

Concurrency Control for Multi-Processor Event-Driven Systems

by

Nickolai Zeldovich

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

© Nickolai Zeldovich, MMII. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis document
in whole or in part.

Author
Department of Electrical Engineering and Computer Science
May 23, 2002

Certified by.....
Robert Morris
Assistant Professor
Thesis Supervisor

Accepted by.....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Concurrency Control for Multi-Processor Event-Driven Systems

by

Nickolai Zeldovich

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes *libasync-mp*, an extension of the *libasync* asynchronous programming library that allows event-driven applications to take advantage of multi-processors by running code for event handlers in parallel. To control 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. Parallelism in existing event-driven applications can be incrementally exposed by assigning different colors to computationally-intensive events that don't share mutable state.

An evaluation of *libasync-mp* shows that applications achieve multi-processor speedup with little programming effort. For example, parallelizing the cryptography in the SFS file server required about 90 lines of changed code in two modules, out of a total of 12,000 lines. Multiple clients were able to read files from this SFS server running on a 4-CPU machine 2.55 times faster than from an unmodified SFS server on one CPU.

Thesis Supervisor: Robert Morris
Title: Assistant Professor

Acknowledgements

I am indebted to my advisor, Robert Morris, for providing the initial impetus for this work and continued suggestions about its implementation.

Thanks to Frank Dabek for his advice and discussions about the design and implementation of this system. Also for his help in benchmarking and evaluating this software.

Most of the work described in this thesis is also presented in “Multiprocessor Support for Event-Driven Programs”, submitted to OSDI 2002. I’m grateful to Frans Kaashoek, Robert, Frank, and David Mazières for their help in writing that paper.

I acknowledge the NMS group at LCS for providing, on short notice, the four-way SMP server used to obtain performance results for this software.

Contents

1	Introduction	7
1.1	Thesis Overview	8
2	Uniprocessor Event-driven Design	9
2.1	<i>libasync</i>	10
2.2	Event-driven Programming	11
3	Multiprocessor Design	15
3.1	Coordinating callbacks	16
3.2	<i>libasync-mp</i> API	18
3.3	Example	19
3.4	Scheduling callbacks	20
4	Implementation	23
5	Evaluation	25
5.1	HTTP server	26
5.1.1	Parallelizing the HTTP server	26
5.1.2	HTTP server performance	28
5.2	SFS server	31
5.2.1	Parallelizing the SFS server	32
5.2.2	Performance improvements	32
6	Related Work	35
7	Conclusions and Future Work	37
7.1	Future Work	37
7.2	Conclusion	38

Chapter 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. Event-based programs are structured as a collection of *callback* functions which a main loop calls when I/O events occur. Threads provide an intuitive programming model, but require coordination of accesses by different threads to shared state, even on a uniprocessor. 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 advantage of multiprocessors.

The contribution of this thesis is *libasync-mp*, an extension of the *libasync* event-driven library [11, 12] that supports event-driven programs on multiprocessors. *libasync-mp* is intended to support the construction of user-level systems programs, particularly network servers and clients; this thesis demonstrates that these applications can achieve performance gains on multi-processors by exploiting coarse-grained parallelism. *libasync-mp* is intended for programs that have natural opportunities for parallel speedup; it has no support for expressing very fine-grained parallelism.

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-mp* provides a simple mechanism to allow the programmer to incrementally add parallelism to uni-processor 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-mp* 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-mp is based on the *libasync* library. *libasync* uses operating system asynchronous I/O facilities to support event-based programs on uniprocessors. The modifications for *libasync-mp* 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-mp* demonstrates that applications achieve multi-processor speedup with little programming effort. As an example, the SFS [12] file server was modified to use *libasync-mp*. This server uses more than 320 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 2.55 times as fast as an unmodified uniprocessor SFS server on one CPU.

Even servers without cryptography can achieve modest speedup, 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 1.54 speedup on four CPUs.

1.1 Thesis Overview

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

Chapter 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 the arrival of network data, a new client connection, completion of disk I/O, or a mouse click, for example. 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 [4], 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). Using an event-driven architecture for such servers allows the servers to keep state for many concurrent activities.

Event-driven programs typically use a library to support the management of events. The 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 much 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 [12, 11]. This sec-

tion describes uniprocessor *libasync* and the programming style involved in using it. Existing systems, such as named [4] and Flash [15], 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 UNIX¹ 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 [11]. 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. For each event, the main loop calls the registered callback. Multiple independent modules can use *libasync* without knowing about each other, which encourages modular design and re-usable 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, data arriving on file descriptors, timer expiration, and UNIX signals. The RPC utility module allows automatic parsing of incoming Sun RPC calls and dispatch to callbacks registered per program/procedure pair. The RPC module also allows a callback to be registered to handle the arrival of the reply to a particular RPC call. Other utility modules initiate activities such as starting a DNS host name lookup, and calling a callback when the activity completes. 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-carrying [19] in the form of the `wrap()` macro [11]. `wrap(fn, x, y)` constructs an anonymous function called a *wrap*. When the wrap is called with

¹*libasync* also runs on Windows using cygwin.

callback wrap ((func *)(), arg1, ..., argN)	Create a callback object
fdcb (int fd, bool read, callback cb)	Call cb when fd is readable/writable
sigcb (int sig, callback cb)	Call cb when signal sig is received
timecb (timespec t, callback cb)	Call cb at time t
amain ()	Start the event loop and poll for events

Table 2.1: Core API for *libasync*

(for example) argument `z`, the wrap calls `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

The core API for *libasync* is shown in Table 2.1, and Figure 2-1 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

```

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 2-1: Outline of a web proxy that uses *libasync*.

Name	#Wraps	Lines of Code
SFS [12]	100	12000
SFSRO [10]	77	7619
Chord [18]	50	5278
CFS [6]	75	3283

Table 2.2: Applications based on *libasync*, along with the approximate number of distinct wraps in each application. The numbers are exclusive of the wraps created by *libasync* itself, which number about 60.

client connections.

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 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 [17], and makes it easy for programmers to compose modules.

A number of applications are based on *libasync*; Table 2.2 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.

Chapter 3

Multiprocessor Design

The focus of this paper is *libasync-mp*, a multiprocessor extension of *libasync*. The goal of *libasync-mp* is to execute event-driven programs faster by running callbacks on multiple CPUs. Much of the design of *libasync-mp* is motivated by the desire to make it easy to adapt existing *libasync*-based servers to multiprocessors.

A server based on *libasync-mp* consists of a single process containing multiple worker threads, one per available CPU. *libasync-mp* maintains a queue of callbacks that need to be run, and each worker thread repeatedly dequeues the next callback and executes it. The worker threads are scheduled by the kernel across multiple CPUs and 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 [1] could be used to dynamically determine the number of available CPUs.

An alternate design might be to run multiple independent copies of an event-driven program on a multiprocessor, one per CPU. This approach might work in the case of a web server, since the processing of different client requests can be made

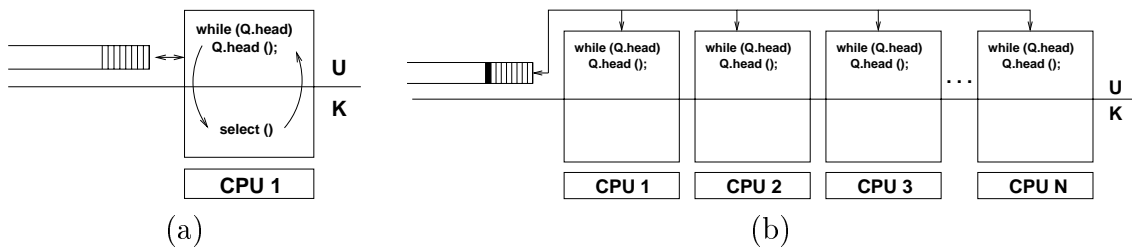


Figure 3-1: The single process event driven architecture (left) and the multiprocess event driven architecture (right).

independent. This approach does not work if the program maintains mutable state that is shared among all 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 decreases in efficiency: a web proxy might maintain a cache of recently accessed pages in main memory. Multiple copies of the proxy could maintain independent caches, but content duplicated in these caches would waste memory. *libasync-mp* allows a single copy of the application to achieve performance improvements similar to those achieved by running multiple copies of an application while protecting access to shared data structures.

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-mp* is motivated by two observations. First, most system software has natural coarse-grain parallelism, because typically client requests are not dependent on each other or each request has a sequence of independent states of processing. 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-mp 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, and their semantics resemble wait channels [13] in many ways. 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-mp*.

Event-driven systems do not inherently preclude the use of other concurrency control mechanisms, such as the conventional mutex locks. However, mutex locks

are almost always blocking: an attempt to obtain a lock held by some other code results in suspension of execution until the lock is available. This property of mutex locks is contrary to the notion of short-running, non-blocking event callbacks: a callback that uses mutex locks for synchronization can block for long periods of time waiting to obtain a mutex lock. Colors avoid callback blocking by exposing the concurrency information to the scheduler, which in turn can make optimal decisions about execution order.

Thread-oriented concurrency control mechanisms, such as mutex locks, are often hard to reason about, and therefore difficult to implement correctly. Engler *et al.* [8] have shown that programming errors associated with mutex locks are common in the Linux kernel, and Savage *et al.* [16] have shown that finding and debugging such problems is very difficult. The coloring concurrency control mechanism is easier to reason about, because colors are applied to existing self-contained units of execution which are often associated with single, well-defined tasks. Because callbacks have a fixed coloring, there are no possible deadlock conditions; the worst case is sequential execution on a single processor, when all callbacks are of the same color. Experience shows that it is easy to correctly color callbacks in applications using *libasync-mp* in order to take advantage of multiple processors.

The fact that color can be applied to a callback almost orthogonally to the callback's code makes it easy to adapt 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, choose a new unique color for the connection and apply that color to the entire sequence of callbacks that process that connection. If a particular stage in request processing shares mutable data among requests (e.g. a cache of web pages), choose a color for that stage and apply 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 modified. This arises when a single callback uses shared data but also has significant computation that does not use shared data. In that case it is typical to split the callback; the first half then uses a special *libasync-mp* call (`cpucb()`) to schedule the second half with a different color.

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 du-

callback <code>cwrap</code> ((func *)(), arg1, ..., argN, Color color)	Create a callback object with the given color.
callback <code>cpwrap</code> ((func *)(), arg1, ..., argN, Color color, int prio)	Create a callback object with the given color and priority.
void <code>cpucb</code> (callback F)	Add <code>cb</code> to the runnable callback queue.

Table 3.1: Additional calls in the *libasync-mp* API.

ration of a callback. However, experience suggests that fine-grained and sophisticated locking, while it may be necessary for correctness with concurrent threads, rarely is 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-mp* API

The API that *libasync-mp* presents differs slightly from that exposed by *libasync*. The `wrap` function described in Section 2 is extended by the `cwrap` function. The new `cwrap` function takes an additional color argument; Table 3.1 shows the prototype for `cwrap`. The color specified at the callbacks 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.

The *libasync-mp* API also includes an optional priority level for callbacks; the `cpwrap` function takes an additional priority value that will be associated with the newly created callback. Priority levels are a hint to the scheduler about the order in which callbacks should be executed; larger priority values indicate a callback that should be executed preferentially over callbacks with lower priority values. Because priority level information is just a hint to the scheduler, applications may not depend on any particular execution order of callbacks based on their priorities. Priority levels can be used to improve overall throughput of the system; their use is described in more detail in Section 3.4.

libasync-mp provides a `cpucb()` function that takes a callback as an argument and puts that that callback directly onto the runnable callback queue. This function can be used to register a callback with a color different from that of the currently executing callback. As commonly used, the `cpucb()` function allows a programmer to split a CPU-intensive callback in two callbacks. One of these callbacks performs computation while the other synchronizes with shared state. To minimize programming errors associated with splitting an existing callback into a chain of `cpucb()` callbacks, *libasync-mp* 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 was found to use a large amount of CPU time and to 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 3-2 shows this change to `req_cb()`. In this example only the `parse_request()` workload is distributed across CPUs. As a further revision, reading requests could be parallelized by adding color arguments to the `fdcb()` calls which register the read request callback.

```

// 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_request_cb(int fd, str inBuf)
{
    parse_request(inBuf, serverName, file);

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

```

Figure 3-2: Changes to the asynchronous web proxy to take advantage of multiple CPUs

3.4 Scheduling callbacks

Each *libasync-mp* worker thread uses a simple scheduler to choose a callback to execute next from the queue. The scheduler considers color restrictions, callback/CPU affinity, and programmer-specified priority level hints. Its design is loosely based on that of the Linux SMP kernel [5].

Figure 3-3 shows the structure of the 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 skips over callbacks whose color makes them not eligible to run; a shared table records the colors of callbacks currently running on other CPUs. When choosing a callback to execute, the worker thread examines the first N eligible callbacks on the queue, and assigns each callback *cb* a weight, computed



Figure 3-3: The callback queue structure in *libasync-mp*. `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.”

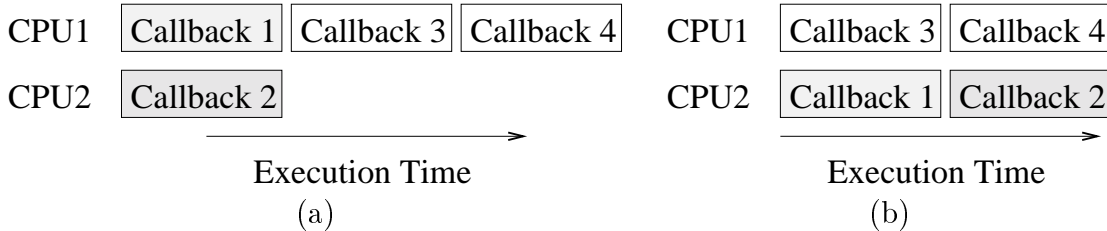


Figure 3-4: Scheduling of callbacks across CPUs, without the use of priorities (left) and with programmer-assigned priority levels (right).

as $w(cb) = cb.priority + (cb.color == last_color ? 1 : 0)$, where *last_color* is the color of the last callback to execute in this worker thread. The worker thread chooses the callback with the highest weight; in case of a tie, the left-most callback is chosen. The parameter *N* is chosen to be a small integer to prevent a linear scan of all runnable tasks.

Different priority levels can be used to increase throughput in situations where a certain color accounts for a much larger fraction of processing time than other colors. For example, suppose the callback queue contains callbacks 1 through 4; callbacks 1 and 2 have colors 1 and 2 respectively, and callbacks 3 and 4 are colored zero. Assume all four callbacks take the same amount of time to execute. Then, in the absence of priority information, callbacks will be executed as shown in Figure 3-4 (a), taking 3 time units to complete. If the programmer specifies a higher priority for zero-colored callbacks (3 and 4), the scheduler will choose to execute callbacks 3 and 4 over callbacks 1 and 2 when possible, as shown in Figure 3-4 (b). This allows the callbacks to complete execution in 2 time units instead of 3.

The priority technique is useful for improving throughput of incrementally parallelized applications. If only a small number of callbacks in the system have been parallelized (colored), then the priority of zero-colored callbacks can be increased to result in better throughput, as demonstrated in Figure 3-4.

The scheduler favors callbacks of the same color as the last to execute on the current CPU in order to increase performance. Callback colors often correspond to particular requests, so *libasync-mp* 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.

The reason that the scheduler favors `cpucb()` callbacks is 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.

Chapter 4

Implementation

libasync-mp is an extension of *libasync*, the asynchronous library [11] distributed as part of the SFS file system [12]. The library runs on Linux, FreeBSD and Solaris. Applications written for *libasync* work without modification with *libasync-mp*.

The worker threads used by *libasync-mp* 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). When a the work queue is empty, a worker thread suspends itself by calling `poll()` on a special pipe file descriptor; when another worker thread puts callbacks on the work queue, it writes dummy data to the pipes of waiting workers.

When *libasync-mp* starts, it adds a “select callback” to the run queue whose job is to call `select()` to detect I/O events. The select callback enqueues callbacks 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 tasks in the 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 two select callbacks guarantees that no worker threads block in `select()` as long as there are callbacks eligible to be executed.

Although programs which use *libasync-mp* should not need to perform fine grained

locking, the *libasync-mp* implementation uses spin-locks internally to protect its own data structures. The most important locks protect the callback run queue, the callback registration tables, and the memory allocator. The reference-counting garbage collector uses atomic increment/decrement instructions.

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

Chapter 5

Evaluation

In evaluating *libasync-mp* we are interested in both its performance and its usability. This section evaluates the parallel speedup achieved by two sample applications using *libasync-mp*, 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-mp*: 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 700 MHz Pentium III Xeon processors. Each processor has 1MB of cache and the system has 1 GB 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-mp* to achieve multi-processor 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. We expect the speedup achieved by this application to be minimal: this experiment represents the baseline performance increases one can expect to achieve using *libasync-mp*.

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. Both of these structures must be protected from simultaneous access.

5.1.1 Parallelizing the HTTP server

Figure 5-1 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 the 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

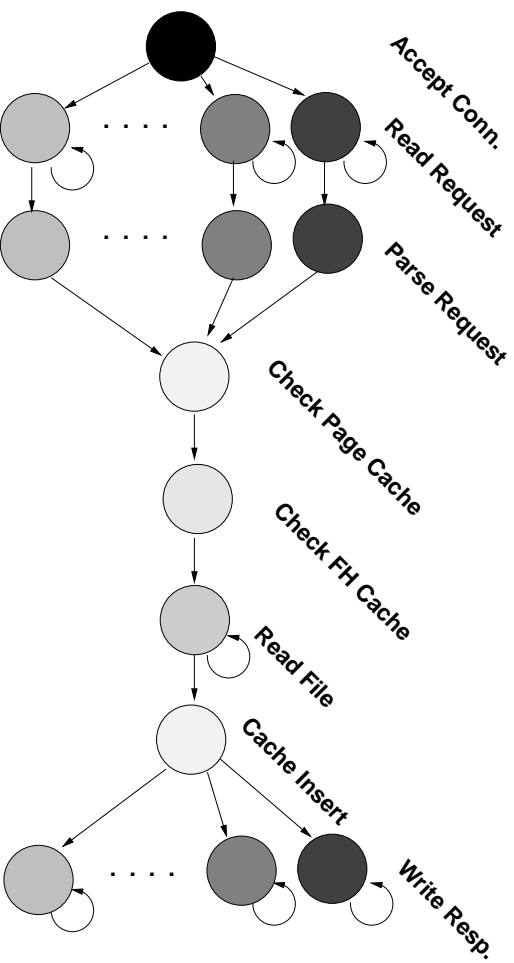


Figure 5-1: The sequence of callbacks executed when the *libasync-mp* web server handles a request for a page not in the cache. Nodes represent callbacks, arrows indicate that the node at the source scheduled the callback represented by the node at the tip. Nodes on the same vertical line are run under distinct colors (and thus potentially in parallel). Labels at the top of the figure describe each step of the processing.

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-mp* although they could be extended to include it.

The server also splits some computation onto 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`.

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 700 4K files; these files fit 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 20MB/s from the web server. Each client made 5 requests over a persistent connection before closing the connection and opening a new one. The servers were started with cold caches and run for 30 seconds under load. The server's throughput was then measured for 20 seconds, to capture its behavior in the steady state.

Figure 5-2 shows the performance (in terms of total throughput) with different numbers of CPUs for the *libasync-mp* web server. Even though the HTTP server has no particularly processor-intensive operations, we can still observe noticeable speedup on a multi-processor system: the server's throughput is 1.36 times greater on two CPUs than it is on one and 1.54 times greater on four CPUs.

To provide an upper bound for the multiprocessor speedup we can expect from the *libasync-mp*-based web server we contrast its performance with N independent copies of a single process version of the web server (where N is the number of CPUs provided to the *libasync-mp*-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-mp* 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-mp* 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-mp* server. The throughput of the *libasync* server was 24.8 MB/s while the *libasync-mp* server's throughput was 22.8 MB/s.

The reduced performance of the *libasync-mp* server is partly due to the fact that many of the *libasync-mp* 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-mp* server. Because the server relies heavily on the reference counted garbage collection provided by *libasync*, it performs a large number (approximately 100 per request) of expensive atomic

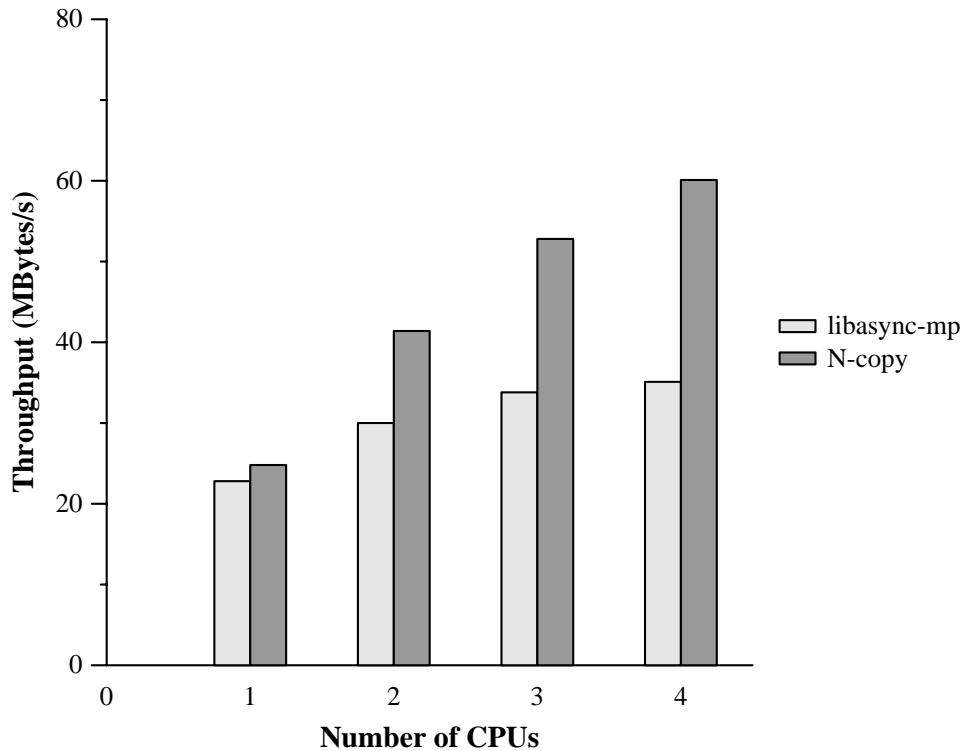


Figure 5-2: The performance of the *libasync-mp* web server serving a cached workload and running on different number of CPUs relative to the performance on one CPU (light bars). The performance of N copies of a *libasync* web server is also shown relative to the performance of the *libasync* server's performance on one CPU (dark bars)

increment and decrement instructions. Locking of shared structures (such as the callback queue) also adds overhead. Profiling revealed that up to 50 percent of total CPU time was spent acquiring mutexes or performing atomic increment/decrement operations when the server is run on four CPUs.

The speedup achieved by multiple copies of a web server represents an upper bound for possible speedup obtained by the *libasync-mp* server. To provide a more realistic performance goal, we compared the *libasync-mp* server with two commonly used HTTP servers. Figure 5-3 shows the performance of Apache 2.0.36 and Flash v0.1_990914 on different numbers of processors. Apache is a multi-process server: it was configured to run with at least 256 servers and up to 512. Flash is an event-driven server; when run on multiprocessors it forks to create N independent copies.

These servers show better absolute performance than the *libasync-mp* server. They also show better speedup than the *libasync-mp* server: Flash achieves 1.73 speedup on four CPUs while the *libasync-mp* server is 1.54 times faster on four CPUs. The

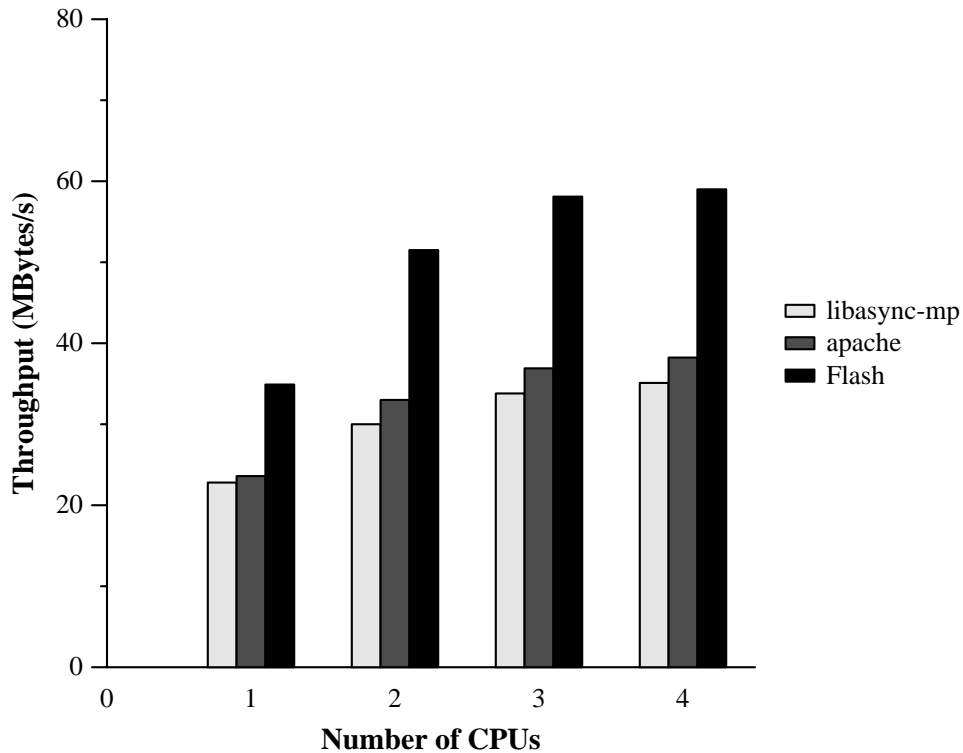


Figure 5-3: The performance several web servers on multiprocessor hardware. Shown are the throughput of the *libasync-mp* based server (light bars), Apache 2.0.36 (dark bars), and Flash (black bars) on 1,2,3 and 4 processors.

differences in multiprocessor speedup as well as absolute performance are due to heavy use of atomic operations.

Like the *libasync-mp* server, Flash and Apache do not show the same speedup achieved by the N-copy server which is 2.50 times faster on four CPUs than on one. 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-mp* can provide stable performance, we measured the web server's throughput with varying numbers of simultaneous clients. Each client repeatedly requests a randomly chosen 4KByte file; the files all fit in the server's cache. Figure 5-4 shows the results. As we expect, the event-driven HTTP

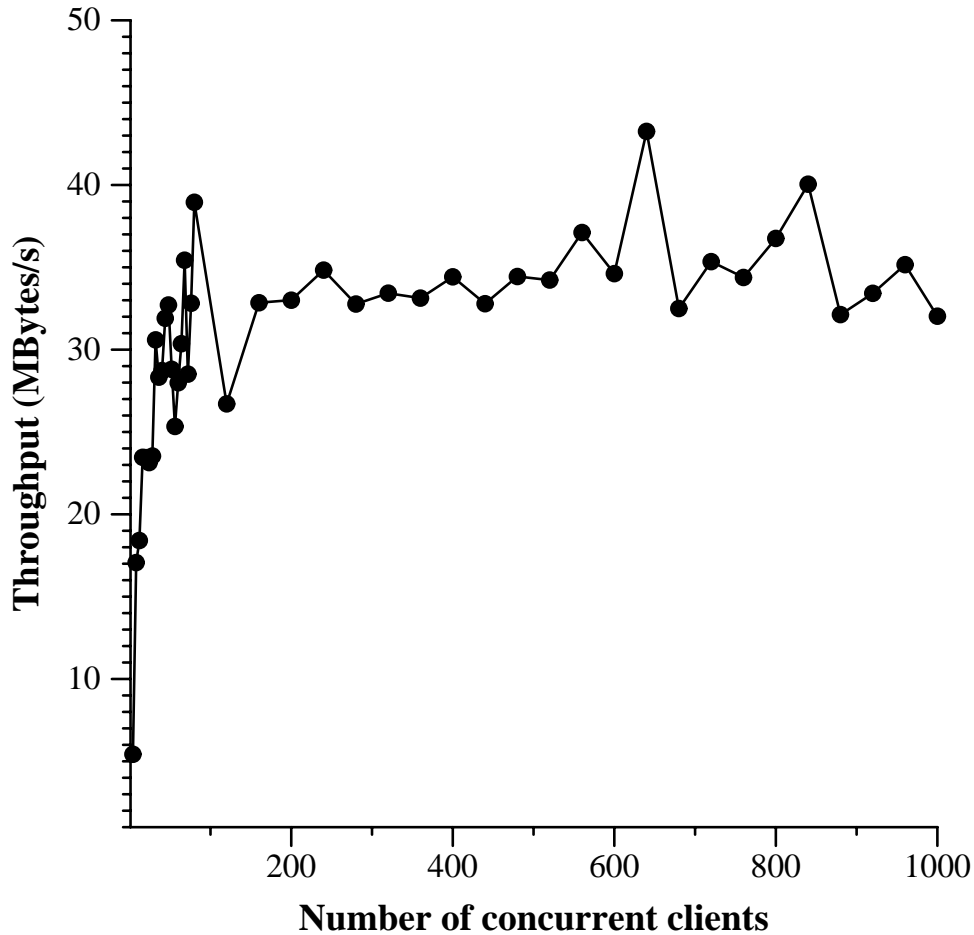


Figure 5-4: The performance of the web server on a cached workload as the number of concurrent clients is varied.

server offers consistent performance over a wide variety of loads.

5.2 SFS server

To evaluate the performance of *libasync-mp* on existing *libasync* programs, we modified the SFS file server [12] to take advantage of a multi-processor 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 work-

loads and therefore we expect that by using *libasync-mp* we can extract significant multiprocessor speedup.

5.2.1 Parallelizing the SFS server

We used the `pct`[3] 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-mp*, 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 default 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

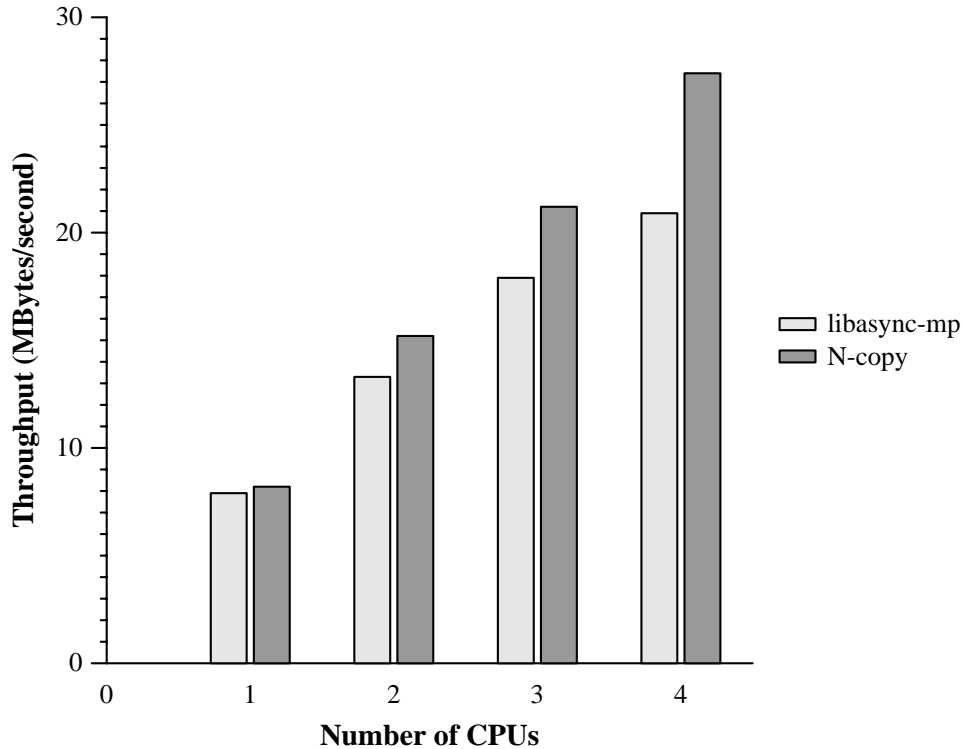


Figure 5-5: Performance of the SFS file server using different numbers of CPUs, relative to the performance on one CPU. The light bars indicate the performance of the server using *libasync-mp*; dark bars indicate the performance of n separate copies of the original server. Each bar represents the average of three runs; the variation from run to run was not significant.

requests over a single TCP connection to the server.

The bars labeled “libasync-mp” in Figure 5-5 show the performance of the parallelized SFS server on the throughput test. On a single CPU, the parallelized server is 0.95 times as fast as the original uniprocessor server. The parallelized server is 1.62, 2.18, and 2.55 times as fast as the original uniprocessor server on two, three and four CPUs, respectively.

The absence of significant speedup for the 4-processor case is due to the way we chose to parallelize the server. 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 (2.55) of the parallelized server. Other activities, such as interrupt handlers and the NFS server, run in parallel with the SFS server and account for the slight increase in performance between 3- and 4-processor cases. 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-mp* could achieve.

The results of this test are labeled “N-copy” in Figure 5-5. The SFS server with *libasync-mp* closely follows the aggregate performance of multiple independent server copies for up to three CPUs. The performance difference for 2- and 3-processor cases is due to the penalty incurred due to shared state maintained by the server, such as file lease data, user ID mapping tables, and so on.

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).

Chapter 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 [14, 7, 9]. This thesis does not attempt to argue that either is superior. Instead, it presents 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 [15]. 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-mp* 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-mp*, 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-mp* uses additional execution contexts to execute callbacks in parallel. *libasync-mp* 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-mp* must also cope with the issue of non-blocking file I/O: *libasync-mp* uses an NFS-loopback server to access files asynchronously. This allows *libasync-mp* 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 [2]. 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 [20]. 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. The major difference between SEDA and *libasync-mp* is that SEDA achieves I/O concurrency with concurrent blocking threads, while *libasync-mp* uses non-blocking callbacks. Both systems 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-mp* tries to preserve the serial callback execution model.

Chapter 7

Conclusions and Future Work

7.1 Future Work

The *libasync-mp* library can be potentially improved in a few ways. The shared callback queue can become a performance bottleneck for systems with large numbers of processors (over four). Implementing per-processor callback queues will allow each processor to execute independently for longer periods of time without synchronizing with other processors. Callback-to-processor affinity can be implemented by placing new callbacks onto the queue of the processor where callbacks of this color have last executed.

Priority levels in *libasync-mp* can be augmented with a better scheduling algorithm that tries to maximize system throughput. By choosing callbacks that have the most frequently occurring color in the callback queue, the scheduler can minimize the probability of color conflicts resulting in idle worker threads in the future. Other more complex maximal matching algorithms could also be applied to devise better scheduling algorithms.

A key feature of *libasync-mp* is the separation of the event-driven core from the callback queueing and execution layer. Current event-driven systems tend to execute event callbacks right away, leaving little room for scheduling policies and priorities. On the other hand, a separate callback queueing and execution layer allows for more involved scheduling policies, since event callbacks, and the event-polling loop itself, can be prioritized and potentially executed out of order. For example, certain callbacks may receive higher priority so that important events are serviced even when the overall system is overloaded and cannot handle the offered load. This allows for

more fine-grained priority control in event-driven systems, as a parallel to priority mechanisms typically available to multi-threaded systems.

Explicit event callback scheduling allows for explicit load-shedding in the face of overload, similar to some of the mechanisms used by SEDA [20]. Typical event-driven systems have no explicit load-shedding strategy; event callbacks are executed as usual, accumulating an excess of events waiting to be serviced, in hopes that the clients will back off due to increased latency. Due to the nature of the event-polling loop, the system is not aware that it is overloaded; all that can be noticed is that there are always events to be serviced. By using a callback queue, such as that used by *libasync-mp*, overload conditions can be observed as the queue length increases past a certain threshold. Overload can be handled by executing a special light-weight *drop* callback for callbacks that are queued beyond a certain depth threshold in the callback queue.

The drop callback would be optionally provided by the application, along with the normal event-handling callback. When executed, the drop callback would clean up any state held by the associated callback and inform the client that the server is unable to service the request due to overload. Such behavior may be better suited for clients in some situations. Additionally, this allows the server to only keep state for clients it expects to service, and free the memory used by clients it does not expect to be able to service.

The callback queue framework can also be used to provide for a more generalized way of handling overlapping disk I/O on UNIX systems than AMPED[15], by spawning more worker threads than the number of available processors.

7.2 Conclusion

This paper describes a library that allows event-driven programs to take advantage of multiprocessors. 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-mp* 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-mp* SFS server running on a 4-CPU machine 2.55 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 1.54 speedup on four CPUs with multiple clients reading small cached files relative to its performance on one CPU.

Bibliography

- [1] T. Anderson, B. Bershad, E. Lazoswka, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.
- [2] The apache web server. <http://httpd.apache.org/>, 2002.
- [3] Charles Blake and Steve Bauer. Simple and general statistical profiling with PCT. To appear in the proceedings of USENIX 2002, 2002.
- [4] J. M. Bloom and K. J. Dunlap. Experiences implementing BIND, a distributed name server for the DARPA Internet. In *Proceedings of the 1986 Summer Usenix Conference*, pages 172–181, 1986.
- [5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O’Reilly, 2001.
- [6] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP’01*, pages 202–215, Banff, Canada, 2001.
- [7] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136. Association for Computing Machinery SIGOPS, 1991.
- [8] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

- [9] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In *Operating Systems Design and Implementation*, pages 101–115, 1999.
- [10] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–196, October 2000.
- [11] David Mazières. A toolkit for user-level file systems. In *Proc. Usenix Technical Conference*, pages 261–274, June 2001.
- [12] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, December 1999.
- [13] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and Kohn S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. 1996.
- [14] John K. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX technical conference, 1996.
- [15] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.
- [16] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [17] G. Steele and G. Sussman. Lambda: The ultimate imperative. Technical Report AI Lab Memo AIM-353, MIT AI Lab, March 1976.
- [18] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, August 2001.
- [19] Simon Thompson. *Haskell, The Craft of Functional Programming*. Addison Wesley, 1996.

- [20] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 230–243, Banff, Canada, October 2001.