FLUXO: A Simple Service Compiler

Emre Kıcıman Benjamin Livshits Madanlal Musuvathi
Microsoft Research

Abstract

In this paper, we propose FLUXO, a system that separates an Internet service's logical functionality from the architectural decisions made to support performance, scalability, and reliability. FLUXO achieves this separation through three mechanisms: 1) a coarse-grained dataflow-based programming model; 2) detailed runtime request tracing to capture workload distributions, performance behavior, and resource requirements; and 3) a set of analysis techniques that determine how to apply simple, parameterized dataflow transformations to optimize the service architecture for performance, scalability, and reliability. In this paper, we describe our vision for how to make Internet services easier to construct, and show how a variety of Internet service performance optimizations may be expressed as transformations applied to FLUXO programs.

1 Introduction

Over the last 10-15 years, our industry has developed and deployed many large-scale Internet services, from e-commerce to social networking sites, all facing common challenges in performance, reliability, and scalability. To address these challenges, developers consistently draw from a relatively small repertoire of techniques, such as replication, tiering, pre-computation, and caching.

Unfortunately, to be properly implemented, these architectural techniques are tightly coupled with service functionality: their effectiveness depends heavily on the service's semantic requirements, its workloads, and its performance, and other runtime and environment characteristics. As a result, developers must have a deep, end-to-end understanding of their system infrastructure when writing the service functionality. Furthermore, modifications to either the underlying infrastructure or the application itself may require significant architectural changes in the deployed service.

Our goal is to change how developers build online Internet services, allowing them to focus on application-level functionality, while orthogonal techniques optimize the service for performance, reliability, and scalability concerns. To this end, we propose FLUXO, a system that *separates* architectural decisions to support performance, reliability, and scalability from service functionality. FLUXO is analogous to an optimizing compiler that takes a high-level program representation and, using program profile data, automatically optimizes the program's execution, relieving the developer from worrying about, say, register allocation, to focus on functionality.

The key to achieving this separation is to capture a representation of those semantic and mechanical details of the application-level service that drive the architectural techniques. To accomplish this goal, FLUXO relies on the following three mechanisms:

1. To simplify reasoning about the service functionality, FLUXO uses a *dataflow model* representation of an Internet service's coarse-grained behavior. To enable optimizing transformation, this dataflow model is annotated with semantic requirements, such as state consistency requirements, component side-effects, and idempotence.
2. To capture program profile data, such as workload distributions, resource usage and performance profiles, FLUXO uses *runtime request tracing* to capture performance characteristics as well as input and intermediate data distributions. For instance, FLUXO can automatically record the sizes of inputs and outputs, and their frequencies to infer the communication cost of components in the dataflow.
3. Finally, FLUXO applies a set of simple, parameterized dataflow program transformations to optimize the service's performance, scalability, and reliability. These optimizations may be parameterized through automated heuristics, simulations, or numerical analyses. Examples of these transformations are described in detail in Section 4.

Unlike previously proposed programming paradigms for Internet services such as Dryad or MapReduce [3, 9], the focus of FLUXO is *online* request processing. This leads to critical distinctions, such as an emphasis on end-to-end request-response latency [4], and many cross-request optimization opportunities.

1.1 Paper Organization

The rest of the paper is organized as follows. Section 2 provides a brief overview of the state of the art in building Internet services. Section 3 talks about the FLUXO architecture and discusses our design choices. Section 4 talks about sample program transformation that are used for optimization. Finally, Section 5 provides a summary of our proposal and outlines future challenges.

2 Background and Motivation

Our motivation for FLUXO came first from an internal survey we conducted of large-scale services at Microsoft. While the provenance of these systems varies greatly—having been built by different groups within Microsoft over more than a decade, or even being brought into Microsoft via acquisitions—we found that all of these systems re-used a small number of architectural patterns. This same observation holds true of publicly available reports of other services' architectures [2, 8, 11, 13].

While describing each of these architectural patterns is outside the scope of this short paper, many of the more prevalent techniques are described by Hamilton [7], and include tiering of services, extensive use of caching, and denormalization of data. These patterns are not simply reused "cookie-cutter," but must be specialized to suit a specific service's requirements and workloads. Understanding the interactions among system components, workloads, and semantic requirements, however, is non-trivial. For instance, we demonstrate in a companion paper [12] that the optimal caching for an Internet service varies significantly with the input workload distribution. Manually maintaining near-optimal caching policies with dynamic changes in the input workload distribution is simply not feasible.

While there are a number of frameworks for easing the building of large-scale interactive services, they are, to our knowledge, focused primarily on reuse of infrastructure rather than separating application-level functionality from architectural decisions. For example, while Java EE (formerly J2EE) provides core caching, tiering and partitioning functionality, developers must still manually decide what and how to cache, tier and partition in their system. These choices are scattered throughout the application-level code [14], making the code hard to understand, maintain, and redeploy. Platform computing services such as Amazon's EC2 and Azure provide elastic compute environments, but do not aid or enforce scalability and performance best practices [1, 10]. Google's App Engine provides a scalable platform for a narrow class of services [6]. SEDA uses a staged event driven architecture to separate application event processing from controllers that handle resource allocation decisions [15]. Dryad and MapReduce achieve many of our goals of separating application-level from scalability and reliability concerns but are scoped to off-line and batch computations instead of interactive services [3, 9].

3 Fluxo Architecture

The main goal of FLUXO is to give the programmer the illusion of writing straight line code for handling a web request and allow the system to handle the complexities arising from the requirements of scalability, high performance, and reliability. The main logic of handling the request is expressed in the form of a dataflow program. FLUXO then performs a series of program transformations that analyze and restructure the input program, resulting finally in a program that can be deployed on a set of physical machines in a datacenter.

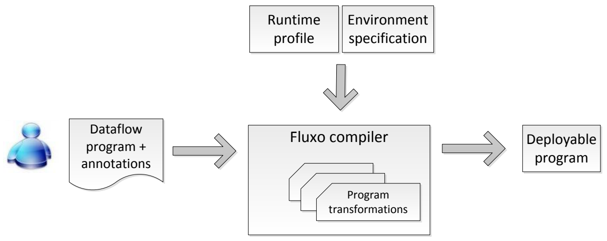

Figure 1: FLUXO architectural diagram.

Akin to a regular compiler, we envision that FLUXO contains both platform-independent transformations that optimize the input program, and platform-specific transformations that map the program components to available physical resources. In addition, FLUXO automatically instruments the program to collect runtime information, such as workload distributions and performance profiles, that can be analyzed to direct future optimizations. These profile-guided transformations may be done off-line, periodically, or continuously, depending on the nature of the transformation. To allow FLUXO to reason about semantic correctness, FLUXO asks the developer to provide semantic annotations that describe attributes such as consistency requirements and side-effects.

Programming Model

A FLUXO graph contains nodes, which perform computation, and typed edges, which represent the flow of data. Execution begins with a *trigger event* such as a web request or timer. The dataflow graph declares input availability requirements of each of its nodes. Nodes wait until all of their declared inputs are available and perform the computation, thereby generating outputs on their outgoing edges. Some nodes are marked as *output nodes*, meaning that their data is sent back through the web interface.

Figure 2 contains an example FLUXO input program implementing a part of a social-news service. The service implements two operations: 1) Given a user id, the service returns an aggregation of messages broadcast by that user's friends; and 2) given a user id, save a message. The dataflow program concisely represents the logic of the service without any reference to optimizations required for scalability, reliability and, performance.

In a FLUXO input program, every request logically executes independently. The only way for a program to exchange data across requests is by explicitly using a soft state or hard state store. However, FLUXO is free to

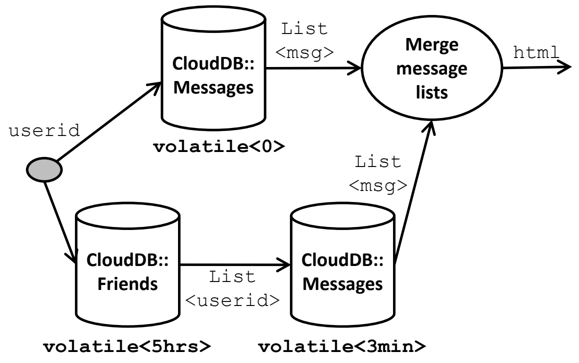

Figure 2: FLUXO example describing a part of a social news service that allows users to broadcast messages to their friends. The service maintains a database of messages broadcast by each user. The service makes a best effort to allow a user to read their own most recent messages, but allows up to 3-minute old reads of messages from friends.

break request isolation as long as the service's semantic requirements are satisfied. In fact, many of our program transformations target cross-request optimizations such as shared caches and batching of common computations.

To simplify the development of FLUXO programs, we provide libraries of reusable components for common tasks such as accessing web services and utilities for manipulating data. Developers may create new components, though FLUXO provides only inter-component optimizations. Existing dataflow-based development frameworks such as Yahoo! Pipes [16] have demonstrated that a wide-variety of services can be built using a small number of standard components.

Annotations

In addition to dataflow, FLUXO asks programmers to add annotations to the components. These annotations allow FLUXO to perform transformations of the original dataflow graph without violating the user-intended semantics. In particular, it is important to specify the side-effects of the computation performed by each component as well as the consistency requirements of the data read by these components. By default, FLUXO assumes that each component is stateless. Components that are stateful are marked with a `volatile` annotation. The annotation contains a time parameter that specifies the maximum allowed staleness of the data read. For example, a `volatile`⟨0⟩ annotation means that component requires the most recent version of the state. In Figure 2, the program requires an up-to-date version of the user's edits but allows the edits of the user's friends to be stale by 3 minutes. Similarly, the set of friends for a given user can be stale by 5 hours. In addition, components that modify state need to be annotated specifying whether the updates performed are `idempotent` or not. This annotation provides FLUXO the flexibility to retry the update on failures. While most annotations are specified per-component, some annotations are program-wide: for example a *closed-world* annotation may specify that external events cannot modify component behaviors, permitting more aggressive optimizations.

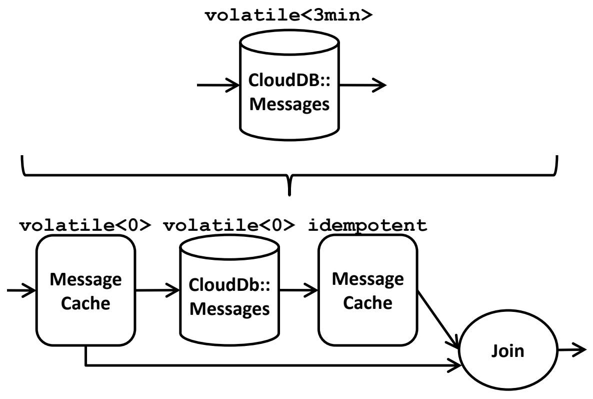

Figure 3: Wrapping a node with a cache

Fault Model

The FLUXO environment assumes that there is an underlying hardware management infrastructure that is responsible for managing the availability of machines, similar to today's utility computing environments [1, 10]. This infrastructure, however, does not need to ensure that the application itself is fault-tolerant. That responsibility, in FLUXO, falls onto the compilation and optimization policies. For example, one part of the compilation process is responsible for replication of component nodes to provide availability in the face of machine failures.

Storage and Consistency Model

Given the proliferation of cloud storage services, FLUXO does not implement its own storage model. However, FLUXO does require programmers to mark nodes with side-effects, as shown above. Currently, FLUXO assumes that the underlying storage-service provides strong consistency semantics for updates performed within a given dataflow component.

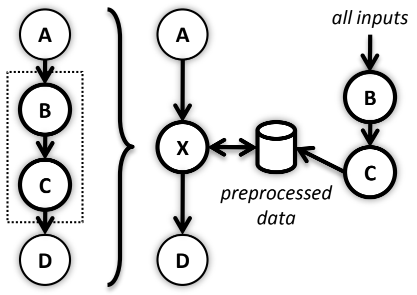

Figure 4: Pre-computing all possible outputs of nodes B and C

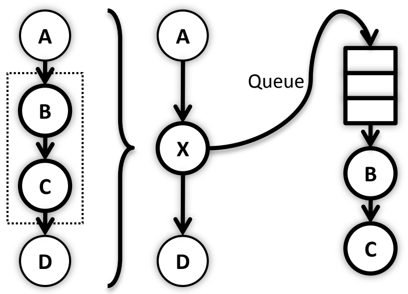

Figure 5: Post-computing all possible outputs of nodes B and C

4 Example Optimizations

The optimizations we consider below target various improvements to overall service latency. These optimizations may be applied based on heuristic analyses, simulation-based analysis, or numerical or queueing models. Each analysis style has its benefits and drawbacks. In general, encoding heuristics based on current practices is straightforward, though not necessarily robust. Simulation-based analysis provides the most accurate analysis of optimization choices [12], but may be computationally intensive. In contrast, numerical and queuing models are often fast, but their abstractions can reduce accuracy.

4.1 Automatic Cache Insertion

The backends of todays Internet services rely heavily on caching at various layers both to provide faster service to common requests and to reduce load on back-end components. In the context of a large-scale Internet service, a cache wraps around the computation and/or I/O performed by one or more components or tiers of the system. For performance, cache contents are usually stored in memory, making them an expensive and scarce resource.

Cache placement is especially challenging given the diversity of workloads handled by widely deployed Internet services such as Hotmail, Live Search, Google Maps, Facebook, and Flickr. To receive the optimal benefit, a cache must be placed with careful comparison of incoming workload distributions, cached data sizes, consistency requirements, component performance, and many other issues.

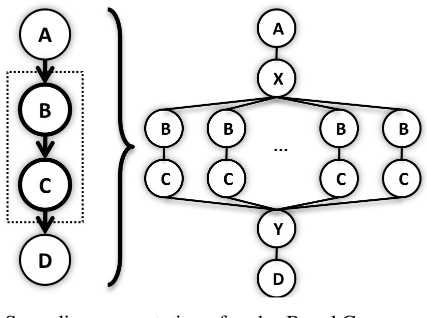

Figure 6: Spreading computation of nodes B and C across machines

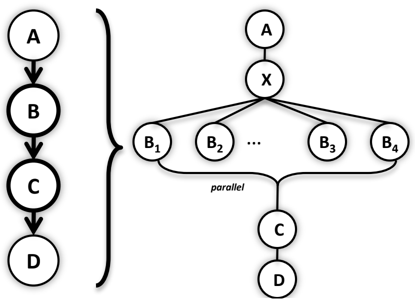

Figure 7: Node B is replicated for fault tolerance. The degree of replication is determined by observed resource requirements and failure rates, as well as reliability targets.

Figure 3 shows a dataflow transformation that adds a cache around a component. The `volatile` annotations in the dataflow components constrain the design of caching policies to satisfy the specified consistency requirements. In particular, FLUXO infers that the entries in the `Edits` cache are valid for at least 3 minutes. Moreover, if the graph contains a *close-world* annotation, FLUXO controls all updates to the `Edits` database. Thus, a cache entry needs to be invalidated only on an update to that entry. Similarly, if a component is stateless, then FLUXO can cache the results of the computations for an arbitrary amount of time, limited only by the opportunity cost of using the cache memory for other viable requests.

The purpose of automatic cache insertion is to determine which node(s) in the dataflow graph should be wrapped with caches. The analysis must estimate the potential performance benefits of cache locations from workload distributions, component performance profiles, and data sizes. The data for the analysis can be automatically culled from runtime instrumentations inserted by FLUXO. One implementation of this analysis is a search through a large state space of possible caching options, where each option is evaluated through a simulation. This simulation technique provides an accurate representation of the inter-dependencies between caches, workloads and component performance profiles; though the cost of exploring a large configuration space solely through simulation is too great to be practical for large programs [12].

4.2 Pre- and post-computation

A second technique used to speed the latency of client requests is to move computation and I/O out of the critical path using pre-computation or post-computation of expensive stages of request processing. Pre-computation is equivalent to 1) identifying the ideal entries in a "cache," 2) pre-filling these entries in a "cache" and, 3) automatically updating the cache to prevent staleness. Figure 4 shows an example of pre-computation. The pre-computation is done offline, where the space of potential inputs to B may be explored and results of going through B and C recorded.

Post-computation means doing just a little bit of work synchronously, in the critical path, and then delegating more heavy-weight tasks for later processing. An example of post-computation is shown in Figure 5. Nodes B and C are replaced by node X that places the computation into a queue to be completed later.

4.3 Partitioning and Tiering

To provide scale-out capabilities, many Internet services partition their request handling, spreading the responsibility for heavy-weight computation or I/O across many machines [5]. As such, partitioning can be combined with tiering such that every tier has its own partitioning scheme on a different set of data. For example, in an e-commerce site, the incoming workload might be randomly partitioned or partitioned based on geography while backend computation and storage tiers are partitioned based on the hashcode of the user's login. An example transformation shown in Figure 6 demonstrates how the heavy-weight computation represented by nodes B and C may be spread across a cluster of machines. Automatic tiering decision can be based on peak memory usage or bandwidth requirements of each component.

4.4 Fault Tolerance

Individual nodes or groups of nodes of a FLUXO program may fail. As described earlier, FLUXO places responsibility for ensuring the service reliability onto the compilation and optimization process. One common pattern for improving availability and performance involves replicating part of a FLUXO program. For instance, a service might send a computation task to several instances of a component, as represented by nodes B_i in Figure 7. The first instance of B to complete will be immediately sent on to node C, while the other outstanding requests will be canceled. As long as at least one of B_i is available and performs quickly, the failures or latencies of other B_i will be hidden. Applying such an optimization requires that the execution of a node B is idempotent.

As a further enhancement, it is possible to have a stateful version of this optimization, where the order in which instances respond would be recorded. That way, we will not attempt to query an instance that likely has failed. Such a stateful optimization would also be able to adaptively load-balance and prefer faster-to-respond instances instead of slower ones.

5 Conclusions and Challenges

This paper proposes FLUXO, a framework for separating Internet service functionality from the architectural concerns of scalability, performance, and reliability. We describe several program transformations in the performance optimization space which show how, through the use of restricted programming model and runtime tracing, optimization decisions can be automated.

We still face several open challenges as we build a practical, deployable version of FLUXO, such as refining the set of semantic annotations through which developers convey information to enable valid program optimizations; ensuring the correctness of program transformations; and reducing the computational requirements of simulation-based analyses. If successful, however, the FLUXO programming model promises to enable almost any programmer to create large-scale, high-performance, reliable Internet services.

References

[1] Amazon. Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[2] R. Bekin and S. Dawson. LinkedIn Communication Architecture. Presentation at JavaOne, 2008.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*, 2004.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of SOSP*, 2007.

[5] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based Scalable Network Services. In *Proceedings of SOSP*, 1997.

[6] Google. Google App Engine. `http://code.google.com/appengine/`.

[7] J. Hamilton. On Designing and Deploying Internet-scale Services. In *Proceedings of LISA*, 2007.

[8] C. Henderson. Flickr and PHP. Presentation to Vancouver PHP Users Group, Aug 2004.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of EuroSys*, pages 59–72, 2007.

[10] Microsoft. Azure Services Platform. `http://www.microsoft.com/azure/`.

[11] T. O'Reilly. Database War Stories #3: Flickr. *O'Reilly Radar*, Apr 2006.

[12] A. Rasmussen, E. Kiciman, B. Livshits, and M. Musuvathi. Short paper: Improving the Responsiveness of Interactive Internet Services with Automatic Cache Placement. In *Proceedings of EuroSys*, 2009.

[13] R. Slobojan. Dan Farino About MySpaces Architecture. *InfoQ*, Nov 2008.

[14] Sun Microsystems. Java EE. `http://java.sun.com/javaee/`.

[15] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[16] Yahoo!, Inc. `http://pipes.yahoo.com/pipes/`, 2008.