Flux: Composing efficient and scalable servers

BRENDAN BURNS
Union College
KEVIN GRIMALDI, ALEXANDER KOSTADINOV,
MATT MEEHAN, GABRIEL TARASUK-LEVIN,
EMERY D. BERGER, and MARK D. CORNER
University of Massachusetts Amherst

Programming high-performance server applications is challenging: it is both complicated and error-prone to write the concurrent code required to deliver high performance and scalability. Server performance bottlenecks are difficult to identify and correct. Finally, it is difficult to predict server performance prior to deployment.

This paper presents Flux, a language that dramatically simplifies the construction of scalable high-performance server applications. Flux lets programmers compose off-the-shelf, sequential C, C++, or Java functions into concurrent servers. The Flux compiler type-checks programs and guarantees that they are deadlock-free. We have built a number of servers in Flux, including a web server with PHP support, an image-rendering server, a BitTorrent peer, and a game server. These Flux servers perform comparably to their counterparts written entirely in C. By tracking hot paths through a running server, Flux simplifies the identification of performance bottlenecks. The Flux compiler also automatically generates discrete event simulators that accurately predict actual server performance under load and with different hardware resources.

Categories and Subject Descriptors: D.3.0 [**Programming Languages**]: General—*Language Design*

General Terms: Languages, Design, Performance

Additional Key Words and Phrases: flow-based, runtime, threads, events, coordination, language

1. INTRODUCTION

Server applications need to provide high performance while handling large numbers of simultaneous requests. Concurrency is required for high performance but introduces errors like race conditions and deadlock that are difficult to debug. The mingling of server logic with low-level systems programming complicates development and makes it difficult to understand and debug server applications. Consequently, the resulting implementations are often either lacking in performance, buggy or both. At the same time, the interleaving of multiple threads of server logic makes

Authors' addresses:
Department of Computer Science, Union College, 807 Union Street, Schenectady, NY, 12305
Department of Computer Science, University of Massachusetts, 140 Governor's Drive, Amherst, MA 01003

```
source Listen ⇒ Image;
Image = ReadRequest → CheckCache
          → Handler
          → Write → Complete;
Handler [_, _, hit] = ;
Handler [_, _, _] =
          ReadInFromDisk
          → Compress
          → StoreInCache;
```

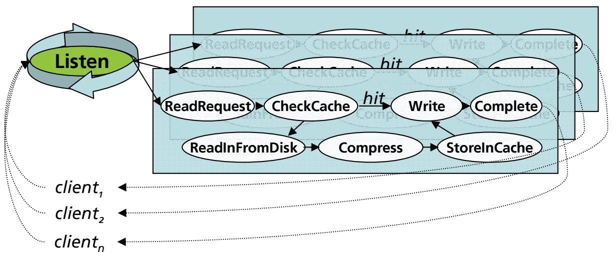


Fig. 1. An example Flux program and a dynamic view of its execution.

it difficult to identify performance bottlenecks or predict server performance prior to deployment. Programming servers is a daunting task.

This paper presents *Flux*, a domain-specific language that addresses these problems in a declarative, flow-oriented language (Flux stems from the Latin word for "flow"). A Flux program describes two things: (1) the flow of data from client requests through nodes, typically off-the-shelf C, C++, or Java functions, and (2) mutual exclusion requirements for these nodes, expressed as high-level *atomicity constraints*. Flux requires no other typical programming language constructs like variables or loops – a Flux program executes inside an implicit infinite loop. The Flux compiler combines C, C++, or Java components into a high performance server using just the flow connectivity and atomicity constraints.

Flux captures a programming pattern common to server applications: concurrent executions, each based on a client request from the network and a subsequent response. This focus enables numerous advantages over conventional server programming:

—**Ease of use.** Flux is a declarative, implicitly-parallel coordination language that eliminates the error-prone management of concurrency via threads or locks. A typical Flux server requires just tens of lines of code to combine off-the-shelf components written in sequential C or C++ into a server application.

—**Reuse.** By design, Flux directly supports the incorporation of unmodified existing code. There is no "Flux API" that a component must adhere to; as long as components follow the standard UNIX conventions, they can be incorporated unchanged.

—**Runtime independence.** Because Flux is not tied to any particular runtime model, it is possible to deploy Flux programs on a wide variety of runtime systems. Section 3 describes multiple runtimes we have implemented: thread-based, thread pool, event-driven, and a hybrid system. We have made numerous improvements to these runtime systems over time with no modifications to the programs themselves.

—**Correctness.** Flux programs are type-checked to ensure their compositions make sense. The atomicity constraints eliminate deadlock by enforcing a canonical ordering for lock acquisitions.

—**Performance prediction.** The Flux compiler can automatically generate a discrete event simulator for Flux programs. As we show in Section 5, this simulator accurately predicts actual server performance.

—**Bottleneck analysis.** Flux servers include lightweight instrumentation that identifies the most-frequently executed or most expensive paths in a running Flux application.

Our experience with Flux has been positive. We have implemented a wide range of server applications in Flux: a web server with PHP support, a BitTorrent peer, an image server, and a "heartbeat" client-server game server. The longest of these consists of approximately 100 lines of Flux code and all perform comparably to their hand-written counterparts.

The remainder of this paper is organized as follows. Section 2 presents the semantics and syntax of the Flux language. Section 3 describes the Flux compiler and runtime systems. Section 4 presents our experimental methodology and compares the performance of Flux servers to their hand-written counterparts. Section 5 demonstrates the use of path profiling and the accuracy of the discrete-event simulator. Section 6 reports our experience using Flux to build several servers. Section 7 presents related work.

2. LANGUAGE DESCRIPTION

To introduce Flux, we use an example application that exercises most of Flux's features. This application is an image server that receives HTTP requests for images that are stored in the PPM format and compresses them into JPEGs, using calls to an off-the-shelf JPEG library. Recently-compressed images are stored in a cache managed with a least-frequently used (LFU) replacement policy.

Figure 1 shows a schematic view of the server execution's flow through the image server, separated into distinct nodes. Each data flow begins with Listen, accepting client connections. Every new connection is a new data flow. The flow proceeds to ReadRequest, where the client's request is parsed and subsequently checked against a cache. If there is a cache hit, the flow can proceed through writing the result to the client and completing the connection. If there is a cache miss the image must be loaded from disk, compressed, and stored in the cache. After this, the result can be written and the connection closed.

Figure 2 shows the Flux code for the image server, which will be referenced in describing the Flux language.

The `Listen` node, declared as a "source node" on Line 2, serves as the start of a flow through the Flux program. Source nodes are executed in an infinite loop, providing the server with an unbounded number of separate concurrent flows.

Flux programs are acyclic. The only loops exposed in Flux are the loops in which source nodes execute. The lack of cycles in Flux allows it to enforce deadlock-free concurrency control. While theoretically limiting expressiveness, we have found cycles to be unnecessary for the Flux implementation of the wide range of servers described in Section 4.

The Flux language consists of a minimal set of features, including **concrete nodes** that correspond to the C or C++ code implementing the server logic, **abstract nodes** that represent a flow through multiple nodes, **predicate types** that implement conditional data flow, **error handlers** that deal with exceptional conditions, and **atomicity constraints** that control simultaneous access to shared state.

```
 1 // source node
 2 source Listen => Image;
 3
 4 // concrete node signatures
 5 Listen () => (int id);
 6
 7 ReadRequest (int id) => (int id, image_tag *request);
 8
 9 CheckCache (int id, image_tag *request)
10         => (int id, image_tag *request);
11
12 Complete (int id, image_tag *request) => ();
13
14 Write (int id, image_tag *request)
15         => (int id, image_tag *request);
16
17 // omitted for space: Compress, StoreInCache, ReadInFromDisk
18
19 // abstract node
20 Image = ReadRequest -> CheckCache -> Handler
21         -> Write -> Complete;
22
23 // predicate type and node
24 predicate inCache TestInCache;
25
26 Handler:[_, inCache]=;
27 Handler:[_, _]=ReadInFromDisk -> Compress -> StoreInCache;
28
29 //atomicity constraints
30 atomic CheckCache:{cache};
31 atomic StoreInCache:{cache};
32 atomic Complete:{cache};
33
34 //error handler
35 handle error ReadInFromDisk => FourOhFour;
```

Fig. 2. An image compression server, written in Flux.

2.1 Concrete Nodes

The first step in designing a Flux program is describing the *concrete nodes* that correspond to C and C++ functions. Flux requires type signatures for each node. A signature is made up of the concrete node's name, its inputs, and its outputs. If a node takes no inputs or outputs, they may be left blank. For example, `ReadRequest`, on Line 7, takes an integer input and outputs an integer and an image tag. Flux uses this information to create the input and output structures used to pass data between nodes at runtime.

While most nodes receive input and produce output, there are two types of nodes that do not follow this convention. *Source* nodes are a special type of node, serving to initiate data flows. As `Listen` is a source node, it is declared on Line 5 to take no input, yet produce output. Similarly, `Complete`, on Line 12, is a *sink* node. It exists at the end of the program flow, accepting inputs without producing outputs.

2.2 Abstract Nodes

A programmer can compose a series of concrete nodes into an *abstract nodes.* These abstract nodes represent a flow of data from concrete nodes to concrete nodes, from concrete to abstract nodes, or abstract nodes to abstract nodes. For instance, `Image` is declared as an abstract node on Line 21 to be a data flow starting at `ReadRequest`, moving to `CheckCache`, then `Handler`, `Write`, and ending with `Complete`. The execution path shows that the output of `ReadRequest` will be the input of `CheckCache`, allowing Flux to confirm that the argument signatures of each node composing an abstract node are correct. Similarly, the inputs and outputs of abstract nodes are dictated by the flow path, giving Flux the ability to implicitly determine an abstract node's type signature if it is not explicitly declared.

2.3 Predicate Types

A client request for an image may result in either a cache hit or a cache miss. Flux incorporates functionality for handling these two cases differently. Instead of exposing control flow directly, Flux lets programmers use the *predicate type* of a node's output to direct the flow of data to the appropriate subsequent node. A predicate type is an arbitrary Boolean function supplied by the Flux programmer that is applied to the node's output.

Using predicate types, a Flux programmer can express multiple possible paths for data through the server. The `predicate` statement on Line 24 binds the type `inCache` to the Boolean function `TestInCache`.

A predicate node is a special abstract node. Its data flow is only used if a series of tests against its outputs are passed. If any of the tests fail, the next case appearing in the program is attempted. Declaring a predicate node takes a name, a test against each of the node's outputs, and a resulting flow path. Underscores are used where no tests are needed. The node `Handler`, on Line 26, checks its second output parameter with the `inCache` predicate type. `inCache`, in turn, uses the `TestInCache` Boolean function. The `TestInCache` function will be applied to the node's second output parameter. If `TestInCache` returns true, then all tests have passed, and `Handler` corresponds to an empty data flow path. Alternatively, if `TestInCache` returns false, then the next case for `Handler` is checked. The second case on Line 27 has no checks at all, serving as a default case that will always be followed if reached. Thus, if `TestInCache` is false, the data flow through `Handler` will be diverted through `ReadInFromDisk`, `Compress`, and finally to `StoreInCache` before resuming its previous flow.

2.4 Error Handling

The dataflow described is appropriate for handling normal web requests. In the event of an invalid page request, however, an error condition outside of the normal flow must be handled. Flux incorporates functionality for handling these errors using *error handlers.*

Flux expects nodes to follow the standard UNIX convention of returning error codes. Whenever a node returns a non-zero value, Flux checks if an error handler has been declared for the node. If none exists, the current data flow is terminated.

An error handler is specified on Line 35. This declaration states that in the event of an error within the `ReadInFromDisk` node, the `FourOhFour` error handler node should be called. In such an event, the `FourOhFour` handler will be passed input from `ReadInFromDisk`, along with the error code. Upon the completion of `FourOhFour`, the data flow terminates.

2.5 Atomicity Constraints

All flows through the image server access a single shared image cache. Access to this shared resource must be controlled to ensure that two different data flows do not interfere with one another.

The Flux programmer specifies such *atomicity constraints* in Flux rather than inside the component implementation. The programmer specifies atomicity constraints by using arbitrary symbolic names. These constraints can be thought of as locks, although this is not necessarily how they are implemented. A node only runs when it has "acquired" all of the constraints. This acquisition follows a two-phase locking protocol: the node acquires ("locks") all of the constraints in order, executes the node, and then releases them in reverse order.

Atomicity constraints can be specified as either *readers* or *writers*. Using these constraints allows multiple readers to execute a node at the same time, supporting greater efficiency when most nodes read shared data rather than update it. Reader constraints have a question mark appended to them ("?"). Although constraints are considered writers by default, a programmer can append an exclamation point ("!") for added documentation.

In the image server, the image compression cache can be updated by three nodes: `CheckCache`, which increments a reference count to the cached item, `StoreInCache`, which writes a new item into the cache, evicting the least-frequently used item with a zero reference count, and `Complete`, which decrements the cached image's reference count. Only one instance of each node may safely execute at a time; since all of them modify the cache, we label them with the same writer constraint (`cache`), on Line 30.

Note that programmers can apply atomicity constraints not only to concrete nodes but also to abstract nodes. In this way, programmers can specify that multiple nodes must be executed atomically. For example, the node `Handler` could also be annotated with an atomicity constraint, which would span the execution of the path `ReadInFromDisk` → `Compress` → `StoreInCache`. This freedom to apply atomicity constraints presents some complications for deadlock-free lock assignment, which are discussed in Section 3.1.1.

2.5.1 *Scoped Constraints.*

While flows generally represent independent clients, in some server applications, multiple flows may constitute a single *session*. For example, a file transfer to one client may take the form of multiple simultaneous flows. In this case, the state of the session (such as the status of transferred chunks) only needs to be protected from concurrent access in that session.

In addition to *program-wide constraints* that apply across the entire server (the default), Flux supports *per-session constraints*. Using session-scoped atomicity constraints increases concurrency by eliminating contention across sessions. Sessions are implemented as hash functions on the output of each source node. The Flux programmer implements a session id function that takes the source node's output as its parameter and returns a unique session identifier, and then adds `(session)` to a constraint name to indicate that it applies only per-session.

2.5.2 *Discussion.* Specifying atomicity constraints in Flux rather than placing locking operations inside implementation code has a number of advantages, beyond the fact that it allows the use of libraries whose source code is unavailable.

Safety. The Flux compiler imposes a canonical ordering on atomicity constraints (see Section 3.1.1). Combined with the fact that Flux flows are acyclic, this ordering prevents cycles from appearing in its lock graph. Programs that use Flux-level atomicity constraints exclusively (i.e., that do not themselves contain locking operations) are thus guaranteed to be deadlock-free.

Efficiency. Exposing atomicity constraints also enables the Flux compiler to generate more efficient code for particular environments. For example, while a multithreaded runtime requires the use of locks, a single-threaded event-driven runtime does not. The Flux compiler generates locks or other mutual exclusion operations only when needed.

Simulation. Expressing atomicity constraints in the Flux language, along with predicate types and error handlers, provides the Flux compiler with complete knowledge of the possible execution paths through a program. Using this information, the Flux compiler automatically generates an accurate discrete event simulator, as discussed in Section 5.1.

Granularity selection. Atomicity constraints let programmers easily choose the appropriate granularity of locking — they can apply fine-grained constraints to individual concrete nodes or coarse-grained constraints to abstract nodes that comprise many concrete nodes—the smallest granularity is at the concrete node level. However, even when deadlock-freedom is guaranteed, grain selection can be difficult: too coarse a grain results in contention, while too fine a grain can impose excessive locking overhead. The simulator provides a valuable tool for the programmer to experiment with lock granularity and find concurrency related performance bottlenecks.

3. COMPILER AND RUNTIME SYSTEMS

A Flux program is transformed into a working server by a multi-stage process. The compiler first reads in the Flux source and constructs a representation of the program graph. It then processes the internal representation to type-check the program. Once the code has been verified, the runtime code generator processes the graph and outputs C code that implements the server's data flow for a specific runtime. Finally, this code is linked with the implementation of the server logic into an operational server. We first describe the compilation process in detail. We then describe the runtime systems that Flux currently supports.

3.1 The Flux Compiler

The Flux compiler is a three-pass compiler implemented in Java, and uses the JLex lexer [Berk and Ananian 2007] in conjunction with the CUP LALR parser generator [Appel et al. 2005].

The first pass parses the Flux program text and builds a graph-based internal representation. During this pass, the compiler links nodes referenced in the program's data flows. All of the conditional flows are merged, with an edge corresponding to each conditional flow.

The second pass decorates edges with types, connects error handlers to their respective nodes, and verifies that the program is correct. First, each node mentioned in a data flow is labelled with its input and output types. Each predicate type used by a conditional node is associated with its user-supplied predicate function. Finally, the error handlers and atomicity constraints are attached to each node. If any of the referenced nodes or predicate types are undefined, the compiler signals an error and exits. Otherwise, the program graph is completely instantiated. The final step of program graph construction checks that the output types of each node match the inputs of the nodes that they are connected to. If all type tests pass, then the compiler has a valid program graph.

The third pass generates the intermediate code that implements the data flow of the server. Flux supports generating code for arbitrary runtime systems. The compiler defines an object-oriented interface for code generation. New runtimes can easily be plugged into the Flux compiler by implementing this code generator interface.

The current Flux compiler supports several different runtimes, described below. In addition to the runtime-specific intermediate code, the Flux compiler generates a Makefile and stubs for all of the functions that provide the server logic. These stubs ensure that the programmer uses the appropriate signatures for these methods. When appropriate, the code generator outputs locks corresponding to the atomicity constraints.

3.1.1 *Avoiding Deadlock.* The Flux compiler generates locks in a canonical order. Our current implementation sorts them alphabetically by name. In other words, a node that has `y,x` as its atomicity constraints actually first acquires `x`, then `y`.

When applied only to concrete nodes, this approach combines with Flux's acyclic graphs to eliminate deadlock. However, when abstract nodes also require constraints, the constraints may become nested and preventing deadlock becomes more complicated. Nesting could itself cause deadlock by acquiring constraints in non-canonical order. Consider the following Flux program fragment:

```
A = B;
C = D;

atomic A: {x};
atomic B: {y};
atomic C: {y};
atomic D: {x};
```

In this example, a flow passing through `A` acquires lock x and then invokes node `B` which in turn acquires lock y. However, a flow through `C` acquires lock y and then invokes `D` which acquires lock x, resulting in a cycle in the locking graph.

To prevent deadlock, the Flux compiler detects such situations and moves up the atomicity constraints in the program, forcing earlier lock acquisition. For each abstract node with atomicity constraints, the Flux compiler computes a constraint list comprising the atomicity constraints the node transitively requires, in execution order. This list can easily be computed via a depth-first traversal of the relevant part of the program graph. If a constraint list is out of order, then the first constraint acquired in a non-canonical order is added to the parent of the node that requires the constraint. This process repeats until no out-of-order constraint lists remain.

For the above example, Flux will discover that node `C` has an out-of-order sequence (`y`, `x`). It then adds constraint `x` to node `C`. The algorithm then terminates with the following set of constraints:

```
atomic A: {x};
atomic B: {y};
atomic C: {x,y};
atomic D: {x};
```

Flux locks are reentrant, so multiple lock acquisitions do not present any problems. However, reader and writer locks require special treatment. After computing all constraint lists, the compiler performs a second pass to find any instances when a lock is acquired at least once as a reader and a writer. If it finds such a case, Flux changes the first acquisition of the lock to a writer if it is not one already. Reacquiring a constraint as a reader while possessing it as a writer is allowed because it does not cause the flow to give up the writer lock.

Because early lock acquisition can reduce concurrency, whenever the Flux compiler discovers and resolves potential deadlocks as described above, it generates a warning message.

3.2 Runtime Systems

The current Flux compiler supports several different runtime systems including the following: one thread per connection, a thread-pool system, an event-driven runtime, and a hybrid of the threadpool and event-driven runtimes. We provide a performance comparison of these in Section 4.

3.2.1 *Thread-based Runtimes.* In the thread-based runtimes, each request handled by the server is dispatched to a thread function that handles all possible paths through the server's data flows. In the one-to-one thread server, a thread is created for every different data flow. In the thread-pool runtime, a fixed number of threads are allocated to service data flows. If all threads are occupied when a new data flow is created, the data flow is queued and handled in FIFO order.

3.2.2 *Event-driven Runtime.* Event-driven systems can provide robust performance with lower space overhead than thread-based systems [Welsh et al. 2001]. An event-driven system allows fine-grain control over event interleaving without any overhead due to context switching. In the event-driven runtime, every input to a functional node is treated as an event. Each event is placed into a queue and handled in turn by a single thread. Additionally, each source node (a node with no input) is repeatedly added to the queue to originate each new data flow. The transformation of input to output by a node generates a new event corresponding to the output data propagated to the subsequent node.

The implementation of the event-based runtime is complicated by the fact that node implementations may perform blocking function calls. If blocking function calls like `read` and `write` were allowed to run unmodified, the operation of the entire server would block until the function returned.

Instead, the event-based runtime intercepts all calls to blocking functions using a handler that is pre-loaded via the `LD_PRELOAD` environment variable. This handler captures the state of the node at the blocking call and moves to the next event in the queue. The formerly-blocking call is then executed asynchronously. When the event-based runtime receives a signal that the call has completed, the event is re-activated and re-queued for completion. Normally, this approach would necessitate the use of a Linux kernel supporting full callback-driven asynchronous I/O. Due to the lack of availability of a complete kernel solution, and in order to avoid being tied to a specific kernel variant, the current Flux event-based runtime instead uses a separate thread to simulate callbacks for asynchronous I/O using the `select` function. A programmer is thus free to use synchronous I/O primitives without interfering with the operation of the event-based runtime.

3.3 Hybrid Runtime

Dispatching a thread to handle each request, even from a thread pool, can introduce overhead. As the number of requests increase, the contention for processor time increases. Alternatively, handling requests with an event-based system can result in new requests being queued up behind pending events, increasing overall latency.

In an effort to address these issues, we developed a hybrid runtime. This multi-threaded event-driven (MTED) runtime uses an event pool and a small number of event-processing threads. As in the event-based runtime, each node is treated as a discrete event. Source nodes can be handled with dedicated threads, as in the threaded runtimes, or with repeated event-generation, as in the event-based runtime.

The hybrid runtime allows Flux the benefit of handling requests simultaneously with threads in addition to the performance improvement inherent in an event-driven runtime.

3.3.1 *Other Languages and Runtimes.* We implemented each of these runtimes in C using POSIX threads and locks. Flux can also generate code for different programming languages. We have implemented a prototype that targets Java, using both SEDA [Welsh et al. 2001] and a custom runtime implementation. The performance and evaluation of a webserver based on the latter can be seen in Section 4.2.

In addition to these runtime systems, we have implemented a code generator that transforms a Flux program graph into code for the discrete event simulator CSIM [Mesquite Software 2007]. This simulator can predict the performance of the server under varying conditions, even prior to the implementation of the core server logic. Section 5.1 describes this process in greater detail.

4. EXPERIMENTAL EVALUATION

To demonstrate Flux's effectiveness for building high-performance server applications, we implemented a number of servers. We summarize these in Table I. Most

Server	**Style**	**Description**	**Flux Lines**	**C/C++ Lines**
Web	request-response	a basic HTTP/1.1 server with PHP	36	386
Image	request-response	image compression server	23	551
BitTorrent	peer-to-peer	a file-sharing server	84	878
Game	heartbeat client-server	multiplayer “Tag” game	54	257

Table I. Lines of code to implement servers using Flux versus C/C++, described in Section 4. C/C++ line counts do not include associated libraries.

server applications can be broadly classified into one of the following categories, based on how they interact with clients: request-response client/server, “heartbeat” client/server, and peer-to-peer. We chose these servers specifically to span the space of possible server applications.

We implemented a server in Flux for each of these categories and compared its performance under load with existing hand-tuned server applications written in conventional programming languages. The Flux servers rely on single-threaded C and C++ code that we either borrowed from existing implementations or wrote ourselves. The most significant inclusions of existing code were in the web server, which uses the PHP interpreter, and in the image server, which relies on calls to the `libjpeg` library to compress JPEG images.

4.1 Methodology

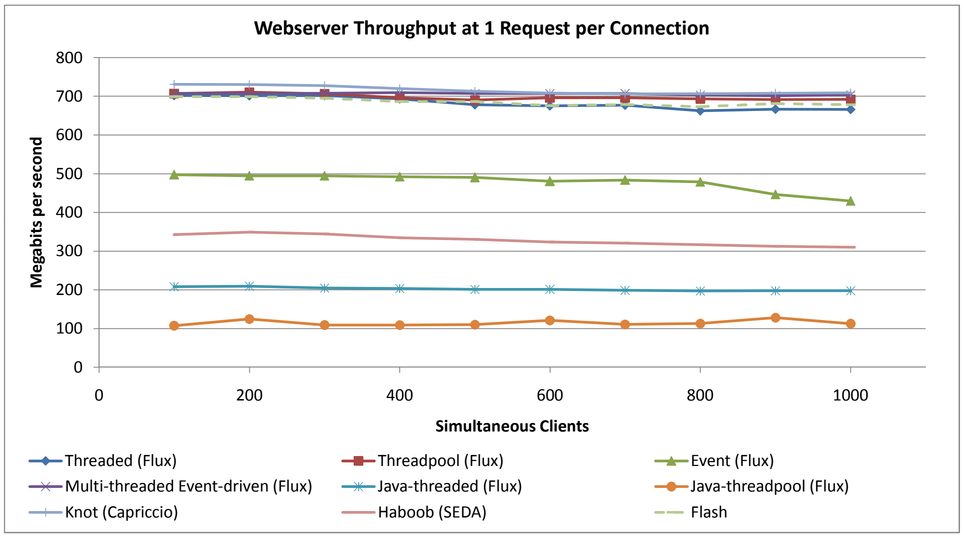
We evaluate all server applications by measuring their throughput and latency in response to realistic workloads under various conditions.

All testing was performed with identical server and client machines, both running Linux version 2.6.15 with Pentium 4 processors (3.0 GHz) and 1GB of RAM each. The machines were connected via a dedicated gigabit Ethernet switch and the client/server distinction was fixed but arbitrary. All server and client applications were compiled using g++ 4.0.3. During testing, both machines were running in multi-user mode with only standard services running.

4.2 Request-Response: Web Server

Request-response based client/server applications are among the most common examples of network servers. This style of server includes most major Internet protocols including FTP, SMTP, POP, IMAP and HTTP. As an example of this application class, we implemented a web server in Flux. The Flux web server implements the HTTP/1.1 protocol and can serve both static and dynamic PHP web pages.

Because webserver performance can be directly affected by user request patterns, we designed and implemented two tests for our webserver. Both tests simulate a varying number of clients requesting files from the server using HTTP/1.1 TCP connections. The tests differ primarily in the sequence of requests and the files requests. The first test pattern is similar to SPECWeb99 [Standard Performance Evaluation Corporation 2007] and requests for files follows the Zipf distribution.



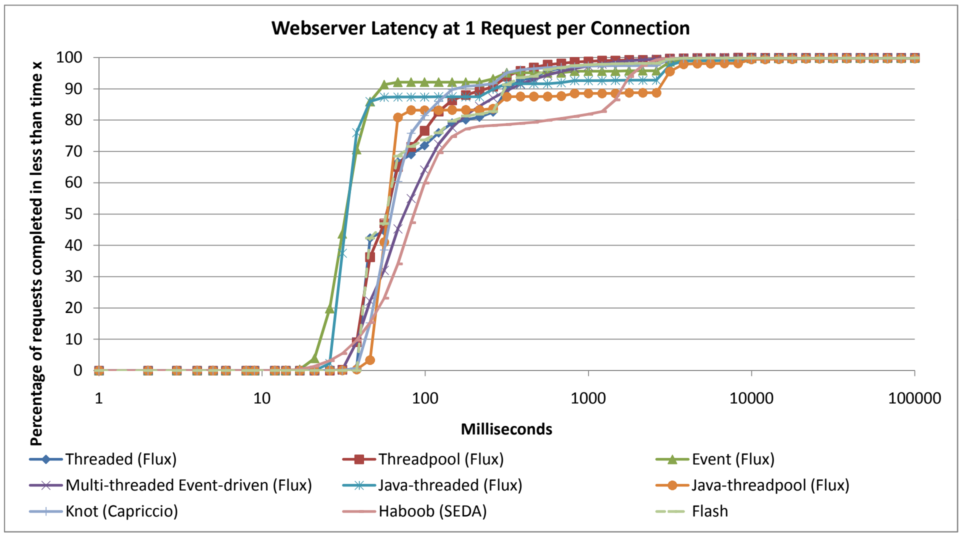
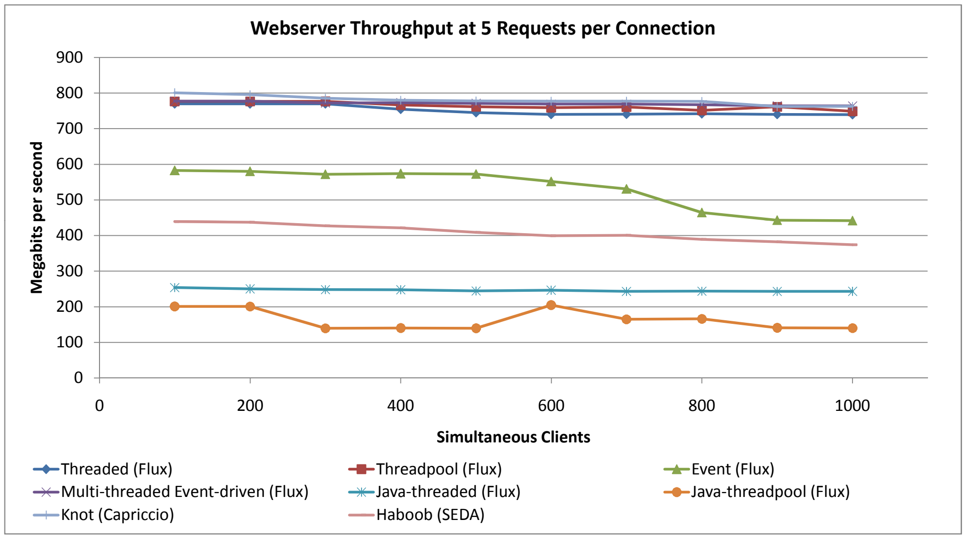


Fig. 3. Comparison of web servers at 1 request per connection (see Section 4.2). Most of the servers achieve the same level of throughput, except the event, SEDA, and Java-based runtime systems, which have substantially lower throughput. The delay of most servers is comparable, with the Java-threaded and event-based Flux servers performing slightly better. The SEDA server has the worst delay characteristics.

The files follow the static portion of the SPECweb benchmark, and their total size is about 32 MB. Because the entire working set of files fits easily into physical memory, this test stresses the host's CPU and RAM.

The second test pattern simulates a webserver with a large enough set of files to prevent effective caching. The files are organized into the same directory structure



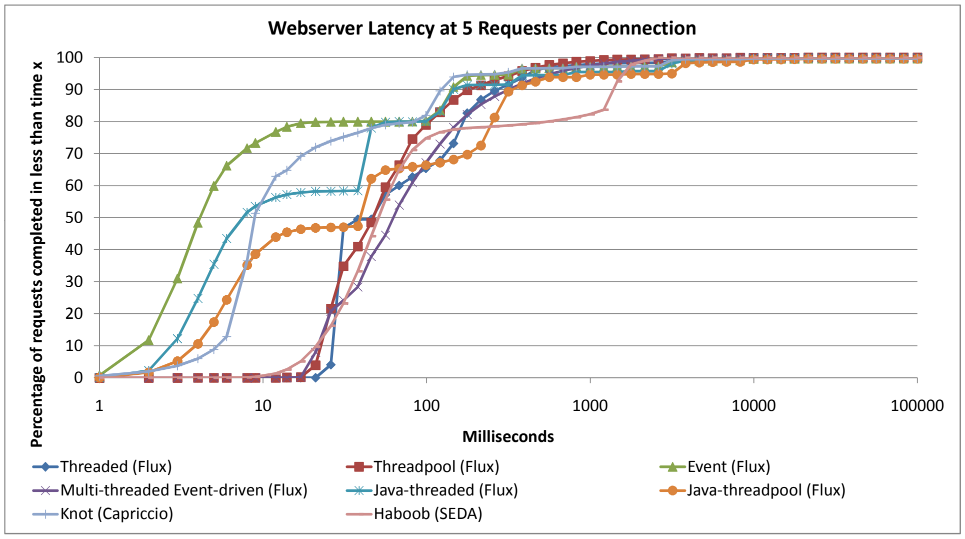
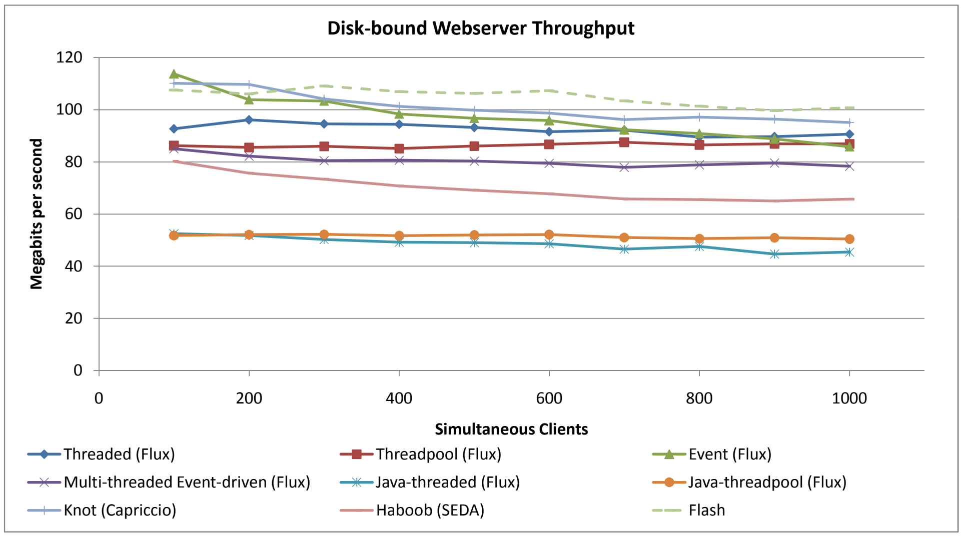


Fig. 4. Comparison of web servers at 5 requests per connection (see Section 4.2). The throughput characteristics are similar to the one-request-per-connection experiment, with most performing equally, except the event, SEDA, and Java-runtime systems. However in the case of delay, the event-based Flux server, performs substantially better than other servers.

as before, but each file is exactly 45 kilobytes in size to reduce performance variance. This brings the total working-set size to approximately 1.1 GB (which will not fit in our test machine's physical memory). To further impair caching, a particular file request is never repeated until each other file has been requested. Because this test severely inhibits the caching of the files, this benchmark primarily stresses disk performance.



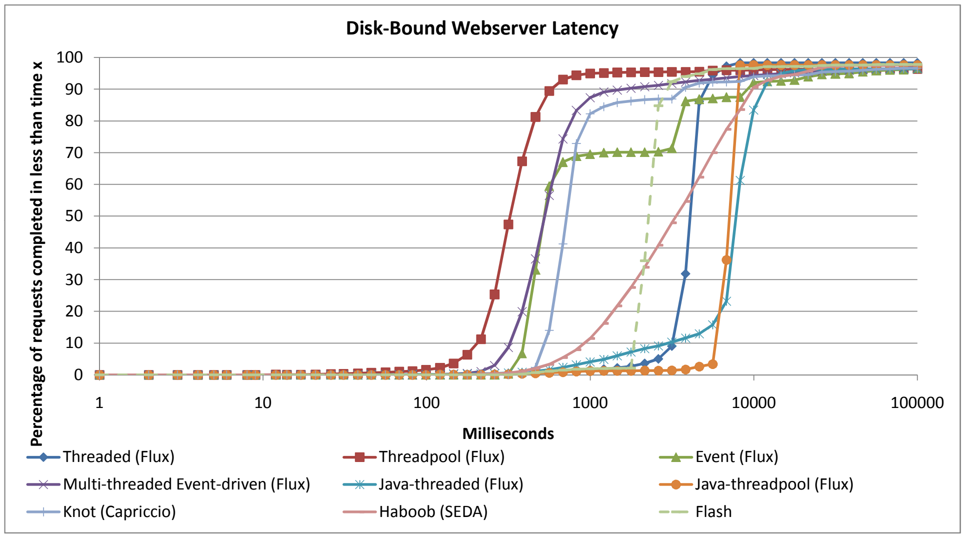
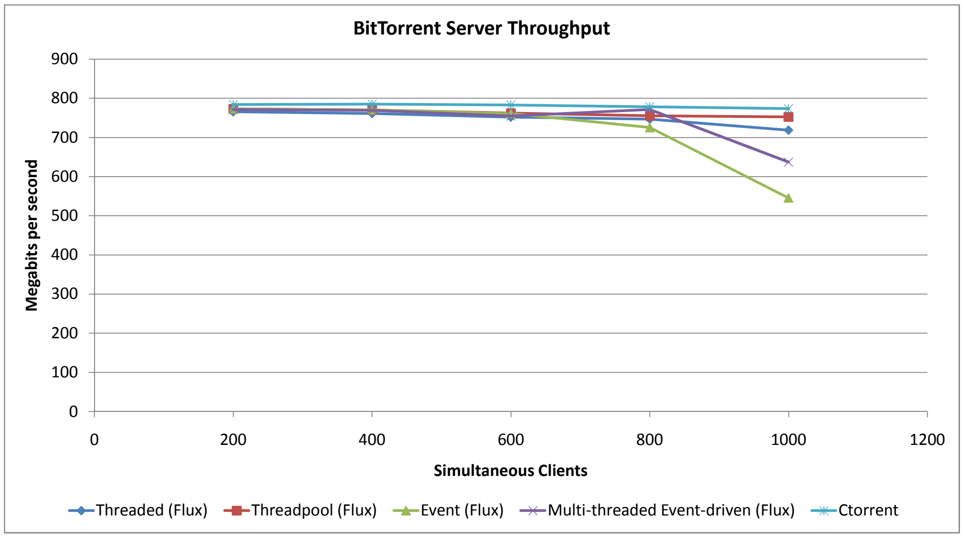


Fig. 5. Comparison of disk-bound web servers (see Section 4.2). In this case, the thread-based runtimes generally show slightly poorer throughput than the event-based runtime, but improved latency.

We compare the performance of the Flux webserver, using varying runtime systems, against the latest versions of the *knot* webserver distributed with Capriccio [von Behren et al. 2003], the *Haboob* webserver distributed with the SEDA runtime system [Welsh et al. 2001], and the *Flash* webserver [Pai et al. 1999]. Because *Flash* does not support HTTP/1.1 style keep-alives, we perform the tests both with and without them.

In the first test using the SPECWeb99 pattern, the simulated clients each connect to the server, issue a single request, disconnect, and then pause for 20 milliseconds



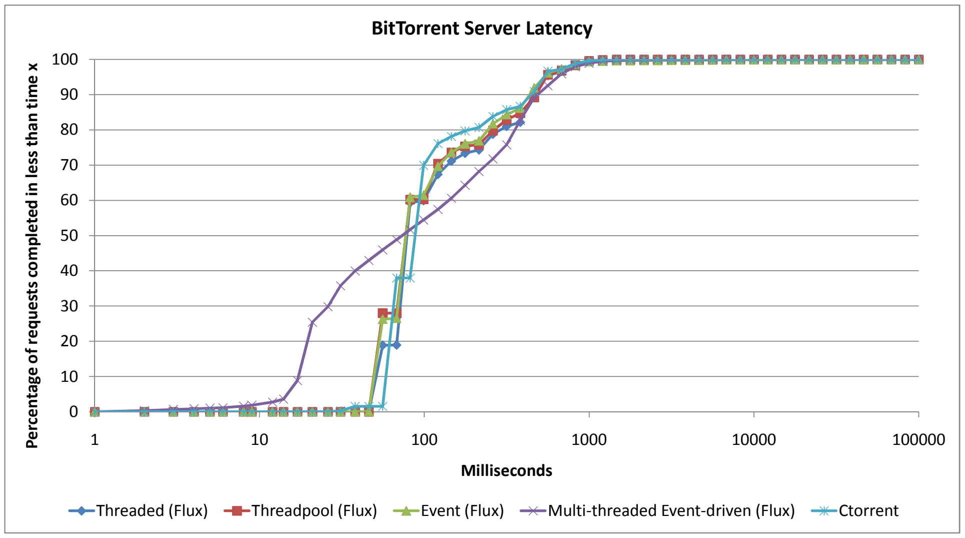


Fig. 6. Comparison of Flux BitTorrent servers with CTorrent (see Section 4.3). This shows that most of the servers perform similarly, although the hybrid thread-event Flux system provides some gains in low-latency, although with a slightly heavier tail.

before repeating these steps. Figure 3 presents the throughput and latency of each server for a range of simultaneous clients. In the second SPECWeb99-like test, the clients each make 5 requests before disconnecting from the server. These results, shown in Figure 4 do not include the *Flash* webserver, as explained above. The final webserver test uses the disk-bound pattern and a single request per connection. Figure 5 shows this test's results. Each of the above graphs was constructed from the average of five different runs of each experiment. All latency graphs represent the runs using 1000 simultaneous clients.

The results show that the Flux webservers using threaded runtimes are comparable to both *Knot* and *Flash* with respect to throughput, with the threadpool runtime often outperforming both with respect to latency. Both threaded runtimes routinely outperform *Haboob*, except for a small percentage of requests which are completed in less time. Previous Flux experiments using LinuxThreads showed a significant performance penalty for using the threaded runtime over the threadpool approach, as the threaded model allocated and freed threads on a per-request basis. With the use of NPTL, this penalty is no longer as pronounced. This was expected, as NPTL reduced thread startup costs was one of NPTL's core design goals [Drepper and Molnar 2005].

The event-based runtime does not match the throughput of the thread-based runtimes in the SPECWeb99-like tests, but performs better than them in the disk-bound scenario. Because the disk-bound scenario makes fewer blocking IO calls, the overhead due to non-native asynchronous IO support is minimized. Similarly, in the disk-bound case, where there are larger requests, the event-based system can dedicate more time to writing rather than dealing with contention among various writers. This results in slightly poorer overall latency, however, as requests are not handled in parallel.

In the disk-bound scenario, the thread-based runtimes generally show slightly poorer throughput than the event-based runtime, but improved latency. This is because Flux does not attempt to schedule threads to prioritize the completion of blocking IO. Rather, all threads are treated equally, allowing each thread to service its request for a brief period of time. As a result, requests see an improvement in latency at the cost of throughput. Attempting to prioritize individual threads would require the runtimes to have detailed knowledge of node implementations. Not only is this contrary to the language-independence goal of Flux, but it is unnecessary, as the included runtimes provide for a level of adaptability that covers various levels of tradeoff between throughput and latency. With the goal of adaptability over specialization we made no effort to counter this inherent tradeoff between throughput and latency.

The hybrid MTED runtime's performance is similar to that of the threadpool runtime. The throughput is slightly higher in the SPECWeb-99 tests, but slightly lower in the disk-bound test. The latency is higher in all cases.

The Java runtimes both suffer from poor throughput, but as they were not optimized and served primarily to demonstrate Flux's adaptability to other base languages, this is to be expected. However, the latency for these servers is often far better than the others, which we attribute to the JVM's thread scheduling policies.

4.3 Peer-to-Peer: BitTorrent

Peer-to-peer applications act as both a server and a client. Unlike a request-response server, they both receive and initiate requests.

We implemented a BitTorrent server in Flux as a representative peer-to-peer application. BitTorrent uses a scatter-gather protocol for file sharing. BitTorrent peers exchange pieces of a shared file until all participants have a complete copy. Network load is balanced by randomly requesting different pieces of the file from different peers.

To facilitate benchmarking, we changed the behavior of both of the BitTorrent peers we test here (the Flux version and CTorrent). First, all client peers are *unchoked* by default. Choking is an internal BitTorrent state that blocks certain clients from downloading data. This protocol restriction prevents real-world servers from being overwhelmed by too many client requests. We also allow an unlimited number of unchoked client peers to operate simultaneously, while the real BitTorrent server only unchokes clients who upload content.

We are unaware of any existing BitTorrent benchmarks, so we developed our own. Our BitTorrent benchmark mimics the traffic encountered by a busy BitTorrent peer and stresses server performance. It simulates a series of clients continuously sending requests for randomly distributed pieces of a 54MB test file to a BitTorrent peer with a complete copy of the file. When a peer finishes downloading a piece of the file, it immediately requests another random piece of the file from those still missing. Once a client has obtained the entire file, it disconnects. This benchmark does not simulate the "scatter-gather" nature of the BitTorrent protocol; instead, all requests go to a single peer. Using single peers has the effect of maximizing load, since obtaining data from a different source would lessen the load on the peer being tested.

Figure 6 compares the latency and network throughput to CTorrent, an implementation of the BitTorrent protocol written in C. Slight modifications had to be made to the CTorrent source code in order to simulate multiple clients from a single machine. The changes had no impact on CTorrent's performance. The goal of any BitTorrent system is to maximize network saturation, and both the CTorrent and Flux implementations achieve this goal. However, the performance of the Flux servers falls off as the number of clients increases, especially the event and MTED runtimes. This performance degradation is likely due to these runtime's reliance on our asynchronous I/O workaround. However, because most real-world BitTorrent peers limit the number of connections below this drop-off point, this has no practical impact.

4.4 Heartbeat Client-Server: Game Server

Unlike request-response client/server applications and most peer-to-peer applications, certain server applications are subject to deadlines. An example of such a server is an online multi-player game. In these applications, the server maintains the shared state of the game and distributes this state to all of the players at "heartbeat" intervals. There are two important conditions that must be met by this communication: the state possessed by all clients must be the same at each instant in time, and the inter-arrival time between states can not be too great. If either of these conditions is violated, the game will be unplayable or susceptible to cheating. These requirements place an important delay-sensitive constraint on the server's performance.

We have implemented an online multi-player game of Tag in Flux. The Flux game server enforces the rules of Tag. Players can not move beyond the boundaries of the game world. When a player is tagged by the player who is "it", that player becomes the new "it" and is teleported to a new random location on the board. All communication between clients and server occurs over UDP at 10Hz, a rate comparable to other real-world online games. While simple, this game has all of the important characteristics of servers for first person shooter or real-time strategy games.

Benchmarking the gameserver is significantly different than load-testing either the webserver or BitTorrent peer. Throughput is not a consideration since only small pieces of data are transmitted. The primary concern is the latency of the server as the number of clients increases. The server must receive every player's move, compute the new game state, and broadcast it within a fixed window of time.

To load-test the game server, we measured the effect of increasing the number of players. The performance of the gameserver is largely based upon the length of time it takes the server to update the game state given the moves received from all of the players, and this computation time is identical across the servers. The latency of the gameserver is largely a product of the rate of game turns, which stays constant at 10Hz. We found no appreciable differences between a traditional implementation of the gameserver and the various Flux versions. These results show that Flux is capable of producing a server with sufficient performance for multi-player online gaming.

5. SIMULATOR AND PROFILING

In addition to its programming language support for writing server applications, Flux provides support for predicting and measuring the performance of server applications. The Flux system generates **discrete-event simulators** that predict server performance for synthetic workloads and on different hardware. It also performs **path profiling** to identify server performance bottlenecks on a deployed system.

5.1 Performance Prediction

Predicting the performance of a server prior to deployment is important but often difficult. For example, performance bottlenecks due to contention may not appear during testing because the load placed on the system is insufficient. System testing on a small-scale system may not reveal problems that arise when the system is deployed on an enterprise-scale multiprocessor.

In addition to generating executable server code, the Flux code generator transforms a Flux program directly into a discrete-event simulator that models the performance of the server. The implementation language for the simulator is CSim [Mesquite Software 2007], a C-based, process-oriented simulator.

In the simulator, CPUs are modeled as resources that each Flux node acquires for a given amount of time. The simulator can either use observed parameters from a running system (per-node execution times, source node inter-arrival times, and observed branching probabilities), or the Flux programmer can supply estimates for these parameters. The simulator can model an arbitrary number of processors by increasing the number of nodes that may simultaneously acquire the CPU resource. When a node uses a given atomicity constraint, it treats it as a lock and acquires it for the duration of the node's execution.

The code in Figure 7 is a simplified version of the CSim code that Flux generates for the CheckCache node from Section 2. The function begins by reserving the cache lock, per the Flux program. The CPU is reserved next. After acquiring both resources, the CPU time is modeled on Line 4 by holding both resources

```
1 void CheckCache(){
2     cache_program_lock->reserve();
3     processor->reserve();
4     hold((CMP_TIME_CPU_CHECKCACHE));
5     processor->release();
6     hold((CMP_TIME_CLOCK_CHECKCACHE - CMP_TIME_CPU_CHECKCACHE));
7     cache_program_lock->release();
8     Handler();
9 }
```

Fig. 7. Sample discrete-event simulation code generated for a Flux node.

for a specified length of time. In this case, the resources are held for the average CPU time spent in `CheckCache`, as observed while profiling execution of the image server. The CPU resource can then be released. On Line 6, the node simulates the time spent with the cache lock without using the CPU. This time is calculated by subtracting the time spent on the CPU from the average time spent in the node, per profiling. Finally, the node releases the cache lock, and moves on to execute the next node in the data flow.

It is important to note that this simulation does not model disk or network resources. While this is a realistic assumption for CPU-bound servers (such as dynamic web-servers), other servers will require more complete modeling. We leave this as a question for future research.

To demonstrate that the generated simulations accurately predict actual performance, we tested the image server described in Section 2. To simulate load on the machine, we made requests at increasingly small inter-arrival times. The image server had eight images, with each request selecting varying sizes (between 1/8th scale and full-size) of a randomly-chosen image for compression. When configured to run with n "clients", the load tester issued requests at a rate of one every $1/n$ seconds. The image server is CPU and lock-bound, with each image taking on average 80 milliseconds to process.

We first measured the performance of this server on a computer with two Intel Xeon 5140 processors, for a total of four cores. Using CPU hotplugging support in the Linux 2.6 kernel, three of the cores were disabled. The observed node runtimes and branching probabilities were then used to parametrize the generated CSIM simulator. We compare the predicted and actual performance of the server by making more processors available to the system. As Figure 8 shows, the predicted results (dotted lines) and actual results (solid lines) match closely, demonstrating the effectiveness of the simulator at predicting performance.

5.2 Path Profiling

The Flux compiler instruments generated servers to simplify the identification of performance bottlenecks. This profiling information takes the form of "hot paths", the most frequent or most time-consuming paths in the server. Flux identifies these hot paths using the Ball-Larus path profiling algorithm [Ball and Larus 1994]. Because Flux graphs are acyclic, the Ball-Larus algorithm identifies each unique path through the server's data-flow graph.

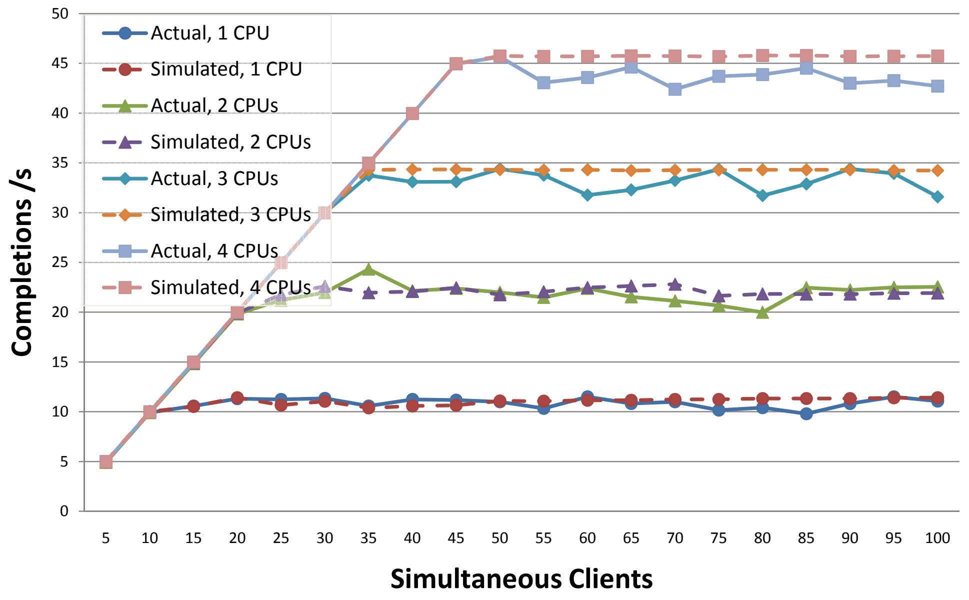


Fig. 8. Predicted performance of the image server (derived from a single-processor run) versus observed performance for varying numbers of processors and load.

Hot paths not only aid understanding of server performance characteristics but also identify places where optimization would be most effective. Because profiling information can be obtained from an operating server and is linked directly to paths in the program graph, a performance analyst can easily understand the performance characteristics of deployed servers.

The overhead of path profiling is low enough that hot path information can be maintained even in a production server. Profiling adds just one arithmetic operation and two high-resolution timer calls to each node. A performance analyst can obtain path profiles from a running Flux server by connecting to a dedicated socket.

To demonstrate the use of path profiling, we compiled a version of the BitTorrent peer with profiling enabled. For the experiments, we used a patched version of Linux that supports per-thread time gathering. The BitTorrent peer was load-tested with the same tester as in the performance experiments. For profiling, we used loads of 25, 50, and 100 clients. All profiling information was automatically generated from a running Flux server.

In BitTorrent, the most time-consuming path identified by Flux was, unsurprisingly, the file transfer path (`Listen` → `GetClients` → `SelectSockets` → `CheckSockets` → `Message` → `ReadMessage` → `HandleMessage` → `Request` → `MessageDone`, 0.295 ms). However, the second most expensive path was the path that finds no outstanding chunk requests (`Listen` → `GetClients` → `SelectSockets` → `CheckSockets` → `ERROR`, 0.016ms). While this path is relatively cheap compared to the file transfer path, it also turns out to be the most frequently executed path (780,510 times, compared to 313,994 for the file transfer path). Since this path accounts for 13% of BitTorrent's execution time, it is a reasonable candidate for optimization efforts.

6. DEVELOPER EXPERIENCE

In this section, we examine the experience of programmers implementing Flux applications. In particular, we focus on the implementation of the Flux BitTorrent peer.

The Flux BitTorrent peer was implemented by two undergraduate students in less than one week. The students began with no knowledge of the technical details of the BitTorrent protocol or the Flux language. The design of the Flux program for the BitTorrent peer was entirely their original work. The implementation of the functional nodes in BitTorrent is loosely derived from the CTorrent source code. The program graph for the BitTorrent server is shown in Figure 9 at the end of this document.

The students had a generally positive reaction to programming in Flux. Primarily, they felt that organizing the application into a Flux program graph prior to implementation helped modularize their application design and debug server data flow prior to programming. They also found that the exposure of atomicity constraints at the Flux language level allowed for easy identification of the appropriate locations for mutual exclusion. Flux's immunity to deadlock and the simplicity of the atomicity constraints increased their confidence in the correctness of the resulting server.

Though this is only anecdotal evidence, this experience suggests that programmers can quickly gain enough expertise in Flux to build reasonably complex server applications.

A user study conducted with Eon [Sorber et al. 2007], a language and runtime system built on Flux, showed that users found the Flux syntax to be intuitive and easy to grasp. Further, inexperienced Eon users were able to write uniformly correct adaptive systems faster than their experienced C user counterparts. Though Eon's features differ from Flux's, our experiences with writing servers using Flux are analogous to those of that study's participants.

7. RELATED WORK

This section discusses related work to Flux in the areas of coordination and data flow languages, programming language constructs, domain-specific languages, and runtime systems.

Coordination and data flow languages. Flux is an example of a *coordination language* [Gelernter and Carriero 1992] that combines existing code into a larger program in a data flow setting. There have been numerous data flow languages proposed in the literature, see Johnston et al. for a recent survey [Johnston et al. 2004]. Data flow languages generally operate at the level of fundamental operations rather than at a functional granularity. One exception is CODE 2, which permits incorporation of sequential code into a dynamic flow graph, but restricts shared state to a special node type [Browne et al. 1989; Browne et al. 2000]. Data flow languages also typically prohibit global state. For example, languages that support *streaming* applications like StreamIt [Thies et al. 2002] express all data dependencies in the data flow graph. Flux departs from these languages by supporting safe access to global state via atomicity constraints. Most importantly, these languages focus on extracting parallelism from individual programs, while Flux describes parallelism across multiple clients or event streams.

Programming language constructs. Flux shares certain linguistic concepts with previous and current work in other programming languages. Flux's predicate matching syntax is deliberately based on the pattern-matching syntax used by functional languages like ML, Miranda, and Haskell [Hudak 1989; Milner 1984; Turner 1985]. The PADS data description language also allows programmers to specify predicate types, although these must be written in PADS itself rather than in an external language like C [Fisher and Gruber 2005]. Flanagan and Freund present a type inference system that computes "atomicity constraints" for Java programs that correspond to Lipton's theory of reduction [Flanagan et al. 2005; Lipton 1975]; Flux's atomicity constraints operate at a higher level of abstraction. The Autolocker tool [McCloskey et al. 2006], developed independently and concurrently with this work, automatically assigns locks in a deadlock-free manner to manually-annotated C programs. It shares Flux's enforcement of an acyclic locking order and its use of two-phase lock acquisition and release.

Related domain-specific languages. Several previous domain-specific languages allow the integration of off-the-shelf code into data flow graphs, though for different domains. The Click modular router is a domain-specific language for building network routers out of existing C components [Kohler et al. 2000]. Knit is a domain-specific language for building operating systems, with rich support for integrating code implementing COM interfaces [Reid et al. 2000]. In addition to its linguistic and tool support for programming server applications, Flux ensures deadlock-freedom by enforcing a canonical lock ordering; this is not possible in Click and Knit because they permit cyclic program graphs. Flux's syntax, flow oriented design, and runtime independence have also been adopted by Eon [Sorber et al. 2007] an energy-aware declarative coordination language for programming perpetual systems.

Runtime systems. Researchers have proposed a wide variety of runtime systems for high-concurrency applications and servers, including SEDA [Welsh et al. 2001], Hood [Acar et al. 2000; Blumofe and Papadopoulos 1998], Capriccio [von Behren et al. 2003], Fibers [Adya et al. 2002], cohort scheduling [Larus and Parkes 2002], and libasync/mp [Zeldovich et al. 2003], whose per-callback *colors* could be used to implement Flux's atomicity constraints. Users of these runtimes are forced to implement a server using a particular API. Once implemented, the server logic is generally inextricably linked to the runtime. By contrast, Flux programs are independent of any particular choice of runtime system, so advanced runtime systems can be integrated directly into Flux's code generation pass.

ACKNOWLEDGMENTS

The authors thank Gene Novark for helping to design the discrete event simulation generator, Vitaliy Lvin for assisting in experimental setup and data gathering, Ting Yang for the design and implementation of the MTED runtime, and Jay Lepreau for serving as our USENIX shepherd.

This material is based upon work supported by the National Science Foundation under CAREER Awards CNS-0347339 and CNS-0447877. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Acar, U. A., Blelloch, G. E., and Blumofe, R. D. 2000. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM Press, New York, NY, USA, 1–12.

Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. 2002. Cooperative task management without manual stack management. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 289–302.

Appel, A. W., Flannery, F., and Hudson, S. E. 2005. CUP parser generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

Ball, T. and Larus, J. R. 1994. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems 16,* 4 (July), 1319–1360.

Berk, E. and Ananian, C. S. 2007. JLex: A lexical analyzer generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

Blumofe, R. D. and Papadopoulos, D. 1998. The performance of work stealing in multiprogrammed environments (extended abstract). In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. ACM Press, New York, NY, USA, 266–267.

Browne, J. C., Azam, M., and Sobek, S. 1989. CODE: A unified approach to parallel programming. *IEEE Software 6,* 4, 10–18.

Browne, J. C., Berger, E. D., and Dube, A. 2000. Compositional development of performance models in POEMS. *The International Journal of High Performance Computing Applications 14,* 4 (Winter), 283–291.

Drepper, U. and Molnar, I. 2005. The native posix thread library for linux.

Fisher, K. and Gruber, R. 2005. PADS: a domain-specific language for processing ad hoc data. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 295–304.

Flanagan, C., Freund, S. N., and Lifshin, M. 2005. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*. ACM Press, New York, NY, USA, 47–58.

Gelernter, D. and Carriero, N. 1992. Coordination languages and their significance. *Commun. ACM 35,* 2, 96.

Hudak, P. 1989. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv. 21,* 3, 359–411.

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

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. 2000. The Click modular router. *ACM Transactions on Computer Systems 18,* 3 (August), 263–297.

Larus, J. R. and Parkes, M. 2002. Using cohort-scheduling to enhance server performance. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 103–114.

Lipton, R. J. 1975. Reduction: a method of proving properties of parallel programs. *Commun. ACM 18,* 12, 717–721.

McCloskey, B., Zhou, F., Gay, D., and Brewer, E. 2006. Autolocker: synchronization inference for atomic sections. In *POPL*, J. G. Morrisett and S. L. P. Jones, Eds. ACM, 346–358.

Mesquite Software. 2007. The CSIM Simulator. `http://www.mesquite.com`.

Milner, R. 1984. A proposal for standard ml. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM Press, New York, NY, USA, 184–197.

Pai, V. S., Druschel, P., and Zwaenepoel, W. 1999. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*.

Reid, A., Flatt, M., Stoller, L., Lepreau, J., and Eide, E. 2000. Knit: Component composition for systems software. In *Proceedings of the 4th ACM Symposium on Operating Systems Design and Implementation (OSDI)*. 347–360.

Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. 2007. Eon: A Language and Runtime System for Perpetual Systems. In *Proceedings of The Fifth International ACM Conference on Embedded Networked Sensor Systems (SenSys '07)*. Sydney, Australia.

Standard Performance Evaluation Corporation. 2007. SPECweb99. http://www.spec.org/osg/web99/.

Thies, W., Karczmarek, M., and Amarasinghe, S. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Grenoble, France.

Turner, D. A. 1985. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*. Springer-Verlag New York, Inc., New York, NY, USA, 1–16.

von Behren, R., Condit, J., Zhou, F., Necula, G. C., and Brewer, E. 2003. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 268–281.

Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 230–243.

Zeldovich, N., Yip, A., Dabek, F., Morris, R., Mazières, D., and Kaashoek, F. 2003. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*. San Antonio, Texas.

Received Month Year; revised Month Year; accepted Month Year

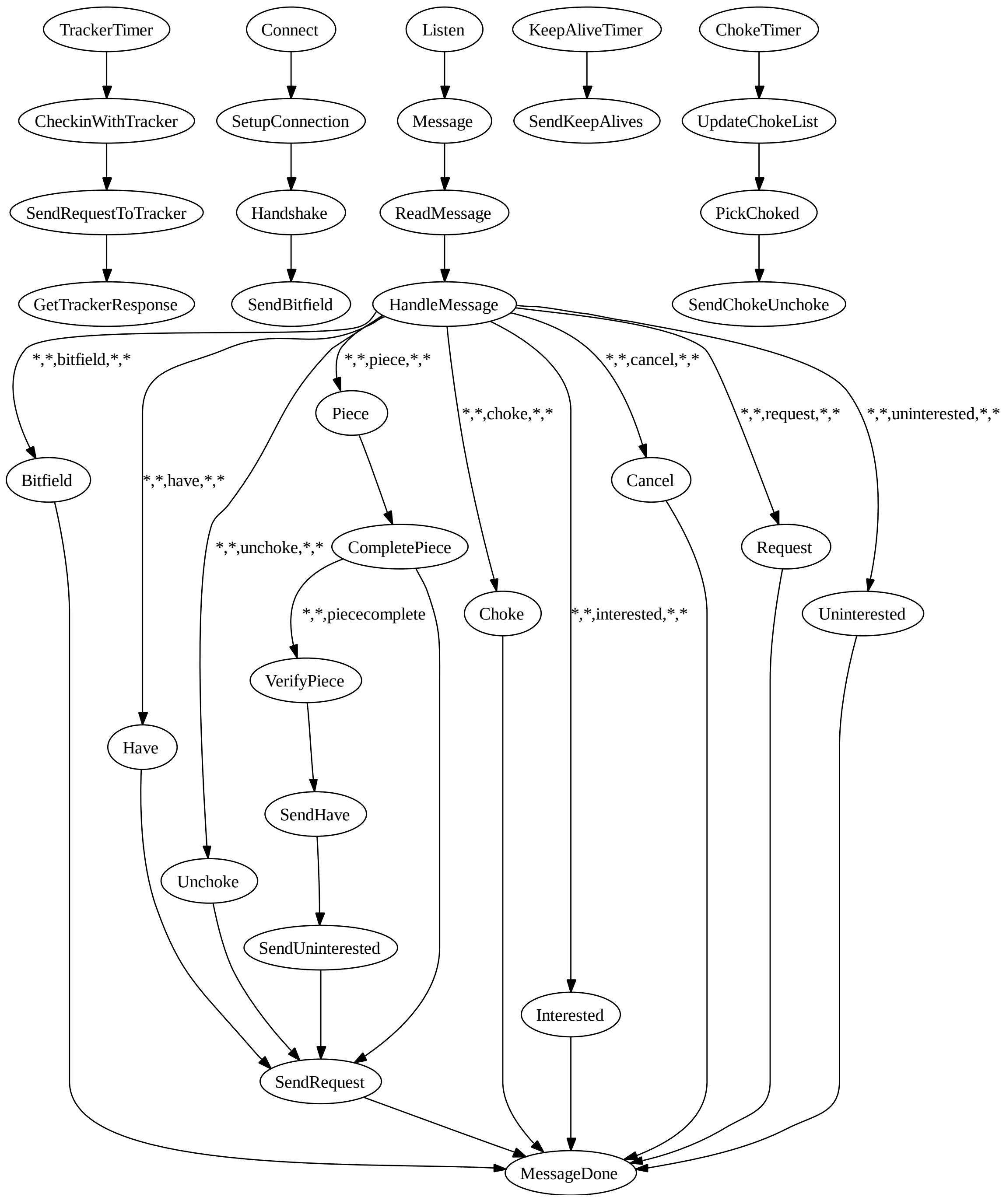


Fig. 9. The Flux program graph for the example BitTorrent server.