libtorque: Portable Multithreaded Continuations for Scalable Event-Driven Programs

Nick Black and Richard Vuduc
Georgia Institute of Technology
nickblack@linux.com, richie@cc.gatech.edu

Abstract

Since before even 4.4BSD or POSIX.1[1], programming via events and continuations has marked best UNIX practice for handling multiplexed I/O[2]. The idiom's programmability was proved sapient astride a decade's architectures and operating systems, as *libevent* [19], *libev*, Java NIO and others achieved ubiquity among network applications. Academic [32] and proprietary systems have responded to multiprocessor economics with threaded callback engines, but such I/O cores remain rare in open source: Firefox uses multiple functionally-decomposed event loops, `nginx` multiple processes with static load distribution, and Apache's `mpm-worker` variant a thread per connection. Economic employment of even COTS (Consumer Off-The-Shelf) hardware already requires concurrent workloads, and manycore's march into the data center still more: dynamic parallelism as rule rather than exception. The community agrees that UNIX network programming must change [6] [13], but consensus of direction remains elusive [14] [29]. We present our open source, portable *libtorque* library, justify the principles from which it was derived, extend previous threaded cores through aggressively exploiting details of memories, processors, and their interconnections (as detected at runtime), and imply a new state-of-the-art in architecturally-adaptive, high-performance systems programming. Built with scalability (in both the large and small), low latency, and faithfulness to UNIX idiom as guiding lights, *libtorque* subsumes the functionality of existing I/O frameworks (for which we provide compatibility wrappers) despite superior performance across most loads and apparatus.

1 Intro

It's a rare and rather lucky program which spends most of its wall time calculating. Whether an interactive application, a desktop widget, or a network server, relative eternities are made up of waiting for, shuffling, and signaling the presence of data. Spinning on event readiness implies ineffective use of processor and power, motivating event registration and data readiness notification schemes (of which blocking I/O—with or without a timeout—can be thought the *uniplex* special case. Each thread has a single channel, bound to a single source). This concept forms the essential *omphalos* of POSIX's system interfaces: Pthread condition variables, asynchronous I/O, and humble `read(2)` can all be interpreted as some mapping between threads and event sources, with the goal always of sleeping as much as local service requirements allow. Every non-trivial program will use them at least once.

Given our definition of blocking I/O, spinless employ of multiple event sources requires either multiple threads or multiplexed, non-blocking (possibly asynchronous, i.e. kernel-demultiplexed) I/O. The former solution, at the cost of $O(n)$ threads and $O(n)$ context switches for n event sources, retains the simplicity and streamline (thanks to the absence of any multiplexing interfaces) of blocking. With the advent of Linux's NPTL and FreeBSD's libthr, this method has seen a resurgence, and even argument *a posteriori* that its performance might exceed that of multiplexed I/O [31]. That this is possible—that $O(n)$ time and space costs are negated by multiplexing overhead, as evidenced in [28]—shows how much room for improving non-blocking I/O solutions exists on modern machines. We enumerate and address five possible overheads: memory effects, multiplex setup ("enplexing"?), synchronization within the system call, walking of the event table, and copying the results to userspace. We illustrate several ways I/O-intensive applications can (and *libtorque* does) make effective use of system architecture properties. We show that a tradeoff exists between dynamic balance under arbitrary load

[1] `select(2)` first showed up in 4.2BSD, `poll(2)` in SVR4.

[2] As canonicalized within the books of W. Richard Stevens [24].

and synchronization costs, and how it might be optimized. The result is as powerful, flexible, and scalable as POSIX.1b Asynchronous I/O [10], Windows's I/O Completion Ports [23] or Solaris's Event Completion Framework [20], but makes better use of modern, complex architectures (see [30] and [9] for the profound effects of multicore and CMT on network servers). We see it as a viable successor to or merge candidate for *libev*, especially given its compatibility interface[3].

2 Architecture and APIs

libtorque assumes exclusive use of all n processors in its cpuset (as inherited from the creating thread, which ought prune processors prior to calling `libtorque_init()` if appropriate). n threads are created via exponential bloom (requiring $O(n)$ work in $O(\log n)$ depth), and use hard affinity to preclude expensive migrations (see [21] and Section 3.4 for justification). Each thread probes its processor and attached memories, collaboratively establishing a map of hardware. Stacks and event queues—optimal in number and placement—are created, and entered via the method of `sigaltstack(2)` trampoline [7]. Threads begin the event loop, the signal mask is reset, and control is returned to the caller.

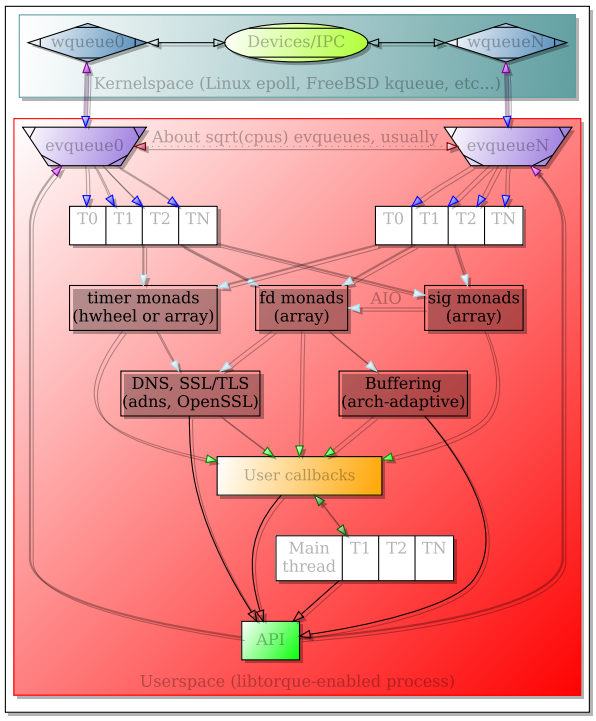

Figure 1: Architecture of a *libtorque*-enabled process.

All entry points are thread-safe[4]. All save the constructor (`libtorque_init()`) and destructors (`libtorque_stop()` and `libtorque_block()`) operate in $O(1)$ time and space. All are lock-free throughout userspace (the kernel itself locks in some cases) and usually wait-free[5]. This arises from the kernel's synchronization of new file descriptors (ensuring no two entities are given the same descriptor), the kernel's synchronization of event notification state with descriptor closings (closing a descriptor atomically removes it from all queues), and the kernel's synchronization of event queue modifications and retrievals.

2.1 Currying as de-layering

Network programming abounds with task idioms. *libtorque* implements several ((de)compression, X.509-based authentication and symmetric encryption via OpenSSL [26], and asynchronous hostname resolution via GNU adns [12]) in a fully layerable fashion reminiscent of SVR4 STREAMS [22]. This is easily achieved via the generic marshaling technique known as *currying*. In addition, several types of buffering are provided. *libtorque*'s extensive system detection allows (possibly NUMA and/or heterogeneous) memories' (usually multiple) page sizes to be taken into consideration for buffer sizing and even placement. Since buffers are highly unlikely to be truly shared among threads or to exhibit temporal locality throughout the threads's life, we *color* both threads' stacks and buffers, of which each thread keeps a properly-formed pool. This aggressive mobilization of the memory hierarchy is a major reason why *libtorque* makes use of hard affinity.

2.2 Callback semantics

In the interests of simplicity, robustness and ease of porting existing code, callbacks are restricted as little as possible. They may spawn child programs or helper threads, close arbitrary file descriptors, and call back into a *libtorque* instance. Fatal signals continue to terminate the process, unless caught elsewhere. Some requirements are unavoidable:

- Blocking calls must be avoided. Numerous systems exist to automatically dispatch blocking I/O [5], and *libtorque* provides the facilities to support them. Any code written with performance in mind is almost certainly already using non-blocking or asynchronous I/O.
- Callbacks must not freely operate on event sources registered with libtorque, save to `close(2)` file descriptors; this is safe due to the atomicity properties described above.

[3] A tradition begun with *libev*'s similar wrapper for *libevent*.
[4] None are async(signal)-safe.
[5] *Lock-free* implies system progression, while *wait-free* requires progression from each thread.

- Obviously, the callback ought not directly modify signal handlers (all non-fatal signals are blocked during callbacks)[6].

3 Implementation

libtorque's implementation is non-trivial. Traditionally free parameters such as buffer sizes, stack sizes, and distribution of queues among threads are derived through system discovery and simple feedbacks. Making optimal use of various operating systems (and versions thereof) required intimacy with a garden of kernel implementations. The result serves as something of a tour of UNIX system API's since the 3rd Single UNIX Specification; much of *libtorque*'s value lies simply in uniting these disparate interfaces.

3.1 Why not POSIX.1b AIO?

Some argue that POSIX.1b asynchronous I/O fits our requirements better than multiplexed I/O. It is true that AIO scales, handles demultiplexing itself, and can be dynamically balanced across threads:

Method	Demux	Thr	Evs	Misc
Blocking	Kernel	1	1	$O(n)$ state
AIO	Kernel	N	N	Restrictions
Multiplex	User	N	N	Complexity

Native AIO on Linux, however, is restricted to those descriptors which support `lseek(2)` (this excludes terminals and sockets, which employ a polling fallback [3]; FreeBSD restricts use to terminals and sockets, excluding disks [25]). Linux has only natively supported *any* AIO since late in the 2.6 development cycle, having emulated it until then. On all systems, AIO is restricted to file descriptors as event sources. These limitations make POSIX.1b unattractive as a core notification mechanism. Performance of the two is comparable (both require a system call to register the event, and at least one context switch before the notification can be received), though AIO could be expected a slight advantage here: when and if it proves desirable, AIO could be used in conjunction with multiplexing. *libtorque* does, of course, support callbacks based on AIO events.

3.2 Multiplexing primitives

The `poll(2)` of POSIX.1-2001 requires an unacceptable $O(n)$ copy of event registration state per invocation. Divergent alternatives exist:

Linux: `epoll(7)` since 2.5.44 [8].
FreeBSD: `kqueue(4)` since 5.0 [17].
Solaris: Eventports since OpenSolaris 10.
Windows: I/O Completion Ports since NT 4.0.

`epoll(7)` is rather more limited than `kqueue(4)`. Most importantly, it explicitly supports only file descriptors. *libtorque* supports events based on file descriptor readiness, signal receipt, network state changes, filesystem changes, AIO and condition variables. Much of this must be emulated on Linux:

Event source	Mechanism	Version
File descriptor	N/A	N/A
Signal/AIO	`signalfd(2)`	2.6.22
Signal/AIO	`epoll_pwait(2)`	2.6.19
Signal/AIO	Self-pipe trick [2]	N/A
Timer	`timerfd(2)`	2.6.25
Timer	POSIX timer	2.6
Timer	Hashed wheel	N/A
Filesystem	`inotify(7)`	2.6.13
Filesystem	`F_NOTIFY`	2.4

libtorque addresses these further differences:

- `kqueue(4)` supports batching of event changes
- `kqueue(4)` can change and retrieve in one call
- `kqueue(4)` has incomplete error reporting
- `epoll_wait(2)` is a cancellation point
- `epoll_wait(2)` is unaffected by SA_RESTART

Clearly, the functionality of `epoll_wait(2)` and `epoll_ctl(2)` can be emulated in terms of `kqueue(4)`, or vice versa. *libtorque*'s event core provides `kqueue(4)`-like semantics, both to take advantage of batching[7] and because no fine-grained action is prompted by a state modification error.

Events are greedily retrieved, up through the space provided. Retrieving more events means fewer system calls overall, but can also result in unnecessary system imbalances without a work-stealing implementation [4] and its attendant locks. Each thread thus initially requests a single event, and uses an algorithm similar to that of TCP's congestion control: request more events when all requested are returned, and request fewer if fewer than that requested were returned. The buffer offsets and number requested are tuned based off the smallest page size and properties of shared caches.

3.3 Thread-safe event retrieval

Avoiding expensive locking operations, and especially contention, is critical in any parallel program. Wrapping the event monads (see Fig. 1) in locks, requiring $O(n)$ `pthread_trylock(3)` operations for n returned events, is clearly undesirable.

[6] Or mess with affinities, install alternate sigstacks, invoke `pthread_exit(3)`, call `exec(2)`, *ad nauseam*...

[7] And also to prevent high-bandwidth connections from bouncing around threads sharing a given queue.

By default, both `epoll(7)` and `kqueue(4)` are *level-triggered*. Under level-triggered semantics, it's more correct to speak of monitoring *states* rather than *events*. An event will be returned so long as the specified case applies to the monitored object. This is entirely unsuitable for threaded operation:

- LT event 1 becomes ready
- Thread A retrieves LT event 1 instance 1
- Thread B retrieves LT event 1 instance 2
- Thread A enters LT event 1 handler
- Thread B enters LT event 1 handler
- *either contention or race follows…*

Adding locks results in spinning: threads loop through event retrieval and attempted lock acquisition. It is similarly unacceptable to refrain from retrieving events until a common producer-consumer buffer is emptied: a heavyweight connection could in this case add significant latency to other processing. This furthermore results in a $O(n)$ walk-and-copy operation from the system call on n monitored descriptors, though this is merely a consequence of existing implementations.

For these reasons, *libtorque* makes exclusive use of *edge-triggered* semantics (on Linux, `EPOLLET`, and on FreeBSD, `EV_CLEAR`). Such semantics truly describe discrete "events"—the event, for instance, of becoming readable. It ought be noted that a readable event source must *become unreadable* before it again *becomes readable*—edge-triggered events imply a contract: the available resource must be fully consumed. To prevent starvation due to a heavyweight connection, *libtorque* embeds a *postponed event queue* within the system event queue, ensuring global flow progression. *libtorque* thus hides edge-triggering's additional complexity.

This eliminates some races, and for certain handlers is sufficient synchronization (consider for instance an `accept(2)` loop which merely calls the concurrency-friendly `libtorque_addfd()`, not touching any other state). In general, this is true for any handler which synchronizes any access to callback state (as implicitly performed by `accept(2)` in our example), and can be indicated to *libtorque* via the `LIBTORQUE_EV_MTSAFE` flag. Otherwise, for instance if `read(2)`ing data into a buffer pointed to by callback state:

- ET event 1 becomes ready
- Thread A retrieves ET event 1 instance 1
- Thread A `read(2)`s all available data
- ET event 1 is automatically rearmed
- Thread B retrieves ET event 1 instance 2
- Thread B `read(2)`s some/all data
- Thread B enters ET event 1 critical section
- Thread A enters ET event 1 critical section
- *either contention or race follows…*

Thankfully, it's once again possible to make use of synchronization implied by the kernel interfaces[8]. If we disabled the event immediately after it was returned, and then reënabled it once truly done with the callback, the race would be eliminated. *Once-triggered* semantics (on Linux, `EPOLLONESHOT`, and on FreeBSD, `EV_ONESHOT`) internally disables the event whenever it's returned, saving this scheme one system call. Once-triggered semantics are orthogonal to edge- vs. level-triggering, though *libtorque* uses only the edge-triggered variant.

3.4 Queue distribution

The distribution of event sources among event queues, and event queues among threads, is a major research question and the focus of ongoing experimentation. This distribution affects the system in several ways:

Balance: Long-term balance among threads is maximized by sharing one event queue among all workers (short-term balance is further a function of the number of events retrieved at once, maximized by serial event retrieval. This is the default model of POSIX.1b AIO). Any thread without work can handle any event.

Locality: Locality of event handling is maximized by associating one event queue with each thread, which will be the only thread to ever handle the event. This is only relevant if locality is being effectively exploited in the first place.

Overhead: Neither `epoll_wait(2)` nor `kevent(2)` can, as of this time, be considered truly scalable for large n. Even assuming improvements in their implementations[9], sharing among more threads leads to more synchronization overhead.

Robustness: It is preferred that *libtorque* threads have exclusive access to their processors, but this might not always be the case. Greater numbers of processors furthermore imply less time before a processor failure. Robustness against lack of access to certain processors increases with greater sharing.

[8] Detailed analysis of the relevant kernel source can be found in `doc/mteventqueues` within a *libtorque* checkout.

[9] Or RCU. Or transactional memory.

At this time, event queues are shared based upon sharing of memories. The lowest-level caches or memories shared by multiple (possibly logical) processors demarcate queue sharing.

4 Future Work

It is expected that the copy of events to userspace will cause three delays: the copies themselves, the calling thread's delay while copies are made, and delay of any contending threads. While the first cannot be eliminated, a lockless ring buffer shared between kernel and userspace could hide the other two delays. Such use has precedent in, for instance, Alexey Kuznetzov's mmap(2)'d packet socket [15], and variants have been developed for modern machines [16]. Profiling of heavily-loaded event retrieval ought be performed at large scales (hundreds of events returned per retrieval, dozens to hundreds of processors), and such a change tested if justified.

Userspace networking stack implementations have been shown to provide excellent performance, primarily through their lack of copies and context switches. With the recent addition of `mmap`'d packet socket-based transmission to the Linux kernel [1], this approach could be the fastest path to true zero-copy networking [27]. *libtorque* could add protocol decomposition to its multiplexing to efficiently provide clients userspace transport demultiplexing [18].

Recent processors have included support for "non-temporal" memory operations [11]. When used, these variants minimize cache pollution. Many I/O operations could make effective use of these methods: not only would large copies lead to fewer evictions (and thus better general performance), but the copies themselves could be enlarged without fear of further disrupting cache (especially for our carefully-placed stacks).

Event sources could be added as *communal* or *solitary*. Sources forming a commune share a great deal of (hopefully read-only) data, and sharing preferences ought follow sharing of memory. Solitary sources share little data, and ought prefer sharing among corresponding members of isomorphism classes within the interconnect.

Unified caches can negate our careful data placement. It ought be possible to detect at least the core code paths traversed by a *libtorque* thread, and integrate their locations into the occupancy map.

References

[1] BAUDY, J. Packet mmap: Tx ring and zero copy. *Linux Kernel Mailing List* (2008).

[2] BERNSTEIN, D. J. The self-pipe trick. `http://cr.yp.to/docs/selfpipe.html`, retrieved 2010-01-19.

[3] BHATTACHARYA, S. Aio fallback for pipes, sockets and pollable fds. *Linux Kernel Mailing List* (2007).

[4] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science* (1994).

[5] CRAMERI, O. liblaiogen: A generic LAIO implementation. Master's thesis, School of Computer and Communications Sciences, EPFL, Lausanne, Switzerland, February 2005.

[6] DREPPER, U. The need for asynchronous, zero-copy network I/O. In *Ottawa Linux Symposium* (July 2006).

[7] ENGELSCHALL, R. Portable multithreading. In *USENIX Annual Technical Conference* (2000).

[8] GAMMO, L., BRECHT, T., SHUKLA, A., AND PARIAG, D. Comparing and evaluating epoll, select, and poll event mechanisms. In *Ottawa Linux Symposium* (2004).

[9] GUO, D., LIAO, G., BHUYAN, L. N., LIU, B., AND DING, J. J. A scalable multithreaded L7-filter design for multi-core servers. In *Symposium On Architecture For Networking And Communications Systems* (2008).

[10] IEEE. *IEEE 1003.1b-1993 Part 1b: Realtime Extensions (Volume 1)*. IEEE Standard Information technology—Portable Operating System Interface (POSIX). IEEE, New York, NY, USA, 1993.

[11] INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. 2B, Instruction Set Reference, N–Z. Intel Press, June 2009.

[12] JACKSON, I. Gnu adns. `http://www.chiark.greenend.org.uk/~ian/adns/`, retrieved 2010-01-26.

[13] JACOBSON, V. Speeding up networking. In *Ottawa Linux Symposium* (July 2006).

[14] KEGEL, D. The C10K problem. `http://www.kegel.com/c10k.html#strategies`, retrieved 2010-01-20.

[15] KUZNETZOV, A. *packet_mmap.txt*. Linux kernel 2.6.32 documentation.

[16] LEE, P. P. C., BU, T., AND CHANDRANMENON, G. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2009).

[17] LEMON, J. Kqueue: A generic and scalable event notication facility. In *USENIX Annual Technical Conference* (2001).

[18] MAEDA, C., AND BERSHAD, B. Protocol service decomposition for high-performance networking. In *ACM Symposium on Operating Systems Principles* (1993).

[19] MATHEWSON, N. Fast portable non-blocking network programming with Libevent. `http://www.wangafu.net/~nickm/libevent-book/`, retrieved 2010-01-19.

[20] MCDOUGALL, R., AND MAURO, J. *Solaris™Internals: Solaris 10 and OpenSolaris Kernel Architecture*, 2nd ed. Prentice-Hall, 2006.

[21] PORTERFIELD, A., FOWLER, R., MANDAL, A., AND LIM, M. Y. Performance consistency on multi-socket AMD Opteron systems. Tech. Rep. TR-08-07, Renaissance Computing Institute, 2008.

[22] RITCHIE, D. A stream input-output system. In *AT&T Bell Laboratories Technical Journal* (1984), no. 63.

[23] RUSSINOVICH, M., SOLOMON, D., AND IONESCU, A. *Windows® Internals*, 5th ed. Microsoft Press, 2009.

[24] STEVENS, W. RICHARD. *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd ed. Addison-Wesley Professional, 2003.

[25] STEVENS, W. RICHARD. *Advanced Programming in the UNIX Environment*, 2nd ed. Addison-Wesley Professional, 2008.

[26] THE OPENSSL PROJECT. OpenSSL: The open source toolkit for SSL/TLS. `http://www.openssl.org/`, retrieved 2010-01-26.

[27] THEKKATH, C., THEKKATH, R. A., NGUYEN, T. D., MOY, E., AND LAZOWSKA, E. D. Implementing network protocols at user level. In *ACM SIGCOMM* (1993).

[28] TYMA, P. Thousands of threads and blocking I/O: the old way to write Java servers is new again. In *Software Development Conference and Expo West* (March 2008).

[29] VARGHESE, G. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*, 1st ed. Elsevier, 2005.

[30] VEAL, B., AND FOONG, A. Performance scalability of a multi-core web server. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2007).

[31] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why events are a bad idea (for high-concurrency servers). In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (May 2003).

[32] WELSH, M. *An architecture for highly concurrent, well-conditioned internet services*. PhD thesis, University of California, Berkeley, August 2002.