Multiprocessor Support for Event-Driven Programs

Nickolai Zeldovich, Alexander Yip, Frank Dabek,
Robert T Morris, David Mazières, Frans Kaashoek

MIT Laboratory for Computer Science

Usenix Technical, June 2003

Introduction

- Many internet servers use an event-driven programming model:
  - Code consists of many callback functions, which are executed when an event occurs
  - Events can be a mouse click, receiving network data, timer expiration, ...
  - Callback functions perform some task and can register other callbacks waiting for new events

What's wrong?

- Callback functions are executed sequentially
  - Code is never executed in parallel
  - Programmer can be confident that his callback is the only one changing the state right now
- But we want parallel execution: it's faster on multiprocessors!
  - Can't just break a fundamental assumption

Carefully breaking the assumption

- Let the programmer say what, if anything, can run in parallel
- Add a color to every callback
  - A color is any integer value
  - Callbacks of the same color can't run in parallel
  - Callbacks of different colors can run in parallel

Where do colors come from?

- Think BSD wait channels
- For example, file descriptor number of client connection, or pointer to shared object
- By default, everything is color zero
  - Programmer has to explicitly break things
- Color collision may reduce performance, but not correctness!

Isn't this already solved?

- Use mutex locks from the threads world?
  - Mutex locks are hard: deadlocks, race conditions
  - Not worrying about concurrency and locking is a big advantage in event-driven programs!
  - Callbacks in event-driven programs should not block; acquiring a mutex does

Why color callbacks?

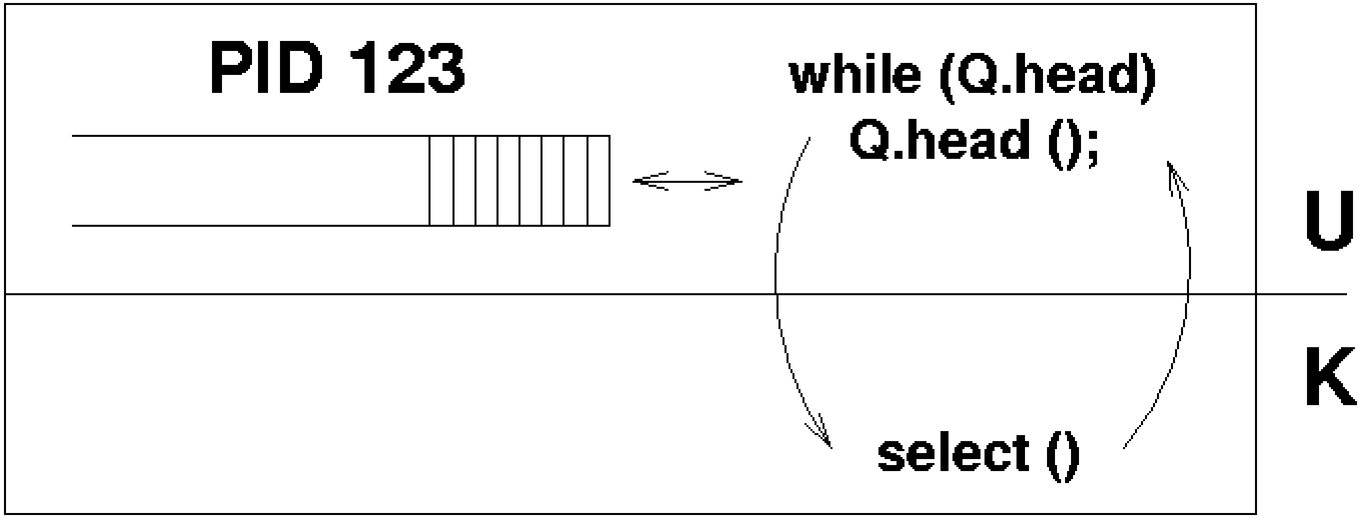
- Two observations:
  - Callbacks typically perform short, well-defined operations associated with a single event
  - Systems software often has natural coarse-grained parallelism (e.g. many independent requests)
- Coordinating parallel execution at the level of callbacks sounds reasonable

What's so great about colors?

- Callback colors let the scheduler make decisions and optimize ahead of time
- Callbacks can be colored incrementally to achieve incremental multiprocessor speedup
  - With threads and mutex locks, it's all-or-nothing
- Less expressive than locking, but that's fine

libasync

- C++ library for event-driven programs
- Provides the main event loop which waits for events and runs callbacks
- Events: signals, timers, socket readable or writable



Useful things in libasync

- Function currying for C++ to save callback state:
  - void cbfunc (char x, int y);

```
callback cb = wrap (&cbfunc, 'A');
cb (7);          /* executes cbfunc ('A', 7) */
```

More useful things

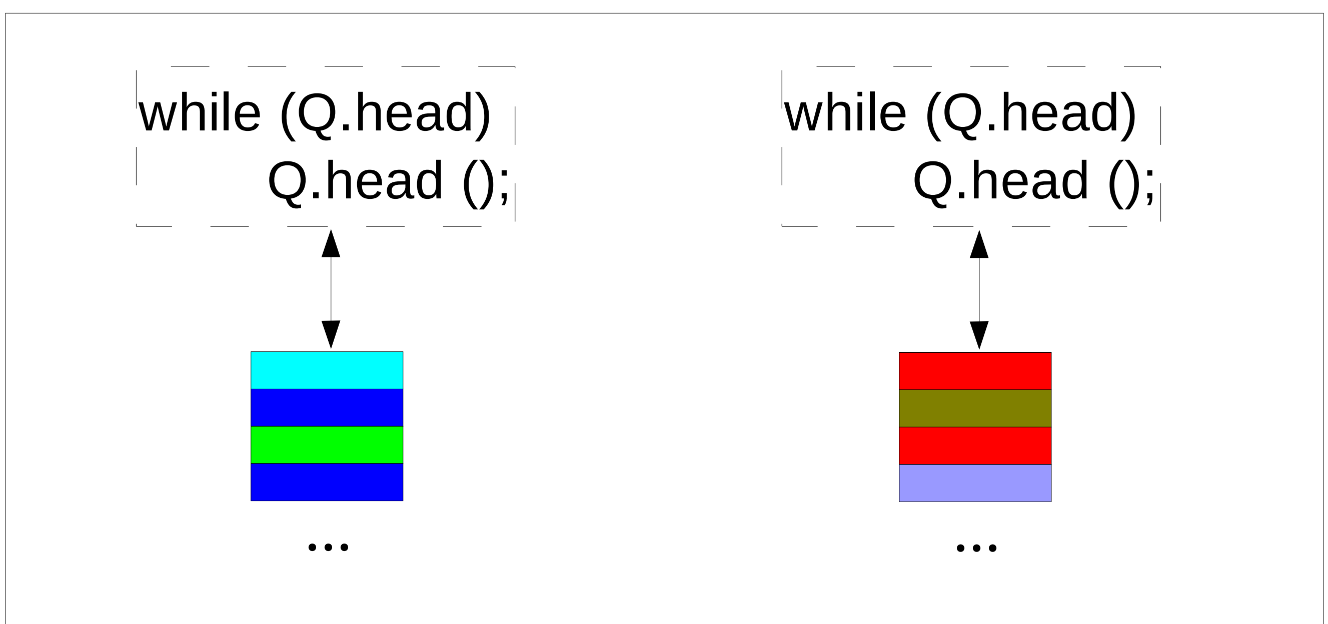
- Common event dispatcher allows modules to co-exist without knowing about each other
  - Great for modularity
- libasync provides additional event-based modules for DNS, SunRPC, NFS, ...

libasync-smp

- Modified version of libasync which can take advantage of multiprocessors
- Implements callback coloring for concurrency control

Design of libasync-smp

- One worker thread and callback queue per CPU
- Worker thread repeatedly chooses a runnable callback from its queue and runs it



Design of libasync-smp

- Worker threads share address space, file descriptors, and signal handlers
- select() call from libasync's event loop is now just another callback on the queue
  - Executed by a worker thread when there are no other callbacks to run
  - Calls select() and enqueues other callbacks as necessary

Where to queue callbacks?

- Mapping of colors to worker threads
  - Callbacks of the same color run in same worker thread
  - Color-to-worker affinity improves cache locality, like thread-to-CPU affinity in kernel scheduler

Scheduling Callbacks

- Preference for callbacks of the same color as the last callback to execute
  - Improves cache locality
- When a worker thread is idle, steal work from other queues
  - Must steal all callbacks of the same color

What to measure?

- How much faster do libasync-smp programs run on *N* CPUs than the same program using libasync on 1 CPU?
- Run *N* copies of libasync version and use aggregate speed of *N* copies as upper bound for libasync-smp performance

What to measure?

- How easy is it to use libasync-smp?
  - Count lines of code changed or written
  - Count number of callbacks colored

Performance Testing

- Experiments done on 4-way 500 Mhz Pentium-3 Linux server, 512MB memory
- Each Linux client has separate gigabit Ethernet link to server
- Tested an HTTP server and SFS (network file system) file server

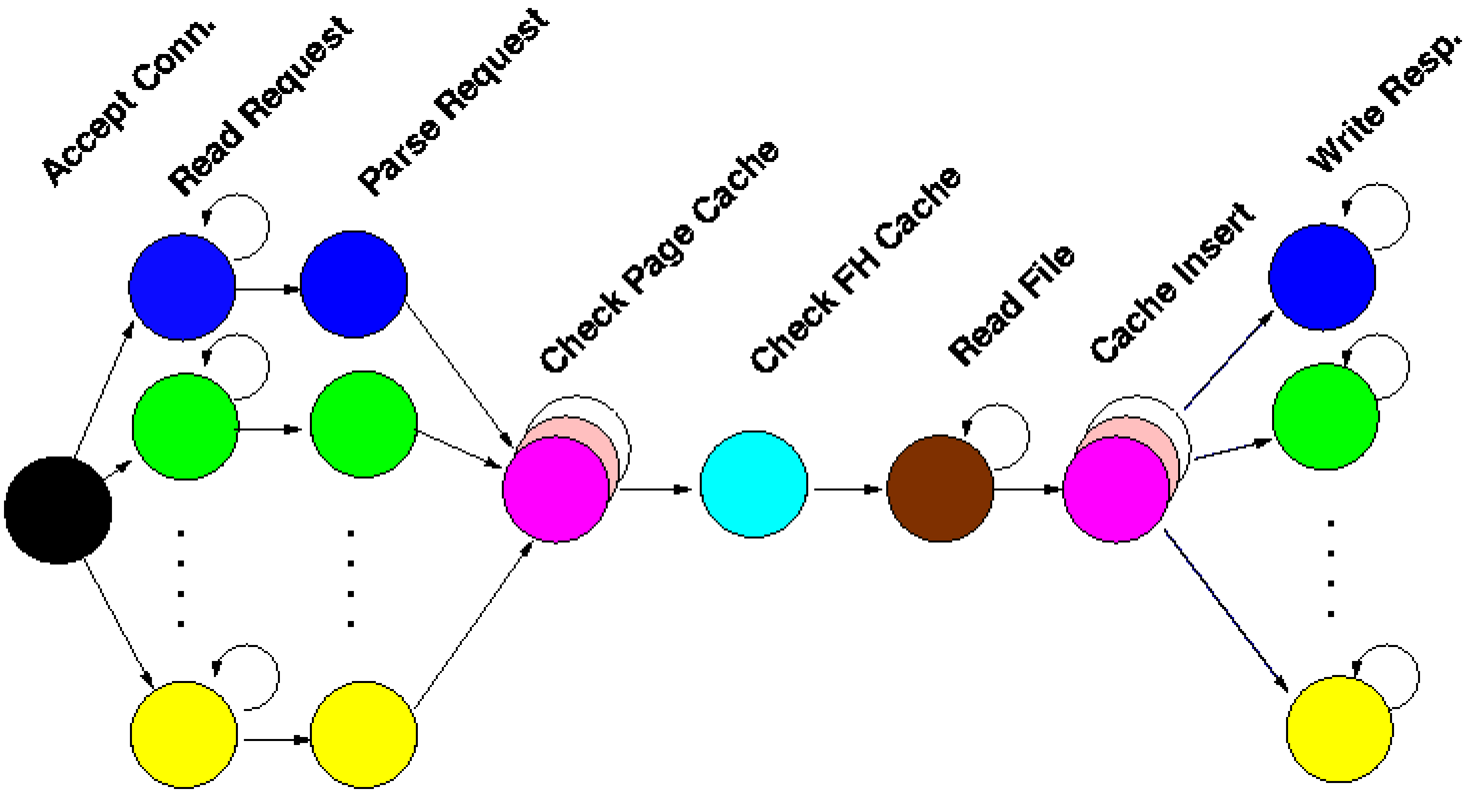
Our HTTP Server

- libasync-based HTTP/1.1 server
- Uses an NFS loopback server for non-blocking disk I/O
- Two shared caches that must be protected from simultaneous accesses:
  - NFS file handle cache
  - Web page cache
    - Actually a small number (10) of independent caches, to allow simultaneous access to different pages

How hard was it?

- Our libasync HTTP server is 1260 lines of code with 39 calls to *wrap* (callback creation)
- 23 callback creation points modified to provide a non-zero color for the callback

HTTP Server Concurrency

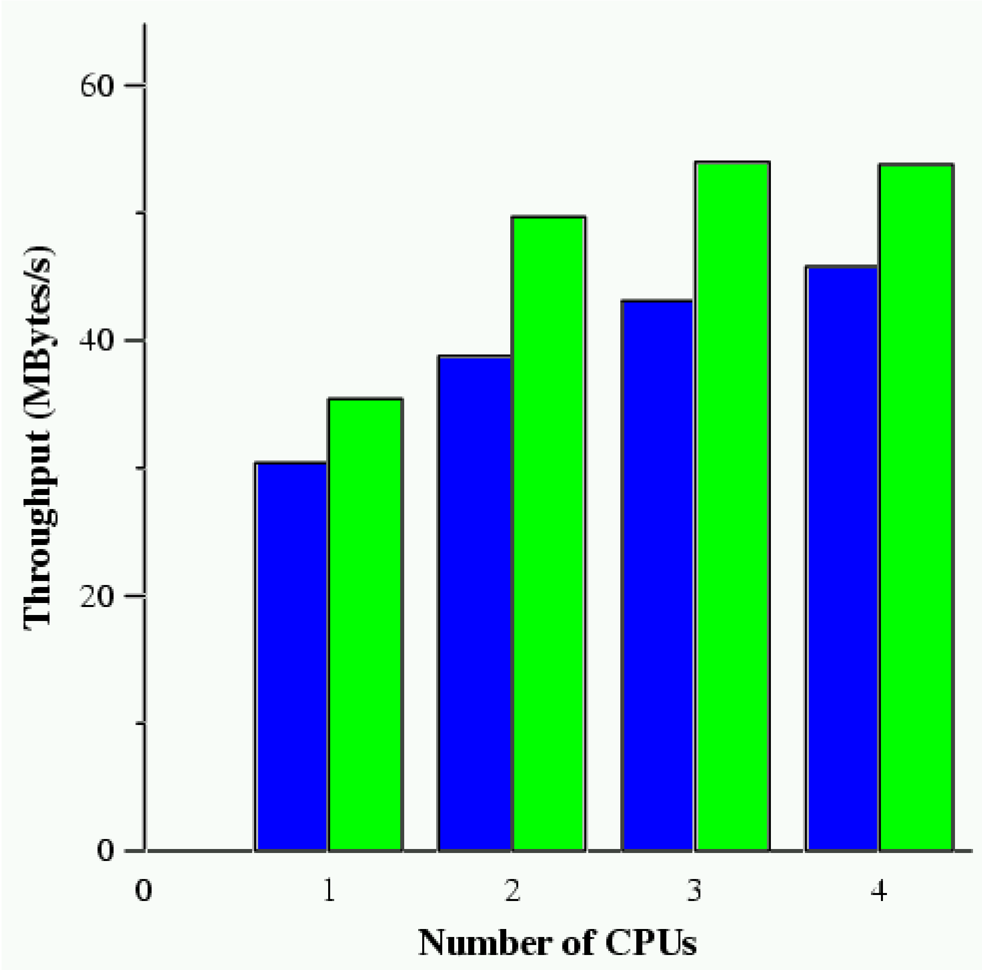


HTTP Servers Tested

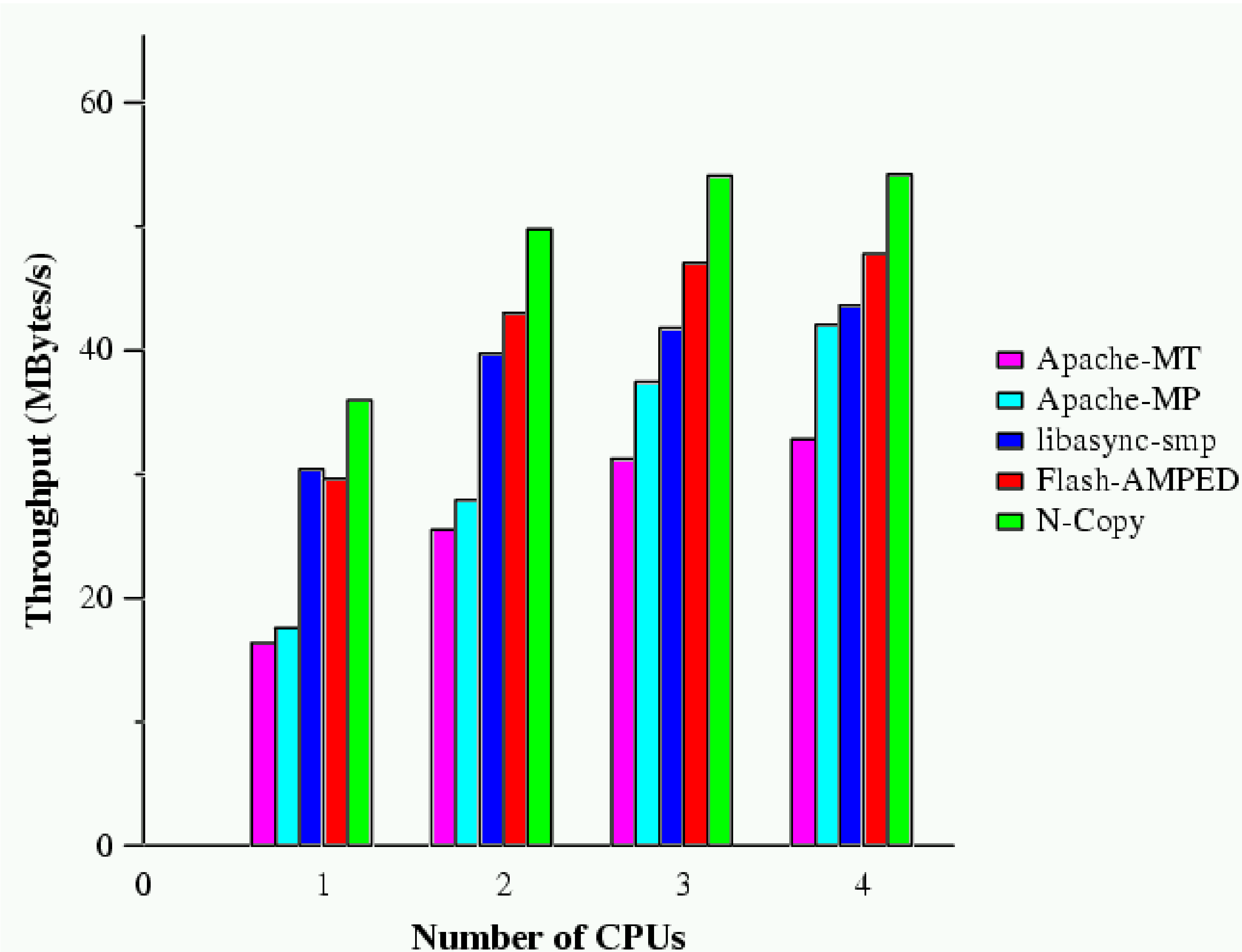
- Compare the performance of these servers:
  - libasync-smp based event-driven server
  - Same web server using unmodified libasync, running a separate copy on each CPU (``N-copy'')
  - Apache 2.0.36
  - Flash v0.1.990914

HTTP: libasync-smp vs. N-copy

- On 1 CPU, libasync-smp thoughput is 0.86 times that of N-copy; on 4 CPUs, it is 0.85 of N-copy
- libasync-smp extracts most of the speedup the OS offers for a web server



HTTP Server Performance



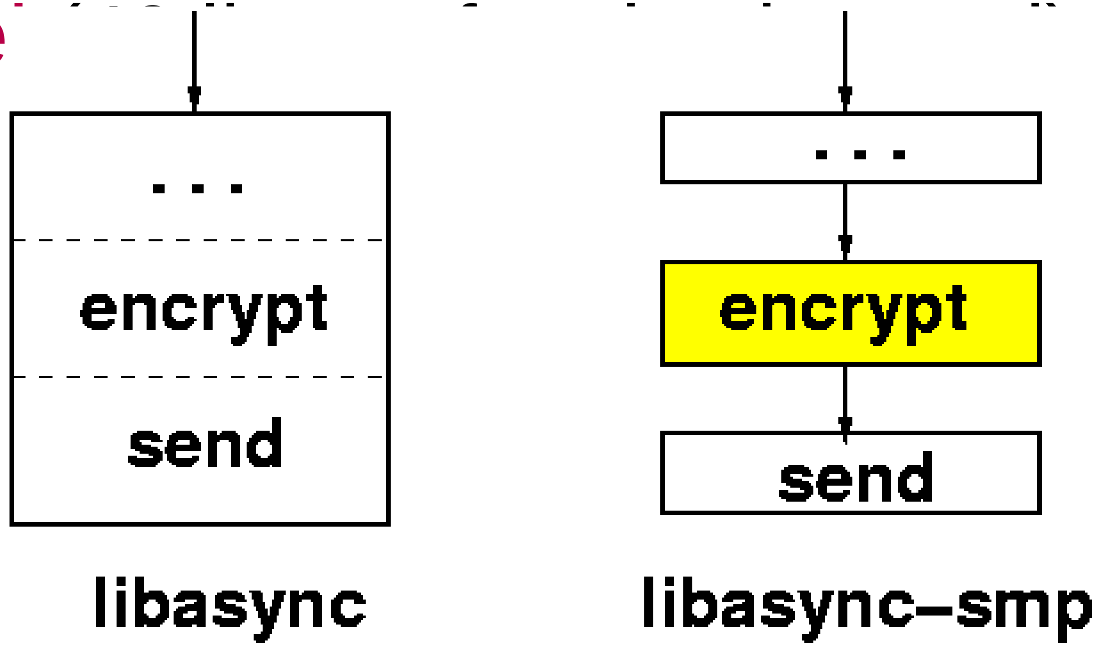
- libasync-smp speedup is 1.5; Flash gets 1.68
- N-copy used by Flash OK for web servers, but not for shared state

SFS File Server

- SFS is a secure network filesystem
- User-level libasync-based SFS file server
- Encrypted (RC4) and authenticated (SHA-1) communication with clients over TCP
- Maintains significant mutable state, such as lease records for client cache consistency

Parallelizing the file server

- Profiling reveals file server is compute-bound due to crypto (75% CPU time spent there)
- Split up the send callback to encrypt in paralle

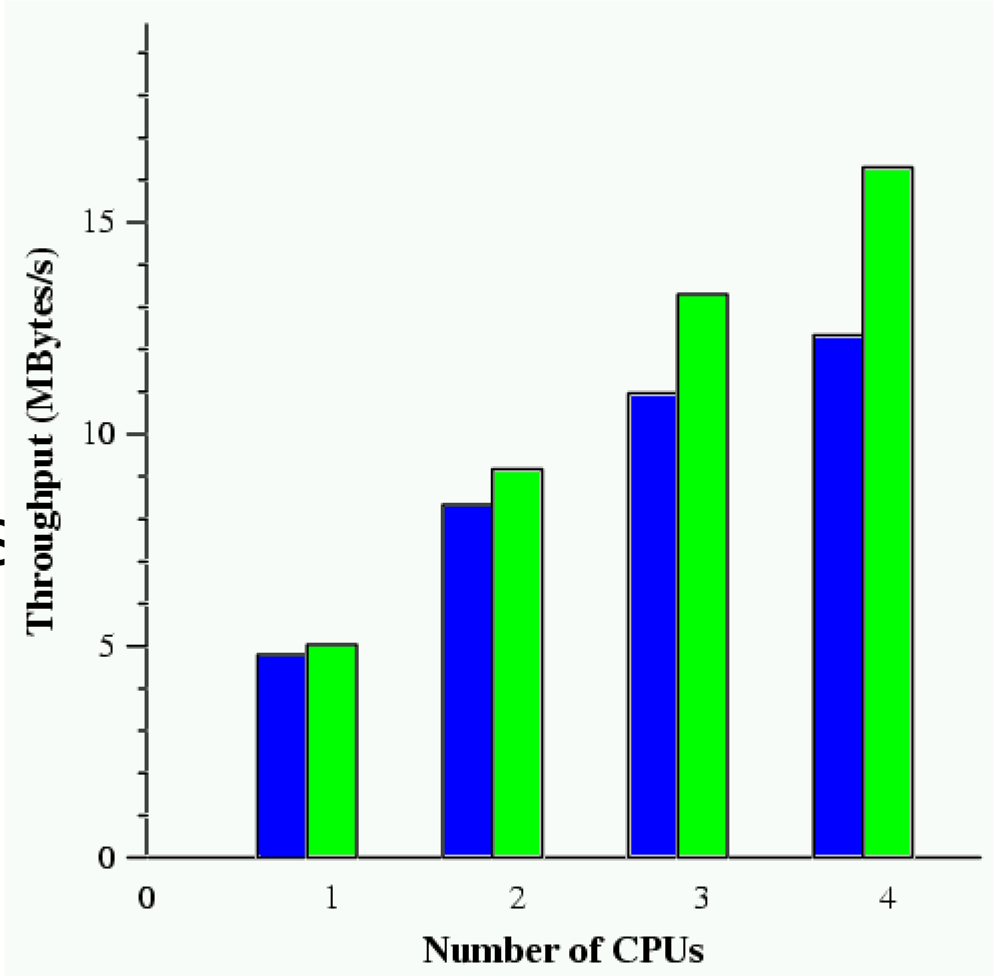


Parallelizing the file server

- Another 50 lines of code changed to similarly color the packet receive code path
- Using libasync-smp, 65% CPU time spent in cryptographic operations
- Maximum theoretical speedup, with as many CPUs as needed, is 1/(1-0.65)=2.85

File server performance

- libasync-smp file server on 4 CPUs is 2.5 times faster than original libasync-based fileserver on 1 CPU
- Close to theoretical maximum speedup of 2.85
- libasync-smp is 0.96 times as fast as libasync-based fileserver on 1 CPU
- N-copy not viable



Conclusion

- Event-driven programs can use colors to specify callbacks to be executed in parallel
- Callbacks in programs can be colored incrementally for incremental speedup
- libasync-smp requires little programming effort to achieve multi-processor speedup

http://www.fs.net/