Multiprocessor Support for Event-Driven Programs

Nickolai Zeldovich,[*] Alexander Yip, Frank Dabek,
Robert T. Morris, David Mazières,[†] Frans Kaashoek
nickolai@cs.stanford.edu, {yipal, fdabek, rtm, kaashoek}@lcs.mit.edu, dm@cs.nyu.edu
MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139

Abstract

This paper presents a new asynchronous programming library (*libasync-smp*) that allows event-driven applications to take advantage of multiprocessors by running code for event handlers in parallel. To control the concurrency between events, the programmer can specify a *color* for each event: events with the same color (the default case) are handled serially; events with different colors can be handled in parallel. The programmer can incrementally expose parallelism in existing event-driven applications by assigning different colors to computationally-intensive events that do not share mutable state.

An evaluation of *libasync-smp* demonstrates that applications achieve multiprocessor speedup with little programming effort. As an example, parallelizing the cryptography in the SFS file server required about 90 lines of changed code in two modules, out of a total of about 12,000 lines. Multiple clients were able to read large cached files from the *libasync-smp* SFS server running on a 4-CPU machine 2.5 times as fast as from an unmodified uniprocessor SFS server on one CPU. Applications without computationally intensive tasks also benefit: an event-driven Web server achieves 1.5 speedup on four CPUs with multiple clients reading small cached files.

1 Introduction

To obtain high performance, servers must overlap computation with I/O. Programs typically achieve this overlap using threads or events. Threaded programs typically process each request in a separate thread; when one thread blocks waiting for I/O, other threads can run. Threads provide an intuitive programming model and can take advantage of multiprocessors; however, they require coordination of accesses by different threads to shared state, even on a uniprocessor. In contrast, event-based programs are structured as a collection of *callback* functions which a main loop calls as I/O events occur. Event-based programs execute callbacks serially, so the programmer need not worry about concurrency control; however, event-based programs until now have been unable to take full advantage of multiprocessors without running multiple copies of an application or introducing fine-grained synchronization.

The contribution of this paper is *libasync-smp*, a library that supports event-driven programs on multiprocessors. *libasync-smp* is intended to support the construction of user-level systems programs, particularly network servers and clients; we show that these applications can achieve performance gains on multiprocessors by exploiting coarse-grained parallelism. *libasync-smp* is intended for programs that have natural opportunities for parallel speedup; it has no support for expressing very fine-grained parallelism. The goal of *libasync-smp*'s concurrency control mechanisms is to provide enough concurrency to extract parallel speedup without requiring the programmer to reason about the correctness of a fine-grained parallel program.

Much of the effort required to make existing event-driven programs take advantage of multiprocessors is in specifying which events may be handled in parallel. *libasync-smp* provides a simple mechanism to allow the programmer to incrementally add parallelism to uniprocessor applications as an optimization. This mechanism allows the programmer to assign a *color* to each callback. Callbacks with different colors can execute in parallel. Callbacks with the same color execute serially. By default, *libasync-smp* assigns all callbacks the same color, so existing programs continue to work correctly without modification. As programmers discover opportunities to safely execute callbacks in parallel, they can assign different colors to those callbacks.

libasync-smp is based on the *libasync* library [16].

[*] Stanford University
[†] New York University

libasync uses operating system asynchronous I/O facilities to support event-based programs on uniprocessors. The modifications for *libasync-smp* include coordinating access to the shared internal state of a few *libasync* modules, adding support for colors, and scheduling callbacks on multiple CPUs.

An evaluation of *libasync-smp* demonstrates that applications achieve multiprocessor speedup with little programming effort. As an example, we modified the SFS [17] file server to use *libasync-smp*. This server uses more than 260 distinct callbacks. Most of the CPU time is spent in just two callbacks, those responsible for encrypting and decrypting client traffic; this meant that coloring just a few callbacks was sufficient to gain substantial parallel speedup. The changes affected 90 lines in two modules, out of a total of about 12,000 lines. When run on a machine with four Intel Xeon CPUs, the modified SFS server was able to serve large cached files to multiple clients 2.5 times as fast as an unmodified uniprocessor SFS server on one CPU.

Even servers without CPU-intensive operations such as cryptography can achieve speedup approaching that offered by the operating system, especially if the O/S kernel can take advantage of a multiprocessor. For example, with a workload of multiple clients reading small cached files, an event-driven web server achieves 1.5 speedup on four CPUs.

The next section (Section 2) introduces *libasync*, on which *libasync-smp* is based, and describes its support for uniprocessor event-driven programs. Section 3 and 4 describe the design and implementation of *libasync-smp*, and show examples of how applications use it. Section 5 uses two examples to show that use of *libasync-smp* requires little effort to achieve parallel speedup. Section 6 discusses related work, and Section 7 concludes.

2 Uniprocessor Event-driven Design

Many applications use an event-driven architecture to overlap slow I/O operations with computation. Input from outside the program arrives in the form of events; events can indicate, for example, the arrival of network data, a new client connection, completion of disk I/O, or a mouse click. The programmer structures the program as a set of callback functions, and registers interest in each type of event by associating a callback with that event type.

In the case of complex event-driven servers, such as named [7], the complete processing of a client request may involve a sequence of callbacks; each consumes an event, initiates some I/O (perhaps by sending a request packet), and registers a further callback to handle completion of that particular I/O operation (perhaps the arrival of a specific response packet). The event-driven architecture allows the server to keep state for many concurrent I/O activities.

Event-driven programs typically use a library to support the management of events. Such a library maintains a table associating incoming events with callbacks. The library typically contains the main control loop of the program, which alternates between waiting for events and calling the relevant callbacks. Use of a common library allows callbacks from mutually ignorant modules to co-exist in a single program.

An event-driven library's control loop typically calls ready callbacks one at a time. The fact that the callbacks never execute concurrently simplifies their design. However, it also means that an event-driven program typically cannot take advantage of a multiprocessor.

The multiprocessor event-driven library described in this paper is based on the *libasync* uniprocessor library originally developed as part of SFS [17, 16]. This section describes uniprocessor *libasync* and the programming style involved in using it. Existing systems, such as named [7] and Flash [19], use event-dispatch mechanisms similar to the one described here. The purpose of this section is to lay the foundations for Section 3's description of extensions for multiprocessors.

2.1 *libasync*

libasync is a UNIX C++ library that provides both an event dispatch mechanism and a collection of event-based utility modules for functions such as DNS host name lookup and Sun RPC request/reply dispatch [16]. Applications and utility modules register callbacks with the *libasync* dispatcher. *libasync* provides a single main loop which waits for new events with the UNIX `select()` system call. When an event occurs, the main loop calls the corresponding registered callback. Multiple modules can use *libasync* without knowing about each other, which encourages modular design and reusable code.

libasync handles a core set of events as well as a set of events implemented by utility modules. The core events include new connection requests, the arrival of data on file descriptors, timer expiration, and UNIX signals. The RPC utility module allows automatic parsing of incoming Sun RPC calls; callbacks registered per program/procedure pair are invoked when an RPC arrives. The RPC module also allows a callback to be registered to handle the arrival of the reply to a particular RPC call. The DNS module supports non-blocking concurrent host name lookups. Finally, a file I/O module allows applications to perform non-blocking file system operations by sending RPCs to the NFS server in the local kernel; this allows non-blocking access to all file system operations, including (for example) file name lookup.

Typical programs based on *libasync* register a callback at every point at which an equivalent single-threaded sequential program might block waiting for input. The result is that programs create callbacks at many points in the code. For example, the SFS server creates callbacks at about 100 points.

In order to make callback creation easy, *libasync* provides a type-checked facility similar to function-currying [23] in the form of the `wrap()` macro [16]. `wrap()` takes a function and values as arguments and returns an anonymous function called a *wrap*. If `w=wrap(fn, x, y)`, for example, then a subsequent call `w(z)` will result in a call to `fn(x, y, z)`. A wrap can be called more than once; *libasync* reference-counts wraps and automatically frees them in order to save applications tedious book keeping. Similarly, the library also provides support for programmers to pass reference-counted arguments to wrap. The benefit of `wrap()` is that it simplifies the creation of callback structures that carry state.

2.2 Event-driven Programming

Figure 1 shows an abbreviated fragment of a program written using *libasync*. The purpose of the application is to act as a web proxy. The example code accepts TCP connections, reads an HTTP request from each new connection, extracts the server name from the request, connects to the indicated server, etc. One way to view the example code is that it is the result of writing a single sequential function with all these steps, and then splitting it into callbacks at each point that the function would block for input.

`main()` calls `inetsocket()` to create a socket that listens for new connections on TCP port 80. UNIX makes such a socket appear readable when new connections arrive, so `main()` calls the *libasync* function `fdcb()` to register a read callback. Finally `main()` calls `amain()` to enter the *libasync* main loop.

The *libasync* main loop will call the callback wrap with no arguments when a new connection arrives on `afd`. The wrap calls `accept_cb()` with the other arguments passed to `wrap()`, in this case the file descriptor `afd`. After allocating a buffer in which to accumulate client input, `accept_cb()` registers a callback to `req_cb()` to read input from the new connection. The server keeps track of its state for the connection, which consists of the file descriptor and the buffer, by including it in each `wrap()` call and thus passing it from one callback to the next. If multiple clients connect to the proxy, the result will be multiple callbacks waiting for input from the client connections.

```
main()
{
  // listen on TCP port 80
  int afd = inetsocket(SOCK_STREAM, 80);
  // register callback for new connections
  fdcb(afd, READ, wrap(accept_cb, afd));
  amain();  // start main loop
}

// called when a new connection arrives
accept_cb(int afd)
{
  int fd = accept(afd, ...);
  str inBuf(""); // new ref-counted buffer
  // register callback for incoming data
  fdcb(fd, READ, wrap(req_cb, fd, inBuf));
}

// called when data arrives
req_cb(int fd, str inBuf)
{
  read(fd, buf, ...);
  append input to inBuf;
  if(complete request in inBuf){
    // un-register callback
    fdcb(fd, READ, NULL);

    // parse the HTTP request
    parse_request(inBuf, serverName, file);

    // resolve serverName and connect
    // both are asynchronous
    tcpconnect(serverName, 80,
               wrap(connect_cb, fd, file));
  } else {
    // do nothing; wait for more calls to req_cb()
  }
}

// called when we have connected to the server
connect_cb(int client_fd, str file, int server_fd)
{
    // write the request when the socket is ready
    fdcb(server_fd, WRITE,
          wrap (write_cb, file, server_fd));
}
```

Figure 1: Outline of a web proxy that uses *libasync*.

When a complete request has arrived, the proxy server needs to look up the target web server's DNS host name and connect to it. The function `tcpconnect()` performs both of these tasks. The DNS lookup itself involves waiting for a response from a DNS server, perhaps more than one in the case of timeouts; thus the *libasync* DNS resolver is internally structured as a set of callbacks. Waiting for TCP connection establishment to complete also involves callbacks. For these reasons, `tcpconnect()` takes a wrap as one of its argument, carries that wrap along in its own callbacks, and finally calls the wrap when the connection process completes or fails. This style of programming is reminiscent of the continuation-passing style [21], and makes it easy for programmers to compose modules.

A number of applications are based on *libasync*; Figure 2 lists some of them, along with the number of distinct calls to `wrap()` in each program. These numbers give a feel for the level of complexity in the programs' use of callbacks.

Name	#Wraps	Lines of Code
SFS [17]	229	39871
SFSRO [13]	58	4836
Chord [22]	65	5445
CFS [10]	87	4960

Figure 2: Applications based on *libasync*, along with the approximate number of distinct calls to `wrap()` in each application. The numbers are exclusive of the wraps created by *libasync* itself, which number about 30.

2.3 Interaction with multiprocessors

A single event-driven process derives no direct benefit from a multi-processor. There may be an indirect speedup if the operating system or helper processes can make use of the multiprocessor's other CPUs.

It is common practice to run multiple independent copies of an event-driven program on a multiprocessor. This *N-copy* approach might work in the case of a web server, since the processing of different client requests can be made independent. The N-copy approach does not work if the program maintains mutable state that is shared among multiple clients or requests. For example, a user-level file server might maintain a table of leases for client cache consistency. In other cases, running multiple independent copies of a server may lead to a decrease in efficiency. A web proxy might maintain a cache of recently accessed pages: multiple copies of the proxy could maintain independent caches, but content duplicated in these caches would waste memory.

3 Multiprocessor Design

The focus of this paper is *libasync-smp*, a multiprocessor extension of *libasync*. The goal of *libasync-smp* is to execute event-driven programs faster by running callbacks on multiple CPUs. Much of the design of *libasync-smp* is motivated by the desire to make it easy to adapt existing *libasync*-based servers to multiprocessors. The goal of the *libasync-smp* design is to allow both the parallelism of the N-copy arrangement and the advantages of shared data structures.

A server based on *libasync-smp* consists of a single process containing one worker thread per available CPU. Each thread repeatedly chooses a callback from a set of runnable callbacks and runs it. The threads share an address space, file descriptors, and signals. The library assumes that the number of CPUs available to the process is static over its running time. A mechanism such as scheduler activations [2] could be used to dynamically determine the number of available CPUs.

There are a number of design challenges to making the single address space approach work, the most interesting of which is coordination of access to application data shared by multiple callbacks. An effective concurrency control mechanism should allow the programmer to easily (and incrementally) identify which parts of a server can safely be run in parallel.

3.1 Coordinating callbacks

The design of the concurrency control mechanisms in *libasync-smp* is motivated by two observations. First, system software often has natural coarse-grained parallelism, because different requests don't interact or because each request passes through a sequence of independent processing stages. Second, existing event-driven programs are already structured as non-blocking units of execution (callbacks), often associated with one stage of the processing for a particular client. Together, these observations suggest that individual callbacks are an appropriate unit of coordination of execution.

libasync-smp associates a *color* with each registered callback, and ensures that no two callbacks with the same color execute in parallel. Colors are arbitrary 32-bit values. Application code can optionally specify a color for each callback it creates; if it specifies no color, the callback has color zero. Thus, by default, callbacks execute sequentially on a single CPU. This means that unmodified event-driven applications written for *libasync* will execute correctly with *libasync-smp*.

The orthogonality of color to the callback's code eases the adaptation of existing *libasync*-based servers. A typical arrangement is to run the code that accepts new client connections in the default color. If the processing for different connections is largely independent, the programmer assigns each new connection a new unique color that applies to all the callbacks involved in processing that connection. If a particular stage in request processing shares mutable data among requests (e.g. a cache of web pages), the programmer chooses a color for that stage and applies it to all callbacks that use the shared data, regardless of which connection the callback is associated with.

In some cases, application code may need to be restructured to permit callbacks to be parallelized. For example, a single callback might use shared data but also have significant computation that does not use shared data. It may help to split such a callback; the first half would use a special *libasync-smp* call (`cpucb()`) to schedule the second half with a different color.

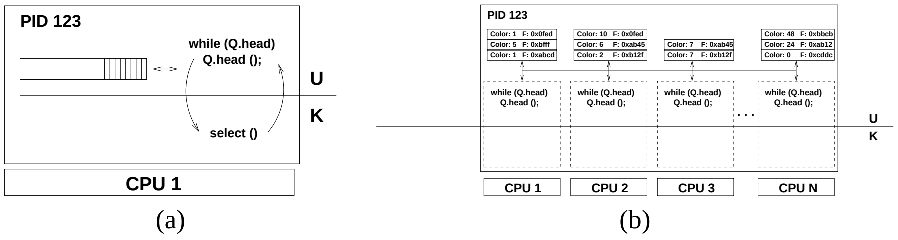

Figure 3: The single process event driven architecture (left) and the *libasync-smp* architecture (right). Note that in the *libasync-smp* architecture callbacks of the same color appear in the same queue. This guarantees that callbacks with the same color are never run in parallel and always run in the order in which they were scheduled.

The color mechanism is less expressive than locking; for example, a callback can have only one color, which is equivalent to holding a single lock for the complete duration of a callback. However, experience suggests that fine-grained and sophisticated locking, while it may be necessary for correctness with concurrent threads, is rarely necessary to achieve reasonable speedup on multiple CPUs for server applications. Parallel speedup usually comes from the parts of the code that don't need much locking; coloring allows this speedup to be easily captured, and also makes it easy to port existing event-driven code to multiprocessors.

3.2 *libasync-smp* API

The API that *libasync-smp* presents differs slightly from that exposed by *libasync*. The `cwrap()` function is analogous to the `wrap()` function described in Section 2 but takes an optional color argument; Table 1 shows the `cwrap()` interface. The color specified at the callback's creation (i.e. when `cwrap()` is called) dictates the color it will be executed under. Embedding color information in the callback object rather than in an argument to `fdcb()` (and other calls which register callbacks) allows the programmer to write modular functions which accept callbacks and remain agnostic to the color under which those callbacks will be executed. Note that colors are not inherited by new callbacks created inside a callback running under a non-zero color. While color inheritance might seem convenient, it makes it very difficult to write modular code as colors "leak" into modules which assume that callbacks they create carry color zero.

Since colors are arbitrary 32-bit values, programmers have considerable latitude in how to assign colors. One reasonable convention is to use each request's file descriptor number as the color for its parallelizable callbacks. Another possibility is to use the address of a data structure to which access must be serialized; for example, a per-client or per-request state structure. Depending on the convention, it could be the case that unrelated modules accidentally choose the same color. This might reduce performance, but not correctness.

libasync-smp provides a `cpucb()` function that schedules a callback for execution as soon as a CPU is idle. The `cpucb()` function can be used to register a callback with a color different from that of the currently executing callback. A common use of `cpucb()` is to split a CPU-intensive callback in two callbacks with different colors, one to perform a computation and the other to synchronize with shared state. To minimize programming errors associated with splitting an existing callback into a chain of `cpucb()` callbacks, *libasync-smp* guarantees that all CPU callbacks of the same color will be executed in the order they were scheduled. This maintains assumptions about sequential execution that the original single callback may have been relying on. Execution order isn't defined for callbacks with different colors.

3.3 Example

Consider the web proxy example from Section 2. For illustrative purposes assume that the `parse_request()` routine uses a large amount of CPU time and does not depend on any shared data. We could re-write `req_cb()` to parse different requests in parallel on different CPUs by calling `cpucb()` and assigning the callback a unique color. Figure 4 shows this change to `req_cb()`. In this example only the `parse_request()` workload is distributed across CPUs. As a further optimization, reading requests could be parallelized by creating the read request callback using `cwrap()` and specifying the request's file descriptor as the callback's color.

3.4 Scheduling callbacks

Scheduling callbacks involves two operations: placing callbacks on a worker thread's queue and, at each thread, deciding which callback to run next.

callback **cwrap** ((func *)(), arg1, arg2, ..., argN, color c = 0)	*// Create a callback object with color* `c`*.*
void **cpucb** (callback cb)	*// Add* `cb` *to the runnable callback queue immediately.*

Table 1: Sample calls from the *libasync-smp* API.

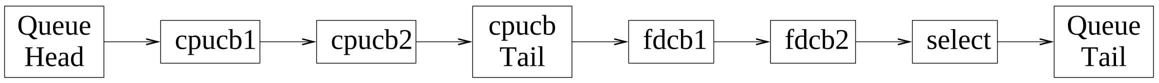

Figure 5: The callback queue structure in *libasync-smp*. `cpucb()` adds new callbacks to the left of the dummy element marked "cpucb Tail." New I/O callbacks are added at "Queue Tail." The scheduler looks for work starting at "Queue Head."

```
// called when data arrives
req_cb(int fd, str inBuf)
{
  read(fd, buf, ...);
  append input to inBuf;
  if(complete request in inBuf){
    // un-register callback
    fdcb(fd, READ, NULL);

    // parse the HTTP request under color fd
    cpucb (cwrap (parse_request_cb, fd, inBuf,
                 (color)fd))
  } else {
    // do nothing; wait for
    // more calls to req_cb()
  }
}

// below parsing done w/ color fd
parse_req_cb (int fd, str inBuf)
{
  parse_request (inBuf, serverName, file);

  // start connection to server
  tcpconnect (serverName, wrap(connect_cb, fd, file));
}
```

Figure 4: Changes to the asynchronous web proxy to take advantage of multiple CPUs

A callback is placed on a thread's queue in one of two ways: due to a call to `cpucb()` or because the *libasync-smp* main loop detected the arrival of an I/O, timer, or signal event for which a callback had been registered. A callbacks with color c is placed in the queue of worker thread $c \bmod N$ where N is the number of worker threads. This simple rule distributes callbacks approximately evenly among the worker threads. It also preserves the order of activation of callbacks with the same color and may improve cache locality.

If a worker thread's task queue is empty it attempts to steal work from another thread's queue [9]. Work must be stolen at the granularity of all callbacks of the same color and the color to be stolen must not be executing currently to preserve guarantees on ordering of callbacks within the same color. *libasync-smp* consults a per-thread field containing the currently running color to guarantee the latter requirement.

When a color is moved from one thread to another, future callbacks of that color will be assigned to the new queue; otherwise, callbacks of the same color might execute in parallel. To ensure that all callbacks with the same color appear on the same queue, the library maintains a mapping of colors to threads: the *n*th element of a 1024 element array indicates which thread should execute all colors which are congruent to $n \pmod{1024}$. This array is initialized in such a way as to give the initial distribution described above.

Each *libasync-smp* worker thread uses a simple scheduler to choose a callback to execute next from its queue. The scheduler considers priority and callback/thread affinity when choosing colors; its design is loosely based on that of the Linux SMP kernel [8].

The scheduler favors callbacks of the same color as the last callback executed by the worker in order to increase performance. Callback colors often correspond to particular requests, so *libasync-smp* tends to run callbacks from the same request on the same CPU. This processor-callback affinity leads to greater cache hit rates and improved performance.

When *libasync-smp* starts, it adds a "select callback" to the run queue of the worker thread responsible for color zero. This callback calls `select()` to detect I/O events. The select callback enqueues callbacks in the appropriate queue based on which file descriptors `select()` indicates have become ready.

The select callback might block the worker thread that calls it if no file descriptors are ready; this would prevent one CPU from executing any other tasks in its work queue. To avoid this, the select callback uses `select()` to poll without blocking. If `select()` returns some file descriptors, the select callback adds callbacks for those descriptors to the work queue, and then puts itself back on the queue. If no file descriptors were returned, a *blocking* select callback is placed back on the queue instead. The blocking select callback is only run if it is the only callback on the queue, and calls `select()` with a non-zero timeout. In all other aspects, it behaves just like the non-blocking select callback.

The use of the two select callbacks along with work stealing guarantees that a worker thread never blocks in `select()` when there are callbacks eligible to be executed in the system.

Figure 5 shows the structure of a queue of runnable callbacks. In general, new runnable callbacks are added on the right, but `cpucb()` callbacks always appear to the left of I/O event callbacks. A worker thread's scheduler considers callbacks starting at the left. The scheduler examines the first few callbacks on the queue. If among these callbacks the scheduler finds a callback whose color is the same as the last callback executed on the worker thread, the scheduler runs that callback. Otherwise the scheduler runs the left-most eligible callback.

The scheduler favors `cpucb()` callbacks in order to increase the performance of chains of `cpucb()` callbacks from the same client request. The state used by a `cpucb()` callback is likely to be in cache because the creator of the `cpucb()` callback executed recently. Thus, early execution of `cpucb()` callbacks increases cache locality.

4 Implementation

libasync-smp is an extension of *libasync*, the asynchronous library [16] distributed as part of the SFS file system [17]. The library runs on Linux, FreeBSD and Solaris. Applications written for *libasync* work without modification with *libasync-smp*.

The worker threads used by *libasync-smp* to execute callbacks are kernel threads created by a call to the `clone()` system call (under Linux), `rfork()` (under FreeBSD) or `thr_create()` (under Solaris).

Although programs which use *libasync-smp* should not need to perform fine grained locking, the *libasync-smp* implementation uses spin-locks internally to protect its own data structures. The most important locks protect the callback run queues, the callback registration tables, retransmit timers in the RPC machinery, and the memory allocator.

The source code for *libasync-smp* is available as part of the SFS distribution at `http://www.fs.net` on the CVS branch mp-async.

5 Evaluation

In evaluating *libasync-smp* we are interested in both its performance and its usability. This section evaluates the parallel speedup achieved by two sample applications using *libasync-smp*, and compares it to the speedup achieved by existing similar applications. We also evaluate usability in terms of the amount of programmer effort required to modify existing event-driven programs to get good parallel speedup.

The two sample applications are the SFS file server and a caching web server. SFS is an ideal candidate for achieving parallel speedup using *libasync-smp*: it is written using *libasync* and performs compute intensive cryptographic tasks. Additionally, the SFS server maintains state that can not be replicated among independent copies of the server. A web server is a less promising candidate: web servers do little computation and all state maintained by the server can be safely shared. Accordingly we expect good SMP speedup from the SFS server and a modest improvement in performance from the web server.

All tests were performed on a SMP server equipped with four 500 MHz Pentium III Xeon processors. Each processor has 512KB of cache and the system has 512MB of main memory. The disk subsystem consists of a single ultra-wide 10,000 RPM SCSI disk. Load was generated by four fast PCs running Linux, each connected to the server via a dedicated full-duplex gigabit Ethernet link. Processor scaling results were obtained by completely disabling all but a certain number of processors on the server.

The server runs a slightly modified version of Linux kernel 2.4.18. The modification removes a limit of 128 on the number of new TCP connections the kernel will queue awaiting an application's call to `accept()`. This limit would have prevented good server performance with large numbers of concurrent TCP clients.

5.1 HTTP server

To explore whether we can use *libasync-smp* to achieve multiprocessor speedup in applications where the majority of computation is not concentrated in a small portion of the code, we measured the performance of an event-driven HTTP 1.1 web server.

The web server uses an NFS loop-back server to perform non-blocking disk I/O. The server process maintains two caches in its memory: a web page cache and a file handle cache. The former holds the contents of recently served web pages while the latter caches the NFS file handles of recently accessed files. The page cache is split into a small number (10) of independent caches to allow simultaneous access [6]. Both of the file handle cache and the individual page caches must be protected from simultaneous access.

5.1.1 Parallelizing the HTTP server

Figure 6 illustrates the concurrency present in the web server when it is serving concurrent requests for pages not in the cache. Each vertical set of circles represents a single callback, and the arrows connect successive callbacks involved in processing a request. Callbacks that can execute in parallel for different requests are indicated by multiple circles. For instance, the callback that reads an HTTP request from the client can execute in parallel with any other callback. Other steps involve access to shared mutable data such as the page cache; callbacks must execute serially in these steps.

When the server accepts a new connection, it colors the callback that reads the connection's request with its file descriptor number. The callback that writes the response back to the client is similarly colored. The shared caches are protected by coloring all operations that access a given cache the same color. Only one callback may access each cache simultaneously; however, two callbacks may access two distinct caches simultaneously (i.e. one request can read a page cache while another reads the file handle cache). The code that sends RPCs to the loop-back NFS server to read files is also serialized using a single color. This was necessary since the underlying RPC machinery maintains state about pending RPCs which could not safely be shared. The state maintained by the RPC layer is a candidate for protection via internal mutexes; if this state were protected within the library the "read file" step could be parallelized in the web server.

While this coloring allows the caches and RPC layer to operate safely, it reveals a limitation of coloring as a concurrency control mechanism. Ideally, we should allow any number of callbacks to read the cache, but limit the number of callbacks accessing the cache to one if the cache is being written. This read/write notion is not expressible with the current locking primitives offered by *libasync-smp* although they could be extended to include it [4]. We did not implement read/write colors since dividing the page cache into smaller, independent caches provided much of the benefit of read/write locks without requiring modifications to the library.

The server also delegates computation to additional CPUs using calls to `cpucb()`. When parsing a request the server looks up the longest match for the pathname in the file handle cache (which is implemented as a hash table). To move the computation of the hash function out of the cache color, we use a `cpucb()` callback to first hash each prefix of the path name, and then, in a callback running as the cache color, search for each hash value in the file handle cache.

In all, 23 callbacks were modified to include a color argument or to be invoked via a `cpucb()` (or both). The web server has 1,260 lines of code in total, and 39 calls to wrap.

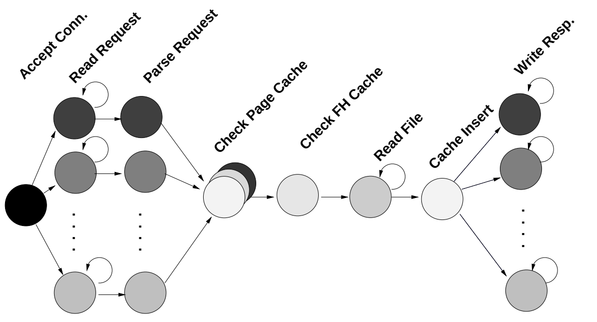

Figure 6: The sequence of callbacks executed when the *libasync-smp* web server handles a request for a page not in the cache. Nodes represent callbacks, arrows indicate that the node at the source scheduled the callback represented by the node at the tip. Nodes on the same vertical line are run under distinct colors (and thus potentially in parallel). The stacked circles in the "Check page cache" stage indicate that a small number of threads (less than the number of concurrent requests) can access the cache simultaneously). Labels at the top of the figure describe each step of the processing.

5.1.2 HTTP server performance

To demonstrate that the web server can take advantage of multiprocessor hardware, we tested the performance of the parallelized web server on a cache-based workload while varying the number of CPUs available to the server. The workload consisted of 720 files whose sizes were distributed according to the SPECweb99 benchmark [20]; the total size of the data set was 100MB which fits completely into the server's in-memory page cache. Four machines simulated a total of 800 concurrent clients. A single instance of the load generation client is capable of reading over 20MB/s from the web server. Each client made 10 requests over a persistent connection before closing the connection and opening a new one. The servers were started with cold caches and run for 4 minutes under load. The server's throughput was then measured for 60 seconds, to capture its behavior in the steady state.

Figure 7 shows the performance (in terms of total throughput) with different numbers of CPUs for the *libasync-smp* web server. Even though the HTTP server has no particularly processor-intensive operations, we can still observe noticeable speedup on a multiprocessor system: the server's throughput is 1.28 times greater on two CPUs than it is on one and 1.5 times greater on four CPUs.

To provide an upper bound for the multiprocessor speedup we can expect from the *libasync-smp*-based web server we contrast its performance with N inde-

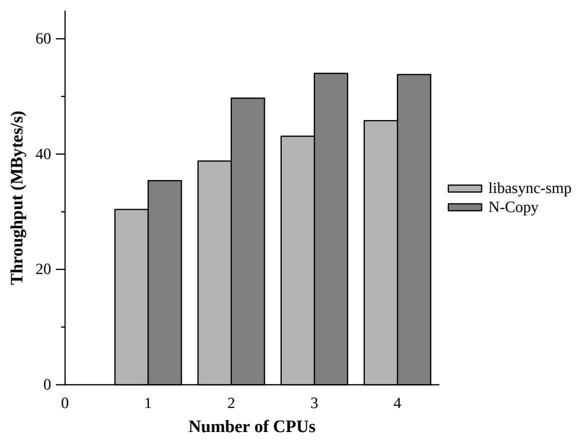

Figure 7: The performance of the *libasync-smp* web server serving a cached workload and running on different number of CPUs relative to the performance on one CPU (light bars). The performance of N copies of a *libasync* web server is also shown relative the performance of the the *libasync* server's performance on one CPU (dark bars)

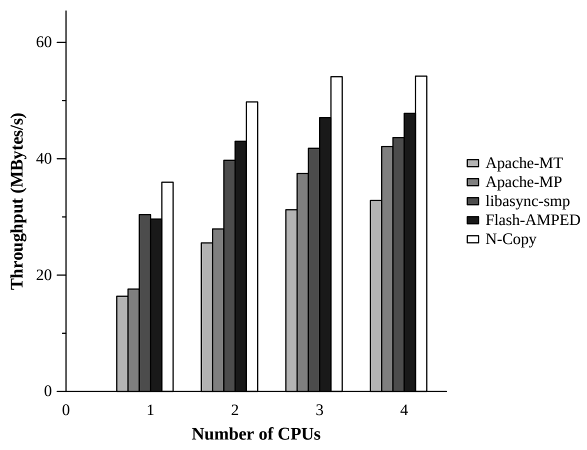

Figure 8: The performance of several web servers on multiprocessor hardware. Shown are the throughput of the *libasync-smp* based server (light bars), Apache 2.0.36 (dark bars), and Flash (black bars) on 1,2,3 and 4 processors.

pendent copies of a single process version of the web server (where N is the number of CPUs provided to the *libasync-smp*-based server). This single process version is based on an unmodified version of *libasync* and thus does not suffer the overhead associated with the *libasync-smp* library (callback queue locking, etc). Each copy of the N-copy server listens for client connections on a different TCP port number.

The speedup obtained by the *libasync-smp* server is well below the speedup obtained by N copies of the *libasync* server. Even on a single CPU, the *libasync* based server achieved higher throughput than the *libasync-smp* server. The throughput of the *libasync* server was 35.4 MB/s while the *libasync-smp* server's throughput was 30.4 MB/s.

Profiling the single CPU case explains the base penalty that *libasync-smp* incurs. While running the *libasync-smp* web server under load, roughly 35% of the CPU time is spent in user-level including *libasync-smp* and the web server. Of that time, at least 37% is spent performing tasks needed only by *libasync-smp*. Atomic reference counting uses 26% of user-level CPU time, and task accounting such as enqueuing and dequeuing tasks takes another 11%. The overall CPU time used for atomic reference counting and task management is 13%, which explains the *libasync-smp* web server's decreased single CPU performance.

The reduced performance of the *libasync-smp* server is partly due to the fact that many of the *libasync-smp* server's operations must be serialized, such as accepting connections and checking caches. In the N-copy case, all of these operations run in parallel. In addition, locking overhead penalizes the *libasync-smp* server: some data is necessarily shared across threads and must be protected by expensive atomic operations although the server has been written in such a way as to minimize such sharing.

Because the N-copy server can perform all of these operations in parallel and, in addition, extract additional parallelism from the operating system which locks some structures on a per-process basis, the performance of the N-copy server represents a true upper bound for any architecture which operates in a single address space.

To provide a more realistic performance goal than the N-copy server, we compared the *libasync-smp* server with two commonly used HTTP servers. Figure 8 shows the performance of Apache 2.0.36 (in both multithreaded and multiprocess mode) and Flash v0.1_990914 on different numbers of processors. Apache in multiprocess mode was configured to run with 32 servers. Apache-MT is a multithreaded version of the Apache server. It creates a single heavyweight process and 32 kernel threads within that process by calling `clone`. The number of processes and threads used by the Apache servers were chosen to maximize throughput for the benchmarks presented here. Flash is an event-driven server; when run on multiprocessors it forks to create N independent copies, where N is the number of available CPUs

The performance of the *libasync-smp* HTTP server is comparable to the performance of these servers: the *libasync-smp* server shows better absolute performance than both versions of the Apache server and slightly lower performance than N-copies of the Flash server.

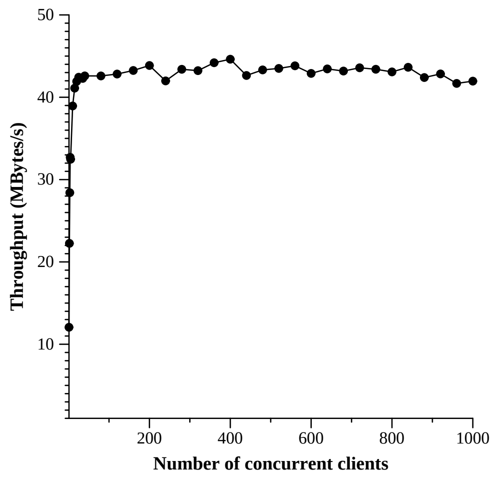

Figure 9: The performance of the web server on a cached workload as the number of concurrent clients is varied.

These servers show better speedup than the *libasync-smp* server: Flash achieves 1.68 speedup on four CPUs while the *libasync-smp* server is 1.5 times faster on four CPUs. Because Flash runs four heavyweight processes, it is able to take advantage of many of the benefits of the N-copy approach: as a result its speedup and absolute performance are greater than that of the *libasync-smp* server. Although this approach is workable for a web server, in applications that must coordinate shared state such replication would be impossible.

Like the *libasync-smp* server, Flash and multiprocess Apache do not show the same performance achieved by the N-copy server. Although these servers fully parallelize access to their caches and do not perform locking internally, they do exhibit some shared state. For instance, the servers must serialize access to the `accept()` system call since all requests arrive on a single TCP port.

The main reason to parallelize a web server is to increase its performance under heavy load. A key part of the ability to handle heavy load is stability: non-decreasing performance as the load increases past the server's point of peak performance. To explore whether servers based on *libasync-smp* can provide stable performance, we measured the web server's throughput with varying numbers of simultaneous clients. Each client selects a file according to the SPECweb99 distribution; the files all fit in the server's cache. The server uses all four CPUs. Figure 9 shows the results. The event-driven HTTP server offers consistent performance over a wide variety of loads.

5.2 SFS server

To evaluate the performance of *libasync-smp* on existing *libasync* programs, we modified the SFS file server [17] to take advantage of a multiprocessor system.

The SFS server is a single user-level process. Clients communicate with it over persistent TCP connections. All communication is encrypted using a symmetric stream cipher, and authenticated with a keyed cryptographic hash. Clients send requests using an NFS-like protocol. The server process maintains significant mutable per-file-system state, such as lease records for client cache consistency. The server performs non-blocking disk I/O by sending NFS requests to the local kernel NFS server. Because of the encryption, the SFS server is compute-bound under some heavy workloads and therefore we expect that by using *libasync-smp* we can extract significant multiprocessor speedup.

5.2.1 Parallelizing the SFS server

We used the `pct`[5] statistical profiler to locate performance bottlenecks in the original SFS file server code. Encryption appeared to be an obvious target, using 75% of CPU time. We modified the server so that encryption operations for different clients executed in parallel and independently of the rest of the code. The resulting parallel SFS server spent about 65% of its time in encryption. The reduction from 75% is due to the time spent coordinating access to shared mutable data structures inside *libasync-smp*, as well as to additional memory-copy operations that allow for parallel execution of encryption.

The modifications to the SFS server are concentrated in the code that encrypts, decrypts, and authenticates data sent to and received from the clients. We split the main send callback-function into three smaller callbacks. The first and last remain synchronized with the rest of the server code (i.e. have the default color), and copy data to be transmitted into and out of a per-client buffer. The second callback encrypts the data in the client buffer, and runs in parallel with other callbacks (i.e., has a different color for each client). This involved modifying about 40 lines of code in a single callback, largely having to do with variable name changes and data copying.

Parallelization of the SFS server's receive code was slightly more complex because more code interacts with it. About 50 lines of code from four different callbacks were modified, splitting each callback into two. The first of these two callbacks received and decrypted data in parallel with other callbacks (i.e., with a different color for every client), and used `cpucb()` to execute the second callback. The second callback remained synchronized with the rest of the server code (i.e., had the de-

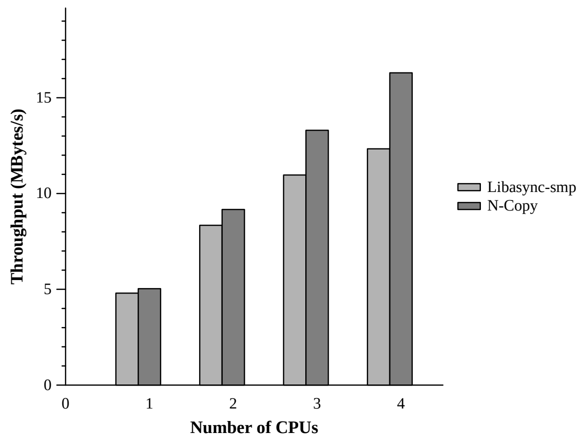

Figure 10: Performance of the SFS file server using different numbers of CPUs, relative to the performance on one CPU. The light bars indicate the performance of the server using *libasync-smp*; dark bars indicate the performance of n separate copies of the original server. Each bar represents the average of three runs; the variation from run to run was not significant.

fault color), and performed the actual processing of the decrypted data.

5.2.2 Performance improvements

We measured the total throughput of the file server to all clients, in bits per second, when multiple clients read a 200 MByte file whose contents remained in the server's disk buffer cache. We repeated this experiment for different numbers of processors. This test reflects how SFS is used in practice: an SFS client machine sends all of its requests over a single TCP connection to the server.

The bars labeled "libasync-smp" in Figure 10 show the performance of the parallelized SFS server on the throughput test. On a single CPU, the parallelized server achieves 96 percent of the throughput of the original uniprocessor server. The parallelized server is 1.66, 2.20, and 2.5 times as fast as the original uniprocessor server on two, three and four CPUs, respectively.

Because only 65% of the cycles (just encryption) have been parallelized, the remaining 35% creates a bottleneck. In particular, when the remaining 35% of the code runs continuously on one processor, we can achieve a maximum utilization of $\frac{1}{0.35} = 2.85$ processors. This number is close to the maximum speedup (2.5) of the parallelized server. Further parallelization of the SFS server code would allow it to incrementally take advantage of more processors.

To explore the performance limits imposed by the hardware and operating system, we also measured the total performance of multiple independent copies of the original *libasync* SFS server code, as many separate processes as CPUs. In practice, such a configuration would not work unless each server were serving a distinct file system. An SFS server maintains mutable per-file-system state, such as attribute leases, that would require shared memory and synchronization among the server processes. This test thus gives an upper bound on the performance that SFS with *libasync-smp* could achieve.

The results of this test are labeled "N-copy" in Figure 10. The SFS server with *libasync-smp* roughly follows the aggregate performance of multiple independent server copies. The performance difference between the *libasync-smp*-based SFS server and the N-copy server is due to the penalty incurred due to shared state maintained by the server, such as file lease data and user ID mapping tables.

Despite comparatively modest changes to the SFS server to expose parallelism, the server's parallel performance was close to the maximum speedup offered by the underlying operating system (as measured by the speedup obtained by multiple copies of the server).

5.3 Library Optimizations

Table 2 shows how much the use of per-thread work queues improves performance. The numbers in the table indicate how fast a synthetic benchmark executes tasks. The benchmark program creates 16 callbacks with unique colors. Each callback performs a small amount of computation, and then registers a child callback of the same color. The benchmark intentionally assigns colors so that all but one of the task queues are populated, in order to explore the effects of work stealing. The benchmark was run with four CPUs.

The first line shows the task rate with a single task queue shared among all the worker threads. The entry shows the task completion rate when using per-thread task queues. The increase in task completion rate is dramatically higher due to better cache locality, and because there is no contention for the task-queue locks. The third line shows the task completion rate when per-thread task free-lists are used in addition to per-thread queues. The fourth configuration adds work stealing between worker threads. Without work stealing, tasks were never run on one of the four CPUs. Work stealing allows the worker thread on that CPU to find work, at the expense of increased contention for the other threads' task queues.

Library Configuration	Tasks/sec
Base	61420
+ Per-thread Queues	240618
+ Per-thread Task Object Freelists	293997
+ Work Stealing	384765

Table 2: A synthetic benchmark shows improved task processing rates as thread affinity optimizations are added.

6 Related Work

There is a large body of work exploring the relative merits of thread-based I/O concurrency and the event-driven architecture [18, 11, 12, 15, 1]. This paper does not attempt to argue that either is superior. Instead, we present a technique which improves the performance of the event-driven model on multiprocessors. The work described below also considers performance of event-driven software.

Pai et al. characterized approaches to achieving concurrency in network servers in [19]. They evaluate a number of architectures: multi-process, multi-threaded, single-process event-driven, and asymmetric multi-process event-driven (AMPED). In this taxonomy, *libasync-smp* could be characterized as symmetric multi-threaded event-driven; its main difference from AMPED is that its goal is to increase CPU concurrency rather than I/O concurrency.

Like *libasync-smp*, the AMPED architecture introduces limited concurrency into an event driven system. Under the AMPED architecture, a small number of helper processes are used to handle file I/O to overcome the lack of non-blocking support for file I/O in most operating systems. In contrast, *libasync-smp* uses additional execution contexts to execute callbacks in parallel. *libasync-smp* achieves greater CPU concurrency on multiprocessors when compared to the AMPED architecture but places greater demands on the programmer to control concurrency. Like the AMPED-based Flash web server, *libasync-smp* must also cope with the issue of non-blocking file I/O: *libasync-smp* uses an NFS-loopback server to access files asynchronously. This allows *libasync-smp* to use non-blocking local RPC requests rather than blocking system calls.

The Apache web server serves concurrent requests with a pool of independent processes, one per active request [3]. This approach provides both I/O and CPU concurrency. Apache processes cannot easily share mutable state such as a page cache.

The staged, event-driven architecture (SEDA) is a structuring technique for high-performance servers [24]. It divides request processing into a series of well-defined stages, connected by queues of requests. Within each stage, one or more threads dequeue requests from input queue(s), perform that stage's processing, and enqueue the requests for subsequent stages. A thread can block (to wait for disk I/O, for example), so a stage often contains multiple threads in order to achieve I/O concurrency.

SEDA can take advantage of multiprocessors, since a SEDA server may contain many concurrent threads. One of SEDA's primary goals is to dynamically manage the number of threads in each stage in order to achieve good I/O and CPU concurrency but avoid unstable behavior under overload. Both *libasync-smp* and SEDA use a mixture of events and concurrent threads; from a programmer's perspective, SEDA exposes more thread-based concurrency which the programmer may need to synchronize, while *libasync-smp* tries to preserve the serial callback execution model.

Cohort scheduling organizes threaded computation into stages in order to increase performance by increasing cache locality, reducing TLB pressure, and reducing branch mispredicts [14]. The staged computation model used by cohort scheduling is more general than the colored callback model presented here. However, the partitioned stage scheduling policy is somewhat analagous to coloring callbacks for parallel execution (the key corresponds to a callback color). Like SEDA, cohort scheduling exposes more thread-based concurrency to the programmer. Cohort scheduling can also take advantage of multiprocessor hardware.

7 Conclusion

This paper describes a library that allows event-driven programs to take advantage of multiprocessors with a minimum of programming effort. When high loads make multiple events available for processing, the library can execute event handler callbacks on multiple CPUs. To control the concurrency between events, the programmer can specify a *color* for each event: events with the same color (the default case) are handled serially; events with different colors can be handled in parallel. The programmer can incrementally expose parallelism in existing event-driven applications by assigning different colors to computationally-intensive events that don't share mutable state.

Experience with *libasync-smp* demonstrates that applications can achieve multi-processor speedup with little programming effort. Parallelizing the cryptography in the SFS file server required about 90 lines of changed code in two modules, out of a total of about 12,000 lines. Multiple clients were able to read large cached files from the *libasync-smp* SFS server running on a 4-CPU machine 2.5 times as fast as from an unmodified uniprocessor SFS server on one CPU. Applications without computationally intensive tasks also benefit: an event-driven Web server achieves 1.5 speedup on four CPUs with multiple clients reading small cached files relative to its performance on one CPU.

Acknowledgments

We are grateful to the many people who aided this work. The NMS group at LCS provided (on short notice) the SMP server used to test an early version of this software. The anonymous reviewers and our shepherd, Edouard Bugnion, provided helpful feedback.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8927.

References

[1] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W., AND DOUCEUR, J. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proc. Usenix Technical Conference* (June 2002).

[2] ANDERSON, T., BERSHAD, B., LAZOSWKA, E., AND LEVY, H. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 53–79.

[3] The Apache web server. http://www.apache.org/, 2002.

[4] BIRRELL, A. An introduction to programming with threads. Tech. Rep. 35, Digital Systems Research Center, Jan. 1989.

[5] BLAKE, C., AND BAUER, S. Simple and general statistical profi ling with PCT. In *Proc. Usenix Technical Conference* (June 2002).

[6] BLASGEN, M., GRAY, J., MITOMA, M., AND PRICE, T. The convoy phenomenon. *Operating Systems Review 13*, 2 (Apr. 1979), 20–25.

[7] BLOOM, J., AND DUNLAP, K. Experiences implementing BIND, a distributed name server for the DARPA Internet. In *Proc. Summer Usenix Conference* (1986), pp. 172–181.

[8] BOVET, D., AND CESATI, M. *Understanding the Linux Kernel*. O'Reilly, 2001.

[9] BURTON, F., AND SLEEP, M. Executing functional programs on a virtual tree of processsors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (Portsmouth, New Hampshire, Oct. 1981).

[10] DABEK, F., KAASHOEK, F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (Banff, Canada, Oct. 2001), pp. 202–215.

[11] DRAVES, R., BERSHAD, B., RASHID, R., AND DEAN, R. Using continuations to implement thread management and communication in operating systems. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 1991), pp. 122–136.

[12] FORD, B., HIBLER, M., LEPREAU, J., MCGRATH, R., AND TULLMANN, P. Interface and execution models in the Fluke kernel. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Feb. 1999), pp. 101–115.

[13] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. Fast and secure distributed read-only fi le system. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2000), pp. 181–196.

[14] LARUS, J., AND PARKES, M. Using cohort scheduling to enhance server performance. In *Proc. Usenix Technical Conference* (June 2002).

[15] LAUER, H., AND NEEDHAM, R. On the duality of operating system structures. In *Proc. Second International Symposium on Operating Systems, IRIA* (Oct. 1978). Reprinted in Operating Systems Review, Vol. 12, Number 2, April 1979.

[16] MAZIÈRES, D. A toolkit for user-level fi le systems. In *Proc. Usenix Technical Conference* (June 2001), pp. 261–274.

[17] MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from fi le system security. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (Kiawah Island, South Carolina, Dec. 1999).

[18] OUSTERHOUT, J. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX technical conference, 1996.

[19] PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An effi cient and portable web server. In *Proc. Usenix Technical Conference* (June 1999).

[20] SPECweb99 design white paper. http://www.specbench.org/osg/web99/docs/whitepaper.html, 2002.

[21] STEELE, G., AND SUSSMAN, G. Lambda: The ultimate imperative. Tech. Rep. AI Lab Memo AIM-353, MIT AI Lab, Mar. 1976.

[22] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference* (San Diego, Aug. 2001).

[23] THOMPSON, S. *Haskell, The Craft of Functional Programming*. Addison Wesley, 1996.

[24] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2001), pp. 230–243.