Fluxo: A System for Internet Service Programming by Non-Expert Developers

Emre Kıcıman, Benjamin Livshits, Madanlal Musuvathi
Microsoft Research, Redmond, WA

Kevin C. Webb
UC San Diego, San Diego, CA

Abstract

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 latency, reliability, and scalability. Over time, a relatively small number of architectural patterns have emerged to address these challenges, such as tiering, caching, partitioning, and pre- or post-processing compute-intensive tasks. Unfortunately, *following* these patterns requires developers to have a deep understanding of the trade-offs involved in these patterns as well as an end-to-end understanding of their own system and its expected workloads. The result is that non-expert developers have a hard time applying these patterns in their code, leading to low-performing, highly suboptimal applications.

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 the use of a restricted programming language designed 1) to limit a developer's ability to write programs that are incompatible with widely used Internet service architectural patterns; and 2) to simplify the analysis needed to identify how architectural patterns should be applied to programs. Because architectural patterns are often highly dependent on application performance, workloads and data distributions, our platform captures such data as a runtime profile of the application and makes it available for use when determining how to apply architectural patterns. This separation makes service development accessible to non-experts by allowing them to focus on application features and leaving complicated architectural optimizations to experts writing application-agnostic, profile-guided optimization tools.

To evaluate FLUXO, we show how a variety of architectural patterns can be expressed as transformations applied to FLUXO programs. Even simple heuristics for automatically applying these optimizations can show reductions in latency ranging from 20-90% without requiring special effort from the application developer. We also demonstrate how a simple shared-nothing tiering and replication pattern is able to scale our test suite, a web-based IM, email, and addressbook application.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SoCC'10, June 10–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0036-0/10/06 ...$10.00.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Data-flow languages

General Terms

Algorithms, Experimentation, Languages, Performance

Keywords

languages for datacenter programming, compiler optimizations

1. INTRODUCTION

As an industry, we have over a decade of experience building large-scale Internet services, from e-commerce to social networking sites, all facing common challenges in latency, reliability, and scalability. Surveying published descriptions of these services, we find that while there are still challenges, many patterns have emerged as *best practices* for architecting well-performing and scalable Internet services. Such practices include tiering, partitioning, and replication for improving reliability and scalability; and caching, pre- and post-processing for reducing the latency of serving requests. Unfortunately, these best practices require developers to have a deep, end-to-end understanding of their own application, its workloads, performance characteristics and the trade-offs involved in different architectural patterns. The result is that many non-expert developers do not take advantage of these patterns and build poor-performing services or none at all.

We believe that these architectural patterns represent concerns that are orthogonal to core application functionality. Our goal is to separate them and introduce them into the program automatically in the same way that an optimizing compiler might improve code efficiency or introduce array bounds checking. This way, the developer is free to focus on the application-level functionality while allowing the compiler to take care of the rest. The challenge is that the use of these architectural patterns is closely tied to application characteristics and workloads. Our insight is that almost all of the factors that inform architectural design decisions are measurable, and that with a few simple restrictions to the programming model, developers can be prevented from writing programs incompatible with common distributed systems architectural patterns.

This paper presents FLUXO, a system that enables exactly this separation of core application functionality from the architectural patterns that help achieve performance, scalability, and reliability. FLUXO primarily targets non-expert developers, allowing them to focus on application functionality with FLUXO taking care of the rest. In other words, we are trying to broaden the developer base rather than replace the expert architect. In FLUXO, we use a *restricted programming model* that simplifies automatic analysis and application of architectural patterns; a data-flow based intermediate

representation for applying compiler optimizations; and an *execution runtime engine* that collects detailed measurements that are fed back to an FLUXO compiler to enable profile-guided optimizations.

As an initial demonstrate of the benefits of FLUXO, we have developed a set of four optimizations, primarily focusing on service latency and scalability. We have applied these optimizations to two sets of benchmarks: a selection of third-party Yahoo! Pipes programs and a set of four FLUXO representative services we have developed in a language we call FLIMP to closely mimic the functionality of large-scale email, address book, and instant messenger services. Our experiments show reductions in latency typically ranging from 20–93%. FLUXO's application of one simple tiering and replication pattern is able to scale our test suite of applications linearly in our experiments.

Contributions. This paper makes the following contributions:

- We propose FLUXO, an optimizing compiler and a runtime for developing large-scale interactive Internet services. To aid with FLUXO service development, we propose a new language called FLIMP that enforces the necessary restricted programming model.
- We propose a set of four performance optimizations focusing on improving overall service latency and scalability. We show how these optimizations can be applied to FLUXO programs automatically.
- Using FLIMP, we develop four services that mirror large-scale real-life instant messaging, email, address book, and authentication back-end services, and evaluate the efficacy of our optimizations on these applications. In addition to these services, we also use several hundred third-party Yahoo! Pipes programs for evaluating our optimizations.
- Through our experiments, we demonstrate that separation of functionality and architectural concerns can be achieved with the help of a profile-driven service compiler. We also show the effectiveness of even simple automated optimizations in practice by describing the results of applying four optimizations in FLUXO.

The rest of the paper is organized as follows. Section 2 provides background on building large-scale Internet services. Section 3 provides an overview of FLUXO. Section 4 gives a brief FLIMP tutorial, while Section 5 goes into the technical details of our optimizations. Section 6 summarizes our experimental results. Finally, Section 7 describes related work, and Section 8 concludes.

2. BUILDING INTERNET SERVICES

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 have 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 [6, 18, 32, 37].

Hamilton [17] and Henderson [19, 20] provide a catalogue of some common patterns. For example, almost all services use some kind of *replication* and *data partitioning* to achieve higher reliability and scale; *caching* to reduce the latency caused by performance bottlenecks; and *pre- and post-processing* to remove computational overhead from the critical path of serving requests. Other common patterns include tiering, data denormalization, retries, and others.

While common, these techniques are not simply reused "cookie-cutter," but must be specialized to suit a specific service's require-

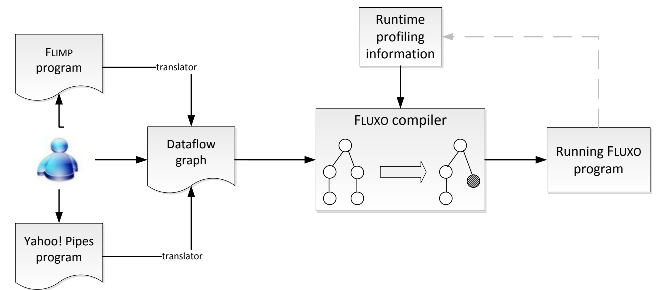

Figure 1: FLUXO architectural diagram.

ments and workloads. Understanding the interactions among system components, workloads, and semantic requirements, however, is non-trivial. For instance, it has been shown before that manually maintaining near-optimal caching policies with dynamic changes in the input workload distribution is simply not feasible [33].

Practical experience with large-scale Internet services presents a number of interesting lessons. The architecture used by Flickr has been described quite extensively [18, 19, 32]. In the case of Flickr, as the server was expanding to support more users, the architecture was redesigned several times to support the growing user pool, with server tiering and data normalization being introduced over time. In the case of LinkedIn [6], the service went through at least four significant architectural revisions over a period of three years. Over time, optimizations such as partial data partitioning, post-computation, and asynchrony were introduced. Similarly, MySpace [37] has altered tiering and caching aspects of their architecture. Needless to say, every single one of these changes was a time- and resource-consuming undertaking.

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 [39], 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 [5, 29]. Google's App Engine provides a scalable platform for a narrow class of services [14]. SEDA uses a staged event driven architecture to separate application event processing from controllers that handle resource allocation decisions [43]. 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 [13, 21].

The lessons learned from the large-scale services described above show that a great deal of complexity is involved in building competitive Internet services. As a result, building such a service is currently outside the reach of non-expert developers. While we may expect that the largest services will always be built by experts, even small-scale services are challenged to achieve low-latency and high-availability. The goal of FLUXO is to enable non-expert developers to build such services that have performance and reliability characteristics approaching those of expert-designed services.

3. OVERVIEW OF FLUXO

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 initially written in FLIMP and converted to a dataflow representation internally. 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 cluster of physical machines in a data center.

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 general, FLUXO optimizations are profile-guided [16]: the FLUXO runtime automatically instruments the program to collect runtime information, such as workload distributions and performance profiles that can be analyzed to direct future program 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 relies on the developer to provide semantic annotations that describe attributes such as consistency requirements and side-effects. Note that a deliberate part of our design is to keep the annotation burden relatively light. Section 4.3 addresses state annotations.

3.1 Dataflow in Fluxo

A commonly-cited benefit of dataflow programming is the ability to extract parallelism from dataflow graphs [23, 31]. This is in part because both control and data dependencies are represented in dataflow *explicitly*, whereas in most general-purpose languages dependency analysis represents a significant stumbling block on the path to automatic parallelization [12]. In a language with pointers or references such as C, C#, or Java, the possibility of *aliasing* complicates the problem even further as dependencies between program variables become harder to extract automatically.

In FLUXO, we largely treat a dataflow representation as a convenient intermediate representation (IR) that simplifies the refactoring and optimization of progams. The majority of experiments in this paper are on FLUXO programs implemented in FLIMP, an imperative language that *compiles down* to this intermediate dataflow representation. As described in Section 4, we restrict FLIMP to preserve many of the benefits of dataflow programs for the analysis of control and data dependencies.

Yahoo! pipes [45] and the now defunct Popfly [30] are two examples of dataflow programming being used for the development of Internet services. To support experimentation with a broader base of programs, we have implemented a front end to FLUXO that reads Yahoo! pipes programs and translates them into our intermediate dataflow representation. As a result, we have been able to successfully apply the same set of optimizations to Yahoo! pipes programs in addition to our own FLIMP programs.

3.2 Restricted Programming Model

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

Logically, the execution model provided by the FLUXO runtime is *turn-based concurrency*. At runtime, FLUXO maintains an *delivery queue of inputs* to be delivered to nodes. At every logical *turn*, an input is picked off the queue and delivered to a node. If all required inputs have been delivered to the node then the logic of the node is executed and one or more outputs is produced. These outputs are then placed on the delivery queue. For the experiments in this paper, our implementation uses a single global delivery queue per machine and multiple execution threads. When dequeueing inputs, the front of the queue is scanned to group together and deliver inputs destined to the same node. While not reported in this paper, we have recently begun tests with SEDA-style stages and resource controllers for delivering inputs and executing node logic [43].

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

Existing dataflow-based development frameworks such as Yahoo! Pipes [45] have demonstrated that a wide-variety of services can be built using a small number of standard components. 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 are free to create new libraries of FLUXO components, for example, to reuse code from existing applications, although FLUXO provides only inter-component optimizations.

3.3 State Semantics

For purposes of optimization, it is crucial to be able to reason about state manipulation. Indeed, for instance, we can only reorder two state writes if they are guaranteed not to create a write-write conflict. The manipulation is made fully explicit in FLUXO by first requiring a declaration of all state shared between requests, including the kind of storage service being used (e.g., soft or persistent storage) and the type of the shared data. State reads are expessed as LINQ queries, enabling programmatic analysis of the query expression. State writes use a similarly analyzable API.

To ensure idempotency of our state operation primitives—namely insert, delete and update—we require that entities must be uniquely identified. While we do not explicitly enforce it in our programming model, we also strongly recommend that FLIMP programs carefully design their data schemas to enable the logic that controls their state updates to also be idempotent. For example, consider a simple FLIMP program intended to increment a counter. If we read the current counter value, increment it in memory, and then update the counter in the sate service, we will have created a dataflow program where each individual element is idempotent, but the whole is not. We would recommend that either the program includes as part of its input a specific version of the counter to read; or the data schema is designed as a log-formatted / append only schema (e.g., updating the counter means writing an "add one" command and each write includes the unique ID of the request generating the write (multiple writes may now occur, but they can be unambiguously merged together later). In either case, the program with respects to own inputs has become idempotent.

3.4 Runtime Metric Collection

The goal of FLUXO is to separate application functionality from architectural patterns and design decisions, but effective implementation of these patterns is dependent on knowledge of application-level factors such as component performance, workload, and data distributions. To help make accurate architectural design decisions while preserving this separation, FLUXO's runtime collects an extensive set of metrics on both nodes and individual requests: (1)

performance latency for each component per turn; (2) size of data flowing across each edge; (3) memory requirements; (4) length of the delivery queue; (5) hash values of the key attributes of data.

These metrics help profile-driven optimization in FLUXO analyze the dataflow program and focus optimizations on performance bottlenecks and sources of unusual latency deviations, as well as calculate expected resource usage of changes to the dataflow program. Metrics of data sizes and the frequency of message passing along an edge help determine the cost of distributing a program across multiple machines. Collecting hashes of data keys as they flow through the system allows calculation of expected cache hit rates and to estimate the quality of data partitioning schemes.

It is worth noting that in most cases analyses based on runtime profiles will only provide estimates of the effects of different architectural patterns. Thus, we envision a bootstrapping cycle of metric collection and optimization with test and real workloads, followed by periodic re-optimizations to keep up with new application features and changes in workloads. While the process of re-deploying an application with a different set of optimizations is necessary, it is a problem outside the scope of this paper.

3.5 Analysis Stage

The analysis stage in FLUXO is responsible for determining how to apply specific architectural optimizations to a program. The data collected by the FLUXO runtime can enable different types of analysis techniques for subsequent optimizations. Depending on the precision requirements of the optimization and the time available to run the analysis, one can pick from several strategies:

Heuristics: Heuristics in FLUXO are attempts to capture and generalize the rules of thumb that Internet service architects use today. These include splitting Internet services into three-tiers, separating persistent state from application logic, load balancing, adding caches to performance bottlenecks, data partitioning per user, etc. [17]. Heuristics are often the simplest to implement and evaluate, without necessarily providing the best outcome. One simple technique for mitigating the worst inaccuracies in heuristic-based analyses is deploy selected optimizations and then back off if they prove detrimental. For an example of a heuristic optimization, see Section 5.2.3 describing a post-computation optimization.

Queuing Models: A FLUXO dataflow graph is amenable to a queuing model analysis, which provides an approximate representation of the performance of a queuing system. The input to such a model consists of throughput and latency statistics for components as well as queue length information. This is all directly captured by our metric collection machinery. We can capture the primary performance effects of changes made to the dataflow graph.

Primary effects would include not running particular components because of a cache, but would often not capture changes because of a different workload mix and other application-specific behaviors. In practice, queuing models often provide accuracies within 10% for throughput estimates and 30% for latency estimates [26, 38].

Simulation: The FLUXO runtime includes a service simulator that can replay modified traces of an application. This feature can be used to evaluate a given optimization to determine its utility [1]. This is because in many cases, analytic models do not provide sufficient precision. For instance, in simulating a caching optimization, we are ultimately interested in the service's end-to-end latency [33].

To simulate a cache hit during a given session, the simulator temporarily adjusts the event stream for that session. Events that would not have occurred because of the cache hit are removed from consideration, and the times on all other events are adjusted to simulate the time savings produced by the cache hit. The simulator records the net decrease in end-to-end latency across all sessions and reports this number as the simulated caching policy's utility. The FLUXO runtime and the FLUXO simulator utilize the same execution runtime and cache implementation, so we believe our simulations to be accurate.

The experiments reported in Section 6 are implemented as simple heuristic analyses. We have reported on the results of simulation-based analyses in the past [33], and while we appreciate their accuracy, we find them prohibitively slow for exploring large-state spaces without additional guidance. In the future, we plan to investigate further the trade-offs among analysis techniques and explore which classes of architectural optimizations are best suited for each.

```
service<http> HelloWorld {
    handler Default( urlargs, cookies ) {
        var name = urlargs["name"];

        name = csharp(name) {@ name.ToUpper(); @};

        return "<html><body>Hello " + name
               + "</body></html>";
    }
}
```

Figure 2: "Hello world" service in FLIMP.

4. FLIMP: THE FLUXO LANGUAGE

While we find dataflow to be an attractive internal representation on which FLUXO can perform optimizations, experience with dataflow graphs suggests that a developer can only reason about graphs of a certain size before visualization becomes a challenge. Once the program gets large enough, traditional encapsulation constructs such as procedures and modules become sorely necessary. This is also bourne out by our observation that, from a sample of several hundred Yahoo! Pipes programs hosted by Yahoo!, most are under 20 nodes in size. This, combined with the familiarity of imperative-style programs for our target developer audience induced us to build a special-purpose imperative programming language that would be easy to map to our internal dataflow representation without extensive program analysis (e.g., easy to discern control and dataflow dependencies).

In this section, we describe FLIMP, a special-purpose language we have developed for programming FLUXO services. FLIMP is the middle ground between the constrained environment of dataflow and the "free-wheeling" world of languages such as C# or Java. Because of the modular structure of FLIMP services, we are able to create services that are several hundred dataflow nodes in size. In the future, we are planning to explore issues of correctness and testing of FLIMP programs in addition to the optimizations that are the subject of this paper.

Our proof of expressiveness for FLIMP comes from the ability to construct useful services such as email and Instant Messenger succinctly; these services are further described in Section 6. At the same time, once automatically translated into dataflow, FLIMP programs are fully amenable to automatic analysis and optimization. To achieve this goal, as mentioned before, we require that all state manipulation be made explicit in FLIMP programs.

4.1 Hello World in Flimp

Figure 2 shows a simple "Hello world" service written in FLIMP. In this section, we examine how it is put together. `HelloWorld` is an `Http` service, which means that the program expects to be accessed via an HTTP interface and calling semantics. The FLUXO

literal	::=	`1234.5` | `"abcd"` | `true` | `false` | `undefined`
expr	::=	*id* | *literal* | (*expr*) | *cond* | *expr* `.` *id* | *expr* `[` *expr* `]` *stateRead* | *stateWrite* | *stateUpdate*
stateRead	::=	`read(` $id_1, \ldots, id_n$`){@...@}`
stateWrite	::=	`insert(` *stateTable*, *expr*`)`
stateUpdate	::=	`update(` *stateTable*, *expr*`)`
cond	::=	$expr_1$ == $expr_2$ |$expr_1$!= $expr_2$
stmt	::=	*block* | *varDecl* | *assignStmt* | *ifStmt* | *foreachStmt* | *returnStmt* | *csharp*
block	::=	{*stmt**}
varDecl	::=	`var` *x* = *expr*`;`
assignStmt	::=	*x* = *y*`;` | $x \leftarrow y$`;`
ifStmt	::=	`if`(*cond*){*stmt*} [`else`{*stmt*}]`;`
foreachStmt	::=	`foreach`(*x* `in` *expr*) *stmt*`;`
returnStmt	::=	`return` *expr*`;`
csharp	::=	[`var` *id* =] `csharp(` $id_1, \ldots, id_n$`){@...@}`
program	::=	`service` *id*{*using**; *handler**}
handler	::=	`handler` *id*($id_1, \ldots, id_n$){*stmt*}
using	::=	`using` *dotNetNamespace*;
stateTable	::=	`state`<*dotNetType*> *persistence* *id*`;`
persistence	::=	`soft` | `persistent("...")`

Figure 3: BNF grammar for FLIMP.

runtime provides a basic Web-based HTTP scaffolding, with fields and links corresponding to individual handlers, etc. This is similar to, for instance, generating a browser-based Web interface from a Web service WSDL description [10]. This scaffolding UI can be overridden by implementing handlers to return custom HTML interfaces. This simple FLIMP service only has a single request *handler* named `Default`; as Section 6 illustrates, our test services have as many as 20 handlers acting as entry points for various kinds of service functionality.

Handlers in FLIMP programs accept named parameters; the code shows how handler parameter `name` is extracted from the `urlargs` array. These are essentially the arguments that are part of an HTTP `GET` request. FLIMP supports so-called `csharp` blocks; these act as an opportunity for the developer to "escape" the constraints of FLIMP and to write C# code instead. Note that *parameter variables* — variable `name` in this case — are being explicitly passed into the `csharp` block from the outside. We assume that, unless stated otherwise, `csharp` blocks have no observable side effects.

As a general rule, while they provide access to useful libraries, `csharp` blocks should be used sparingly, as they can limit optimization opportunities. Furthermore, because they are currently embedded without analysis or sandboxing, reliance on C# libraries and code may introduce subtle hard-to-detect side effects.

4.2 Flimp Constructs Examined

Figure 3 shows a BNF grammar for FLIMP. In this section, we examine some of the more unusual language constructs in turn:

`var` **introduction**: FLIMP handlers can introduce and redefine variables. These variables are untyped; we use runtime checks to make sure that, for instance, the argument to iterate over for a `foreach` statement is a set. When translating FLIMP into dataflow, we effectively introduce a form of a single static assignment form [2]. For example, in the case of an `if` node, we put a `ConditionalGate` node at the merge point for all variables redefined within the `if` or `else` branch; this corresponds to gated SSA φ-nodes in compiler parlance [42].

Loads and stores: FLIMP supports both array and field loads (i.e. `x = y.f` or `x = y[i]`) but not stores such as `x.f = y`. This means that we obviate the need for generalized pointer analysis to figure out which locations a store statement could be affecting, greatly simplifying reasoning about code.

`csharp` **blocks**: As mentioned above, FLIMP provides facilities for calling into C# code. Note that *arguments* may be passed into C# as well. Optionally, the result of a `csharp` block may be assigned to a fresh variable. This explicit form or argument passing and return makes it easy to extract data dependencies, making standard compiler analyses such as reaching definitions easy to construct.

`foreach` **loops**: FLIMP's explicit dependencies make it possible to *parallelize* `foreach` loops. The absence of pointers in FLIMP along with explicit state manipulation makes write-write and write-read conflict detection easy. A special *append* form of assignment,

$$set \ \leftarrow \ value$$

a form of list comprehension [25], allows us to compute sets for use after the loop *without* introducing dependencies between iterations.

```
// persistent table of users
state<AuthSQLUser> persistent(
        "AuthSQLUsersDataContext",
        "AuthSQLUsers",
        "Data Source=A2738424\SQLEXPRESS...")
Users;

// soft-state table of conversations
state<IMConversation> soft Conversations;
```

Figure 4: State declarations.

4.3 State Manipulation

State manipulation is made explicit in FLIMP. In fact, the developer is required to specify the data tables that they are using in their service, explicitly denoting whether they are part of persistent or soft state. Examples of state declaration and state manipulation are shown in Figures 4 and 5. The code in Figure 4 declares both soft and persistent SQL-backed state in FLIMP. For persistent state, we specify the database access string as part of the declaration. In both cases, the .NET type of the data to be stored is explicitly specified; this is so that the FLUXO runtime can instantiate the right kind of objects and to serialize them properly.

Figure 5 demonstrates how to perform state reads (in this case, reads from the table of users) and how to create a new user in the table. For read operations we support LINQ queries [3], giving FLIMP developers the full power of LINQ. Because LINQ queries are both structured and quite explicit, they are amenable to automatic analysis. For instance, we can automatically determine which *columns* of which state tables may be accessed by a given handler. This could be useful if we try to parallelize handler execution and are interested in avoiding read-write conflicts. Note that we only allow `select` LINQ expressions inside `read` statements.

4.4 Translating Flimp into Dataflow

As noted above, FLIMP code must be translated into one or more dataflow graphs before it can be optimized and executed by FLUXO. Translation begins with the instantiation of a FLIMP service — the translator constructs a new service container, which acts as a namespace for all of the service's handlers and state tables, and it examines the service's declaration and instantiates the corresponding service type. When instantiated, a service automatically creates a Web server that listens for requests and directs them to the correct handler. Next, the translator uses the state table declarations to initialize database connections and build the service's

```
// find all users by name
var presences = read(Users, presenceUser) {@
  from user in Users
  where string.Equals(user.Value.Username,
                                    presenceUser)
  select user
@};

// insert a new user into the Users table
var newUser = csharp(username, password, userID) {@
  return new IMUser((string) username,
              (string) password, (string) userID);
@};
insert(Users, newUser);
```

Figure 5: State manipulation.

```
 1. handler GetPresence( urlargs, cookies ) {
 2.     var presenceUser = urlargs["presenceUser"];
 3.
 4.     if (presenceUser == undefined) {
 5.         return "Missing parameter: presenceUser";
 6.     }
 7.
 8.     var presences = read(Users, presenceUser) {@
 9.     from user in Users where
10.     string.Equals(user.Value.Username, presenceUser)
11.         select user
12.     @};
13.
14.     var result = csharp(presences) {@...@};
15.
16.     return result;
17. }
```

Figure 6: `GetPresence` handler from Instant Messenger.

state tables. Finally, the translator reads handler declarations and constructs their dataflow graph equivalents.

Given the FLIMP code for a handler, the handler translation process begins by parsing the code and building an abstract syntax tree (AST). Each statement in the FLIMP AST corresponds to a node in the resulting dataflow graph. Edges are added between nodes as a result of explicit data dependencies. Data dependencies are easy to construct because FLIMP programs list *uses* explicitly; for instance, a LINQ block lists the arguments it uses.

Example 1. In the example in Figure 6, the top two nodes (`argsnode` and `literalnode`) are implicitly created to produce handler arguments and literal values, respectively. Line 2 of the handler in Figure 6 produces a member-lookup node called `member` − 65, which looks up the `presenceUser` argument. The resulting value is delivered over a dataflow edge as an input to any other nodes that reference it (in this case, an `EqualityTest` on line 4 and a `LINQ` state read node on line 8).

FLIMP's `csharp` and state manipulation statements (such as the `LINQ` state read on line 8) require extra effort to convert into dataflow graph nodes because their actions are dependent on their embedded C# and LINQ code. For such statements, an auxiliary step is added to the translation process in which the embedded code is first compiled in memory and wrapped inside of a placeholder node. The placeholder is added to the graph, and it simply calls into the embedded code to produce its results. `Linq71` is an example of such a node.

In addition to defining nodes and dataflow dependencies, some FLIMP statements such as `if` and `return` alter control flow. To prevent the unwanted execution of nodes that belong to untaken conditionals or occur after `return` statements, we add additional

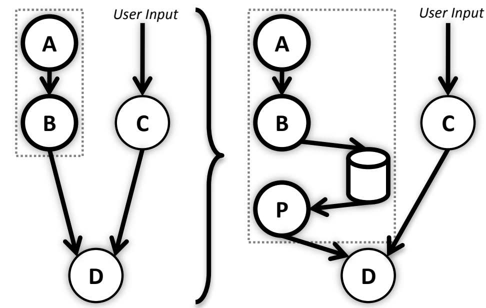

Figure 8: The constant propagation transformation. `P` represents a `constantprop` node.

input dependencies to trigger the execution of such nodes. Nodes that are dependent on a conditional statement, like the `if` on line 4 of Figure 6, receive a dependency on a conditional trigger whose value is determined by the execution of the condition. The trigger's output is only delivered to nodes on the side of the branch that is taken. Nodes on the other side of the branch will never receive the trigger, and as a result they will never execute. □

5. OPTIMIZATIONS

Architectural optimizations described in this section play a key role in optimizing large-scale deployed web sites such as Flickr and MySpace. For each optimization in this section, Figure 7 shows common pitfalls that inexperienced developers face that a particular optimization prevents.

FLUXO views optimizations in terms of dataflow graph transformations; we feel that this gives FLUXO approach a degree of uniformity. In compiler terminology, applying FLUXO optimizations can be thought of as intermediate representation (IR) rewrites. For instance, FLUXO can optimize dataflow graphs optained from both Yahoo! Pipes and FLIMP using the same set of transformations. Note that the optimizations described in this section affect how the resulting distributed system is put together. For instance, a post-computation optimization will create a worker task that will be run after the fact, independently of the web request that created it and perhaps executing on a different machine.

5.1 Optimizations as Graph Transformations

The transformation process consists of two phases: graph *analysis* and graph *rewriting*. Given an initial set of input graphs, a transformer uses the analysis phase to determine *where* the input graphs should be rewritten to implement the optimization. Depending on the nature of the optimization, such determinations could be based on manually-created specifications or an automated analysis of the information contained in the input graph structure, input FLIMP code, or recorded runtime statistics. As part of the analysis process, a transformer must ensure that its determinations are valid according to any constraints that are imposed by the dataflow graph semantics. Caches, for example, should not contain nodes that have non-deterministic outputs; failing to consider this constraint may lead to an *unsound* optimization.

After completing the analysis phase, a transformer transitions to the rewriting phase, in which it focuses on *how* it should modify the input graphs to apply the optimization. During the rewriting phase, a transformer modifies the input graphs by adding and removing nodes and edges. In the rest of this section we consider different types of optimization we have implemented in FLUXO. Section 5.2 focuses on latency-reducing optimizations. Section 5.3 focuses on optimizations that improve service scalability.

Optimization	Pitfall
Constant propagation	Precomputation is a critical aspect of building a low-latency service. Some application frameworks [14] even limit the amount of time or the number of data store or other API calls one can make while handling a web request. In order not to exceed these bounds, developers are forced to figure out ways to precompute common queries [15, 22]. However, they have to manually decide which queries need to be precomputed and how, and often must rewrite code to execute as a precomputation in a different context.
Caching	A common mistake that trips inexperienced developers when invalidating caches is not invalidating computations that are currently "in flight" and are based upon out-of-date data. An optimization designer will need to take care of this possibility, but getting it right once is better that relegating this responsibility to the developer [18, 37].
Post-computation	Many time-consuming tasks can be done after the fact, offline or using a map-reduce job. For instance, thumbnailing an image submitted to an application like Facebook does not need to be done until later: the request can be returned to the user immediately [27]. The decision of what to post-compute, by when it must be computed is left to the developer.
Scale-out	Shared-nothing scale-out architectures rely on separating persistent state from stateless computation. A common mistake for non-experts is the accidental introduction of state by using local disk or static in-memory variables. More complicated scale-out patterns that involve data and workload partitioning provide additional difficulties for non-experts as they must reason about user workloads, data distributions, query workloads, and the performance of the backend database or storage system.

Figure 7: Optimizations described in this section and their effect in practice.

5.2 Latency Optimizations

End-user responsiveness is often cited as a characteristic responsible for success of a particular Internet application of site [7, 8, 36]. Our first set of optimizations focuses on approaches that reduce the end-to-end request latency.

5.2.1 *Constant Propagation*

The first optimization we consider is a special form of constant propagation, a commonly-used compiler optimization. Our optimization separates a dataflow graph's nodes into two types: those that are dependent on user input and those that are not. Using methods that are analogous to compiler constant propagation [2, 9], this optimization performs graph transformations to improve execution latency. For example, our experience with Yahoo! Pipes suggests that many Pipes graphs consist of two independent branches. One branch is responsible for fetching data feeds with *statically known* URLs and applying operators to their output. The second branch prompts the user to enter a parameter, which is used as a filter or search term. Eventually, the two branches are merged, producing the final output. Figure 8 depicts such an example, in which we consider the first branch to be *constant* because it can be executed without requiring any user interaction. The insight behind this optimization is that any such constant subsections of the dataflow graph can be safely computed before the arrival of any user queries, subject to freshness requirements, just like a typical optimizing compiler would statically perform and eliminate computations that only rely on constants.

To perform this optimization, the graph transformer begins by annotating each node in the graph with a boolean indicating whether or not the node or any of its predecessors vary their behavior as a result of user inputs. We annotate nodes using a depth-first search, running it backward from the final output node. As the search progresses, it marks every constant node that has only input-dependent children. Any such marked node exists at the border between a constant sub-graph and the original graph. Note that the same analysis can be also performed using a standard dataflow formulation as described in Aho et. al. [2].

For each of the marked border nodes, the graph transformer creates a new `constantprop` node that will represent the entire sub-graph rooted at the marked node. The transformer gives the `constantprop` node a reference to the constant sub-graph and then rewrites the original graph to contain the `constantprop` node in place of the constant sub-graph that it represents.

Each `constantprop` node is executed immediately. During the first execution, a `constantprop` node executes its sub-graph and stores the final result. It also starts a *timer* that will periodically re-

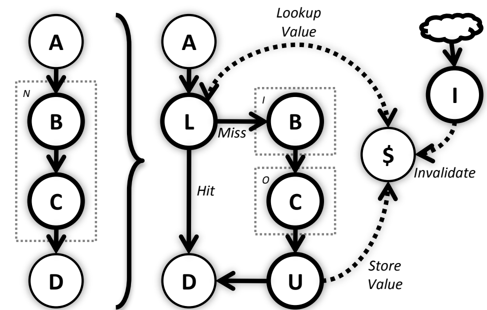

Figure 9: The caching transformation. Nodes `L`, `U`, and `I` represent lookup, update, and invalidate nodes.

schedule execution of the sub-graph to ensure that stored versions of external data sources (such as RSS feeds) do not become stale.

Upon receiving a user request, execution of the original graph immediately retrieves the stored result for each of its `constantprop` nodes. Thus, user-perceived computation time is reduced as a result of constant sub-graph pre-computation.

5.2.2 *Caching*

Our next optimization, caching, seeks to reduce latency and improve overall throughput by eliminating redundant computation. The caching optimization operates over a sub-graph of the original, unoptimized graph. For a candidate sub-graph, we define N to be the set of nodes contained in that sub-graph.

We also define I to be the subset of N that receive inputs from nodes that are outside of N and O to be the subset of N that send outputs to nodes that are outside of N, as displayed in Figure 9. Note that we have to be careful when considering what can be a valid cache: for instance, we need to ensure there is no way to get to an internal node within a cache without visiting one of the inputs. We can use dominance and post-dominance from graph theory [2] to encode the necessary conditions:

1. $\forall n \in N : I \ dom \ n$;
2. $\forall n \in N : O \ pdom \ n$;
3. $\forall n \in N$, n is deterministic and side-effect free.

Our implementation currently supports two forms of analysis for placing caches. The first method uses manually-created specification files to indicate which subsections of a graph should be cached. The second form of analysis uses simple heuristics to automatically place caches around expensive regions of a graph.

As shown in Figure 9, given a tuple of node sets $\langle N, I, O\rangle$ that is suitable to cache, the cache transformer's rewriting phase begins

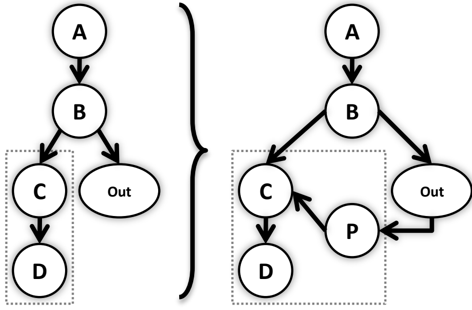

Figure 10: The post-computation transformation. `P` represents a `postcompute` node.

by inserting a `lookup` node in front of I's incoming edges. Next, it inserts a `update` node behind O's outgoing edges. Finally, the transformer adds an edge from the `lookup` node to each of O's outgoing edge destinations. The `lookup` and `update` nodes share a common in-memory dictionary for storing cached values.

This configuration ensures that all inputs to the cached sub-graph are first sent to the `lookup` node, which may find the sub-graph's result to be in one of three states:

1. **Available**: The result is already stored in the cache. The lookup module outputs the stored value, bypassing execution of the sub-graph.
2. **Unavailable**: The result is not stored in the cache, and it must be produced by executing the sub-graph. The lookup module passes its inputs through to the sub-graph, which executes normally. The result of the execution is captured by the update module, which stores the result in the cache.
3. **In production**: The result is not yet stored in the cache, but another request has already triggered its production. Rather than executing the sub-graph a second time, the request will yield until the value becomes available in the cache.

The first validity constraint ensures that, after the rewriting phase, every node in N receives inputs only from the `lookup` node or other nodes in N. Likewise, the second constraint guarantees that all nodes in N deliver outputs only to the `update` node or other nodes in N. Together, these conditions prevent the cached sub-graph from directly interacting with the rest of the original graph. Finally, the third validity constraint precludes the caching of nodes that might indirectly cause side-effects.

It should be noted that, when a cached sub-graph includes a read from a state table, writes to that table may need to invalidate cached results. To ensure correct results, the cache transformer may insert `invalidate` nodes after such writes to remove stored values from a cache. In many cases, such writes exist in other graphs altogether. Note that caches may become stale, which is why we support a *time-to-live* annotation. Entries in the cache are automatically flushed once their time-to-live is exceeded.

5.2.3 Post-computation

The goal of the post-computation transformer is to reduce user-perceived latency by *delaying* the execution of non-critical nodes until after the user has received a response from a service. Intuitively, the transformer seeks to shorten the critical path by deferring the execution of nodes that do not return results to the user. One common example of post-computation is sending email. Typically upon sending an email, a user's web-based mail client will immediately inform the user that the message has been sent, despite the fact that it may be several seconds or even minutes before the message is transmitted via SMTP to the recipient's mail server. In this example, the mail client is post-computing delivery of the email to eliminate the delivery delay from the user's critical path latency.

The validity constraint for post-computation dictates that post-computed nodes do not affect the results that are returned to the user. The analysis phase of our post-computation transformer is fully automated. Given an input dataflow graph, it begins by determining which nodes in the graph influence the user's result. To perform this analysis, the transformer searches the graph backwards, starting from each return node. All of the nodes that it encounters in the search are marked, indicating that they have an output path that reaches a return node. Any nodes that are not marked after the search is complete are known to have no output path that reaches a return node. Such nodes are suitable for post-computation.

Having determined the set of nodes to be post-computed, the transformer's rewriting phase begins by adding a single `postcompute` node to the graph as depicted in Figure 10. The `postcompute` node receives incoming edges from all of the graph's return nodes. Next, the transformer adds an edge from the `postcompute` node to each of the nodes that were unmarked after the analysis phase.

In this implementation of post-computation, the introduction of the `postcompute` node and its edges creates a new chain of dataflow dependencies — the unmarked nodes cannot execute until they receive an input from the `postcompute` node, which in turn cannot execute until a return node has executed. One alternative implementation is to use a persistent queuing mechanism to send the task to a background worker node to be computed later. In either implementation, the nodes that are suitable for post-computation have been removed from the user's critical path.

It is important to note that the post-computation transformer does not change the semantics of FLUXO's execution model. Any node in the model can execute as soon as its inputs become available. Even without the post-computation optimization applied, nodes may still execute after the execution of a return node when the return node becomes runnable first. The post-computation transformation simply guarantees that return nodes will execute first with respect to nodes that do not produce user-visible results.

State write nodes are common candidates for post-computation. In some situations, however, it may be necessary to ensure that a write to state has *completed* before returning a result to the user. To satisfy this scenario, all state writing nodes produce a value that can be read by other nodes in order to create the data dependencies necessary to restrict post-computation.

5.3 Scale-Out Optimizations

The final optimization we present is a transformer that applies a simple, shared-nothing replication pattern to scale up an Internet service. This shared-nothing pattern implements a two-tiered architecture, where all stateful components are placed in a back-end storage tier, and all other components are placed in a front-end tier. None of the front-end nodes communicate with each other, so this tier of the system can be scaled up simply by adding additional machines. Our tiering algorithm relies solely on information derived from the static dataflow graph to separate stateless from stateful components. The performance profiling information we gather at runtime can be analyzed using simple queuing models to determine an appropriate replication factor [38].

Other commonly implemented patterns for scaling out a service architecture include more complex multi-tiered architectures and also tree-based aggregation structures to handle processing of larger scale data. While the FLIMP implementations of these scale-

	Average	Median	StdDev	Min	Max
Nodes	11.3	8	10.0	2	103
Edges	12.4	8	13.3	1	136

Figure 11: Sizes of dataflow graphs produced from Yahoo! Pipes.

Service	Lines of code	Hand-lers	Number of nodes	edges	State tables	reads	writes
Auth	116	4	62	168	1	4	2
InstantMessenger	481	20	268	692	3	12	10
AddressBook	380	9	209	606	2	12	8
Mail	188	4	93	273	2	5	3

Figure 12: Static service statistics

out techniques, as well as optimizations for improving reliability, such as the addition of explicit retry operators, remain future work, we believe that each is implementable using FLIMP's basic abstractions, though in some cases, such as the dependence of tree-based architecture on homomorphic functions, the optimization may require additional semantic annotations.

6. EXPERIMENTAL EVALUATION

This section is organized as follows. Section 6.1 talks about our experimental methodology, benchmark selection, and choosing appropriate workloads. Section 6.2 describes the effect of latency optimizations. Section 6.3 talks about scalability optimizations.

6.1 Experimental Methodology

To evaluate the effectiveness of FLUXO on a broad range of scenarios, the current FLUXO implementation supports two set-ups. The first one is for executing FLIMP services, the second one is for executing Yahoo! Pipes programs. FLUXO uses the same optimization and runtime layer for both set-ups with different frontends and component libraries. Our FLIMP component library includes, for example, the components that implement FLIMP's primitive conditionals and comparisons, while our Yahoo! Pipes component library includes Pipes' higher-level components such as `FetchFeed` or `YahooSearch` modules to fetch data from an arbitrary RSS feed or obtain data by performing a Yahoo! search. In total, the FLIMP library contains 15 components and the library for Yahoo! Pipes contains 26 components. This is in addition to generic modules, such as the ones use for caching, that are are shared by both types of programs to make it possible for optimizations to work uniformly.

6.1.1 Yahoo! Pipes Programs

Yahoo! Pipes is a tool for composing mash-ups of Web content. A user creates a pipe by connecting small, single-purpose modules using a graphical Web-based editor. Modules are classified into several categories such as data sources (e.g., Fetching from RSS feeds, Flickr, or Yahoo Search), user inputs, operators (e.g., Loop, Sort, Union, etc.), and data manipulation modules for dealing with items like URLs and dates. Similar to the notion of pipes in a Unix shell, the idea behind Yahoo! Pipes is that powerful results can be achieved by connecting many simple pieces. Users can choose to "publish" the pipes they create, making them publicly accessible. Anyone can "clone" a published pipe to modify its behavior or use it as a sub-component in a new pipe. More than 50,000 pipes have been published to date.

Such Pipes programs are interesting for our evaluation of FLUXO because they demonstrate user-created, desirable functionality already expressed in a dataflow language. However, while Yahoo! Pipes use a combination of client-side and server side execution, our pipes execution engine does all the work on the server, without relying on client-side JavaScript. To experiment with FLUXO, we have downloaded a set of 998 pipes programs from the set hosted at `pipes.yahoo.com`. Most of these are implemented as a stand-alone pipes, written in a JSON format, many also depend on sub-pipes that we downloaded as separate JSON files. As mentioned earlier, we developed a JSON parser mapping the input into a dataflow representation in FLUXO. Our reason for choosing Yahoo! Pipes for the purposes of experimentation was to run our experiments on unmodified third-party programs. While we based some of our experiments on Pipes, as can be seen from Figure 11, most Yahoo! Pipes programs are relatively small with a median size of only 8 nodes. Beyond the size and complexity limitations found in these programs, two other issues present an obstacle to easy experimentation.

First, Yahoo! Pipes often require user input to run, which makes automation and repeatable timing measurements difficult to achieve. In many cases, we need to decide what the proper *workload mix* might be for a particular user input. For instance, when entering a search query term, what distribution should we use? How about entering a zip code? The second issue that often presents a challenge with Pipes programs is their reliance on *external input*. Modules such as `FetchFeed` and `Flickr` are routinely used to fetch external data. However, many Pipes rely on feeds that are no longer valid, complicating running these programs.

6.1.2 FLIMP Benchmarks

To address both the issues of scale and the challenges of workload generation and replayability, we have designed and experimented with services written in FLIMP, in addition to Yahoo! Pipes. These FLIMP services are a cooperative set of four services whose aim is to demonstrate the construction and optimization opportunities of larger, more realistic Web services in FLUXO. A summary of information about these services in shown in Figure 12. Column 2 shows the size of each service in terms of the number of lines of FLIMP code. As can be seen from the table, we are able to express complex services succinctly, in only several hundred lines of code. Column 3 lists the number of handlers that constitute entry points into each service. Finally, columns 4 and 5 list the sizes of the dataflow graphs that are produced from these services. Columns 6–8 show information about the amount of state used by each service. These four services have several state tables and about a dozen read and write statements each.

The `Auth` service supplies basic authentication functionality and acts as the foundation for the other three services. Users interact with the authentication service by registering/deleting persistent accounts and by logging in to the service to obtain account credentials for identifying themselves to other services. The authentication service also supplies the other services with a procedure for obtaining user account information.

The next service provides users with instant messaging (IM) functionality. On the backend, it is a relatively simple service that keeps only soft state to store tables consisting of active users, active conversations, and pending messages. It defines handlers for setting/retrieving user status information, initiating conversations, sending/receiving messages, and retrieving an HTML user interface (described below).

The third service supports the others by supplying users with an address book for recording contacts. The address book service is designed to provide functionality similar to that of Microsoft's Address Book Clearing House (ABCH) service [41]. The address book keeps limited persistent state in the form of tables that record contact entries and contact groups. The address book service de-

fines handlers for creating, retrieving, manipulating, and deleting contacts and contact groups.

The final service adds support for offline messaging that is similar to email. The service consists of two persistent tables for storing message contents and message deliveries. It defines handlers for sending, retrieving, and deleting messages.

Users interact with the services though a Web-based interface that is provided by the instant messaging service. A user starts this Web-based interface by executing the IM service's `GetInterface` handler, which responds with the Web application user interface, implemented in HTML and JavaScript code. Initially, the interface requires the user to connect to the authentication service to obtain account credentials. After successfully authenticating, the interface uses the obtained credentials to retrieve the user's contacts from the address book service. The contacts are presented as a "buddy list", from which the user can select message recipients. User actions performed on the web application can send requests to the instant messaging service, the offline messaging service, or the address book service. In the background, the web interface automatically polls the IM service and offline messaging service for new conversations or messages and notifies the user.

6.1.3 *Workload Generator for* FLIMP *Services*

To properly exercise these test FLIMP services we have developed, we implemented a *workload generation engine*. The generator drives execution of our services by simulating a specified number of concurrent client sessions. A simulated client periodically chooses and executes a handler chosen from one of three services: InstantMessenger, AddressBook, and Mail. An invariant we preserve is that a client maintains at most one outstanding request for each service. As part of our experimental setup, every request's end-to-end execution latency is measured. Upon completing a request, a client will allow a short "cooldown" time to pass before executing the next request to that service.

The choice of which action a client should perform next for a given service is determined by the client's current state and the simulator's *workload mix* distribution. The client state is used to determine the set of actions that are currently available, which prevents a client from choosing to execute an impossible action such as removing a contact when its contact list is currently empty.

The workload mix specifies the relative weights associated with each action. The generator reads its workload mix from a simple specification file. Once a client has verified its set of available actions, it generates a random value between zero and the sum of the available action weights. The random value is then mapped to its corresponding action, which is selected for the next request to be issued by the client.

We have based workload mixes we have used for experiments on real usage data from Windows Messenger and Hotmail. The workload produced by this specification contains a total of 92,379 requests calling one of 20 FLIMP handlers. This workload leads to a total of 12,8581 state operations. Of these 83,818 or 65% are reads, and the other 44,763 are writes.

6.2 Latency Optimization Experiments

User-perceived latency is an important metric in determining the success of an Internet service [7, 8, 36]. Several of FLUXO's automatically applied optimizations serve to improve the latency of user requests. This section demonstrates that FLUXO positively impacts user request latency by experimentally quantifying application latency reductions over a varying input workload. Overall, we see an order-of-magnitude decrease in latency for constant propagation, caching, pre- and post-computation optimizations.

6.2.1 *Constant Propagation*

We have applied constant propagation to almost 1,000 Pipes programs. We have discovered that for over 500 of these programs at least one node, typically a `FetchFeed`, can benefit from constant propagation. For some programs, this can be as many as 10 to 15 nodes. Figure 13 shows the outcome of applying constant propagation to several representative Yahoo! Pipes programs on end-to-end program latency. For this experiment, we have chosen several Pipes programs that require no user input that needs to be typed in and can benefit from the constant propagation optimization. The number of nodes that constant propagation applies to ranges from 1–4, as shown in column 2. In all cases, these programs relied on either a `FetchFeed` or a `FetchSite` node to fetch data from an external server such as a blog or a news feed. Because these sites often do not change very rapidly, applying constant propagation has obvious benefits. To simulate multiple users using these programs, we run each program a total of 20 times. In these experiments, the value computed for the optimized subgraph is saved away and is refreshed at a rate of every 2 minutes.

Columns 4–6 of figure Figure 13 show the average latency before and after constant propagation, as well as the reduction in latency. The reduction is latency is quite significant, exceeding 90% for 3 out of 5 Pipes programs. A 93% reduction for the "Parenting 24/7" Pipe means savings as significant about 7 seconds on average. As it turns out, this particular pipe fetches data on parenting-related topics from 10 different news and medical information sites, resulting in both high overall latency and also high latency variance.

6.2.2 *Caching*

Figure 14 displays the results of caching transformations applied to selected handlers from the FLIMP example services described in Section 6.1.2. The input to this optimization is a caching policy specification that identifies the subgraphs around which a cache needs to be inserted, the cache size, the eviction policy, and a staleness parameter indicating the maximum time entries stay in the cache before being evicted. The caching policy can either be provided manually or can be generated by a prior automatic analysis.

For this experiment, we identified a favorable caching policy as follows. First, we ran with a default caching policy that inserts a cache around all LINQ blocks that access the backend database. This heuristic is justified by the fact that the access to the database is the latency bottleneck in our experiments. Using the cache-hit rate statistics and the resulting latency improvements obtained during this run, we generated a caching policy that inserts caches only at those LINQ blocks that result in a net-positive latency improvements.

The table in Figure 14 shows the end-to-end server-side latency improvements achieved with this caching policy over the base policy that adds no caches. The table shows both the mean and the 95th percentile latency observed over four independent runs of the workload described in Section 6.1.3. The simple cache-policy described above provides up to 50% latency savings on some handlers, while the overhead of caching reduces the latency by 8% on some other handlers.

6.2.3 *Post-computation*

Figure 14 also describes the latency improvements obtained by the post-computation optimization. For this optimization, we automatically inferred computation that can be deferred based on the technique described in Section 5.2.3. We performed a two-step experiment as in the caching experiments above. In the first run, we performed the optimization on all the handlers indicated by our automatic analysis. In subsequent runs, we applied the transformation

	Constprop node statistics		Average latency			Latency StdDev			Latency 95 Percentile		
Name of the pipe	**#**	**Node types**	**Before**	**After**	**Savings**	**Before**	**After**	**Increase**	**Before**	**After**	**Savings**
The Joy of Tech	3	FetchFeeed, Rename, Regex	81	78	4%	200	198	-1%	94	78	17%
Metafilter Current Posts	4	FetchFeeed, Sort, Rename, Regex	4,811	1,642	66%	431	1,286	198%	5,684	1,642	71%
zeropunctuation feed	2	FetchFeed, Filter	676	61	91%	457	266	-42%	1,655	61	96%
Parenting 24/7	2	FetchSiteFeed, Filter	7,562	506	93%	418	1,787	327%	8,443	507	94%
Del.icio.us Popular	1	FetchFeed	3,904	337	91%	353	791	124%	4,103	337	92%

Figure 13: Effect of constant propagation on latency optimizations. Latency numbers are shown in *ms*.

	Base		Caching				Post-computation			
Service::Handler	**Mean**	**95%**	**Mean**	**Savings**	**95%**	**Savings**	**Mean**	**Savings**	**95%**	**Savings**
AddressBook::AddContact	107	279	72	32%	498	-79%	127	-19%	374	-34%
AddressBook::AddGroup	48	164	24	49%	264	-61%	55	-15%	231	-41%
AddressBook::GetContacts	51	271	41	20%	261	4%	51	0%	247	9%
AddressBook::GetGroups	36	128	37	-2%	262	-104%	37	-3%	233	-81%
AddressBook::RemoveContact	84	277	76	9%	403	-46%	52	38%	240	13%
AddressBook::RemoveGroup	78	241	74	5%	228	5%	69	12%	231	4%
AddressBook::UpdateContact	46	164	45	1%	249	-52%	40	13%	234	-43%
AddressBook::UpdateGroup	54	197	52	4%	288	-47%	50	7%	237	-21%
Auth::GetUserID	73	270	42	42%	276	-2%	77	-6%	320	-18%
Auth::RegisterAccount	9	7	9	0%	15	-112%	9	2%	6	14%
Auth::VerifyAccount	19	24	17	13%	36	-52%	15	22%	23	2%
InstantMessenger::Connect	5	11	5	0%	16	-49%	6	-29%	64	-482%
InstantMessenger::Disconnect	1	2	1	0%	7	-219%	1	5%	7	-223%
InstantMessenger::GetMessages	77	350	64	17%	286	18%	63	19%	285	19%
InstantMessenger::GetPresence	49	220	47	3%	261	-19%	39	21%	220	0%
InstantMessenger::GetUpdates	65	318	45	30%	261	18%	50	24%	257	19%
InstantMessenger::SendInvite	62	213	31	50%	322	-51%	65	-5%	233	-9%
InstantMessenger::SendMessage	108	379	73	33%	260	31%	65	40%	232	39%
InstantMessenger::SetPresence	76	263	76	0%	326	-24%	54	29%	245	7%
Mail::DeleteMessage	47	252	38	20%	211	16%	35	25%	174	31%
Mail::GetMessage	245	387	265	-8%	536	-39%	246	0%	438	-13%
Mail::GetMessageList	317	550	322	-2%	549	0%	331	-4%	545	1%
Mail::SendMessage	94	327	84	11%	283	14%	76	19%	284	13%

Figure 14: Savings with the caching and post-computation optimizations.

only on those handlers for which the optimization was beneficial in the first run. The table shows that post-computation provides as much as 40% improvement in latency.

6.3 Scalability

Another critical aspect of large systems is the way in which they scale when run on large, parallel clusters. As with latency, FLUXO's transformations automatically improve service scalability by replicating state and distributing execution across multiple machines. Figure 15 confirms that our application of the simple automated application of the shared-nothing pattern of tiering and replicating a service scales our test suite. In these experiments, we measure the number of requests that can be processed per second as the size of the execution cluster increases. The results show that scaling from a single-machine configuration to a 4-node configuration provides almost linear improvement at its peak performance.

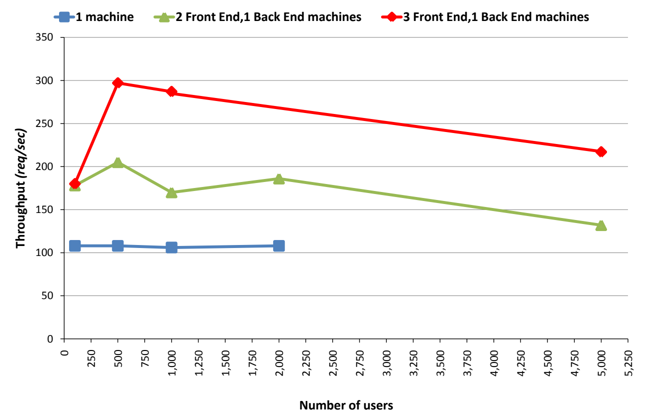

Figure 15: Throughput as a function of the number of users.

7. RELATED WORK

Previous work [24] has introduced FLUXO's high-level architecture and described several classes of optimizations. Many programming languages that are widely deployed in enterprise environments provide frameworks to simplify the development of scalable system architectures. Examples of such systems include J2EE [39], SCALA [35], and Ruby-on-rails [34], which provide APIs and infrastructure that facilitate distribution, communication, and management for large, Web-based services. These systems focus on component re-use rather than enabling the separation of functionality from architectural performance and scalability. Developers must still manually determine how to best use the provided infrastructure, which complicates application-level code and hampers attempts to re-achitecture the service in response to changing deployment conditions.

MapReduce [13], Dryad [21], and Hadoop [40] are systems that ease the development of large, data-intensive parallel computations. Similar to FLUXO, these systems automatically distribute and replicate data for parallel execution. However, they primarily focus on *bulk processing* tasks in which throughput, rather than latency, is the primary performance metric. In other words, end-users do not generally interact with such systems. In contrast, FLUXO

optimizations presented in this paper primarily focus on reducing the end-to-end latency of user requests.

MapReduce Online [11] modifies Hadoop to pipeline data between operators. The modifications allows users to receive "early returns" as Hadoop computes aggregate results. The system also supports continuous queries for monitoring and stream processing. The staged event driven architecture (SEDA) [43] separates application event processing from controllers that dynamically handle resource allocation decisions.

P2 [28] is a system in which developers specify overlay networks using a declarative language. Like FLUXO, P2 compiles its high-level language into an optimized dataflow graph for execution. P2 primarily differs from FLUXO by targeting overlays and by applying optimizations that more closely resemble database query optimizations. More recent efforts [4] focus on applying P2's declarative language to simplify the construction of a Hadoop-compatible MapReduce implementation.

The Hilda [46] project provides developers with a declarative high-level language for developing data-driven web applications. Hilda provides a compiler that translates Hilda programs into Java Servlet code. The Scalable Games Language (SGL) [44] proposes utilizing data management techniques to improve the AI in computer games. The language consists of SQL statements, `let`-statements, and conditionals, which are translated into relational algebra and optimized using standard database optimizations.

8. CONCLUSIONS

FLUXO is a system to enable non-expert developers to build performant and scalable distributed Internet services. FLUXO broadens Internet service development by allowing developers to focus on application functionality, with architectural issues being handled by profile-driven optimizers written by experts. At its core, FLUXO is an optimizing compiler that uses a restricted programming model and runtime profiling to create a logical separation between the core functionality of the service and its architectural patterns. This separation allows a wide class of programmers to build scalable and reliable web services.

To demonstrate the viability of separating architectural decisions from application logic, this paper presents four optimizations that we have applied to two classes of Internet services, existing third-party Yahoo! Pipes programs and a test suite of four realistic services. Our experiments show these application-agnostic optimizations reducing latency from 20–90% without requiring the developer's assistance or awareness. Similarly, FLUXO's application of one simple tiering and replication pattern is able to scale our test suite of applications.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of VLDB*, pages 496–505, 2000.

[2] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[3] J. Albahari and B. Albahari. *LINQ Pocket Reference*. O'Reilly Media, 2008.

[4] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. Hellerstein, and R. Sears. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of EuroSys*, 2010.

[5] Amazon. Amazon elastic compute cloud (EC2). `http://aws.amazon.com/ec2/`.

[6] R. Bekin and S. Dawson. LinkedIn communication architecture. Presentation at JavaOne, 2008.

[7] C. Bouras, A. Konidaris, and D. Kostoulas. Predictive prefetching on the web and its potential impact in the wide area. *World Wide Web*, 7(2):143–179, 2004.

[8] J. Brutlag. Speed matters for Google Web search. `http://code.google.com/speed/files/delayexp.pdf`, June 2009.

[9] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the Symposium on Compiler Construction*, pages 152–161, 1986.

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. WSDL: Web services description language. `http://www.w3.org/TR/wsdl`, Mar. 2001.

[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[12] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of OSDI*, 2004.

[14] Google. Google app engine. `http://code.google.com/appengine/`.

[15] Google. Google app engine (second look). `http://dumpstuffhere.blogspot.com/2008/07/google-app-engine.html`, 2008.

[16] R. Gupta, E. Mehofer, and Y. Zhang. Profile guided compiler optimizations. In *The Compiler Design Handbook*, pages 143–174, 2002.

[17] J. Hamilton. On designing and deploying internet-scale services. In *Proceedings of LISA*, pages 1–12, 2007.

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

[19] C. Henderson. *Building Scalable Web Sites: Building, scaling, and optimizing the next generation of web applications*. O'Reilly Media, Inc., 2006.

[20] C. Henderson. Scalable Web Architectures: Common Patterns and Approaches, September 2008.

[21] 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.

[22] James. Google app engine followup. `http://dumpstuffhere.blogspot.com/2008/07/google-app-engine-followup.html`, 2008.

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

[24] E. Kiciman, B. Livshits, and M. Musuvathi. Fluxo: A simple service compiler. In *Proceedings of HotOS*, 2009.

[25] G. Lapalme. Implementation of a "lisp comprehension" macro. *SIGPLAN Lisp Pointers*, IV(2):16–23, 1991.

[26] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[27] Livejournal. `http://www.slideshare.net/miyagawa/how-we-build-vox`, 2007.

[28] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of SOSP*, 2005.

[29] Microsoft. Azure services platform. `http://www.microsoft.com/azure/`.

[30] Microsoft. `http://www.popfly.com/`, 2008.

[31] W. A. Najjar, E. A. Lee, and G. R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907 – 1929, 1999.

[32] T. O'Reilly. Database war stories #3: Flickr. *O'Reilly Radar*, Apr 2006.

[33] A. Rasmussen, E. Kıcıman, B. Livshits, and M. Musuvathi. Short paper: Improving the responsiveness of interactive Internet services with automatic cache placement. In *Proceedings of EuroSys*, 2009.

[34] Ruby on Rails. `http://rubyonrails.org`, 2009.

[35] Scala. `http://www.scala-lang.org`, 2009.

[36] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in Web search. `http://en.oreilly.com/velocity2009/public/schedule/detail/8523`, May 2009.

[37] R. Slobojan. Dan Farino: About MySpaceŠs architecture. *InfoQ*, Nov 2008.

[38] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proceedings of NSDI*, pages 71–84, 2005.

[39] Sun Microsystems. Java enterprise edition (J2EE). `http://java.sun.com/javaee/`.

[40] The Hadoop Project. `http://hadoop.apache.org`, 2009.

[41] P. Thurrott. MSN: The inside story. `http://www.winsupersite.com/showcase/msn_inside_03.asp`, May 2005.

[42] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the International Conference on Supercomputing*, pages 414–423, 1995.

[43] 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.

[44] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *Proceedings of SIGMOD*, 2007.

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

[46] F. Yang, J. Shanmugasundaram, M. Riedewald, J. Gehrke, and A. Demers. Hilda: A high-level language for data-driven web applications. Technical report, TR2005-1991, Computer Science Department, Cornell University, 2005.