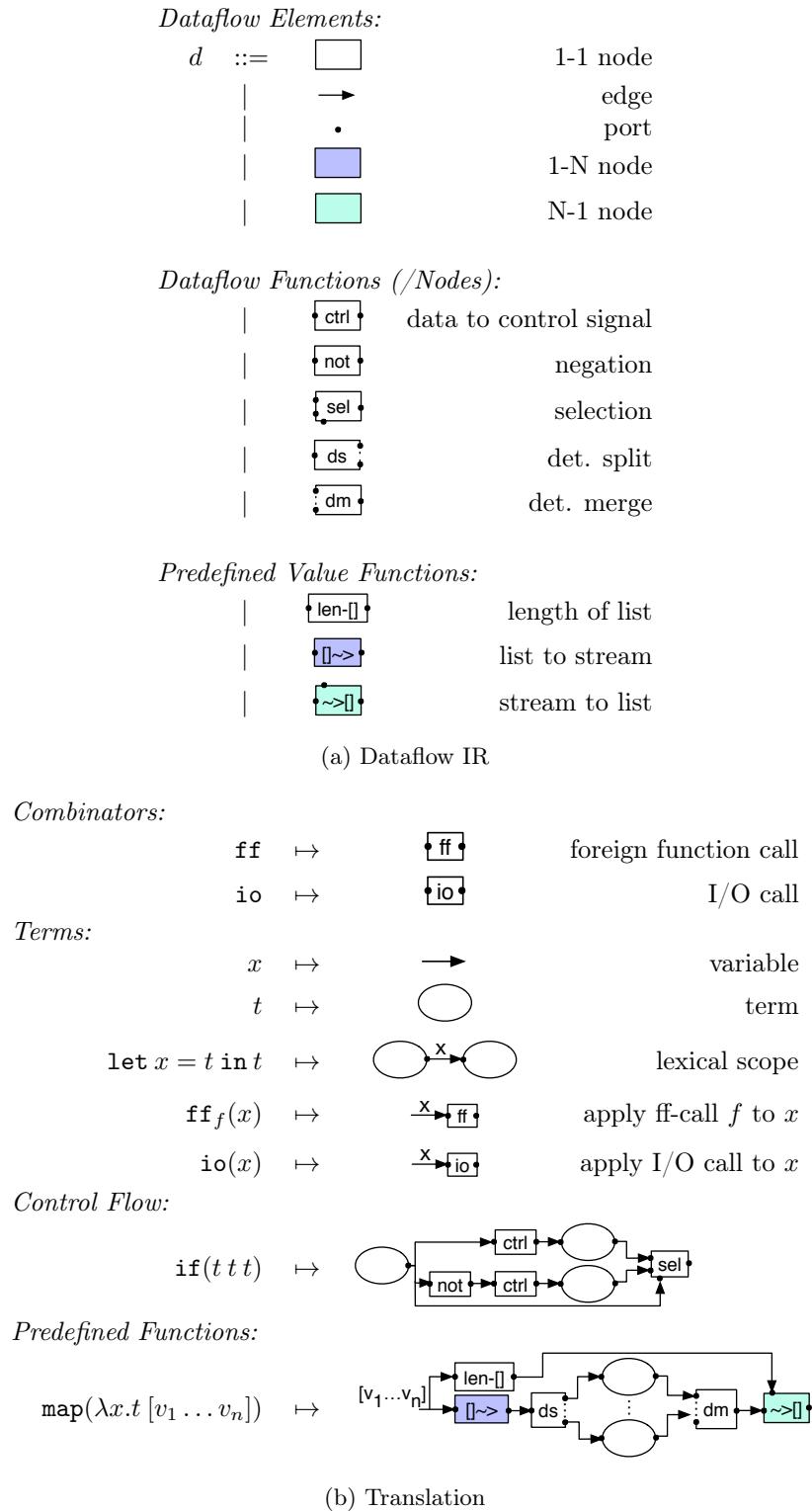


Figure 4.7: Definition of the dataflow IR and the translation from the expression IR to the dataflow IR.

**Lowering Expressions to Dataflow** From an expression in EDD form we can easily derive the corresponding dataflow graph, as shown in Figure 4.7b. Each term on the right-hand side of a binding form translates to a node. In EDD form this can only be an application of a combinator, a conditional or a call to one of the pre-defined functions. The definition of dataflow nodes perfectly matches the semantics of our  $\text{ff}$  and  $\text{io}$  combinators because both may have side-effects. Since both

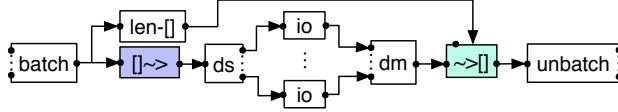


Figure 4.8: The dataflow graph for concurrent I/O.

require only a single argument, the corresponding nodes define one input port and one output port with a 1–1 correspondence. Each variable translates into an arc. More specifically, each application to a variable creates a new edge at the same output port. An unlabeled ellipse denotes an abstract term which translates to a subgraph of the final dataflow graph. In order to translate control flow into dataflow, we turn the result of the first term into a control signal and send it to the subgraph from the second term. The subgraph for the third term receives the negated result as a signal. Therefore, the subgraph for the second term computes a result only if the subgraph of the first term emitted a `true` value. Otherwise, the subgraph for the third term performs the computation. The translation of the `map` function first streams the input list and then dispatches the values to the replicas of the subgraph derived from  $t$ . The subgraph results are again merged and then turned back into a list that is of the same length as the input list. Computation among the replicas of the subgraph of  $t$  can happen concurrently. To achieve maximum concurrency, the number of subgraphs needs to be equal to the size of the input list. This is impossible to achieve at compile-time because the size of the list is runtime information. So we need to make a choice for the number of replicas at compile-time and route the length of the input list to the node that constructs the output list. For both terms, the conditionals and the `map` function, we assume that the terms passed as arguments do not contain any free variables. The approaches to do so can be found elsewhere [215]. In this paper, however, we focus on concurrency transformations.

### 4.3.2 Concurrent I/O

We introduce concurrency for the `bio` 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  $y_1 \dots y_n$  such that

```

let  $y_{1\dots n}$  = bio( $x_1 \dots x_n$ ) in
let  $y_1$  = nth(1  $y_{1\dots n}$ ) in ... let  $y_n$  = nth( $n$   $y_{1\dots n}$ ) in  $e$ 
::= let  $y_1 \dots y_n$  = bio( $x_1 \dots x_n$ ) in  $e$ 

```

In languages such as Haskell and Clojure, destructuring desugars into calls to `nth` just as in our expression IR. But the destructuring version captures very well the concept of multiple output ports for the `bio` node in the dataflow graph, one for each value in the list. This makes the resulting dataflow graph more concise.

Since the definition of `bio` uses `map` we can directly translate it into dataflow as shown in Figure 4.8. The resulting graph expresses the concurrency of the I/O calls but the `~~>[]` node requires each I/O call to finish before it emits a result. That is, the computation that depends on I/O calls that finished early is blocked waiting for slower I/O calls that it does not depend upon.

In order to get rid of this dependency, we remove the `~~>[]` node and change the `unbatch` node to receive individual I/O responses instead of the full list. Since the `unbatch` does not require all I/O responses to perform this operation, it can forward results as soon as it receives them.

As an example, Figure 4.9 shows the dataflow graph of the EDD expression (4.1) that computes the main pane.

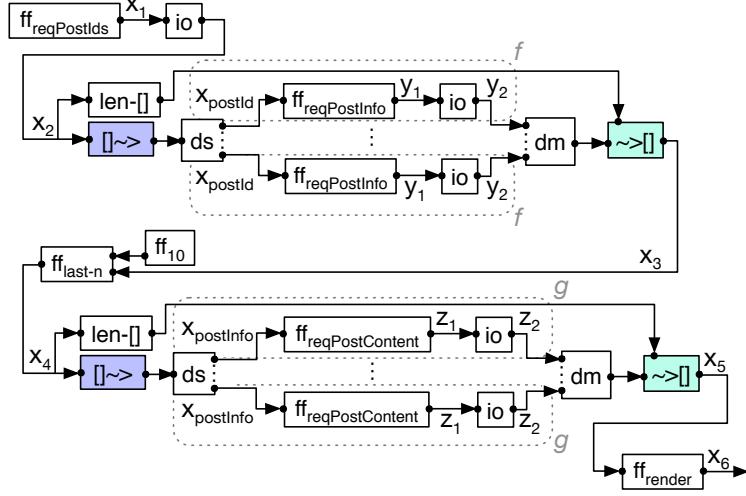


Figure 4.9: The dataflow graph for EDD expression 4.1 from Section 4.2.1 that computes the main pane.

## 4.4 RELATED WORK

As of this writing, two solutions exist in the public domain for optimizing I/O in micro-service-based systems. Facebook’s Haxl [155], 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 public domain is Muse [130, 129], written in Clojure. Muse implements a so-called free monad [207] 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 [211], Flume [39] or StreamFlex [198]. 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 [117]. 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.

## 4.5 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 4.10 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:

Terms:

$$\begin{aligned}
 \text{x} &\mapsto x \\
 (\text{fun } [x] \ t) \equiv \#(t) &\mapsto \lambda x. t \\
 (t \ t) &\mapsto t \ t \\
 (\text{let } [x \ t] \ t) &\mapsto \text{let } x = t \text{ in } t \\
 (\text{if } t \ t \ t) &\mapsto \text{if}(t \ t \ t) \\
 (\text{f } x_1 \dots x_n) &\mapsto \text{ff}_f(x_1 \dots x_n) \\
 (\text{io } x) &\mapsto \text{io}(x)
 \end{aligned}$$

---

Figure 4.10: Mapping the terms of the Clojure-based Ÿauhau language to our expression IR.

Table 4.1: Execution times of the Blog Example

Version	seq	base	batch	full
Time [ms]	275 ± 25	292 ± 19	79 ± 21	66.5 ± 5.2

```
(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<sup>3</sup>. Table 4.1 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 4.2.

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.

#### 4.5.1 Microservice-like Benchmarks

Due to missing public domain benchmarks, we use a framework for generating synthetic microservices [91]. 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, a random level graph has an edge  $(v, w)$  with probability  $2^{l(v)-l(w)} < 1$ .

---

<sup>3</sup><https://tud-ccc.github.io/yauhau-doc/>

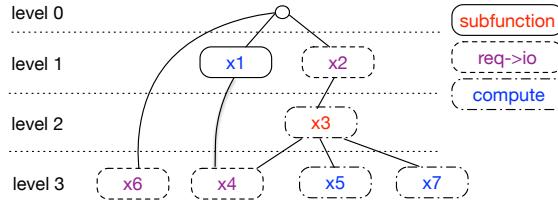


Figure 4.11: An example of a Level Graph.

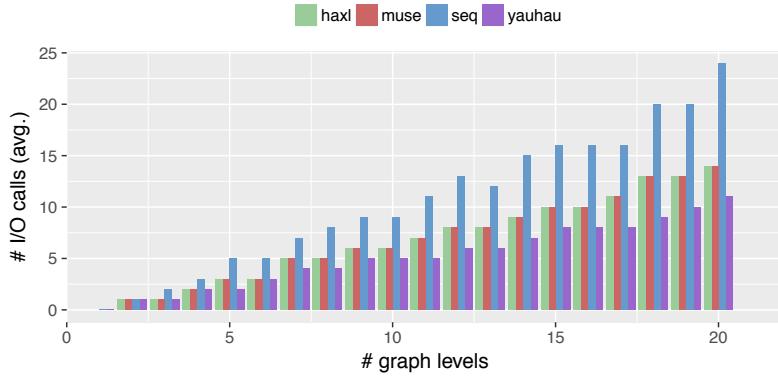


Figure 4.12: Results: Baseline.

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`(`ff_compute`), `req->io` (to simulate `io(ff_req*(...))`), `subfunction` and `map`. 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 4.11 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.

### 4.5.2 Experimental Setup

We designed three experiments to measure different aspects of Yauhau and compare it 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 [155]. We performed every measurement 20 times with different (fixed) random seeds and report the averages.

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

Table 4.2: Experimental Setup Level Graphs

exp.	indep. var.	dep. var.	# lvl.	pr. subf.
baseline	# lvl.	# I/O calls	1-20	0
conc. I/O	# lvl.	latency	1-20	0
modular	pr. subf.	# I/O calls	20	0-0.4

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 `io` 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 4.2 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.

### 4.5.3 Results

Figure 4.12 shows the baseline comparison, see Table 4.2. The plot shows that batching in Ÿauhau is superior to Haxl and Muse because our rewrites manage to batch across levels in the code. In this set of experiments, Ÿauhau 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 `io` 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 [158], 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 4.13 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 `io` 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 4.14. 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

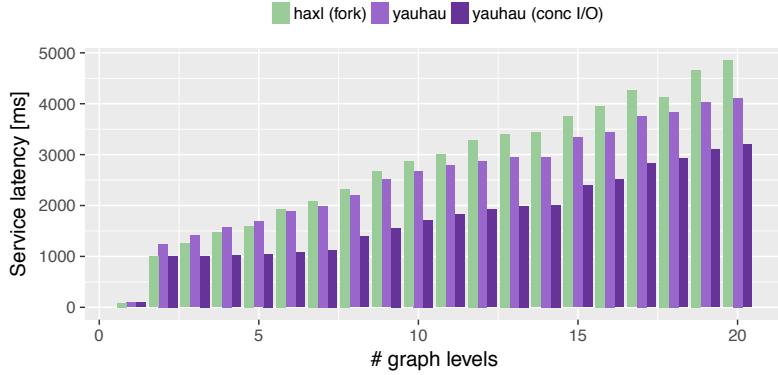


Figure 4.13: Results: Concurrent I/O.

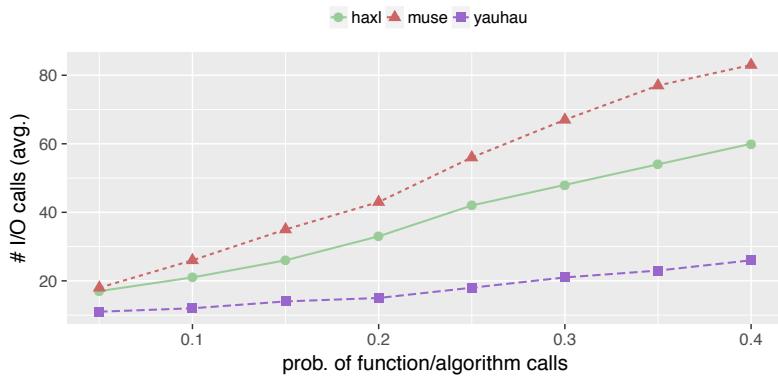


Figure 4.14: Results: Modular.

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 Yauhau 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.

## 4.6 CONCLUSION AND FUTURE WORK

Todays 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 Yauhau 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 Yauhau 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 Yauhau performs 21% less I/O than runtime approaches such as Haxl and Muse.

Yauhau 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.

**Postscript** Our EDD form is well-known in the functional programming language community as *administrative normal form (ANF)* [79]. However, we are unaware of a compiler that exploits ANF to derive a dataflow graph. Note that this paper, did not refer to stateful functions but rather to stateless ones. This was due to the fact that the nature of the functions did not matter for the transformations that we were targeting in this paper. We claim that the transformations remain valid even when functions have side-effects to their own private state. We have presented and formalized recently [74, 73] that a stateful function in our programming model behaves like a state monad (in Haskell) [223, 141]. That is, although the monad represents a computation with side-effects, it integrates into the program as a side-effect free function and preserves determinism. This is possible because side-effects are only local which is also true for our stateful functions. Based on this reasoning, it is possible to show that the transformations are semantic-preserving even when the programming model contains stateful functions. In fact, we also restricted ourselves to programs that only perform `read` operations. Supporting `write` operations remains an open research topic.



# 5 A FRAMEWORK FOR THE DYNAMIC EVOLUTION OF HIGHLY-AVAILABLE DATAFLOW PROGRAMS

**Prelude** This section covers the paper with the same title co-authored by Pascal Felber that I presented at the International Middleware Conference in 2014 [75]. The work addresses dataflow runtime system in highly loaded servers. Instead of parallelism, it presents an algorithm for the dynamic evolution of the program. Dynamic evolution, i.e., updating a program while it is executing without using additional hardware, is a particularly important problem. Many software systems today are provided as software as a service in a cloud-based setting. If the same system can service many customers then this reduces the deployment costs. At the same time, this puts much more load on a single program such that downtimes for updates are not an option anymore. This paper argues that a program that executes on the abstraction of a dataflow graph can update, extend or delete subgraphs without requiring any additional synchronization. Therewith, we deliver a very important argument for a dataflow-based runtime system.

## 5.1 INTRODUCTION

**Context and Motivations.** The recent evolutions in micro-processor design [47] have brought increasing numbers of cores into modern computers. While this provides higher computing power to applications, it also introduces several challenges related to concurrent programming. First, we need the right tools to develop concurrent applications that are both correct and efficient, i.e., can exploit the parallel computing capacities of the cores. Second, the added complexity of concurrent programs make debugging, testing, and software evolution much harder than for sequential programs, in particular when dealing with dependable systems.

In this paper, we tackle the problem of software evolution for applications that have availability requirements and must remain operational on a 24/7 basis. This is an important challenge because on the one hand multi-cores provide us with additional resources to scale up, e.g., by dedicating additional cores for concurrently servicing more clients; on the other hand, when an application must be updated to a new version, orchestrating the restart of all the components running on multiple cores and/or nodes is very challenging if service must *not* be interrupted. Therefore the “live update” problem can be studied from a new perspective in the light of modern multi-/many-core and Cloud architectures.

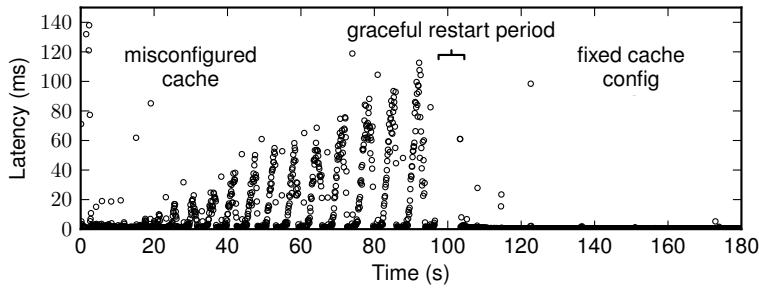


Figure 5.1: Graceful restart of a Jetty web server with the default configuration and 38 concurrent clients requesting one out of 10000 files of 20 kB size randomly every 20 ms.

To illustrate the impact of software reconfiguration on availability, we have experimented with Jetty’s web server,<sup>1</sup> a widely used application designed for servicing thousands of requests per second in highly concurrent settings. For every reconfiguration or activation of a new feature, Jetty needs to be gracefully shut down and restarted (new connection requests are dropped but requests in progress are allowed to finish). Restarts result in loss of computational state and persistent connections, hence penalizing request latency and throughput, and ultimately leading to customer dissatisfaction. Figure 5.1 shows a scenario in which the default configuration of Jetty’s cache size (2,048 files) leads to increasing delays for clients. This issue could be resolved by increasing the cache size (10,000 files), but resulted in a measured downtime of 8 s which is substantial considering an average request latency of 1 ms. Hence, an execution environment that allows us to update or reconfigure critical application services with negligible overheads and no interruption of service is highly desirable.

**Live Update.** Early approaches to live update rely on restarting a new instance of the program and transferring its state [105]. However, the costs of redundant hardware and the overheads of state transfer and synchronization can be substantial [107]. Focus has therefore turned towards the live update of running programs in-place, coined as *dynamic software updates* (DSU) [116]. Exchanging and altering an executing program without breaking correctness a particularly challenging [194]. In this paper, we use the notion of program state consistency to reason about update correctness [89]. The live update process identifies three key challenges when dealing with object-based systems as considered here: state transfer, referential transparency, and mutual references [78].

The first step is to find a quiescent state where an update can be applied safely. This is especially challenging in highly concurrent systems where many threads<sup>2</sup> access shared state at any given time. Hence, before performing a *state transfer*, we need to identify (i) the threads accessing the shared state to be updated and the (ii) time at which these accesses occur. We refer to these two elements of state quiescence detection as *participants* and *appointment* respectively.

Prominent approaches for live update systems force developers to introduce barriers into the program to achieve quiescence [107, 184]. In addition to the blocking behaviour (i.e., forcing threads to idle during update) and the performance penalty during normal execution, this approach is hard to scale to large multi-threaded programs because of their complexity and the risk of synchronization hazards (e.g., deadlocks) [90]. Further, while the timeliness of updates has been identified as a requirement [166], no live update mechanism we are aware of is based on a solid definition of time.

Another important challenge is to preserve *referential transparency* in the process of updating all pointers to the new state, i.e., one must still be able to consistently access state while the update is in progress.

Live updates become even more complicated when *mutual dependencies* between the components exist. Consider the example of a simple web server handling requests concurrently in a pipeline parallel fashion, as illustrated in Figure 5.2. If we want to change the processing of input/output from blocking to non-blocking (NIO), it is not sufficient to halt only the *load* and *reply* components because data containing file content might exist in between both. After the

<sup>1</sup><http://www.eclipse.org/jetty/>

<sup>2</sup>We use the terms of *threads* and *processes* interchangeably to refer to concurrent execution units.

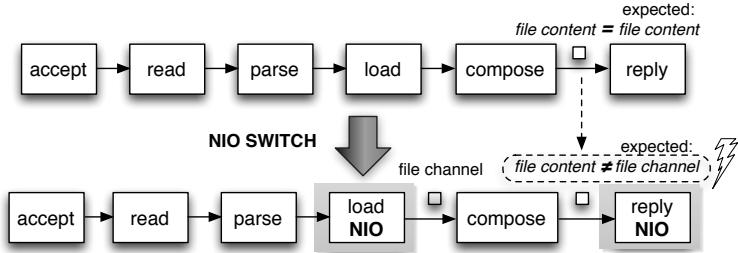


Figure 5.2: This simple HTTP server handles requests concurrently in a pipeline parallel fashion. To switch the server from blocking I/O to non-blocking I/O (NIO), we need to change the `load` to retrieve a file channel rather than the file content and the `reply` to use NIO’s `sendfile` primitive.

update the `reply` would not be able to handle these requests.<sup>3</sup> Even event-based approaches that support non-blocking live updates of individual components still have to block when the update spans more than one component and do not address the problem of undelivered messages [89].

Another desirable property for live updates is to provide *location transparency*, i.e., seamlessly handle updates of components irrespective of whether they are executing locally or remotely. This is particularly important in cloud computing environments where servers are virtualized and services can be dynamically relocated to other machines, e.g., for load balancing or fault tolerance.

Finally, most live update systems deal with small security patches. Support for complex updates to also enhance the system has gained attention only recently [184, 89]. In order to fully enable dynamic evolution of programs, updates must not be limited to new features inserted at well-defined points but must support *structural program changes*.

**Contributions and Roadmap.** It has been shown that structured software design helps providing safe and flexible live update capabilities [90]. In light of these observations, we do not aim at bringing live update capabilities to any existing programs or languages. We instead take the opportunity to argue for a different type of programming model [47] that can naturally solve the aforementioned challenges of dynamic software evolution: (data)flow-based programming (FBP) [163]. In addition to providing powerful abstractions for multi-core programming and for implicit parallel and distributed applications, we strongly believe that FBP offers a promising foundation to incorporate live updates.

In this paper we make several contributions. We first evaluate the FBP concepts with regard to the requirements of dynamic software evolution (Section 5.2). We then introduce in Section 5.3 a live update algorithm based on Lamport’s logical time to coordinate mutual dependency updates in a non-blocking fashion. We present in Section 5.4 FBP extensions for supporting scalable live updates that require no programmer intervention. In Section 5.5, we extend our algorithm by enabling unrestricted live updates even to the structure of the dataflow graph itself. We describe the implementation of our approach in the *Ohua* dataflow framework in Section 5.6 and present a case study for the dynamic development of the web server from Figure 5.2 in Section 5.7. We also evaluate the efficiency, timeliness and overheads of our algorithm for different types of updates. We finally discuss related work in Section 5.8 and conclude in Section 5.9.

## 5.2 FBP FOR LIVE UPDATES

FBP can be found at the heart of many advanced data processing systems today where parallel and concurrent processing is key for scalability: IBM’s data integration systems [123], database engines [63, 93], data streaming [40, 212], declarative network design [151] and even web servers [229]. Our Ohua dataflow framework targets a more general approach by introducing FBP as a parallel programming model, similar to Cilk [28], but in a truly implicit way: The dataflow dependency graph is derived from a program written in the functional language Clojure. Dataflow dependency graphs are widely considered to be a sound basis for parallel computations and their reasoning [4]. Respectively, dataflow represents a good abstraction for our live update algorithms. This section evaluates flow-based programming as a core foundation to support live updates.

<sup>3</sup>As a matter of fact, restarting the whole server would also involve discarding such state.

```

1 ; classic Lisp-style
2 (ohua (reply (compose (load (parse (read (accept "80"))))))))
3
4 ; or using Clojure's threading macro to improve readability
5 ; (which transforms at compile-time into the above)
6 (ohua (-> "80" accept read parse load compose reply))

```

---

Figure 5.3: Ohua-style HTTP Server Algorithm in Clojure.

```

1 class Reply extends Operator {
2     // stateless operator
3     @Function Object[] reply(Socket s, String resp) {
4         OutputStreamWriter writer = new OutputStreamWriter(s.getOutputStream());
5         writer.write(resp); // send response
6         writer.close();
7         s.close();
8         return null;
9     }
10 }

```

---

Figure 5.4: Ohua-style Operator Implementation in Java.

### 5.2.1 Referential Transparency by Design

In FBP, an algorithm is described in a directed acyclic<sup>4</sup> *dataflow graph* where 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 arc is connected to exactly one input port and one output port. An operator continuously retrieves data one packet at a time from its input ports and emits (intermediate) results to its output ports.

In Ohua, the dataflow graph is not explicitly created in the program or via a visual tool as is the case in other dataflow systems. Instead, it is derived from a functional program implemented in Clojure where functions represent operators implemented in Java. The language separation is meant to help the developer to understand the difference between functionality and algorithm. The algorithm of the web server example of Figure 5.2 is given in Listing 5.3.

In the classic dataflow approach [16, 62], operators are fine-grained stateless instructions. In contrast, FBP operators are small functional code blocks, which are allowed to keep state. This programming model is similar to message-passing with actors, which currently gains momentum in languages such as Scala [101]. FBP cleanly differentiates between functionality such as loading a file from disk and the web server algorithm: the former is used to implement the latter. FBP algorithms make reasoning about complex programs easier by hiding the implementation details (functionality). An operator neither makes any assumptions nor possesses any knowledge about its upstream (preceding) or downstream (succeeding) neighbours. Therewith, operators are context-free and highly reusable. As an example, Listing 5.4 shows the implementation of the `reply` operator in Ohua.

Finally, an FBP program defines the communication between operators, which at runtime translates into data dependencies, via arcs at compile-time rather than at runtime. This solves the problem of concurrent state accesses and allows for an implicit concurrent race-free program execution. Ohua’s programming model is similar to that of purely functional languages that provide referential transparency by design, with the major difference that operators are allowed to keep state. Operators do not however share state, only pointers in the realm of an operator can point to the state that changes during a live update. For example, Ohua operators can choose whether they want their state to be managed by the runtime engine or on their own. In the latter case, Ohua requires getter and setter functionality for state access. This allows us to update all

---

<sup>4</sup>While FBP does not define restrictions on the graph structure, we restrict ourselves to acyclic graphs in this paper for simplicity reasons.

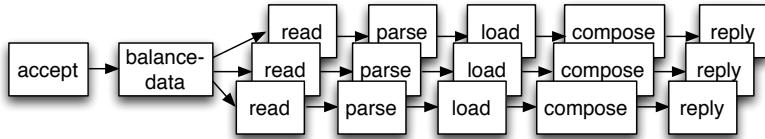


Figure 5.5: HTTP server handling requests in 3-way parallel.

```

1 (ohua
2   (let [accepted (accept 80)]
3     (balance 3 accepted (-> read parse load write reply)))

```

Figure 5.6: Explicit Data Parallelism via Clojure Macros.

operator-managed state pointers via a single function call and respectively preserves the referential transparency property.

Note that state handling is not exclusively related to live updates but is also required for other tasks, e.g., checkpointing to implement fault tolerance or operator migration to adapt to varying workloads. The state of a program is not only defined over the state of the operators but also includes the in-flight packets among the arcs. Packets stored inside the arcs correspond to function calls in the synchronous programming model and therefore do not count towards program state, which is sensitive to live updates. Nevertheless, as seen in the NIO Switch example of Figure 5.2, in-flight packets become problematic upon mutual dependencies. We defer this problem to the next section and assume for now that all arcs are drained before such an update can proceed.

### 5.2.2 Sequential Operator State Access

Although operators are executed concurrently, which introduces implicit pipeline parallelism, concurrent execution of the same operator is not allowed in FBP. Therewith, the only remaining case where shared state might exist is when replicating an operator to process packets in parallel. Figure 5.5 shows an example of the web server graph answering requests in a 3-way parallel fashion. FBP does not however address data parallelism as it has no knowledge on whether a part of a flow graph can be replicated by splitting the packet flow. Therefore, this form of parallelism cannot be directly exploited by Ohua's execution engine. It requires either the help of the developer or a graph inspection algorithm. Note that, in both cases, parallelism is introduced at the level of the flow graph and not its runtime representation. For instance, Listing 5.6 relies on a user-defined `balance` macro (code omitted) to insert a `balance-data` operator, replicate the code provided as the last parameter three times, and create data dependencies to set up the graph in Figure 5.5.

Although, the mapping of operator instances to threads is dynamic, no concurrent execution of the same operator instance is permitted. Each operator in the graph becomes an independent instance at runtime. Respectively, operator state access remains strictly sequential and therewith one can know at any point in time which thread is executing an operator and accessing its operator state, thus precisely identifying the participants in the state quiescence problem.

## 5.3 A TIME-BASED ALGORITHM

In order to safely apply a live update, we must first find the right point in time when operator state is quiescent. Certainly the scheduler has all execution knowledge of which operator is currently being executed on which thread and which operator is currently inactive. There are, however, several arguments against involving the scheduler. First, FBP does not declare any scheduler interface or other scheduling details. Hence, a scheduler-based solution cannot directly leverage FBP abstractions and would lack generality. Second, FBP does not state how often and when an operator might return control to the scheduler. As such either the scheduler must forcefully interrupt operator execution or timeliness of a live update cannot be guaranteed. Most importantly,

the mutual dependency problem can not be solved using the scheduler, as already explained in the NIO Switch example of Figure 5.2. Even when the scheduler blocks execution of both the `load` and the `reply`, in order to make sure both operator states are quiescent, packets in between these two operators expect the old blocking I/O API. The simple solution of draining the packets among the arcs between the `load` and the `reply` once again forces the `load`, the `compose` and the `reply` to idle while applying the update. Hence, a notion of time with respect to the computation is required to define a solid concept of timeliness for live updates and perform non-blocking mutual dependency updates.

### 5.3.1 The Right Point in Computation Time

Another way to reason about a dataflow graph without violating FBP principles is in terms of a distributed system, i.e., a set of independent processes (with operators as functionality) communicating with each other by exchanging messages (information packets) via FIFO channels.

Reasoning about time in a distributed system based on the happened-before relationship of events was described in Lamport’s seminal paper [137]. It provides the foundation for Chandy and Lamport’s algorithm to capture a global snapshot of a distributed system [41], which defines the notion of computation time and explains the associated reasoning about the states of the participating processes. The central idea is to inject a *marker* into the dataflow to mark a specific time  $t$  in computation in a decentralized fashion. The marker travels along the FIFO channels of the system. On arrival at a process, the local computational state is at time  $t$  as defined by the happened-before relationship of message arrival events. The process then captures its state and propagates the marker to all outgoing links. A global snapshot is consistent with the states of all processes gathered at time  $t$ . The algorithm is solely based on the marker concept and the happened-before relationship of message arrival events at the processes. Further, it operates in a purely non-blocking fashion as a natural part of data processing and allows us to specify a concrete point in computation time for operators to capture their state, or in our case apply an update.

### 5.3.2 Marker-based Update Coordination

Our approach to decentralized update coordination is presented in Algorithm 2. The marker propagation for mutual dependencies adheres to the principles of Lamport’s snapshot algorithm to preserve consistency. We extend the algorithm to not only capture the state of an operator on marker arrival, but also update it. An update is injected as an *update marker* in the dataflow graph (see Section 5.4.2). The marker contains the unique identifier of the operator to be updated as well as the functionality to update its state and the operator implementation. Operators propagate the update marker from their input to their output ports through the flow graph (Lines 16–22). Whenever the marker encounters the target operator, the update is performed inline with the computation (Lines 2–6) at a time  $t$  in computation.

In order to coordinate a mutual dependency update, we piggyback in an update marker  $m$  as many markers as there are mutually dependent downstream operators to be updated at the same point in time. Once  $m$  has been applied, these markers are propagated downstream at time  $t$  in the computation (Lines 8–11). Hence, in order to switch our HTTP server flow to NIO (see Figure 5.2), we submit one marker for the `load`, which piggy backs another one for the `reply` operator. At time  $t$  the marker arrives at the `load` operator. The update is performed and the marker targeted for the `reply` is propagated downstream. On its way to its destination the remaining update marker defines a clear point in time at which the update of the `reply` is safe to be performed. All packets sent by the `load` operator at an earlier time  $s < t$  that require the old blocking API are located downstream of the marker while all packets sent at a later time  $u > t$  that require the new NIO API are upstream. Safety of the update process is guaranteed by the FIFO ordering of packets inside the arcs and immediate propagation of arriving markers inside the operators (see Lines 17–19). Finally, the last update marker reaches the `reply` operator at time  $t$  and performs the update. Both updates were performed without blocking any of the operators.

---

**Algorithm 2:** Marker-based Update Algorithm

---

**Data:** operator  $o := (\mathbb{I}, \mathbb{O})$  consisting of input and output ports; marker  $m$  arriving on input port  $i \in \mathbb{I}$  at time  $s \leq t$ ;  $j$  the joined representation of all arrived markers  $m$  (with the same id)

```

1 if  $m$  arrived on all  $k \in \mathbb{I}$  then
2   if  $m.target = o.id$  then                                // operator update ( $s = t$ )
3     oldState  $\leftarrow o.getState();$ 
4     updatedState  $\leftarrow m.updateState(oldState);$ 
5     updatedOp  $\leftarrow m.updateOperator(o);$ 
6     updatedOp.setState(updatedState);
7     if  $j.depends \neq \emptyset$  then
8       for dependency marker  $c \in j.depends$  do           // coordination (enforce update
9         consistency:  $s = t$ )
10        |  $c.setDependent(true);$ 
11        |  $o.broadcastInOrder(c);$ 
12      end
13    else
14      |  $j.setCleanup(true);$ 
15      |  $o.broadcastOutOfOrder(j);$ 
16    end
17  else
18    if  $j.isDependent()$  then                            // marker propagation
19      |  $o.broadcastInOrder(j);$                            // consistency ( $s = t$ )
20    else
21      |  $o.broadcastOutOfOrder(j);$                       // timeliness ( $s < t$ )
22    end
23  foreach  $k \in \mathbb{I}$  do  $k.unblock();$ 
24 else
25  if  $m.isCleanup()$  then                                // termination
26    |  $o.broadcastOutOfOrder(m);$ 
27    foreach  $k \in \mathbb{I}$  do  $k.unblock();$ 
28    ;
29     $j \leftarrow \emptyset;$                                 // drop all subsequently arriving markers  $m$ 
30  else
31    |  $j.depends \leftarrow j.depends \cup m.depends;$ 
32    if  $m.isDependent()$  then
33      | |  $i.block();$ 
34      | |  $j.setDependent(true);$ 
35    else
36      | | // initial marker detected, no mutual dependency upstream
37    end
38  end
39 end

```

---

### 5.3.3 Marker Joins

Up to now, we assumed that operators only have one input port. Typically, operators with more than one input port merge or join the data arriving from the different input ports. So does our algorithm with respect to the piggy-backed markers and its type (Lines 29, 32). According to Lamport's algorithm, operators with more than one input port have to wait for the update marker to arrive at all input ports before it can be propagated (Line 1). Until this point in time, packets from input ports that have already received the marker must not be dequeued (Lines 30–35). Note that the algorithm only blocks input ports that received a marker with a dependency to an upstream update (Lines 9, 30–33).<sup>5</sup> Although all input ports have to see a marker before the propagation is safe, input ports without an upstream update are not blocked as packets arriving

---

<sup>5</sup>In our algorithm, *blocking* does not lead to idling threads but rather restricts the functionality available for execution.

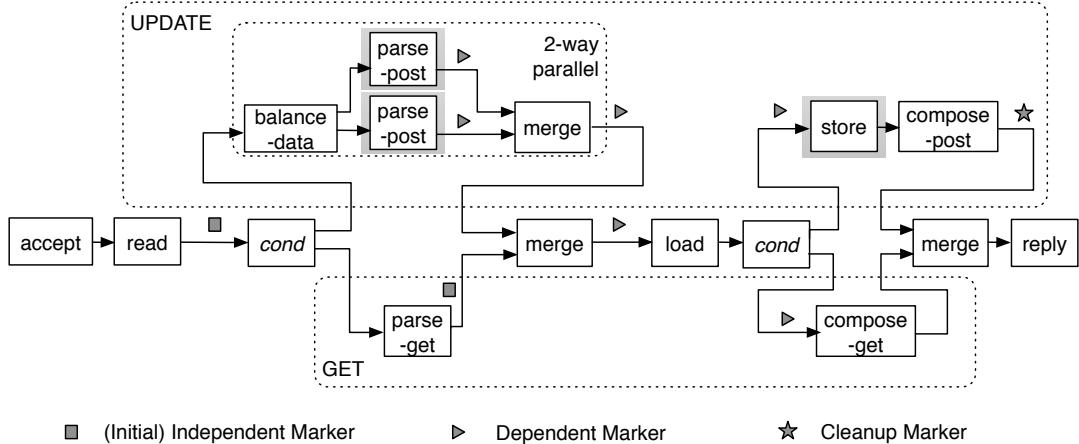


Figure 5.7: An extended version of our HTTP server that additionally supports UPDATE requests to the files it provides. An UPDATE of a file shares functionality with a GET whenever possible and returns the previous version of the updated file.

on these ports are not influenced by the downstream update (Lines 34–36). This seems counter-intuitive because arcs represent data dependencies and hence a downstream operator should always depend on all operators upstream. We describe such a scenario in the context of the version of our HTTP server in Figure 5.7 that was enhanced to handle updates to the provided resources. Similar to the semantics of data structures in common languages, our server does not only update a file but also returns the content of the previous version. As such, GET and UPDATE functionality share operators of the flow graph, e.g., `accept`, `read`, `load` and `reply`. Both `cond` operators dispatch data based on the type of the request. Since, parsing the new content for a resource update might require more time, we execute this step in a two-way parallel fashion. A mutual dependency update for the functionality of `parse-post` and `store` operators is injected as an independent marker. After updating the `parse-post` operators the adjacent downstream `merge` performs a marker join of two dependent markers. In the second `merge` that funnels UPDATE and GET requests to the `load` we encounter the situation where the initial marker from the GET branch does not define an upstream dependency (Lines 33–35). In this case it is safe to allow further propagation of GET requests because their further processing remains unaffected by the second part of the mutual dependency update. Finally, after the dependent marker has arrived at the `store` and finished the update, we send a marker to clean up existing markers in the graph (Lines 12–15). This is required because our propagation algorithm has no knowledge of the structure of the data flow graph. Therefore, propagation happens via broadcasting the markers. The cleanup marker resolves the case in our HTTP flow graph where the dependent marker arrives from the GET branch at the last `merge`. Here the `merge` must block this port as it has no knowledge of the location of the searched target and might be required to coordinate the arriving markers towards a defined time  $t$ . If no marker would arrive among the input ports, the algorithm would deadlock. To avoid this, we propagate a cleanup marker out of band to penalize processing as little as possible. Once the `merge` receives this marker it unblocks all ports and signals that no coordination towards an update of marker  $m$  is required any more (Lines 25–28). It cleans all received update information and drops all future markers of type  $m$ .

### 5.3.4 Deadlocks

The arrival of a packet on an input port is controlled by the operator algorithm, which requests packets via its input ports. Therefore, a marker can only arrive at an input port if there are no more packets in front of it, i.e., it is located at the head of the queue associated with the incoming arc. The introduction of blocking into marker propagation is an invasive step that directly influences the operator algorithm. The marker join algorithm is non-deterministic by nature: it allows arbitrary dequeuing behaviour from all input ports that have not seen the marker yet and blocks all dequeuing from the rest. As a result, the operator algorithm must adapt to this aspect, yet it does not have the notion of a blocked port. To an operator, a blocked port looks like one with no data currently available. It may therefore decide to back off and delay processing until data becomes available instead of querying another input port. This deterministic behaviour can lead to deadlocks when the operator algorithm decides to wait for data to arrive on a blocked port.

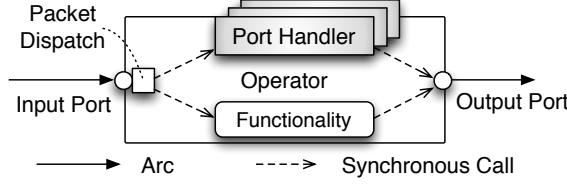


Figure 5.8: Port handler extensions for FBP operators.

The classic example for this type of problematic join is a deterministic merge with packets forwarded in a predefined order. In practice, however, non-deterministic merges are far more common when combining packets from a parallel pipeline. For the rest of this paper we assume that the execution semantics of the language avoid this type of deadlock.

### 5.3.5 State

Referential transparency and state transfer during an update are respectively solved by FBP and our marker-based approach. The challenge of mutual dependency updates is to find all state in the system that was derived from the program version before the update and either flush it before applying the update, or update it. The marker makes sure that old state is flushed from the (dependent) incoming arcs before the dependent operator is updated. Still, some old state may hide in any of the operators in between the mutually dependent operators. The update manager has therefore to decide whether the old state is considered harmful to the system. This might be the case if the update fixes a bug where state of other operators was corrupted and needs to be corrected. In such a case all these (downstream) operators need to be included into the update. In our future work, we intend to provide more advanced solutions to remove this burden from the user applying the update.

### 5.3.6 Timeliness

Finally, the moment in computation time  $t$  when an update is to be applied must not match the time  $v$  when the initial update marker  $m$  is inserted into the flow graph. If  $m$  travels through the graph in FIFO order with the data, it arrives at its target operator at time  $v$ . In Algorithm 2 the operator is updated on arrival such that  $t = v$ . It is straightforward to extend the algorithm to wait for a certain amount of time and apply the update at a later point  $t > v$ . Note, however, that this is possible only if the marker is independent, i.e., the current update is not dependent on an upstream update. The important insight is that FIFO marker propagation is only required to coordinate mutual dependencies (Lines 17–21). When finding the update target, we do not need to adhere to this rule and we can utilize out-of-order propagation possibilities among the arcs (Line 14), such as out-of-band processing among TCP connections if available. As a result, it is possible to deliver timely updates even at  $t < v$ .

## 5.4 FBP EXTENSIONS

This section describes the extensions to incorporate our marker-based update algorithm into the FBP model in a scalable and location transparent manner.

### 5.4.1 Structured Operators and Packet Dispatch

The FBP model consists of three major abstractions: arcs, operators, and a dataflow graph. A scalable design must preserve low complexity in terms of operator implementation and algorithm construction. Existing FBP systems focus primarily on the actual functionality. We reach beyond by structuring an operator into *operator functionality* (the operator code), *port handlers*, and a

```

1 class UpdateHandler extends PortHandler {
2     List<InputPort> blocked;
3     void arrived(InputPort i, UpdateMarker m) {
4         // implementation of the update algorithm
5     }
}

```

Figure 5.9: Update port handler stub in Ohua.

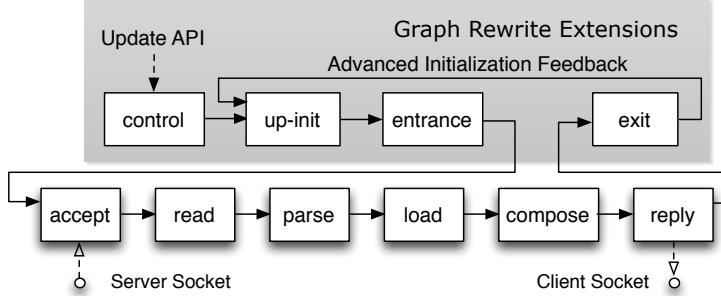


Figure 5.10: Pre-runtime rewriting of the HTTP flow graph.

*packet dispatch mechanism*, as depicted in Figure 5.8. While the operator functionality is provided in the classical way, the port handlers and the packet dispatcher are features of the runtime system that each operator instance inherits. Port handlers are meant to provide common functionality to all operators and therefore treat them as black boxes with respect to the implemented functionality. Still, the structure of an operator, i.e., its input and output ports, is known to a port handler.

We also refine the notion of packets by distinguishing between *meta-data* and *data packets*. Markers are examples of the former class, while the latter correspond to information packets as known from FBP. Each port handler registers for one or more types of meta-data packets. The packet dispatcher extends packet retrieval and makes sure handlers take responsibility for meta-data packets they have registered for. In contrast, data packets are always dispatched to the operator functionality. Since port handlers are meant as an extension of the operator functionality, they also consist of sequential context-free code blocks. The programming interface for port handlers is illustrated in Listing 5.9.

One handler is allowed to register at to multiple input ports. We further require that all calls in the realm of an operator are synchronous and the packet dispatcher is stateless, such that all dequeue operations among input ports preserve the FIFO processing order of the packets in the data flow graph.

This enhanced operator structure, together with the packet classification and automatic dispatch, provide a powerful extension framework to implement not only our coordinated update algorithm but also runtime features such as checkpointing or logging in a scalable and distributed manner.

#### 5.4.2 Dataflow Graph Transformations

The major strength of FBP resides in the abstraction of the flow graph itself. In essence, the arcs of the graph define the data dependencies between the operators at compile-time rather than at runtime. FBP does not define the implementation (array, shared memory queue, TCP connection, etc.) of the arcs but only their (FIFO) semantics. This provides independence from the actual execution infrastructure. The clear structural decomposition of the algorithm into small context-free operators allows the runtime system to exploit pipeline parallelism on any distributed architecture, e.g., multi-core, cluster, WAN, etc. Since our algorithm and its extensions strictly adhere to these principles, they inherit this location transparency property. Nevertheless, the injection of the update marker into the flow graph still requires knowledge on the current location of all *source operators*, i.e., operators without incoming arcs.

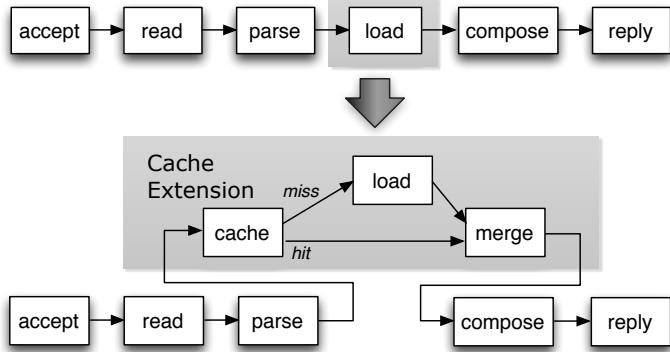


Figure 5.11: Extension of the HTTP flow graph with a Cache.

### Flow Graph Entrance and Exit

We address this problem via rewriting of the dataflow graph. Just like the above extensions to the operator, this rewriting neither influences the construction of the dataflow graph nor the functionality of operators or even the design of port handlers. It can be applied before runtime to the entire dataflow graph and, as such, it is independent of the actual algorithm implemented by the structure and operators of the graph and leaves both unchanged.

Our rewriting, as depicted in Figure 5.10, wraps the flow graph with an **entrance** operator and an **exit** operator. The former has outgoing arcs to all source operators in the flow graph while the latter is connected via incoming arcs to all operators that have no outgoing arcs, also referred to as *target operators*. All arcs and operators added during rewriting are valid FBP components and can be executed by the very same runtime system. Computation however only take place in the original graph and all extension operators remain silent at runtime. There is consequently no reason for the runtime system to move these operators, e.g., in order to balance the computation across multiple processes or nodes in a cluster. Therefore, there is no need anymore for our update marker injection to identify the source operators in the flow graph because we created a single entry point: the **entrance** operator. Scalability to a large number of source or target operators depends on the arc implementation. For example, in Ohua, a network arc maps to a ZeroMQ<sup>6</sup> connection and a fast message broker network that seamlessly scales to thousands of concurrent connections.

Finally, an additional **control** operator provides an API to submit update requests from the external world. The operator translates these requests into markers and sends them downstream the dataflow graph.

### Advanced Transformations

Our rewriting is not limited to the three aforementioned operators. In Figure 5.10, we also added an **up-init** operator that performs offline preparation of the updates. Furthermore, when the original flow graph is split and deployed on multiple nodes, each of the subflows represents a valid dataflow graph by itself and is eligible for rewriting. While the **control** operator remains on the initial node, the **up-init** operator can be transposed within the subflows to achieve local initialization of the updates. Since, no additional arcs from inside the original flow graph to the local **up-init** exist, the update algorithm needs to be extended. When an update marker arrives at the target operator it does not apply the update but sets a flag in the marker and sends it downstream. The marker propagates through the rest of the flow graph until it reaches the **exit** operator. Upon identifying the flag, the **exit** operator notifies the local **up-init** via a feedback arc. From that point on the algorithm works as before: the **up-init** performs the initialization and forwards the marker into the subflow where it hits the target operator (again) and applies the update.

<sup>6</sup><http://zeromq.org/>

## 5.5 STRUCTURAL UPDATES

Up to this point we have addressed updates spanning one or multiple operators, which are akin to updates of small functions in a program. The step from live updates to dynamic development does however require the capability to update the program, i.e., the dataflow graph, itself. In this section our focus extends to structural updates to the flow graph. It is important to stress that no new FBP features are added, as restructuring is solely based on the FBP extensions and our marker-based update algorithm. It follows that, when a port handler is executing, all input and output ports of the associated operator are inactive and can thus be rewired. We further assume that all structural changes are validated before submission by an algorithm that verifies that the update does not lead to violations at any of the downstream operator interfaces.

For example, in Figure 5.11 we extend our simple HTTP server flow with a `Cache` operator to remove disk I/O and improve request latency. The packets sent down the `hit` arc must adhere to the same structure as packets coming from the `Load` operator. As validation of a flow graph extension is a compile-time concern, it can be performed as a check before the actual structural rewriting is applied. Hence, rewriting is valid if the resulting dataflow graph contains neither unconnected arcs nor orphaned ports.

### 5.5.1 Coordinated Reconnections

Update markers for operator changes carry either a completely new functionality or a property of the existing operator functionality to be changed. Structural updates are also marker-based and, as such, they adhere to our notion of computation time and must not break the FBP abstractions. Changes can only be applied to the local operator, i.e., a port handler may only disconnect incoming arcs of the local input ports.

We classify three structural change operations that alter the flow graph structure: delete, insert, and replace. Each of these operations is essentially composed of a series of disconnection and reconnection steps that must be coordinated across the flow graph. For example, the deletion of the subgraph  $o_i \rightarrow \dots \rightarrow o_n$ , as depicted in Figure 5.12, involves a disconnection step for detaching the incoming arc  $x$  from the input port of  $o_i$  and a reconnection step to reconnect it to the input port of  $o_{n+1}$ . To coordinate these steps, we define a *reconnection marker* that contains a set of reconnections. A *reconnection* is a projection from one input port to another, and a set of reconnections is always bijective.

The pseudo-code for the reconnection operations is shown in Algorithm 3. It focuses solely on the update part, as we already covered marker propagation in Algorithm 2 (including offline initialization of the flow graph extension). Each reconnection starts by disconnecting the target input port from its incoming arc (Line 3). Disconnected arcs that are not part of the flow graph to be replaced or deleted need to be reconnected either to the new flow graph (Line 6) or to an input port in the existing flow graph (Line 13). In the latter case the arc is attached to a dependent marker and propagated downstream the port (Lines 8–10). Dependent reconnections of arcs from the new flow graph are also coordinated through the existing graph (Lines 16–19). Note finally that reconnections inside the marker are order-sensitive, e.g., an operation that disconnects the incoming arc from an input port and reconnects it to the new flow graph must be performed before another operation that reconnects an arc to that same input port.

### 5.5.2 Distributed Rewriting

Figure 5.13 illustrates another coordinated insertion, but in a distributed context. It performs a two phase process for enhancing our web server with proxy functionality located at a different cluster node to load balance disk I/O. The first phase deploys the proxy flow. The actual structural rewriting happens in the second phase, where we extend the existing flow graph to not only dispatch existing requests to the proxy flow but also join proxied requests back into the pipeline. The structural rewriting performs the first insertion at the `Load` operator with two reconnections. Afterwards, a marker delivers the pending two reconnections to the `Reply` operator. All the original operators in the existing flow are preserved and both parts of the rewriting happen at the same time in computation.

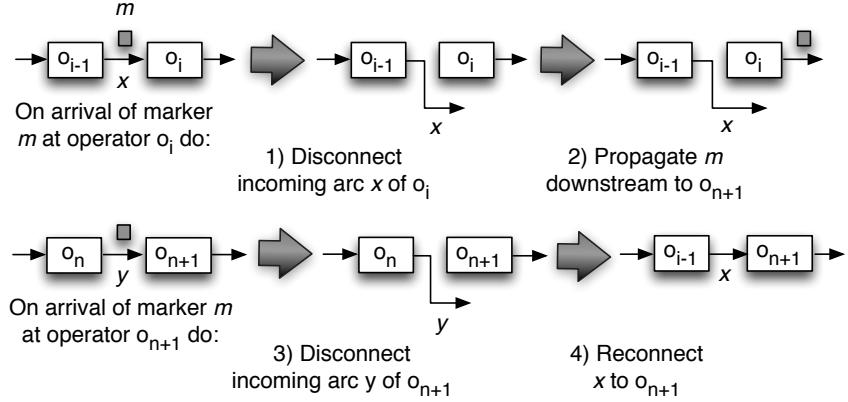


Figure 5.12: Algorithm steps for deleting the subgraph from operator  $o_i$  to operator  $o_n$ .

---

### Algorithm 3: Structural Update Algorithm

---

**Data:** operator  $o := (\mathbb{I}, \mathbb{O})$  and a marker  $m$  containing reconnections  $\mathbb{R}$  such that  $m.target = o.id$ ;

```

1 for reconnection  $r \in \mathbb{R}$  do                                // operator update
2   if  $r.in \in \mathbb{I}$  then
3     arc  $\leftarrow r.in.disconnect();$ 
4     if  $r.arc = 0$  then
5       if  $r.new \neq 0$  then                                // insert (arc to new subflow)
6          $r.new.connect(arc);$ 
7       else                                              // deletion (pending reconnection in old flow)
8          $r.dependent.reconnection.arc \leftarrow arc;$ 
9          $r.dependent.setDependent(true);$ 
10         $o.broadcastInOrder(r.dependent);$ 
11      end
12    else                                              // finalize reconnection
13       $r.in.connect(r.arc);$ 
14    end
15  else                                              // propagate dependent reconnections
16    for dependency marker  $c \in m.dependents$  do
17       $c.setDependent(true);$ 
18       $o.broadcastInOrder(c);$ 
19    end
20  end
21 end

```

---

The proxy rewrite also serves as a more concrete example of an extension that is deployed in a distributed fashion. Our rewriting algorithms do not make any assumptions on the location and deployment of the flow graph to be inserted. The deployment algorithm running underneath the FBP abstraction must however be able to cope with resulting deployment changes. For example, in Figure 5.12, if operators  $o_{i-1}$  and  $o_i$  were deployed together in the same process, then arc  $x$  would map to a simple queue. If operator  $o_{n+1}$  were located on in different process or even on a different node, then  $x$  would have to be converted to an IPC or TCP connection. Our algorithm does not make any assumption on the deployment of the flow graph to be inserted. This is a problem in the underlying implementation of the deployment algorithm, which is outside the scope of this paper.

We furthermore require that reconnections at a single operator are performed at the same time in order to keep computation consistent throughout the rewriting process. Therefore, operators with more than one incoming arc have to perform a marker join before reconnections on the local input ports are performed. This ensures that there exists a time  $t$  at which rewriting logically took place for this operator. Note that a new portion of the data flow graph starts to actively participate in computation as soon as the first arc has been reconnected, i.e., even before rewriting is fully complete. This allows us to perform structural changes to the flow graph in a purely non-blocking manner.

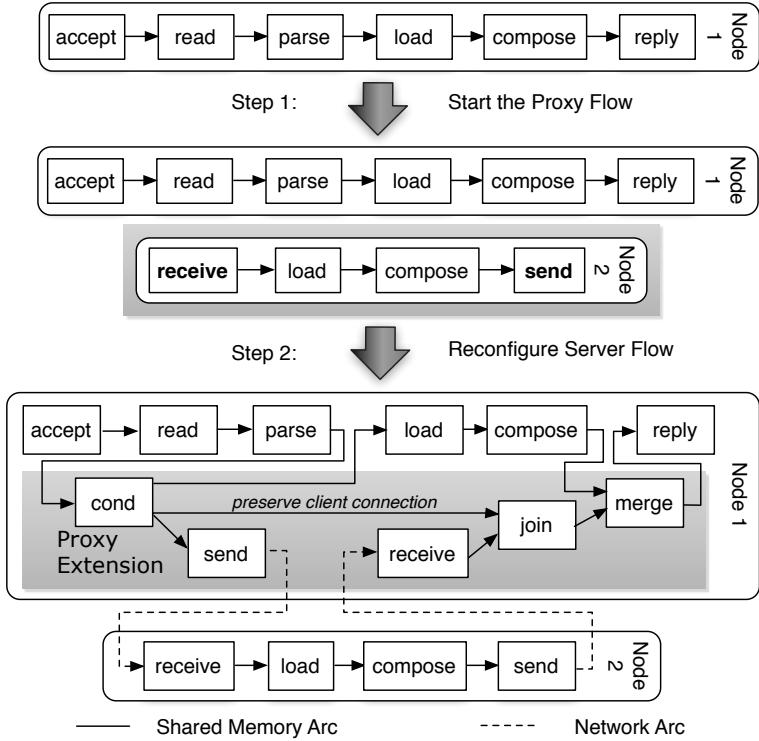


Figure 5.13: Proxy extension of the HTTP flow graph.

```

1 (update [com.server/reply com.server.update/reply (new ReplyStateTransfer)]
2   [com.server/load com.server.update/load])

```

Figure 5.14: The NIO-switch realized with Ohua.

## 5.6 PROGRAMMING MODEL

The key requirement in the definition of a programming model for live updates is certainly simplicity. It must allow the developer to submit its updates in a context-free fashion without having to reason about concurrency or locality. Ohua achieves this goal by defining a single `update` function. We first describe the update model related to operator and mutual dependency and later focus on the algorithm.

### 5.6.1 Operators and Mutual Dependencies

Listing 5.14 shows the code to submit the NIO-switch to the executing web server.

Note that the new code is defined in a different namespace than the old one. This is beneficial because dynamically evolving a program does not implicitly mean that the replaced code is buggy and needs to be discarded, e.g., some functions might be swapped with others for performance reasons on specific architectures. Ohua detects whether the update references an operator or a function. In the former case, Ohua takes care of finding all operators of the specified type in the currently executing flow graph and creates the necessary update requests. Mutual dependency updates are composed by submitting update pairs, as shown for the I/O switch in Listing 5.14. Dependencies are detected automatically and therefore the order of operator updates can be arbitrary. Furthermore, if state needs to be adapted to the new version of an operator (as exemplified for the `reply` operator in Listing 5.14), it is possible to specify a `state transfer` object that implements a predefined interface to return an updated state on the basis of the old version.

```

1 ; clojure namespace definition
2 (ns com.server)
3
4 ; web server defined in a function
5 (ohua
6   (defn web-server [port]
7     (-> port accept read parse load compose reply)))

```

---

Figure 5.15: The web server defined as a function in Ohua.

```

1 ; Ohua import to use the defined web server function
2 (ohua :import [com.server])
3
4 ; configure and launch the web server
5 (ohua (com.server/web-server "80"))

```

---

Figure 5.16: Web-server invocation in Ohua.

### 5.6.2 Algorithm Functions

Operators in Ohua represent functionality while the algorithm is written in Clojure. A program can be composed of many smaller algorithms encapsulated within an *algorithm function*. In all aspects related to the algorithm description, Ohua strictly follows the LISP programming model and syntax of Clojure. Listing 5.15 shows the definition of the web server algorithm as a function.

Finally, the invocation in Listing 5.16 assigns a port for the server to listen on.

To illustrate the update process, consider the scenario from Figure 5.10 of incorporating a cache to the web server. This process entails two steps. First, the developer writes an enhanced version of the web server algorithm utilizing a cache (operator), as shown in Listing 5.17.

Thereafter, this updated function is for all invocations of the web server in the running program. Listing 5.18 illustrates the submission of the update.

This simple model relieves the developer from the burden of specifying what exactly has changed in the algorithm and what insertions or deletions need to be performed in order to incorporate the update with the executing flow graph. Instead, Ohua replaces all subflows resembling to invocations of the old function with the new version along the guidelines of Algorithm 3. This function-level approach of updating the structure of a flow graph avoids the above identified deadlock problem for functions without additional I/O channels. The marker join is coordinate at the non-deterministic function entry that gathers the arguments. Additional marker joins inside the function are then coordinate towards that entry join. Due to space limitations, we delay a detailed analysis and a thorough discussion of more advanced updates of algorithm functions in Ohua to future work.

```

1 (ns com.server.update)
2
3 ; the web server algorithm utilizing the cache
4 (ohua
5   (defn web-server-with-cache [port]
6     (let [res-ref (-> port accept read parse)]
7       (let [res (cache res-ref)]
8         (-> (if (= res nil) (load res-ref) res)
9             compose reply)))

```

---

Figure 5.17: Cache-based Web Server.

---

```
1 (update [com.server/web-server com.server.update/web-server-with-cache])
```

---

Figure 5.18: Cache Insertion Update.

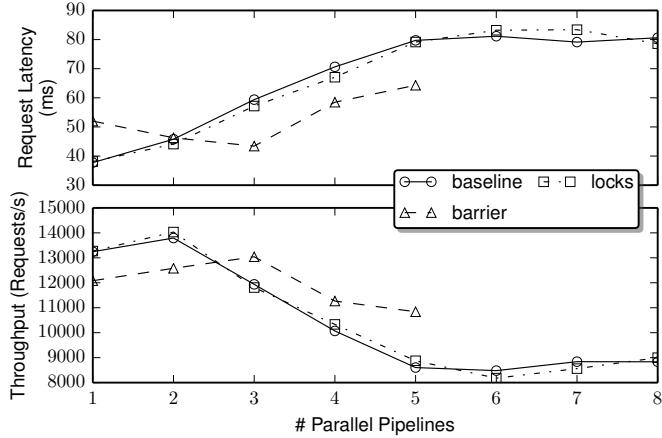


Figure 5.19: Runtime overhead using locks and barriers to simulate update points.

### 5.6.3 Integration with Clojure Programs

The actual insertion of new code within a running program is already supported by existing tools.<sup>7</sup> Clojure programs typically execute in a “read-evaluate-process-loop” (REPL), which is similar to a command line interface that can launch programs in another thread or even JVM process. Attaching to an executing REPL is also functionality that is readily available<sup>8</sup>, therefore loading new operators and algorithm functions packaged in jars is straightforward.

## 5.7 EVALUATION

In order to evaluate our evolution framework, we implemented the algorithms presented in this paper in Ohua and dynamically developed our flow graph with the NIO, cache, and proxy extensions described above (see Figures 5.2, 5.11 and 5.13). We followed the same methodology as other similar frameworks [164] to analyse web server performance. We deployed our Ohua web server on one cluster node, and clients were evenly spread across 19 other machines. The nodes in the cluster were equipped with 2 Intel Xeon E5405 CPUs with 4 cores each, 8 GB of RAM, Gigabit Ethernet, and SATA-2 hard disks. Note that the following experiments are not meant to evaluate the scalability of the web server deployment but focus on the performance of the live updates.

### 5.7.1 Runtime Overhead Evaluation

We first investigate the naive approach of inserting update points within the operators in order to perform an analysis of the runtime overhead. We used two flavours of update points: barriers and locks. One should point out that neither approach can handle on their own the type of updates that we addressed in this paper. Even for the simple mutual dependency update of switching the web server to use NIO, barriers would deadlock and locks would fail due to the packet inconsistency illustrated in Figure 5.2.

We do not explicitly consider the runtime overhead of Ohua as the introduced port handler switch barely encompasses a simple conditional expression at each input port, which is a vital part

---

<sup>7</sup><https://github.com/pallet/alembic>

<sup>8</sup><https://github.com/djpowell/liverepl>

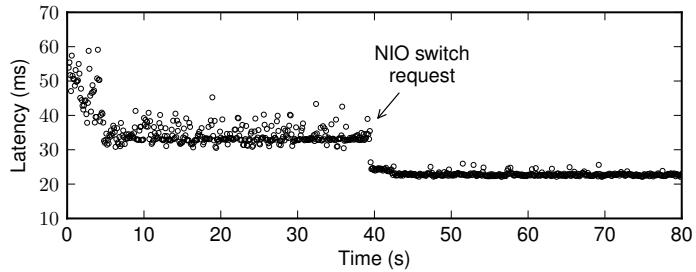


Figure 5.20: Request latency impact during a coordinated switch of the Load and Reply to use NIO’s sendfile primitive.

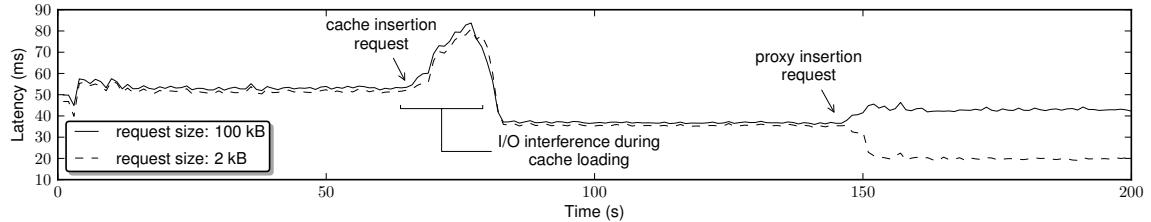


Figure 5.21: Reconfiguration experiment for cache and proxy insertion into the HTTP server flow graph.

of the runtime engine. In order to introduce more parallelism into the experiment, we parallelized the whole web server pipeline and assigned each `accept` with a different server socket. 1,000 clients evenly distribute their requests among sockets and ask for one out of 10,000 files of 512 bytes each. Update points are shared across operators of the same type to simulate updates on the level of functions.

Mean latency and request throughput, reported in Figure 5.19, indicate that the overhead for introducing locks is not visible even with 8 threads competing for a single lock. In contrast, barrier coordination exhibits overhead for a single and two parallel pipelines. Once the performance is capped by the I/O operations in the system, barriers seem to indeed have a beneficial effect on I/O coordination, which results in improved latency and throughput. The deadlock problem mentioned earlier does however render the barriers approach infeasible. Even in our simple experiment, which was designed to favour barriers, we were unable to execute a successful run above 5 parallel pipelines, i.e., 5 different server sockets, without encountering deadlocks.

### 5.7.2 Coordinated NIO Switch

For the coordinated switch of our web server to support the NIO API, we concurrently executed 30 clients issuing requests with a delay of 1 ms. As previously, requests were distributed evenly at random across a set of 10,000 files of 50 kB stored on disk. Figure 5.20 shows request latency over the runtime of the experiment, averaged every 100 ms. At 40 s into the experiment, we performed the coordinated update of the Load and Reply operators. The graph shows that request latency drops immediately from an unstable average of 33 ms to a stable average of 21 ms with very few outliers. Note that there is no spike when the switch is performed. The update does not block ongoing messages and successive requests instantaneously benefit from the switch.

### 5.7.3 Dynamic HTTP Server Development

To support our claim that our update algorithms can sustain the evolution from simple live updates to dynamic development of highly concurrent systems, we perform the cache insertion and proxy extension in a single experiment, one after the other, on the initial version of the web server without

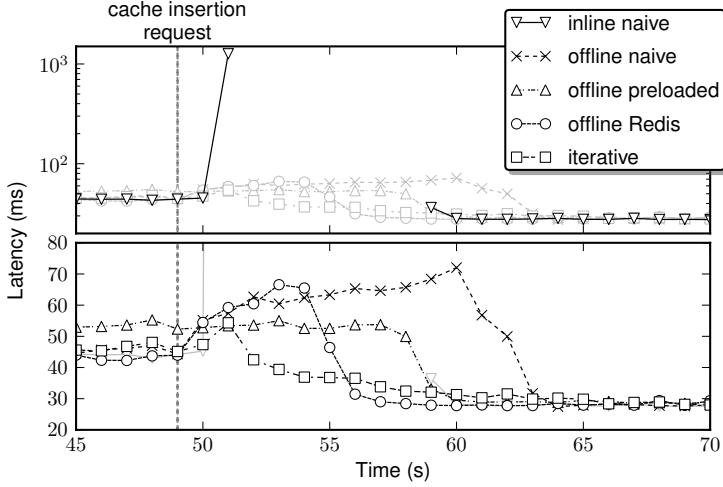


Figure 5.22: Various strategies to fill the cache during update.

NIO. We consider 30 clients requesting files of size 100 kB and another 30 clients issuing requests for files of size 2 kB. All clients pause 20 ms between consecutive requests.

In this dynamic evolution scenario, updates aim at improving request latency as much as possible. Therefore, we first insert a cache over all 20,000 files 65 s into the computation, which removes disk I/O overheads and reduces latency of both request types by 16 ms. Yet, due to the pipelined execution model, requests for small files are penalized by requests for larger files. The second rewriting operation after 145 s removes this penalty by inserting a proxy that dispatches requests for small files to a proxy server located on a different cluster node. Thereafter, small requests are processed in about 19 ms while requests for larger files have an average latency of 42 ms.

Note the small increase in request latency for large files by about 4 ms after the proxy insertion. This is due to the fact that we continue to use the same `Reply` operator for both request types and small file now compete with larger files, which results in this penalty on both sides. Note that we could easily remove this bottleneck by dynamically introducing separate `Reply` and `Send` operators for small-files.

#### 5.7.4 Cache Loading Strategies

Although the proxy insertion happens again without any noticeable impact on request latency, this is not the case for the cache rewriting. The increase in latency for about 15 s corresponds to the time where the new cache loads the files into memory. Although this happens in the `up-init` operator, outside of the graph part performing the computation, it nevertheless competes for disk I/O with the concurrent handling of requests. The developer needs to be aware of potential I/O contention and plan its update accordingly.

In Figure 5.22 we investigate different strategies to fill the cache. For simplicity we used a single set of 30 clients requesting a 50 kB file randomly out of 10,000 files every 20 ms. The upper plot shows that the naive approach, which does not use the `up-init` operator but inlines cache loading with request handling, blocks the requests and results in a downtime of almost 9 s. In the lower graph, we plot different strategies for loading the cache offline from request handling in the `up-init` operator. We compare the naive strategy of loading directly from disk (offline-naive) to the two alternatives of (1) loading the files from the Redis<sup>9</sup> in-memory key value store, and (2) having all files already preloaded into memory. While the former approach already greatly improves cache loading time, the latter demonstrates that our rewriting algorithm does not degrade performance when no contention exists. The slight latency overhead we can observe results from the heavily loaded JVM process in the preloaded case, but we can avoid it by loading files on demand (iterative approach shown in the graph).

<sup>9</sup><http://redis.io>

## 5.8 RELATED WORK

Research on DSU systems is often classified with respect to their target domain: programming languages, distributed systems, and operating systems. The more important classification however is based on the programming model, especially with respect to concurrency and parallelism.

**Disruptive Updates.** Procedural and object-oriented programming models such as C and Java support parallel execution explicitly. Updates have to follow the very same programming model in order to reason about the state of the program and enforce state quiescence. Therefore, recent approaches introduce synchronization in the form of update points [107, 184]. They either use barriers or define a more relaxed form of synchronization to remove runtime overhead during normal execution and avoid deadlocks. Update points require the update author to reason about the different states of the program. The state space increases with the degree of concurrency in the program and becomes even more complicated when additional update points are introduced to increase the timeliness of updates.

The alternative strategy of automatically inserting update points involves heavy static analysis and code restructuring, which again makes them scale poorly [106]. Early evaluation of Rubah, a DSU system for Java, reports 8 % overhead during normal processing and requires updates in the range of seconds. This is because Rubah does not support fine-grained online updates and must perform a full heap traversal to find the objects to be updated. Hence the larger the heap, the longer the traversal. This is a major concern because programs that rely on DSU instead of classical replication approaches typically have a large amount of state. Recent work [90] showed that update points are often limited to only the original version of the program and therefore the update process does not scale with the program evolution.

Our FBP-based algorithms do not suffer from any of these problems and every newly inserted operator or subflow can again be updated on-the-fly.

**Cooperative Updates.** The event-driven programming model [89] is better suited to reason about state and provide cooperative rather than disruptive updates [90]. This model essentially facilitates the development of a distributed system with independent processes communicating via a common message-passing substrate. Updates of one or multiple processes are coordinated by an update manager. The benefit is that single process updates are performed without blocking the other processes. In contrast, updates spanning multiple processes reach state quiescence by blocking all affected processes. Event-driven programming uses a processing loop similar to the operators in FBP. The model does not address, however, the nature of the IPC and the consistency problem of packets in transit during a coordinated update. The only work we are aware of that addresses this issue delays processing to drain packets between the two components until the whole update can be applied atomically [217].

Although less relevant to our work, one can finally mention, in the area of data streaming, the so-called teleport messages used to coordinate reconfiguration of parallel operators [213]. In software defined networks, consistent updates of switch configurations are coordinated based on tagged packets to achieve consistency [191].

We enhance the work on live updates by defining the concept of marker joins, which is vital for the definition of a solid notion of time in the program execution, and introduce a scalable language-based approach that applies mutual dependency updates in a non-blocking and infrastructure independent algorithm.

## 5.9 CONCLUSION AND FUTURE WORK

In this paper, we extended the dataflow-based programming model [163] to support live updates as well as the dynamic evolution of highly concurrent programs during runtime. We rely on a well-defined notion of time to reason about update consistency in the live update process and to perform structural modifications to the flow graph in a non-blocking fashion, even with mutual dependencies across many program parts. Since we based our extensions solely on the abstractions of the flow-based programming model, our algorithms work independently of the underlying architectures, e.g., multi-core systems or clusters. We conclude that FBP is appealing as a programming abstraction not only because it helps develop scalable programs for modern multi-core systems, but also because it is instrumental in solving many of the problems encountered with explicit parallelism at the language level.

**Postscript** This paper shows that a dataflow-based runtime enables non-blocking dynamic software updates. However, it does not provide a thorough programming model for updates. Such a model is subject of future work. Note that the update algorithms from this paper apply broadly to every runtime system that adheres to dataflow principles. This includes the runtime system for our stateful functional programming model with dataflow nodes for conditionals and `smap`.

# 6 CONCLUSION

In this thesis, we proposed *stateful functional programming (SFP)*, an implicit parallel programming model that incorporates state. A *stateful function* call owns its private state for the entire time of the program’s execution. An *algorithm* composes stateful functions. This distinction between algorithms and stateful functions allows us to remove all other concurrency aspects from the programming model. We do not require the abstraction of a task or an IStructure to synchronize data access. Parallelism is truly implicit in our programming model. SFP programs may very well be compiled into a purely sequential program; they only provide the opportunity for the compiler to turn them into concurrent ones. As a matter of fact, we have shown in this thesis that our notion of algorithms already exists in many programs. So, writing SFP algorithms is nothing new to programmers in mainstream languages such as Java and C++. Our case study on internal parallelism of big data systems identified two design patterns: the iterator and observer design pattern. In combination with a loop construct, stacked iterators or observers implement a pipeline but most of the compilers do not parallelize it. Compilers that try to do so, often do not take state into account or require it to have special properties such as commutativity [220, 153]. In fact, the abstraction of loops as streams requires that the stages in the pipeline are essentially stateless to run them in a data parallel fashion [39]. All of these approaches do not support the construction of full-blown graphs that incorporate control flow and state. SFP algorithms do without requiring the programmer to learn a new concurrency abstraction or new declarative language. The loops (our `smap`), variable bindings and (stateful) function calls in our algorithms are fundamental language concepts. They are part of all mainstream imperative languages and even of the main functional ones.

Conceptually, a stateful function is present in many programming languages. For example, a stateful function maps to a class with a public function in object-oriented languages such as Java and C++. A struct with an associated function in a procedural language such as C and Rust is also a stateful function. Even the well-known state threads from Haskell fit our definition of a stateful function [141]. On a more abstract level, actors may also be seen as stateful functions. Actors have been studied recently as a stateful programming model for multi-cores due to their encapsulation of state [49]. But they are missing a composition language. This loose coupling was essential for the highly distributed settings they were designed for to tolerate node failures and frequently joining and leaving nodes. These aspects are not so important on a multi-core machine where several actors often implement an algorithm. Our SFP programming model can provide this composition and even further hide several aspects from the Actors programming model, such as for example mailboxes, from the programmer. Even microservices fit our model of a stateful function. Algorithms could compose microservices in backend server infrastructures of internet companies or in adaptive AutoSAR applications for embedded devices.

To foster this broad applicability, our programming model is best integrated into existing languages as an EDSL (embedded domain specific language) with a standalone compiler for algorithms. The compiler presented in this thesis is based on the call-by-need lambda calculus and translates algorithms into dataflow graphs. It uses the administrative-normal form known from compilers for functional languages as its intermediate representation to perform optimizations and to lower the code to dataflow [79]. The current compiler for `Ohua`, our implementation of the

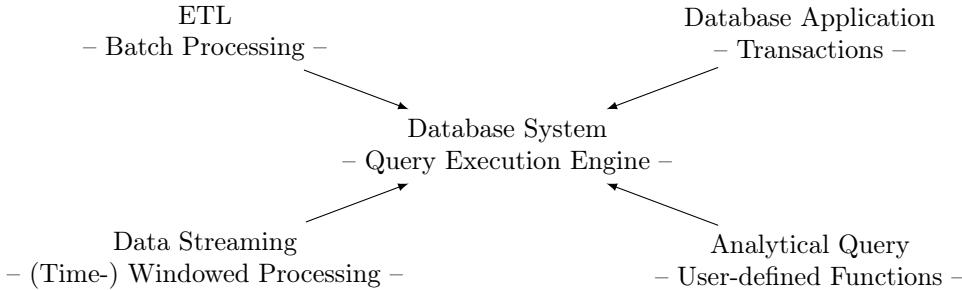


Figure 6.1: The data management ecosystem.

SFP programming model, is written in Haskell and features a full Rust integration<sup>1</sup>. It generates Rust code for algorithms such that Rust’s borrow checker verifies data-race freedom across stateful functions. Since our compiler derives the dataflow graph at compile-time, we can perform optimizations across the whole program. This is not possible for task-based programming models that build the dataflow graph at runtime. As an example, we looked at optimizing I/O in the particular context of a microservice-based architecture. In this domain, the need for concurrent I/O spawned a whole new area of new programming models for event-driven programming. The associated asynchronous events require a heavy restructure of the code and introduce new problems such as stack ripping. Even more new concepts such as fibers were proposed to bring back the simple semantics of synchronous (blocking) I/O calls. No new concepts are necessary in our SFP programming model. Concurrent I/O perfectly aligns with the concurrent execution in the rest of the dataflow graph. On top of that, we show that our compile-time I/O optimizations are superior to runtime-based approaches.

Runtime optimizations included the mapping of dataflow graph nodes to threads and especially non-blocking live updates in high-throughput, low latency server architectures. A primary concern to move from monolithic single server-based architectures to microservices is the need to update the system or parts thereof often without a noticeable impact on latency. This was also a main design principle behind actor-based languages such as Erlang. A single actor or microservice can easily be updated without impacting the rest of the system. In fact, the update appears as an atomic operation to the rest of the system. But often algorithms stretch many actors or microservices. Expanding this atomic update operation is much more complicated. Concurrent stateful programs running on multi-core hardware face the very same problem just on a different level. We have shown that there exists a live update algorithm on the foundation of the dataflow graph which is non-blocking and guarantees consistency. That is, without halting the system our update algorithm moves the system from one consistent state to another for node updates and even structural updates of the dataflow graph.

## 6.1 OPPORTUNITIES FOR SFP IN THE DOMAIN OF DATA MANAGEMENT SYSTEMS

We see this work as the foundation for many interesting aspects to research in the future. In this final section, we look at application domains where our programming model could provide benefits over state-of-the-art. For applications where our programming model is not yet sufficient, we derive the necessary additions in terms of the language, the compiler and the runtime system.

In this thesis, we studied stateful functional programming in the context of server and big data systems, i.e., data processing systems. As such, it is only natural to ask what benefits our programming model can provide in the overall context of data management systems. The ecosystem for data management encompasses systems for loading data into a data warehouse, so called ETL (extract, transform, load) systems, data streaming systems, a query execution engine and database applications (see Figure 6.1). All of these systems are tightly associated with a dataflow execution model.

<sup>1</sup><https://ohua-dev.github.io/ohua/>

**Reductions and Shared State** The dataflow graph of an ETL job is often developed in a GUI which becomes problematic for bigger graphs [26]. The concise algorithm abstraction of our programming model can be a solution to this problem. Note that this also accounts for dataflow applications on embedded devices which follow the very same development style. Data streaming systems are once more an incarnation of the dataflow model. Current state-of-the-art, such as Flink, uses the abstraction of a stream with support for higher-order functions rather than an implementation of the continuous query language (CQL) [11]. Our algorithms can produce a much more fine-grained dataflow graph for data streaming systems that incorporates state. Furthermore, the programming model for Flink data streams does not incorporate a notion of time. Time is important in these systems to implement so called windowed aggregations, i.e., aggregations over a time window. The code of the current state-of-the-art benchmark for streaming computation engines mixes threads and locks with dataflow in order to implement this functionality [48]. In order to address these short-comings, the stateful functional programming model needs to evolve beyond `smap`. Consider for example the dataflow graph for the current state-of-the-art data streaming benchmark depicted in Figure 6.2a. In order to support this data processing pipeline, our programming model needs to support reductions and shared state. The `Filter` node is essentially a reduction which we did not define yet. Our model can also provide a clean implementation without the need for threads and locks if it accommodates the notion of shared state depicted in Figure 6.2b. One function can aggregate the newly arriving data into the state and another one emits the current state upon a time request. The time signal is then nothing but another input source to the program. Due to the shared data access, the program is non-deterministic which is represented by the non-deterministic merge that the compiler needs to insert (Figure 6.2c).

**Data Parallelism and Explicit State** Once reductions and shared state are supported, the pipeline from the benchmark can be implemented concisely. But the produced dataflow graph will scale poorly because it only exploits pipeline parallelism. Most of the speed in data streaming comes straight from data parallelism. In order to transform a node towards a data parallel execution, the compiler needs to know whether its function is stateless or contains state that can be partitioned. Currently, state is implicit and the compiler would have to query the host language in order to understand such properties. It would be better if the programming model supports a more explicit notion of state. This would enable the compiler to directly differentiate between stateful and stateless functions, the latter of which are applicable to a data parallel execution. Identifying state that can be partitioned is likely to involve a type system with types for state that can be partitioned.

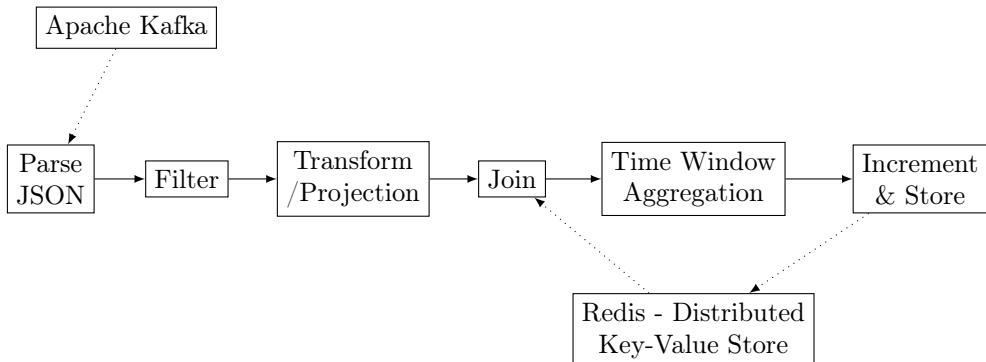
Data parallelism over partitioned state is also the key concept for scalability in database engines. Our `smap` function can already be used to implement simple operations such as scanning the records of a table and projecting onto some columns. Reductions, that can execute in parallel, would be the foundation to implement filtering or summation. Shared state allows to implement one of the most important but probably also one of the most challenging operators: the join. At this point, the (research) question arises: Is it possible to use our model as a foundation to implement a database engine with only a fraction of the code of state-of-the-art systems?

**Higher-Level Compilation** Optimizations play an important role in the scalability of database and data streaming engines. Since they are at the dataflow graph level, the more general ones apply to our execution model as well. Fusion is one such optimization that evaluates the performance of two consecutive operators<sup>2</sup> and decides whether these should be merged into a single operator. This turns costly inter-operator communication into a cheap function call. The compiler of the operator code (i.e., the C++ or Java compiler) can then further optimize the code across these two operators. In the database literature, this is known as (operator) fusion [117]. In the (functional) programming language literature exists a compiler optimization called stream fusion also known as deforestation [88, 132]. Compilers for imperative languages such as Java and C/C++ refer to this optimization as loop fusion. The goal is to remove materializations in between consecutive calls to higher-order functions such as `map`, `filter`, etc. The simplest example turns the composition `(map g) . (map f)` into `map (f . g)`. This optimization also directly applies to our programming model and the presented compiler. Connecting these two notions of fusion is novel work. Only recently, researchers discovered this connection but only evaluated different implementation styles [195]. They did not address the inherent trade-off between (pipeline) parallelism and materialization.

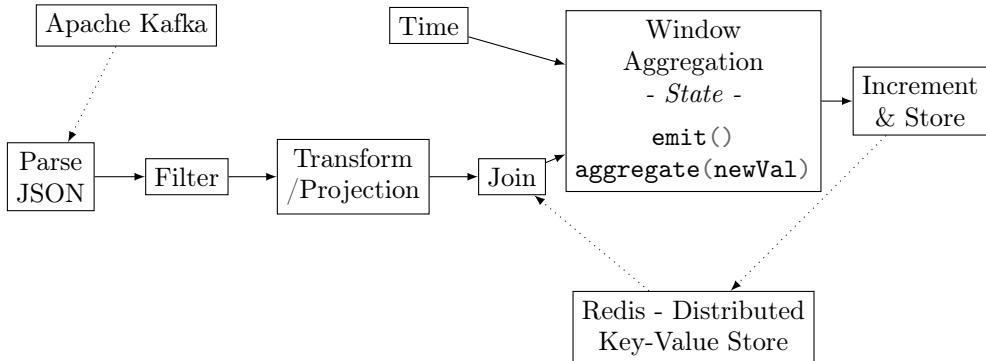
Our proposed programming model can also help to automatically parallelize user-defined functions (UDFs) in a database system. A database engine executes queries very fast because the SQL

---

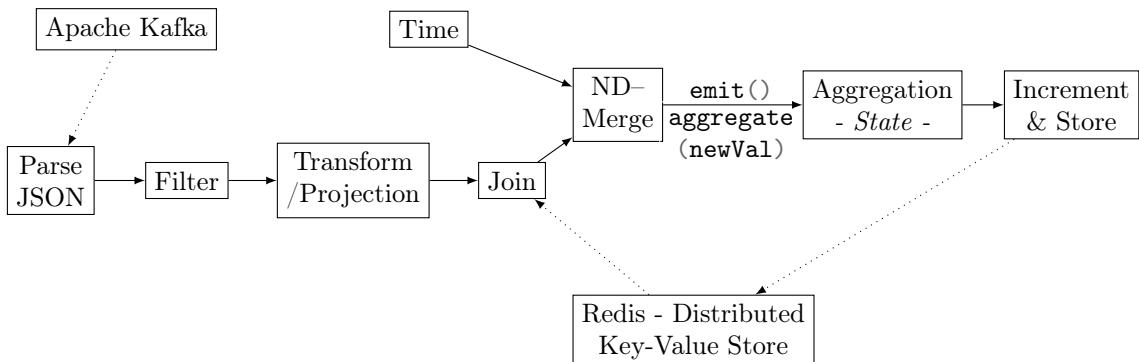
<sup>2</sup>Nodes in a database dataflow graph are called operators.



(a) Current version with threads and locks inside the **Time Window Aggregation** node.



(b) Explicit **Time** and shared state of the **Time Window Aggregation** node.



(c) Explicit dataflow synchronization via a non-deterministic merge.

---

Figure 6.2: Making time, shared data and synchronization explicit in the current state-of-the-art benchmark for streaming computation engines [48].

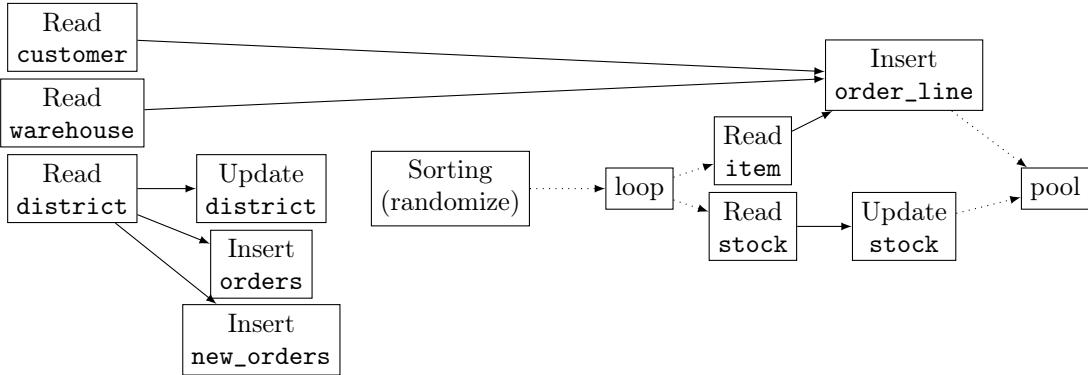


Figure 6.3: Inherent data flow between the database calls in the `NewOrder` query of the TPC-C benchmark.

constructs (`SELECT`, `FROM`, `WHERE`, etc.) translate into a standard set of operators (`Scan`, `Join`, etc.). Parallel implementations of these operators exist for decades now and allow the execution to scale across cores or even nodes in a cluster. But a call to a user-defined function in a query refers to an opaque piece of code to the database system and as such the query optimizer can not derive any parallelism. But user-defined functions are gaining importance because of two reasons. Data extracted from the internet is often semi-structured, i.e., `JSON`, `XML`, etc., and needs to be prepared before it can be used in a query [53, 96]. User-defined functions are now often written in languages such as `JavaScript` which have a much richer library system than `SQL`. To guarantee fast query execution, database vendors have to tell their clients that they better not use these libraries and perform analytics in `SQL` instead. On the other hand, the programming languages and libraries of UDFs enable whole new analytical applications [225]. One can even argue that the big success of Map/Reduce was in part because of the programming model which incorporated two types of user-defined functions, `map` and `reduce`. Our presented compiler translates algorithms into dataflow graphs, i.e., into the very same representation that an `SQL` compiler translates queries. We can exploit this fact to make user-defined functions scale. An `SQL` compiler could hook the dataflow graph of the user-defined function into the dataflow graph of the query that calls it. The `SQL` compiler knows how to parallelize the operators of the query while our compiler knows how to parallelize the stateful functions of the UDF. When combined, this leads to a compiler that allows even queries with large user-defined functions to scale.

SFP also has the potential to speedup database applications. Such applications are similar in spirit as the microservices that we studied for our I/O optimizations. Existing approaches leverage program analysis on (Java) source code to introduce laziness into the execution that allows to batch and prevents unnecessary database interactions at runtime [44, 46, 45]. A recent approach introduces an EDSL in Scala based on higher-order functions. The associated compiler moves parts of the application into stored procedures in the database system and enforces the automatic construction of index structures to speed up query execution. None of these approaches addresses the inherent concurrency that is present in these applications. Figure 6.3 shows the dataflow graph for the database calls in the `NewOrder` transaction of TPC-C, a benchmark for OLTP (online transaction processing) systems [43]. The graph contains ample opportunities for concurrency. It is presumably better to exploit the concurrency inside a single transaction rather than across transactions because it shortens the duration of a single transaction. Shorter transactions decrease the probability for lock contention inside the database system and in turn lead to higher transaction throughput.

**Recursion** In order to also address applications that were implemented using memory transactions, our programming model has to support recursion. Transactional memory is often used to exploit parallelism in *irregular applications*, i.e., applications that deliver non-deterministic results by design [134]. An example of such an application is the labyrinth benchmark contained in STAMP, a well-established benchmark suite for transactional memory [161]. It uses Lee's routine algorithm to find the shortest-distance path between a start and an end point in a grid such that paths do not share fields with other paths [226]. Many solutions are possible for a set of start and end points. The algorithm finds one of them using a transactional approach. When a transaction aborts, the changes are rolled back and the transactions re-executes with new data. This is where the recursion is. It is essentially a work-queue algorithm that adds aborted transactions to the queue and finishes when the queue is empty. When our programming model supports shared state and recursion then it is reasonable to ask whether it can even replace transactional memory.

## 6.2 OTHER DIRECTIONS

There are of course more applications that are worthwhile exploring using the idea of SFP. For example, functional reactive programming builds on top of a dataflow graph abstraction and may also be covered by our model. With respect to the aspect of providing safety in terms of data race freedom, interesting directions are the recent advances in linear type systems [25] and safe languages such as Rust [128]. Finally, we did not yet get to look into efficient scheduling algorithms for the dataflow graph. We think there is a lot of potential in leveraging the rich body of work on scheduling from the embedded computing community, e.g., for static dataflow graphs [146], for more dynamic models like KPN [37, 38], and for energy-efficient adaptive execution [92]. The applications, in the above studied and mentioned domains do not fall into this category for various reasons but parts of the dataflow graph may do. For such parts, a pre-computed schedule might be more efficient than work-stealing.

SFP is a foundation that is worthwhile and fun to explore. For every researcher who will take on this journey: I hope it will be as educative for you as it was for me.

# BIBLIOGRAPHY

- [1] *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
- [2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.
- [3] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into data at facebook. *Proc. VLDB Endow.*, 6(11):1057–1067, 2013.
- [4] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA ’11*, pages 499–518. ACM.
- [5] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. ATC, 2002.
- [6] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 4 2007.
- [7] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.
- [8] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS ’67 (Spring)*, pages 483–485, New York, NY, USA, 1967. ACM.
- [9] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke. Bridging the gap between hpc and big data frameworks. *Proc. VLDB Endow.*, 10(8):901–912, Apr. 2017.
- [10] Apache. Hadoop. <http://hadoop.apache.org/>.
- [11] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
- [12] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, May 1997.
- [13] Z. M. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’95*, pages 233–246, New York, NY, USA, 1995. ACM.
- [14] J. Armstrong. The development of erlang. ICFP, 1997.

- [15] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent programming in erlang. 1993.
- [16] Arvind and D. E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow architectures. Annual Reviews Inc., Palo Alto, CA, USA, 1986.
- [17] Arvind, K. P. Gostelow, and W. Plouffe. Indeterminacy, monitors, and dataflow. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 159–169, New York, NY, USA, 1977. ACM.
- [18] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, Oct. 1989.
- [19] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.
- [20] J. Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, Aug. 1978.
- [21] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013.
- [22] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, Mar. 2017.
- [23] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568, Berlin, Heidelberg, 1991. Springer-Verlag.
- [24] M. Beck and K. Pingali. From control flow to dataflow. In *Proceedings of the International Conference on Parallel Programming (ICPP)*, ICPP '90, 1990.
- [25] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, Dec. 2017.
- [26] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Köller, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel. Knime-the konstanz information miner: version 2.0 and beyond. *AcM SIGKDD explorations Newsletter*, 11(1):26–31, 2009.
- [27] C. Bienia and K. Li. Characteristics of workloads using the pipeline programming model. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, pages 161–171, Berlin, Heidelberg, 2012. Springer-Verlag.
- [28] 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.
- [29] R. L. Boccino Jr, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 4–4, 2009.
- [30] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008.
- [31] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 2011.
- [32] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. 1994.
- [33] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasilar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, Aug. 2010.
- [34] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

- [35] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.
- [36] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, Sept. 2008.
- [37] J. Castrillon and R. Leupers. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. Springer, 2014.
- [38] J. Castrillon, R. Leupers, and G. Ascheid. MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, Feb. 2013.
- [39] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’10, pages 363–375, New York, NY, USA, 2010. ACM.
- [40] 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.
- [41] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 1985.
- [42] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI ’06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.
- [43] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study. *ACM SIGMOD Record*, 39(3):5–10, 2011.
- [44] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment*, 5(11):1471–1482, 2012.
- [45] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). *ACM Transactions on Database Systems (TODS)*, 41(2):8, 2016.
- [46] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. Using program analysis to improve database applications. *IEEE Data Eng. Bull.*, 37(1):48–59, 2014.
- [47] A. A. Chien. Pervasive parallel computing: An historic opportunity for innovation in programming and architecture. PPoPP ’07, New York, NY, USA, 2007. ACM.
- [48] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.
- [49] S. Clebsch and S. Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. In *ACM SIGPLAN Notices*, volume 48, pages 553–570. ACM, 2013.
- [50] S. Clebsch, J. Franco, S. Drossopoulou, A. M. Yang, T. Wrigstad, and J. Vitek. Orca: Gc and type system co-design for actor languages. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):72, 2017.
- [51] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.
- [52] T. P. P. Council. Tpc-h benchmark specification. Published at <http://www.tcp.org/hspec.html>, 21:592–603, 2008.
- [53] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226. ACM, 2016.

- [54] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998.
- [55] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.
- [56] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. Cpu db: Recording microprocessor history. *Commun. ACM*, 55(4):55–63, Apr. 2012.
- [57] U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson. Data integration flows for business intelligence. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT ’09, pages 1–11, New York, NY, USA, 2009. ACM.
- [58] 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. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ ’14, pages 39–50, New York, NY, USA, 2014. ACM.
- [59] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI’04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [60] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon’s highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP ’07, pages 205–220, New York, NY, USA, 2007. ACM.
- [61] J. Dennis. A parallel program execution model supporting modular software construction. In *Proceedings of the Conference on Massively Parallel Programming Models*, MPPM ’97. IEEE Computer Society.
- [62] J. B. Dennis. Data flow supercomputers. *Computer*, 1980.
- [63] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *VLDB*, VLDB, 1986.
- [64] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO ’07, pages 21–33, Washington, DC, USA, 2007. IEEE Computer Society.
- [65] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 3–14. ACM, 2014.
- [66] E. W. DIJKSTRA. Two starvation free solutions to a general exclusion problem, 1978.
- [67] A. Discolo, T. Harris, S. Marlow, S. P. Jones, and S. Singh. Lock free data structures using stm in haskell. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS’06, pages 65–80, Berlin, Heidelberg, 2006. Springer-Verlag.
- [68] J. Donham. Introducing stitch. Technical report, 2014. [Online; accessed 4-May-2017].
- [69] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why stm can be more than a research toy. *Commun. ACM*, 54(4):70–77, Apr. 2011.
- [70] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous i/o for event-driven servers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.
- [71] M. Eriksen. Your server as a function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS ’13, pages 5:1–5:7, New York, NY, USA, 2013. ACM.
- [72] S. Ertel, J. Adam, and J. Castrillon. Supporting fine-grained dataflow parallelism in big data systems. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM’18, pages 41–50, New York, NY, USA, 2018. ACM.

- [73] S. Ertel, J. Adam, N. A. Rink, A. Goens, and J. Castrillon. Category-theoretic foundations of "stclang: State thread composition as a foundation for monadic dataflow parallelism", 2019.
- [74] S. Ertel, J. Adam, N. A. Rink, A. Goens, and J. Castrillon. Stclang: State thread composition as a foundation for monadic dataflow parallelism. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, pages 146–161, New York, NY, USA, 2019. ACM.
- [75] S. Ertel and P. Felber. A framework for the dynamic evolution of highly-available dataflow programs. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 157–168, New York, NY, USA, 2014. ACM.
- [76] S. Ertel, C. Fetzer, and P. Felber. Ohua: Implicit dataflow programming for concurrent systems. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 51–64, New York, NY, USA, 2015. ACM.
- [77] S. Ertel, A. Goens, J. Adam, and J. Castrillon. Compiling for concise code and efficient i/o. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 104–115, New York, NY, USA, 2018. ACM.
- [78] N. Feng, G. Ao, T. White, and B. Pagurek. Dynamic evolution of network management software by software hot-swapping. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, 2001.
- [79] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.
- [80] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in mantiscore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 119–130, New York, NY, USA, 2008. ACM.
- [81] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 37–44. ACM, 2007.
- [82] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 221–230, June 2010.
- [83] T. A. S. Foundation. Apache hbase. <https://hbase.apache.org/>, 2017. Accessed: 2017-03-22.
- [84] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. PLDI '98.
- [85] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayananurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: The pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, Aug. 2009.
- [86] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 231–242, Washington, DC, USA, 2011. IEEE Computer Society.
- [87] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 229–240, New York, NY, USA, 2013. ACM.
- [88] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232. ACM, 1993.
- [89] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. ASPLOS '13, New York, NY, USA, 2013. ACM.
- [90] C. Giuffrida and A. S. Tanenbaum. Cooperative update: A new model for dependable live update. HotSWUp '09, New York, NY, USA, 2009. ACM.

- [91] A. Goens, S. Ertel, J. Adam, and J. Castrillon. Level graphs: Generating benchmarks for concurrency optimizations in compilers. In *Proceedings of the 11th International Workshop on Programmability and Architectures for Heterogeneous Multicores, co-located with 13th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, MULTIPROG 2018, Jan. 2018.
- [92] A. Goens, R. Khasanov, M. Hähnel, T. Smejkal, H. Härtig, and J. Castrillon. Tetris: a multi-application run-time system for predictable execution of static mappings. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES'17)*, SCOPES '17, pages 11–20, New York, NY, USA, June 2017. ACM.
- [93] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 102–111, New York, NY, USA, 1990. ACM.
- [94] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.
- [95] J. Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986.
- [96] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923. ACM, 2015.
- [97] J. L. Gustafson. Reevaluating amdahl’s law. *Commun. ACM*, 31(5):532–533, May 1988.
- [98] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.
- [99] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, Proceedings of the IEEE, pages 1305–1320, 1991.
- [100] M. Hall, D. Padua, and K. Pingali. Compiler research: The next 50 years. *Commun. ACM*, 52(2):60–67, Feb. 2009.
- [101] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 2009.
- [102] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 1985.
- [103] T. Harris and S. Singh. Feedback directed implicit parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 251–264. ACM.
- [104] W. Hasenplaugh, A. Nguyen, and N. Shavit. Quantifying the capacity limitations of hardware transactional memory. In *7th Workshop on the Theory of Transactional Memory (WTTM)*, 2015.
- [105] C. Hayden, E. K. Smith, M. Hicks, and J. Foster. State transfer for clear and efficient runtime upgrades. HotSWUp '11.
- [106] C. M. Hayden, K. Saur, M. Hicks, and J. S. Foster. A study of dynamic software update quiescence for multithreaded programs. HotSWUp '12, Piscataway, NJ, USA, 2012. IEEE Press.
- [107] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. OOPSLA '12, New York, NY, USA, 2012. ACM.
- [108] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298. ACM, 1984.
- [109] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Parallel Architectures and Compilation Techniques (PACT), 2008 International Conference on*, pages 260–269. IEEE, 2008.

- [110] J. He, P. Wadler, and P. Trinder. Typecasting actors: From akka to takka. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA ’14, pages 23–33, New York, NY, USA, 2014. ACM.
- [111] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.
- [112] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA ’93, pages 289–300, New York, NY, USA, 1993. ACM.
- [113] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
- [114] C. Hewitt, P. Bishop, and R. Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.
- [115] R. Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS ’08, pages 1:1–1:1. ACM.
- [116] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, Nov. 2005.
- [117] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, Mar. 2014.
- [118] C. A. R. Hoare. Communicating sequential processes, 1978.
- [119] G. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, Sept. 1997.
- [120] J. Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.
- [121] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, 2000.
- [122] N. Hunt, P. S. Sandhu, and L. Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *Proceedings of the 2011 15th Workshop on Interaction Between Compilers and Computer Architectures*, INTERACT ’11, pages 63–70. IEEE.
- [123] IBM. Infosphere datastage data flow and job design. <http://www.redbooks.ibm.com/>, July 2008.
- [124] P. Jiang and X. S. Liu. Big data mining yields novel insights on cancer. *Nat Genet*, 47(2):103–104, 02 2015.
- [125] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
- [126] W. M. Johnston, J. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.
- [127] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. Bluedbm: An appliance for big data analytics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA ’15, pages 1–13, New York, NY, USA, 2015. ACM.
- [128] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, Dec. 2017.
- [129] A. Kachayev. Muse. Technical report, 2015. [Online; accessed 4-May-2017].
- [130] A. Kachayev. Reinventing haxl: Efficient, concurrent and concise data access. Technical report, 2015. [Online; accessed 4-May-2017].
- [131] A. Khrabrov and E. De Lara. Accelerating complex data transfer for cluster computing. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud’16, pages 40–45, Berkeley, CA, USA, 2016. USENIX Association.
- [132] O. Kiselyov, A. Biboudis, N. Palladinos, and Y. Smaragdakis. Stream fusion, to completeness. In *ACM SIGPLAN Notices*, volume 52, pages 285–299. ACM, 2017.

- [133] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.
- [134] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Cascajal. How much parallelism is there in irregular applications? In *ACM sigplan notices*, volume 44, pages 3–14. ACM, 2009.
- [135] L. Kuper and R. R. Newton. Lvars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84, New York, NY, USA, 2013. ACM.
- [136] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
- [137] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.
- [138] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9:690–691, September 1979.
- [139] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.
- [140] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, Apr. 1979.
- [141] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM.
- [142] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.
- [143] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.
- [144] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.
- [145] 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.
- [146] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.
- [147] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.
- [148] F. Li, A. Pop, and A. Cohen. Automatic extraction of coarse-grained data-flow threads from imperative programs. *IEEE Micro*, 32(4):19–31, July 2012.
- [149] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM.
- [150] J. Liu, E. Racah, Q. Koziol, and R. S. Canon. H5spark: bridging the i/o gap between spark and scientific data formats on hpc systems. *Cray user group*, 2016.
- [151] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SOSP*, 2005.
- [152] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.
- [153] S. Manilov, C. Vasiladiotis, and B. Franke. Generalized profile-guided iterator recognition. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 185–195, New York, NY, USA, 2018. ACM.

- [154] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing mapreduce for multicore architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep.*, 2010.
- [155] S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’14, pages 325–337, New York, NY, USA, 2014. ACM.
- [156] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: Better strategies for parallel haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell ’10, pages 91–102, New York, NY, USA, 2010. ACM.
- [157] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell ’11, pages 71–82, New York, NY, USA, 2011. ACM.
- [158] S. Marlow, S. Peyton Jones, E. Kmett, and A. Mokhov. Desugaring haskell’s do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pages 92–104, New York, NY, USA, 2016. ACM.
- [159] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS’15, pages 14–14, Berkeley, CA, USA, 2015. USENIX Association.
- [160] E. Meijer. Subject/observer is dual to iterator. 2010 Conference on Programming Language Design and Implementation (PLDI), Fun Ideas and Thoughts Session, 2010.
- [161] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. Citeseer, 2008.
- [162] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.
- [163] J. P. Morrison. *Flow-Based Programming*. Nostrand Reinhold, 1994.
- [164] D. Mosberger and T. Jin. Httpperf—;a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, Dec. 1998.
- [165] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI 08: Eighth Symposium on Operating Systems Design & Implementation*. USENIX, December 2008.
- [166] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. PLDI ’09, New York, NY, USA, 2009. ACM.
- [167] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992.
- [168] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud’hommeaux, H. W. Lie, and C. Lilley. Network performance effects of http/1.1, css1, and png. In *Proceedings of the ACM SIGCOMM ’97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM ’97, pages 155–166, New York, NY, USA, 1997. ACM.
- [169] R. S. Nikhil. *Id language reference manual*. Laboratory for Computer Science, MIT, July 1991.
- [170] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 385–398, 2013.
- [171] S. Okur and D. Dig. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE ’12, pages 54:1–54:11. ACM.
- [172] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen. A study and toolkit for asynchronous programming in c#. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1117–1127, New York, NY, USA, 2014. ACM.

- [173] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.
- [174] J. Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, volume 5. San Diego, CA, USA, 1996.
- [175] J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, Jan. 1996.
- [176] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.
- [177] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.
- [178] V. Pankratius, F. Schmidt, and G. Garretón. Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 123–133, Piscataway, NJ, USA, 2012. IEEE Press.
- [179] M. P. Papazoglou and W.-J. Heuvel. Service oriented architectures: Approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, July 2007.
- [180] D. Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–32, 53, July 2010.
- [181] C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (stm): Dissecting haskell stm applications on a many-core environment. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 67–78, New York, NY, USA, 2008. ACM.
- [182] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. pages 295–308. ACM Press, January 1996.
- [183] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: Moving bindings to give faster programs. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 1–12, New York, NY, USA, 1996. ACM.
- [184] L. Pina and M. Hicks. Rubah: Efficient, general-purpose dynamic software updating for java. HotSWUp '13.
- [185] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.
- [186] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.
- [187] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using dope: The degree of parallelism executive. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 26–37. ACM.
- [188] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. Ieee, 2007.
- [189] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.
- [190] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM.

- [191] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. *SIGCOMM '12*, New York, NY, USA, 2012. ACM.
- [192] P. V. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004.
- [193] A. Sbırlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, LCTES '12*, pages 61–70. ACM.
- [194] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw.*, 1993.
- [195] A. Shaikhha, M. Dashti, and C. Koch. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming*, 28, 2018.
- [196] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '95*, pages 204–213, New York, NY, USA, 1995. ACM.
- [197] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
- [198] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: High-throughput stream programming in java. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07*, pages 211–228, New York, NY, USA, 2007. ACM.
- [199] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. *ECOOP*, 2008.
- [200] L. Stadler, C. Wimmer, T. Würthinger, H. Mössenböck, and J. Rose. Lazy continuations for java virtual machines. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 143–152. ACM, 2009.
- [201] G. L. Steele, Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP '09*, pages 1–2, New York, NY, USA, 2009. ACM.
- [202] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A high-performance I/O architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.
- [203] X. Su, G. Swart, B. Goetz, B. Oliver, and P. Sandoz. Changing engines in midstream: A java stream computational model for big data processing. *Proc. VLDB Endow.*, 7(13):1343–1354, Aug. 2014.
- [204] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. Gramps: A programming model for graphics pipelines. *ACM Transactions on Graphics (TOG)*, 28(1):4, 2009.
- [205] M. A. Suleiman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10*, pages 147–156. ACM, 2010.
- [206] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.
- [207] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.
- [208] D. Syme, T. Petricek, and D. Lomov. The f# asynchronous programming model. In R. Rocha and J. Launchbury, editors, *Practical Aspects of Declarative Languages*, pages 175–189, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [209] S. Tasharofi, P. Dinges, and R. Johnson. Why do scala developers mix the actor model with other concurrency models? *ECOOP '13*.
- [210] 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.

- [211] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. CC, 2002.
- [212] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. CC '02, London, UK, UK, 2002. Springer-Verlag.
- [213] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. PPoPP '05, New York, NY, USA, 2005. ACM.
- [214] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.
- [215] R. Townsend, M. A. Kim, and S. A. Edwards. From functional programs to pipelined dataflow circuits. CC 2017, pages 76–86, New York, NY, USA, 2017. ACM.
- [216] A. Trivedi, P. Stuedi, J. Pfefferle, R. Stoica, B. Metzler, I. Koltsidas, and N. Ioannou. On the [ir]relevance of network performance for data processing. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 126–131, Berkeley, CA, USA, 2016. USENIX Association.
- [217] Y. Vandewoude, P. Ebriaert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.*, 33(12), Dec. 2007.
- [218] 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. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, GreenCom'10, pages 229–236, 2010.
- [219] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.
- [220] T. J. K. E. von Koch, S. Manilov, C. Vasiladiotis, M. Cole, and B. Franke. Towards a compiler analysis for parallel algorithmic skeletons. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 174–184, New York, NY, USA, 2018. ACM.
- [221] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.
- [222] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990.
- [223] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.
- [224] P. Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.
- [225] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, et al. The myria big data management and analytics system and cloud services. In *CIDR*, 2017.
- [226] I. Watson, C. Kirkham, and M. Luján. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 388–398. IEEE Computer Society, 2007.
- [227] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *ACM SIGPLAN Notices*, volume 40, pages 439–453. ACM, 2005.
- [228] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.
- [229] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. SOSP, 2001.

- [230] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou. Non-determinism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 44–53. ACM, 2014.
- [231] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI’12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.
- [232] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazieres, and M. F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference, General Track*, pages 239–252, 2003.

## Declaration of contribution

**Paper:** Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. In Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15). ACM, New York, NY, USA, 51-64.  
DOI: <https://doi.org/10.1145/2807426.2807431>

## Contributions following the CRediT<sup>1</sup> model

Role	Authors
Conceptualization	Sebastian Ertel, Pascal Felber
Data curation	Sebastian Ertel
Formal analysis	Sebastian Ertel
Funding Acquisition	Christof Fetzer
Investigation	Sebastian Ertel
Methodology	Sebastian Ertel
Project administration	Christof Fetzer, Pascal Felber
Resources	Christof Fetzer
Software	Sebastian Ertel
Supervision	Christof Fetzer, Pascal Felber
Validation	Sebastian Ertel, Christof Fetzer, Pascal Felber
Visualization	Sebastian Ertel
Writing – original draft	Sebastian Ertel
Writing – review & editing	Pascal Felber

I declare that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Sebastian Ertel,

Christof Fetzer,

Pascal Felber,

<sup>1</sup>The CRediT model is the preferred model for publications in the *Science* magazin:  
<https://www.sciencemag.org/authors/science-journals-editorial-policies>  
<https://casrai.org/credit/>

## Declaration of contribution

**Paper:** Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Supporting Fine-grained Dataflow Parallelism in Big Data Systems. In Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'18), Quan Chen, Zhiyi Huang, and Pavani Balaji (Eds.). ACM, New York, NY, USA, 41-50.

DOI: <https://doi.org/10.1145/3178442.3178447>

## Contributions following the CRediT<sup>1</sup> model

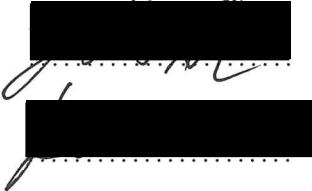
Role	Authors
Conceptualization	Sebastian Ertel
Data curation	Justus Adam
Formal analysis	Sebastian Ertel
Funding Acquisition	Jeronimo Castrillon
Investigation	Sebastian Ertel, Justus Adam
Methodology	Sebastian Ertel
Project administration	Jeronimo Castrillon
Resources	Jeronimo Castrillon
Software	Sebastian Ertel, Justus Adam
Supervision	Jeronimo Castrillon
Validation	Jeronimo Castrillon
Visualization	Sebastian Ertel
Writing – original draft	Sebastian Ertel
Writing – review & editing	Jeronimo Castrillon

I declare that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Sebastian Ertel,



Justus Adam,



Jeronimo Castrillon,



<sup>1</sup>The CRediT model is the preferred model for publications in the *Science* magazin:  
<https://www.sciencemag.org/authors/science-journals-editorial-policies>  
<https://casrai.org/credit/>

## Declaration of contribution

**Paper:** Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. 2018. Compiling for concise code and efficient I/O. In Proceedings of the 27th International Conference on Compiler Construction (CC 2018). ACM, New York, NY, USA, 104-115.

DOI: <https://doi.org/10.1145/3178372.3179505>

## Contributions following the CRediT<sup>1</sup> model

Role	Authors
Conceptualization	Sebastian Ertel, Andrés Goens
Data curation	Justus Adam
Formal analysis	Andrés Goens
Funding Acquisition	Jeronimo Castrillon
Investigation	Justus Adam, Sebastian Ertel, Andrés Goens
Methodology	Sebastian Ertel, Andrés Goens
Project administration	Jeronimo Castrillon
Resources	Jeronimo Castrillon
Software	Sebastian Ertel, Justus Adam, Andrés Goens
Supervision	Jeronimo Castrillon
Validation	Andrés Goens, Jeronimo Castrillon
Visualization	Sebastian Ertel
Writing – original draft	Sebastian Ertel, Andrés Goens (evaluation section)
Writing – review & editing	Andrés Goens, Jeronimo Castrillon

I declare that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

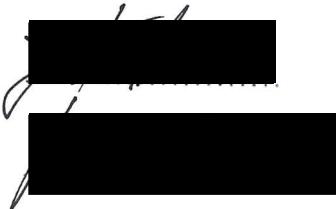
Sebastian Ertel,



Andrés Goens,



Justus Adam,



Jeronimo Castrillon,

<sup>1</sup>The CRediT model is the preferred model for publications in the *Science* magazine:  
<https://www.sciencemag.org/authors/science-journals-editorial-policies>  
<https://casrai.org/credit/>

## Declaration of contribution

**Paper:** Sebastian Ertel and Pascal Felber. 2014. A framework for the dynamic evolution of highly-available dataflow programs. In Proceedings of the 15th International Middleware Conference (Middleware '14). ACM, New York, NY, USA, 157-168.

DOI: <https://doi.org/10.1145/2663165.2663320>

## Contributions following the CRediT<sup>1</sup> model

Role	Authors
Conceptualization	Sebastian Ertel, Pascal Felber
Data curation	Sebastian Ertel
Formal analysis	Sebastian Ertel
Funding Acquisition	Pascal Felber
Investigation	Sebastian Ertel
Methodology	Sebastian Ertel
Project administration	Pascal Felber
Resources	Pascal Felber
Software	Sebastian Ertel
Supervision	Pascal Felber
Validation	Sebastian Ertel, Pascal Felber
Visualization	Sebastian Ertel
Writing – original draft	Sebastian Ertel
Writing – review & editing	Pascal Felber

I declare that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Sebastian Ertel,

Pascal Felber,

<sup>1</sup>The CRediT model is the preferred model for publications in the *Science* magazin:  
<https://www.sciencemag.org/authors/science-journals-editorial-policies>  
<https://casrai.org/credit/>