Dynamic Development Support for Highly Concurrent Programs in the Ohua Data Flow Engine

Sebastian Ertel
University of Technology, Dresden
Dresden, Germany
sebastian.ertel@tu-dresden.de

Michael J. Beckerle
Waltham, MA, USA
michael.beckerle@alum.mit.edu

ABSTRACT

Most programs that require the full capability of multi-core architectures in order to achieve scalability address very challenging tasks. In the era of cloud computing, web technologies and big data these programs are often required to be online 24/7. Nevertheless, these programs need to be enhanced with new features or might have bugs to be fixed. Hence, a runtime system is required that allows for dynamic development without halting the executing program. While famous scripting languages like Python or JavaScript already provide such a feature, these languages were not designed for highly concurrent programming.

Introducing dynamic development into a highly concurrent runtime system is a challenging task that we address in this paper. We present our dataflow-based execution engine, Ohua, as a promising approach to write and execute highly concurrent programs for the future multi-core era. Furthermore, we extend the principles of flow-based programming in order to create a runtime extension framework that, due to the dataflow abstractions, enables an easy incorporation of new runtime features such as dynamic development into the engine.

1. INTRODUCTION

The dawn of the multi-core era introduced the notion of parallel program execution where many processes or threads try to solve one global problem. Synchronization techniques such as locks or transactional memory are meant to help programmers to turn sequential programs into bug free and highly concurrent scalable versions. The future in micro processor design[8] predicts a further increase in the number of cores and therewith concurrency. And while concurrent programming itself remains extremely challenging, even more difficult in this context seems dynamic program evolution.

1.1 Dynamic Development

Most of today's programs are not designed to be one-time executables. In the era of big data and cloud computing, most programs are meant to run for a long time. Consider the example of a web server depicted by the data flow graph in Figure 1. Inarguably, a web server is one of the most common programs with very high concurrency demands to service concurrent requests and the requirement to provide continuous service 24/7. Nevertheless even such a program has to grow with new features, optimizations or might have bugs that require fixing. For instance, the famous Apache web server[1] is widely used in highly concurrent settings servicing thousands of requests per second but for every reconfiguration or activation of a new feature (module) it needs to be gracefully[1] shut down and restarted. Restarts unnecessarily may lose computational state or persistent connections, penalize request latency and throughput which ultimately translate into customer dissatisfaction. Hence, an execution environment that allows a developer to grow and change programs online without imposing large overheads is highly desirable.

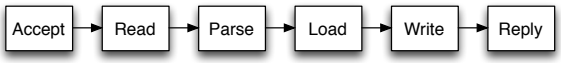

Figure 1: Basic data flow for a simple HTTP server.

1.1.1 Problem Definition

Exchanging and altering an executing program is a challenging task which mainly concerns preserving program correctness[21]. In [12], the authors identify three problems for object(/component)-based systems: referential transparency, state transfer and mutual references. For a concurrent shared memory program, we add the *consistency problem*. While the referential transparency problem addresses the references to the component which is to be updated, the consistency problem concerns the aspect of understanding which execution unit (thread/process) accesses this object at the time of an update. Finding these interactions without program knowledge resembles into implicit parallel program execution which is known to be hard. On the other hand, protecting every object access via locks is expensive.

1.2 FBP To The Rescue

A closer look at future microprocessor architecture reveals that their design starts to look more and more like the "old" data flow machines from the 1970s and 1980s that were introduced by massive parallel processing pioneers such

[1]Child processes are shutdown only when they have finished handling all their current requests and restarted by the parent process immediately.

LaME'13 July, 1^{st} 2013, Montpellier, France.
.

as Arvind, Culler[5] and Dennis[9]. The according data flow languages and the concept of flow-based programming (FBP)[20] compose programs out of small independent functional blocks, called operators, into a directed graph that describes the data flow of information packets between operators explicitly via message-passing. Hence, it provides the missing information required to solve the above consistency problem. FBP can be found at the core of most of the advanced data processing systems today where parallel and concurrent processing is key for scalability. For example, IBM's DataStage and InfoSphere products[17, 16] are state-of-the-art systems for data integration that are purely based on the FBP concepts. Database systems[10, 13] as well as the adjunct field of data stream processing[6, 3] also describe their algorithms in a directed graph like structure. The declarative design of network protocols such as for instance described in the Overlog system[19] are data flow graphs by definition. The NoFlo framework [2] is a JavaScript implementation of FBP for easy construction of backend web services and even the construction of highly scalable web services[25] comes back to the basic FBP principles.

1.3 Contributions

In the spirit of the new multi-core era, we take the opportunity for innovative thoughts in the programming language community[8] and consider FBP as the most promising concept to build highly scalable concurrent programs sequentially with a clear separation of concern between algorithm and functionality. Our data flow engine *Ohua* strictly adheres to the FBP principles and extends them such that the implementation of new runtime features like online operator reconfiguration or exchange to enable dynamic development become as straight forward as writing new operators. In detail, we contribute:

1. A generic extension concept for FBP operators that allows Ohua to mixin new functionality into existing and future operators.
2. A runtime extension framework based on FBP principles that allows Ohua to compose new runtime features for any data flow graph.
3. The implementation of an online reconfiguration algorithm that allows to update or exchange operators dynamically during execution.

Since all of these extensions are implemented by taking the FBP principles by heart, a new runtime feature in Ohua is composed purely of sequential code and makes no assumptions on the execution context.

1.4 Outline

The rest of the paper starts with a brief introduction to the core principles of FBP and their implementation in our Ohua data flow engine. Afterwards, Section 4 presents our advanced FBP concept and the resulting runtime extension framework. Afterwards, we use this framework to build our online reconfiguration runtime feature. Related work is presented in Section 6, before we conclude.

2. FBP ESSENTIALS

This section introduces the basic abstractions and key insights that are necessary in order to follow the construction of the runtime extension framework in Ohua. For a thorough review on FBP, we refer the interested reader to [20].

2.1 Concepts

In flow-based programming, an algorithm is described in a directed acyclic[2] *data flow graph* where the edges are referred to as *arcs* and vertices are referred to 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 a single arc while on the other hand an output port can be the source of multiple arcs. The *operator algorithm* continuously retrieves data one packet at a time from one of its input ports and emits (intermediate) results to its output ports. The output port broadcasts the data packets to all of its outgoing arcs. Finally, an operator is context-free. That is, it neither makes any assumptions nor possess any knowledge about its upstream (preceding) or downstream (succeeding) neighbours. This basically resembles the concept of a library and therewith the operator concept inherits the benefit of high reusability.

2.2 Key Strength

Apart from the high reusability of operators in the construction of the flow graph algorithms, 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 explicitly rather than implicitly. The concept of implicit data access (i.e. shared memory), which causes many problems in today's system design such as lost updates, mutual exclusion, deadlocks etc., is omitted. Furthermore, FBP does not define the implementation (array, shared memory queue, TCP connection etc.) of the arcs but only their (FIFO) semantics. The result is execution infrastructure independence. Accompanied by the clear structural decomposition of the algorithms into small context-free tasks (operators), this allows the runtime system to exploit pipeline and data parallelism on any distributed architecture (i.e. multi-core, cluster, WAN etc.).

Implementing reconfiguration support into a runtime that executes highly concurrent programs requires 1) knowledge about who exactly accesses the component to be altered and 2) a mechanism to fit this operation into the execution schedule. Knowledge about the interactions of the operators is provided by the explicit nature of the flow graph. The latter aspect is subject of our runtime extension framework.

3. OHUA SIMPLIFIED

Ohua is our implementation of an advanced data flow execution engine which strictly adheres to the above principles. Operators are implemented as classes in Java while the specification of the flow graph is given in XML. At runtime this specification is passed to the Java-based execution engine which instantiates the listed operators and connects them via arcs. Execution happens in three phases: *initialization*, *computation* and *shutdown*. During initialization all operators are required to claim all resources such as I/O connections or build the internal data structures that they need to perform their computation. Only when all operators successfully finished this process, computation starts. Respec-

[2]We constrain ourselves to acyclic graphs for reasons of simplicity and space limitations of the paper. FBP though does not define any restrictions on the graph structure.

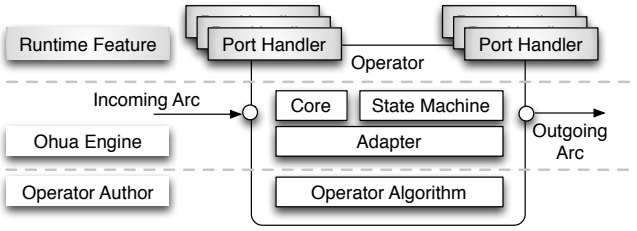

Figure 2: Anatomy of an Ohua operator.

tively, the shutdown phase triggers a coordinated resource release across the flow graph. Ohua has many advanced capabilities, but for this paper we use a simplified execution model also known from other dataflow-like implementations [22] which allows us to continue focusing on the construction of our runtime extension framework. In this simplified version, each operator of the submitted flow graph executes on a single thread. It accounts that an operator is never executed by more than one thread but all operators can execute concurrently. Arcs are implemented via blocking queues, that block either the enqueue or dequeue operation on a full or empty queue, respectively. Executing each operator on its own thread imposes the maximum amount of concurrency into the flow graph and therewith does not restrict our runtime extension framework to any specific concurrent execution setting. Since the following algorithms are implemented on the FBP abstraction of a data flow graph, any execution model is supported.

4. ADVANCED FBP CONCEPTS

The main extension point in FBP is an operator. In the following section, we introduce new extension points along the FBP guidelines of sequential code and execution context independence to enable easy implementation of new runtime features, such as online operator reconfiguration.

4.1 Runtime Extension Framework

The development process of a specific flow graph is guided by looking at the specific functionality that is implemented by each of the operators. Respectively, a runtime extension framework that is supposed to enhance capabilities of all existing and future operators needs to treat them primarily as black boxes. Furthermore, the framework needs to be independent of any specific flow graph structure in order to apply for any existing or future flow graphs. In order to accomplish both of these goals, we introduce concepts that lend themselves towards the concepts of mixin inheritance and runtime polymorphism.

4.1.1 Operator Internals

Figure 2 provides an in-depth view on the internal structure of an operator in Ohua. There exist two layers: 1) the *engine layer* and 2) the *operator author layer*. The engine part of an operator is provided by the Ohua execution engine for every operator and consists of the *core* and a *state machine*. The latter is a vital part in more advanced execution scenarios to realize cooperative operator scheduling. In our basic execution model it keeps track of whether the operator is in initialization, computation or shutdown state with respect to the execution phases. The core part incorporates and controls the operator structure such as input and output ports as well as the associated services to retrieve and emit data packets. An *adapter* hides these internals and provides access to these well-defined services for the specific *operator algorithm* implementation as provided by the operator author. As a result, whenever the operator algorithm interacts with its ports to retrieve or emit data, it gives control back to the engine part of the operator. This is an essential aspect that makes the following extensions possible. Finally, an operator algorithm defines a set of properties that are assigned at data flow design time and used by the operator algorithm at runtime. For example, the list of headers to be used for HTTP response creation are supplied to the Write operator of our web server flow graph from Figure 1 as part of the XML flow graph description.

4.1.2 Operator Mixins

Up to this point, the operator structure resembles to known FBP. The *port handlers* introduce the concept of mixin-style inheritance to operators as they introduce new functionality by treating them mainly as black boxes. They have full knowledge of the operator structure but do not have the possibility to change it. Neither do port handlers have access to the state machine of the operator. However, port handlers do have full access to the operator algorithm including making changes to it. Operators inherit the functionality of port handlers by exposing an event notification mechanism for packet arrival and emission. A port handler registers for the events it is interested in and gets called by the operator core whenever such an event has occurred. During such a notification the port handler additionally has access to the packet that triggered the event. Note, that the code of a port handler is just an extension of the operator functionality and as such is purely sequential by FBP design. Whenever the operator algorithm retrieves the next packet from an input port, the registered port handlers run first before the core returns the packet to the operator algorithm. Similarly, on packet emission the registered packets are notified before the operator core enqueues the packet into the outgoing arc. Ohua does not set a limit on the number of port handlers per port. Port handlers on the same port are typically independent of each other and are executed in the order in which they are mixed into the operators during flow graph initialization. During this initialization process, a port handler is created and registered to each operator. While one port handler can register for many events from input and output ports, it remains operator local in order to adhere to the context-free nature of FBP operators.

4.1.3 Packet Dispatch

The port handler concept is implemented as a visitor pattern on the packets. This is similar to the concept of pattern matching in Actors as found in languages such as Erlang or Scala. A port handler implements the interface methods for the packet types it is interested in. Therefore, Ohua extends the classical packet concept in FBP by differentiating between two types of packets:

- *Data packets* are instantiated and interpreted at the operator author layer and contain data records belonging to the program of the data flow graph.
- *Meta-data packets* are instantiated and interpreted at the engine and runtime feature layer and contain meta information of the current execution.

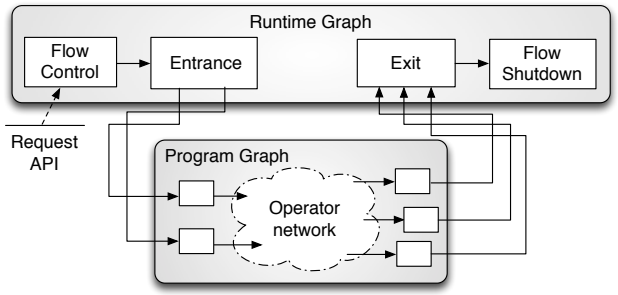

Figure 3: Runtime representation of a submitted data flow graph.

An example of a meta-data packet is the end-of-stream (EOS) packet responsible for signalling the coordinated shutdown of the flow graph. Meta-data packets use the existing abstraction of the data flow graph and travel along the defined arcs. Meta-data packets are never dispatched to an operator algorithm. However, port handlers are allowed to register for the arrival of both, data as well as meta-data packets. Routing of the meta-data packets is the responsibility of the associated port handlers. Routing of the data packets is handled by the operator algorithms.

4.1.4 Overlay Network Algorithms

Respectively, the design of a new runtime feature consists of at least one new type of port handler and a new type of meta-data packet. The key observation at this point is that, due to its abstract nature, a data flow graph can also be viewed as a network of operator nodes in the sense of the distributed computing community. Hence, a new runtime feature consisting of a set of port handler objects deployed on various nodes in the network of a data flow graph forms an *overlay network*[11] as the handlers of a single runtime feature only see the (meta-data) messages created by themselves plus other messages that they registered for at the underlying network.

This powerful observation allows us to utilize algorithms and concepts from the distributed systems community on the Ohua runtime engine executing on a multi-core machine or even a cluster thereof. Furthermore, we can now use the seminal work of Lamport and Chandy[7] to reason about the consistency of our program. This notion of consistency is essential to our dynamic development framework in order to solve the last problem of creating a mechanism to fit the reconfiguration actions into the execution schedule without sacrificing the consistency of the program state. The result is that changes can be applied when they travel along the arcs of the system in line with data flow and get applied as a processing step inline with the computation.

4.2 Runtime Feature API

The final challenge is the interaction with the highly concurrent execution environment. For example, reconfiguration requests for our dynamic development framework originate outside of the Ohua runtime engine. In order to become a valid part of the execution, they have to be convert into meta-data packets and become part of the data flow.

4.2.1 Runtime Components

We solve this problem the FBP way by extending the concept of operators and ports. Ohua differentiates between two types of components (for operators and ports):

- A *program component* is defined by the author of the data flow graph resembling the algorithm of the program.
- A *runtime component* is part of the Ohua runtime system and not part of the available operator suite for algorithm construction.

Note that the execution model does perform this differentiation and respectively runtime operators are executed just as program operators are. Before execution starts, Ohua performs a graph rewrite as depicted in Figure 3 that creates a *runtime graph* composed of all runtime operators. The Entrance operator provides a central entry for meta-data that originates outside of the *Program Graph*. It is connected to runtime ports on all source operators. On every interaction of an operator algorithm with the core of a source operator, the runtime port is checked for new data. Similarly, the Exit operator creates a central exit for meta-data results. Finally, the Flow Control operator provides a general (non-blocking) API not only to steer the execution by submitting a "start computation" or "finish computation" request as depicted in Listing 1 but to submit arbitrary requests to an executing data flow.

Listing 1: Ohua's Flow Graph Execution Management API

```
1 OhuaProcessRunner runner = new
      OhuaProcessRunner("web-server-graph.xml");
2 runner.submit(new Request(Type.INITIALIZE));
3 runner.submit(new Request(Type.START_COMPUTE));
4 // do something else here
5 runner.submit(new Request(Type.FINISH_COMPUTE));
6 runner.submit(new Request(Type.SHUTDOWN));
```

These requests are converted into meta-data packets and forwarded to the Entrance to enter the data flow. Therewith, requests become valid parts of the data flow execution without the necessity of invasive interruption code or additional synchronization techniques. Therefore, all further injection of meta-data packets into the flow graph happen in accordance with the semantics of the data flow graph abstraction. Respectively, the Flow Shutdown operator waits to receive the EOS packets from all the target operators. The Ohua engine now can easily determine when processing has finished and a coordinated flow graph shutdown can be initiated by just waiting for the Process Shutdown operator to (receive the EOS packet and) finish its computation. Once more, no additional synchronization, coordination or interruption code is required.

5. ONLINE RECONFIGURATION

Based on these concepts, we extend the runtime to allow reconfiguration of operator properties and even exchange of operators, online.

5.1 Overlay Network Implementation

Our runtime feature is composed of a new port handler and a corresponding configuration meta-data packet, to carry

the reconfiguration information to the target. The class implementation of this new message is given in Listing 2[3].

Listing 2: Reconfiguration meta-data packet.

```
1 class ReconfigMsg implements MetaDataPacket{
2   enum Category{ PROPERTY, OPERATOR}
3   String _targetOp, _propertyRef = null;
4   Object _reconfigValue = null;
5   Category _category = null;
6 }
```

The meta-data packet requires to specify whether the reconfiguration request addresses a property or an operator exchange along with references to identify the targeted operator and property. Finally, it carries the updated property or operator algorithm implementation. These packets are interpreted by the according port handler of Listing 3.

Listing 3: Reconfiguration port handler.

```
1 class ReconfigHandler implements PacketHandler{
2  void notifyArrival(InputPort port, ReconfigMsg
       packet) {
3   Operator op = port.getOwner();
4   if(!op.getName().equals(packet._targetOp)){
5    // propagate via broadcast
6    for(OutputPort out : op.getOutputPorts()){
7     out.emit(packet);}
8   }else{
9    switch(packet._category) {
10    case PROPERTY:
11     // update the property via reflection
12     RelectionUtils.updateProperty(packet.
           _propertyRef, op, packet._reconfigValue);
13     break;
14    case OPERATOR:
15     // initialize and exchange the operator algorithm
16     OperatorAlg newAlg = packet._reconfigValue;
17     newAlg.init();
18     op.setAlgorithm(newAlg);}}}
19 }
```

The port handler is only interested in the packets of the reconfiguration overlay and therefore an instance of it is mixed into the functionality of each operator in the flow graph during initialization. Whenever a configuration packet arrives at an operator, the configuration handler checks whether the operator is the target of this configuration. If so then it either updates the property of an operator algorithm via reflection on the class hierarchy or exchanges the operator algorithm entirely. Otherwise, the configuration packet is broadcasted to all downstream neighbours. If a request was handled or the marker reaches the Exit, it is dropped. Hence, the reconfiguration algorithm itself makes no assumptions on the graph structure or operator implementation and is therewith applicable to any data flow graph and any operator.

5.2 API

Finally, reconfiguration requests are issued from outside the Ohua execution engine. Therefore, no specific system operators are required but instead the request API is used to convert change requests into meta-data packets and inject them into the flow graph.

[3]We omit visibility quantifiers in our code listings for brevity reasons.

5.2.1 Property Update Request

Listing 4 shows an example for redefining the headers that the HTTP response of our web server contains. The new map of headers is wrapped into a packet and finally submitted to the data flow graph. The packet contains the identifier of the target operator for this packet and a reference to the operator property to be reconfigured. Once the packet reached the Write operator, the according port handler applies the property via reflection and drops the packet.

Listing 4: Reconfiguration request for the response headers.

```
1 ReconfigMsg packet = new ReconfigMsg();
2 packet._category = ReconfigMsg.PROPERTY;
3 packet._targetOp = "Write";
4 packet._propertyRef = "properties.headers";
5 packet._reconfigValue = Collections.singletonMap
      ("From", "sebastian@ohua.com");
6 runner.submit(new Request(Type.FLOW_INPUT,
      packet);
```

5.2.2 Operator Exchange Request

During processing, there might be many reasons why an operator algorithm wants to be exchanged, for example a faster implementation is available or a bug was fixed. Listing 5 exchanges the file-based Load with a database Load operator in order to improve request latency by removing disk seek times and benefiting from index structures and caching of the database system.

Listing 5: Exchanging the file with a DB load operator.

```
1 ReconfigMsg packet = new ReconfigMsg();
2 packet._category = ReconfigMsg.OPERATOR;
3 packet._targetOp = "Load";
4 packet._reconfigValue = new
      DatabaseLoadOperator("jdbc:derby://localhost
      :1527:/http");
5 runner.submit(new Request(Type.FLOW_INPUT,
      packet);
```

The exchange functionality is limited to the operator structure, i.e. the new operator algorithm most adhere to the operator structure. For example, the Load operator exposes only one input port and therefore can not be exchanged with an operator algorithm requiring two or more input ports. Note, the operator exchange as portrayed in the above program code is not applicable to allow the exchange of operators that might be located on different physical nodes in a cluster. But a reflection-based implementation is straight forward and therefore omitted for clarity reasons.

6. RELATED WORK

A detailed summary on dataflow languages is given in [18]. Yet, none of these languages nor current state-of-the-art such as Kilim[23] or StreamIt[24] provide a runtime extension concept that allows to dynamically exchange properties or operators. In [24] the authors of the StreamIt compiler briefly mention a re-initialization mechanism for operators (called filters in StreamIt) and parts of a flow graph that is incorporated with the messaging system but omit

any more details. Meta components (such as Ohua's system components) are common constructs in flow-based programming[20]. Yet, we are unaware of an FBP system that introduces the dynamic component change concepts of Ohua to enable online evolution of highly concurrent programs.

Actors and mixin inheritance are known concepts and have recently gained popularity due to their implementation in the Scala programming language[14]. Actors and the message-passing programming model is known from Erlang[4]. Erlang also allows to send new functions via messages to processes which can then perform the code exchange. This model most closely resembles our online reconfiguration feature. However, Erlang requires each process that needs this capability to implement it on its own. In Ohua it is a runtime feature which is implemented once and inherited by all operators. Clojure[15] also supports dynamic development but we were unable to find a detailed explanation on whether exchanging functions during highly concurrent execution of a program is possible. Scripting languages such as Perl, Python and JavaScript also provide features to exchange functions and operators at runtime but were not designed to execute programs concurrently.

7. CONCLUSION AND FUTURE WORK

In this paper, we addressed the problem of interfacing and exchanging parts of highly concurrent programs, online. Therefore, we introduced a runtime extension framework for languages based on the abstract principles of dataflow and flow-based programming as a promising paradigm for future multi-core programming. We showed that mixin-style inheritance to enhance any operator with new functionality in combination with actor-like message dispatching support allows for easy implementation of new runtime features as an overlay network over any data flow graph. Our graph rewrite allows not only for these extensions to introduce their own operators but to become a natural part of the whole runtime system. On the basis of these concepts, we have shown the implementation of our online reconfiguration runtime feature, which allows to update operator properties or even complete operator algorithms at runtime, as a first step towards dynamic development capabilities. Our goal is to evolve this runtime feature to handle stateful operators and support dynamic development of the whole flow graph.

8. REFERENCES

[1] Apache, apache http version 2.2, documentation, stopping and restarting. `http://httpd.apache.org/docs/2.2/stopping.html`.

[2] Noflow, flow-based programming for node.js. `http://noflojs.org/`.

[3] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. SIGMOD. ACM, 2003.

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

[5] Arvind and D. E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow architectures, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[6] 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.

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

[8] A. A. Chien. Pervasive parallel computing: an historic opportunity for innovation in programming and architecture. PPoPP. ACM, 2007.

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

[10] 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*, 1986.

[11] D. Doval and D. O'Mahony. Overlay networks: A scalable alternative for p2p. *Internet Computing, IEEE*, 7(4):79–82, 2003.

[12] 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*, pages 63–76, 2001.

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

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

[15] R. Hickey. The clojure programming language. DLS. ACM, 2008.

[16] IBM. Infosphere streams. `http://www-01.ibm.com/software/data/infosphere/streams/`.

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

[18] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004.

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

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

[21] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw.*, 10(2):53–65, Mar. 1993.

[22] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throughput stream programming in java. OOPSLA. ACM, 2007.

[23] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. ECOOP. Springer-Verlag, 2008.

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

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