A Framework for the Dynamic Evolution of Highly-Available Dataflow Programs

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

Pascal Felber
Institut d'informatique
Université de Neuchâtel, Switzerland
pascal.felber@unine.ch

ABSTRACT

Many distributed applications deployed on the Internet must operate continuously with no noticeable interruption of service. Such 24/7 availability requirements make the maintenance of these application difficult because fixing bugs or adding new functionality necessitates the online replacement of the software version by the new one, i.e., a "live update". Support for "live update" is therefore essential to allow software evolution of critical services. While the problem of live update has been widely studied and several techniques have been proposed (e.g., using group communication and replication), we propose in this paper an original approach for the dataflow-based programming model (FBP). An interesting property of FBP is its seamless support for multi- and many-core architectures, which have become the norm in recent generation of servers and Cloud infrastructures. We introduce a framework and new algorithms for implementing coordinated non-blocking updates, which do not only support the replacement of individual software components, but also modifications of structural aspects of the applications independently of the underlying execution infrastructure. These algorithms allow us to transparently orchestrate live updates without halting the executing program. We illustrate and evaluate our approach on a web server application. We present experimental evidence that our live update algorithms are scalable and have negligible impact on availability and performance.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Programming—*Dataflow languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

1. INTRODUCTION

Context and Motivations. The recent evolutions in microprocessor design [6] 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.

Middleware '14, December 08 - 12 2014, Bordeaux, France

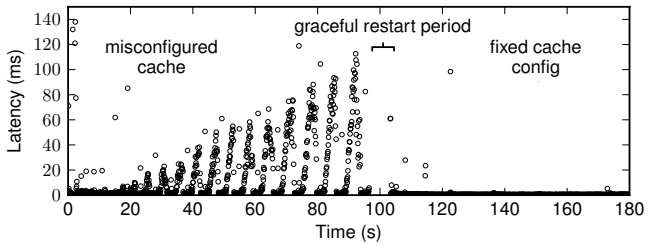

Figure 1: Graceful restart of a Jetty web server with the default configuration and 38 concurrent clients requesting one out of 10000 files of 20 kB size randomly every 20 ms.

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.

To illustrate the impact of software reconfiguration on availability, we have experimented with Jetty's web server,[1] 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 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

[1] http://www.eclipse.org/jetty/

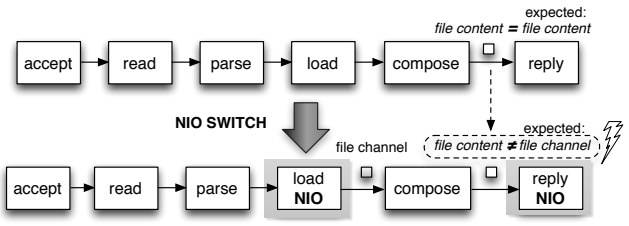

Figure 2: This simple HTTP server handles requests concurrently in a pipeline parallel fashion. To switch the server from blocking I/O to non-blocking I/O (NIO), we need to change the `load` to retrieve a file channel rather than the file content and the `reply` to use NIO's `sendfile` primitive.

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 [14]. However, the costs of redundant hardware and the overheads of state transfer and synchronization can be substantial [16]. Focus has therefore turned towards the live update of running programs in-place, coined as *dynamic software updates* (DSU) [17]. Exchanging and altering an executing program without breaking correctness a particularly challenging [26]. In this paper, we use the notion of program state consistency to reason about update correctness [10]. The live update process identifies three key challenges when dealing with object-based systems as considered here: state transfer, referential transparency, and mutual references [9].

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[2] 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 [16, 24]. In addition to the blocking behavior (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) [11]. Further, while the timeliness of updates has been identified as a requirement [23], 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 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 update the `reply` would not be able to handle these requests.[3] 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 [10].

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 [24, 10]. 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 [11]. 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 [6] that can naturally solve the aforementioned challenges of dynamic software evolution: (data)flow-based programming (FBP) [21]. 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 2). We then introduce in Section 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 4 FBP extensions for supporting scalable live updates that require no programmer intervention. In Section 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 6 and present a case study for the dynamic development of the web server from Figure 2 in Section 7. We also evaluate the efficiency, timeliness and overheads of our algorithm for different types of updates. We finally discuss related work in Section 8 and conclude in Section 9.

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 [18], database engines [8, 12], data streaming [4, 27], declarative network design [20] and even web servers [30]. Our Ohua dataflow framework targets a more general approach by introducing FBP as a parallel programming model, similar to Cilk [3], but in a truely 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 compu-

[2]We use the terms of *threads* and *processes* interchangeably to refer to concurrent execution units.

[3]As a matter of fact, restarting the whole server would also involve discarding such state.

tations and their reasoning [1]. 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.

2.1 Referential Transparency by Design

In FBP, an algorithm is described in a directed acyclic[4] *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 2 is given in Listing 1.

Listing 1: Ohua-style HTTP Server Algorithm in Clojure

```
1 ; classic Lisp−style
2 (ohua
3   (reply (compose (load (parse (read (accept "80")))))))
4
5 ; or using Clojure's threading macro to improve readability
      (which transforms at compile−time into the above)
6 (ohua
7   (−> "80" accept read parse load compose reply))
```

In the classic dataflow approach [2, 7], 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 [13]. 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 2 shows the implementation of the `reply` operator in Ohua.

Listing 2: Ohua-style Operator Implementation in Java

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

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

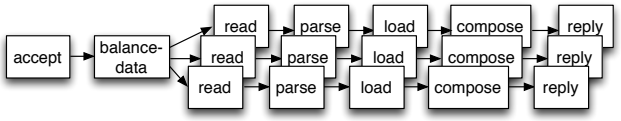

Figure 3: HTTP server handling requests in 3-way parallel.

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 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 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.

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 3 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 3 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 3.

Listing 3: Explicit Data Parallelism via Clojure Macros

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

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.

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 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.

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 [19]. It provides the foundation for Chandy and Lamport's algorithm to capture a global snapshot of a distributed system [5], 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.

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

3.2 Marker-based Update Coordination

Our approach to decentralized update coordination is presented in Algorithm 1. 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 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

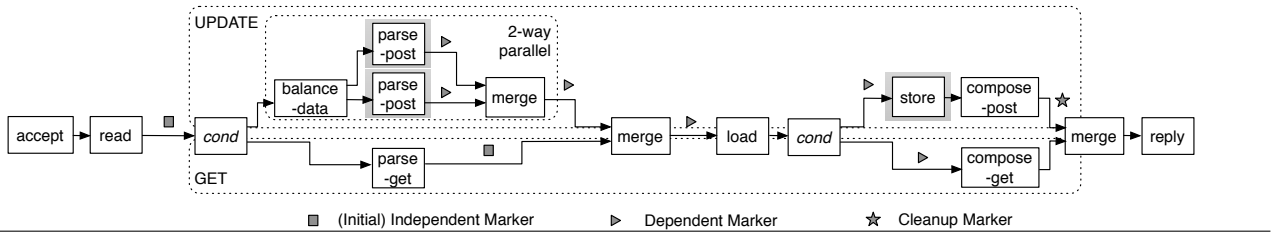

Figure 4: An extended version of our HTTP server that additionally supports UPDATE requests to the files it provides. An UPDATE of a file shares functionality with a GET whenever possible and returns the previous version of the updated file.

computation (Lines 8–11). Hence, in order to switch our HTTP server flow to NIO (see Figure 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.

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 piggybacked 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).[5] 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 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 4 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 the `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]In our algorithm, *blocking* does not lead to idling threads but rather restricts the functionality available for execution.

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.

The classic example for this type of problematic join is a deterministic merge with packets forwarded in a predefined

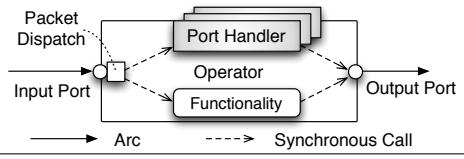

Figure 5: Port handler extensions for FBP operators.

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.

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.

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 1 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 $t < v$.

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.

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 *packet dispatch mechanism*, as depicted in Figure 5. 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 4.

Listing 4: Update port handler stub in Ohua

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

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.

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.

4.2.1 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

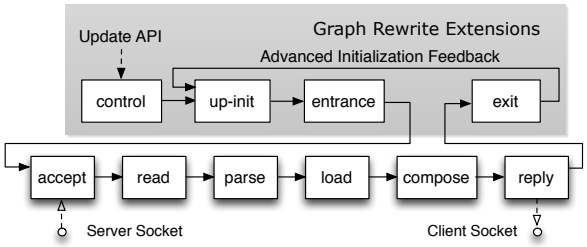

Figure 6: Pre-runtime rewriting of the HTTP flow graph.

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 6, 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[6] 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.

4.2.2 Advanced Transformations

Our rewriting is not limited to the three aforementioned operators. In Figure 6, 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

[6]http://zeromq.org/

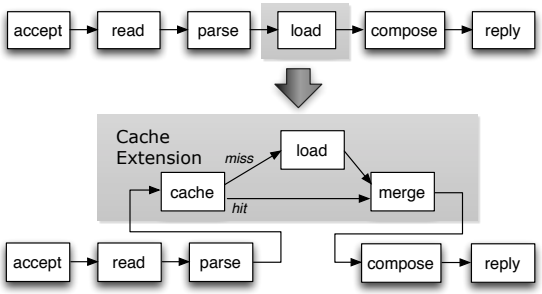

Figure 7: Extension of the HTTP flow graph with a `Cache`.

hits the target operator (again) and applies the update.

5. STRUCTURAL UPDATES

Up to this point we have addressed updates spanning one or multiple operators, which are akin 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 7 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.

5.1 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 $o_i \rightarrow \ldots \rightarrow o_n$, as depicted in Figure 8, 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

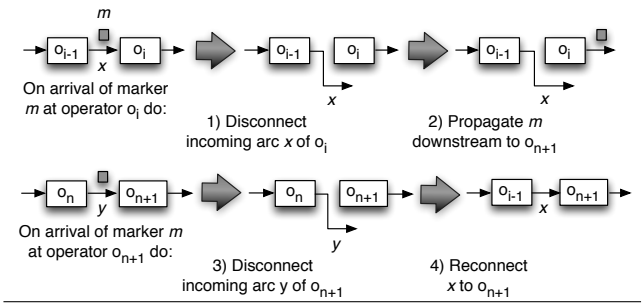

Figure 8: Algorithm steps for deleting the subgraph from operator o_i to operator o_n.

in Algorithm 2. It focuses solely on the update part, as we already covered marker propagation in Algorithm 1 (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.

5.2 Distributed Rewriting

Figure 9 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.

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 8, 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.

Algorithm 2: 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 ∈ ℝ do                          // operator update
2      if r.in ∈ 𝕀 then
3          arc ← r.in.disconnect();
4          if r.arc = 0 then
5              if r.new ≠ 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 ← 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 ∈ m.dependents do
17             c.setDependent(true);
18             o.broadcastInOrder(c);
19         end
20     end
21 end
```

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.

6.1 Operators and Mutual Dependencies

Listing 5 shows the code to submit the NIO-switch to the executing web server.

Listing 5: The NIO-switch realized with Ohua.

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

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

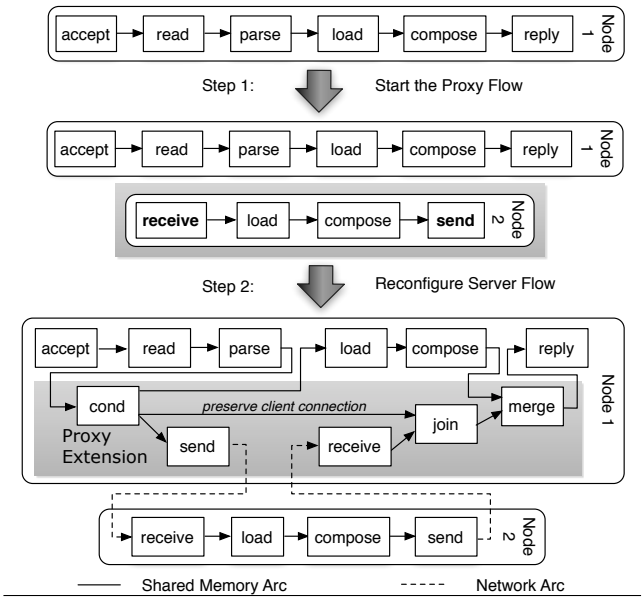

Figure 9: Proxy extension of the HTTP flow graph.

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. 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), 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.

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 6 shows the definition of the web server algorithm as a function.

Listing 6: The web server defined as a function in Ohua.

```
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)))
```

Finally, the invocation in Listing 7 assigns a port for the server to listen on.

Listing 7: Web-server invocation 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"))
```

To illustrate the update process, consider the scenario from Figure 6 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 8.

Listing 8: Cache-based Web Server.

```
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)))
```

Thereafter, this updated function is for all invocations of the web server in the running program. Listing 9 illustrates the submission of the update.

Listing 9: Cache Insertion Update.

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

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 2. 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.

6.3 Integration with Clojure Programs

The actual insertion of new code within a running program is already supported by existing tools.[7] 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[8], therefore loading new operators and algorithm functions packaged in jars is straightforward.

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 2, 7 and 9). We followed the same methodology as other similar frameworks [22] to analyse web server performance. We deployed our Ohua web server on one cluster node, and clients were

[7] https://github.com/pallet/alembic
[8] https://github.com/djpowell/liverepl

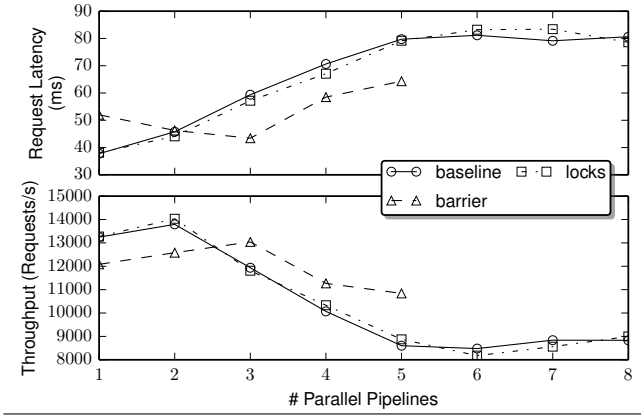

Figure 10: Runtime overhead using locks and barriers to simulate update points.

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.

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 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 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 10, 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.

7.2 Coordinated NIO Switch

For the coordinated switch of our web server to support the NIO API, we concurrently executed 30 clients issuing

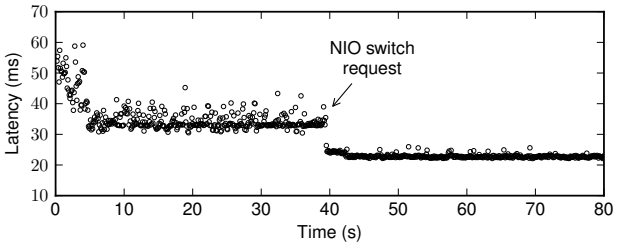

Figure 11: Request latency impact during a coordinated switch of the Load and Reply to use NIO's sendfile primitive.

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 11 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.

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 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.

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

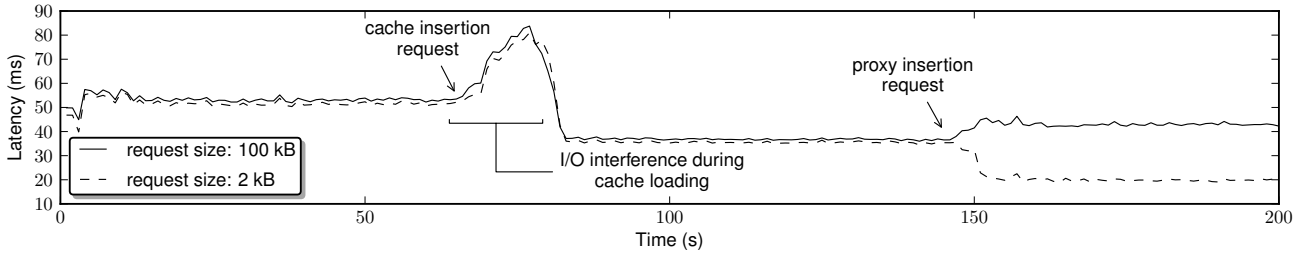

Figure 12: Reconfiguration experiment for cache and proxy insertion into the HTTP server flow graph.

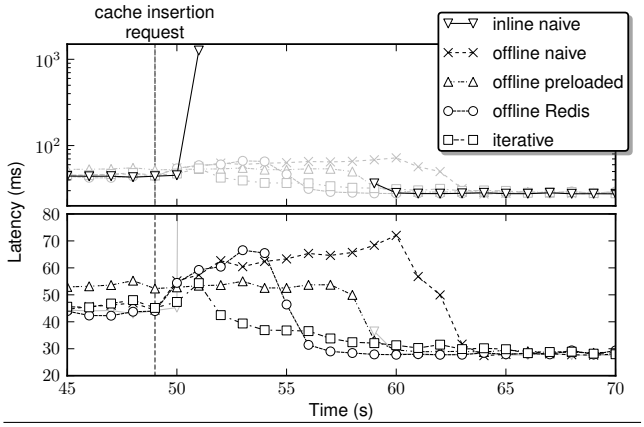

Figure 13: Various strategies to fill the cache during update.

potential I/O contention and plan its update accordingly.

In Figure 13 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[9] 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).

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 [16, 24]. 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 [15]. 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 [11] 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 [10] is better suited to reason about state and provide cooperative rather than disruptive updates [11]. 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 [29].

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 [28]. In software defined networks, consistent updates of switch configurations are coordinated based on tagged packets to achieve consistency [25].

[9]http://redis.io

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.

9. CONCLUSION AND FUTURE WORK

In this paper, we extended the dataflow-based programming model [21] 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.

10. REFERENCES

[1] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. OOPSLA '11, New York, NY, USA, 2011. ACM.

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

[3] 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, New York, NY, USA, 1995. ACM.

[4] 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. ACM, 2003.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 1985.

[6] A. A. Chien. Pervasive parallel computing: An historic opportunity for innovation in programming and architecture. PPoPP '07, New York, NY, USA, 2007. ACM.

[7] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, Nov. 1980.

[8] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. VLDB '86.

[9] 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.

[10] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. ASPLOS '13, New York, NY, USA, 2013. ACM.

[11] C. Giuffrida and A. S. Tanenbaum. Cooperative update: A new model for dependable live update. HotSWUp '09, New York, NY, USA, 2009. ACM.

[12] G. Graefe. Encapsulation of parallelism in the volcano query processing system. SIGMOD. ACM, 1990.

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

[14] C. Hayden, E. K. Smith, M. Hicks, and J. Foster. State transfer for clear and efficient runtime upgrades. HotSWUp '11.

[15] 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.

[16] 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.

[17] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, Nov. 2005.

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

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.

[20] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. SOSP '05, New York, NY, USA, 2005. ACM.

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

[22] D. Mosberger and T. Jin. Httperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3), Dec. 1998.

[23] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. PLDI '09, New York, NY, USA, 2009. ACM.

[24] L. Pina and M. Hicks. Rubah: Efficient, general-purpose dynamic software updating for java. HotSWUp '13.

[25] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. SIGCOMM '12, New York, NY, USA, 2012. ACM.

[26] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw.*, 1993.

[27] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. CC '02, London, UK, UK, 2002. Springer-Verlag.

[28] 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.

[29] Y. Vandewoude, P. Ebraert, 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.

[30] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. SOSP '01, New York, NY, USA, 2001. ACM.