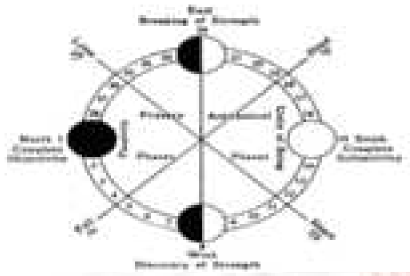
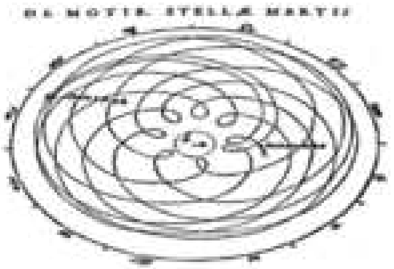
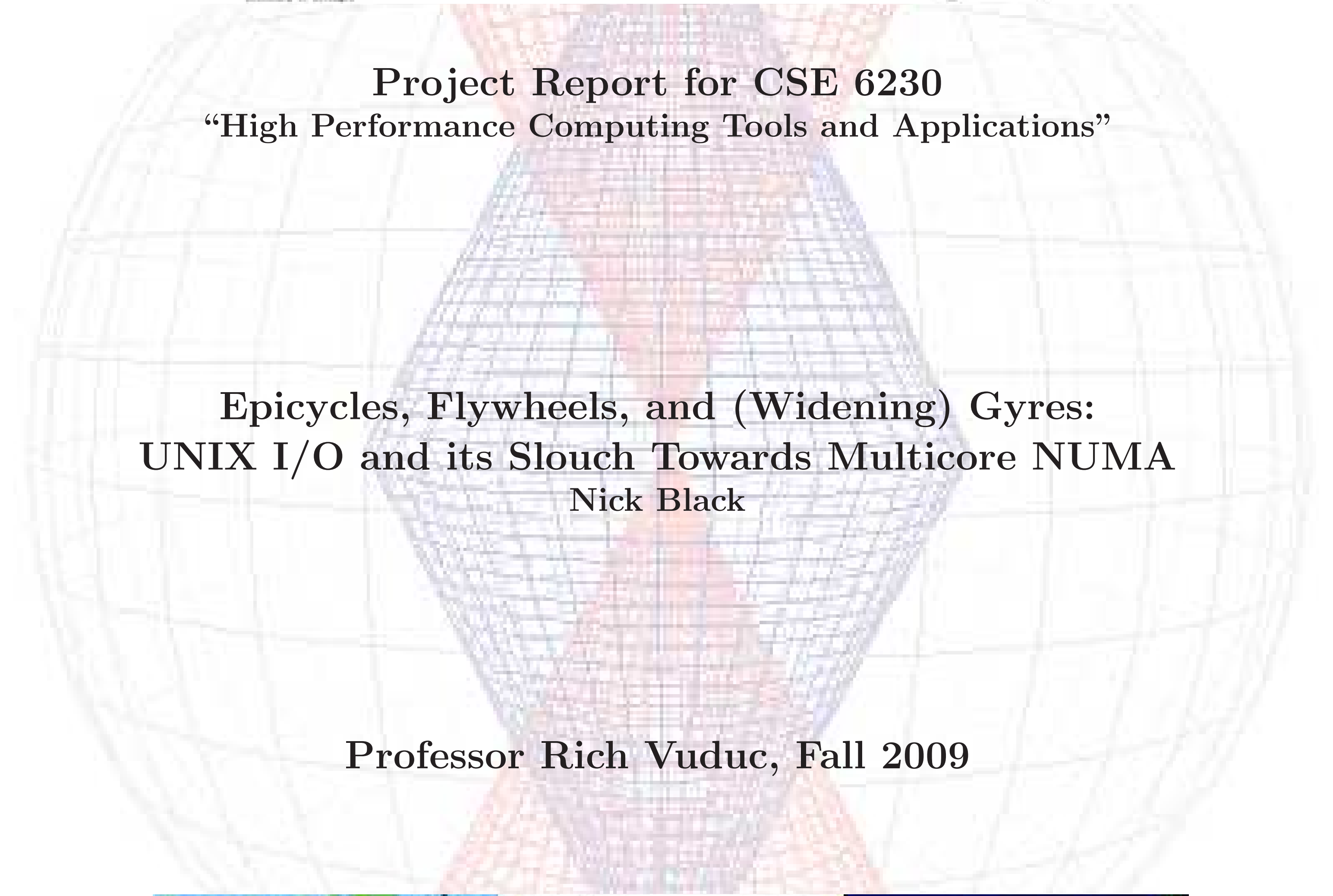






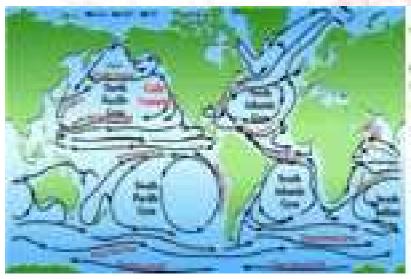
Project Report for CSE 6230

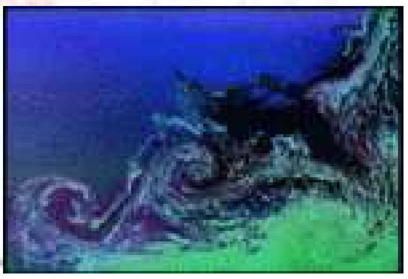
"High Performance Computing Tools and Applications"

Epicycles, Flywheels, and (Widening) Gyres: UNIX I/O and its Slouch Towards Multicore NUMA

Nick Black

Professor Rich Vuduc, Fall 2009





"Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning."—Winston Churchill (1942-11-10)

1 Evaluation Redux

The libtorque project proposal listed a number of high-level objectives. As of 2009-12-10, work has been broken down into 75 bugs of widely ranging complexity (35 remain open[1]). System detection on x86 is complete and authoritative. The basic event wrapper set has been defined and implemented, and "first light" (a cpu-balanced, parallel event-driven application[2], despite no concept of threading calls in the application itself) was achieved 2009-11-17. These two subsystems, currently isolated, will be joined together over the course of next semester's CSE 8903.

The lofty end goals (high-performance, easily-programmed UNIX I/O) of this project are open-ended[3]. Major practical goals were described in the proposal's *"Evaluation"* section, but can be summed up in two statements:

1. An event-driven, (independent) continuations-passing-style implementation ought see performance scale linearly across hardware when augmented with libtorque[4]: **libtorque scales arbitrarily.**
2. The same callbacks, manually threaded using existing event libraries, ought be strictly outperformed by a single libtorque context (despite the simpler interface). As flow uniformity decreases, this performance gap ought increase: **libtorque is arbitrarily dynamic, despite efficient implementation.**

Quantifying and indeed even qualifying success will require the actual manually-threaded implementations for fair comparison. libtorque now provides the minimum functionality to perform such comparisons, once the alternates are written and measurement infrastructure is established. Coarse *per se* analysis via `oprofile` (and, yes, `top`) *has* suggested scaling through two CPUs, but breakdown as flows diverge. The first criterion thus appears met through $n = 2$, but the second criterion (arbitrary dynamism) has not yet been achieved. Furthermore, it can be expected that larger values of n will effect breakdowns in scalability (exploring this will require more substantial hardware).

The author has became aware of two closed-source solutions similar to libtorque:

- Windows Input/Output Completion Ports
- the Solaris Event Completion Framework

Both employ continuation-passing and event distribution as a solution to I/O parallelism (the flexibility and efficiency of their distribution mechanics are unknown, but we can assume them similar to our expectations for libtorque). This seems confirmation of the event API's applicability to the problem; it is thus our clever use of hardware (and open source licensing, of course) which will distinguish libtorque's approach.

Both the Ruby and Tcl projects are involved in similar efforts; the author is in close communication with core Ruby developers, and hopes libtorque might fulfill their needs. If successful, this would lead to substantial distribution of libtorque as a dependency and a dramatic increase in testing exposure. Placing libtorque in the Ruby core is thus a major short-term goal, and rather mundane work (packaging etc) has occupied most recent project time[5].

[1]This does not imply 53% completion. Bug dependency trees unfold as a higher-level bug is investigated (libtorque's longest bug dependency chain is currently 7th order).

[2]The `torquessl` SSL echo server, to be precise.

[3]More correctly, bounded by theoretical maxima which will be nigh-impossible to reach. Establishing these theoretical maxima (indeed establishing their *units*) might emerge as a substantial research extension.

[4]This ought be independent of continuation granularity—even a blocking I/O implementation ought see linear improvement for uniformly distributed flows!

[5]I'm up in your internets, getting built on your servers, w00t.

2 Methodology

A reductionist implementation suggests five distinct, linearly-dependent phases:

1. Implement a single-threaded callback engine.
2. Expose an event wrapper API, built atop the callback engine.
3. Parallelizing the callback engine.
4. System discovery and associated callback engine optimization.
5. Expose a richer client API, built atop system discovery.

This may well have achieved the main goals more quickly, but another ordering exists, one which maximizes external utility. The system discovery component is tremendously useful by itself; despite representing only an optimization overall, it was thus implemented (and made publicly available) first. Releasing a "1.0" of the library has an operational effect: the shared object version steps forward into 1.0, and thus an implicit contract regarding API (newer 1.x versions may augment, but not otherwise modify, the API). It was thus desirable to define the minimal API as early as possible.

That API exists, and is exported by *libtorque.h*. Several proof-of-concept tools have been developed or are planned: `torquessl` has been mentioned, while parallel `inotifty`, packet queueing to userspace, large-scale dns resolution/testing are in the works. The author hopes to embed libtorque into Fyodor's popular and versatile `nmap` tool soon after a 1.0 release (`nmap` holding a cherished place in my heart, and being in desperate need of better parallelism).

3 Toward a 1.0

A 1.0 release is targeted around Christmas[6]. The major efforts of this next semester will be:

- Explore scalability and engage any bottlenecks we find. I believe resource bottlenecks have been largely anticipated: file descriptors, timers, and signals are respectively $O(lgn)$, $O(1)$ and $O(1)$ on CPUs (all are $O(1)$ on memory nodes). I have proved the userspace, at all scales, to be weakly wait-free in the limit[7].
- Begin quantifying event distribution. I'd very much like to include the Windows and Solaris contenders in this measurement. I don't have any idea of how Windows programming works, and had hoped to avoid it forever. It appears to be a prerequisite for use in any number of projects, however (Ruby and nmap being two). Argh...
- Start coming up with the expanded client API. This is largely disconnected from other aspects of the project, meaning it can be arbitrarily delayed. It's also, however, a fascinating subcomponent and potentially a real differentiator. Furthermore, it's another useful component in and of itself. I'd like to get working on this sooner rather than later; it seems the real HPC core.

[6]The documentation needs be brought up-to-date, but the infrastructure is there; libtorque(3) is attached.

[7]That is, whenever there's more time spent handling than dequeueing events (true for all libtorque cases save trivial work or grossly inefficient kernelspaces (yes, `select` or `poll` constitute the grotesque), it's *unlikely* that threads contend. Progress cessation can result only due to external suspension of a thread holding a lock; aside from this possibility, libtorque is strongly lock-free.

```
#ifndef LIBTORQUE_LIBTORQUE
#define LIBTORQUE_LIBTORQUE

#ifdef __cplusplus
extern "C" {
#endif

#include <signal.h>

struct itimerspec;
struct libtorque_ctx;
struct libtorque_cbctx;

// Initialize the library, returning 0 on success. No libtorque functions may
// be called before a successful call to libtorque_init(). libtorque_init() may
// not be called again until libtorque_stop() has been called. Implicitly, only
// one thread may call libtorque_init().
// Create a new libtorque context on the current cpuset. The best performance
// on the widest set of loads and requirements is achieved by using one
// libtorque instance on as many uncontested processing elements as possible,
// but various reasons exist for running multiple instances:
//
// - Differentiated services (QoS): Sets of connections could be guaranteed
//     sets of processing elements
// - Combination of multiple (possibly closed-source) libtorque clients
// - libtorque's scheduling, especially initially, is likely to be subobtimal
//     for some architecture + code combinations; certain situations might be
//     improved using such a technique (I've not got examples, and such cases
//     ought be controllable via hints/feedbacks/heuristics).
// - Trading performance for fault-tolerance (putting less-essential events in
//     their own libtorque "ring" in case of lock ups in buggy callbacks, using
//     distinct alternate signal stacks...).
// - Trading performance for priority separation (see sched_setscheduler()).
//
// If multiple instances are used, the highest performance will generally be
// had running them all with as large a cpuset as possible (ie, overlapping
// cpusets are no problem, and usually desirable). Again, make sure you really
// want to be using multiple instances.
struct libtorque_ctx *libtorque_init(void)
        __attribute__ ((visibility("default")))
        __attribute__ ((warn_unused_result))
        __attribute__ ((malloc));

// Multiple threads may add event sources to a libtorque instance concurrently,
// so long as they are not adding the same event source (ie, the callers must
// be able to guarantee the signals, fds, whatever are not the same). The
// registration implementation is lock- and indeed wait-free.

// Read callbacks get a triad: our callback state, and their own. Ours is just
// as opaque to them as theirs is to us.
typedef int (*libtorquercb)(int,struct libtorque_cbctx *,void *);
typedef int (*libtorquewcb)(int,void *);

// Invoke the callback upon receipt of any of the specified signals.
int libtorque_addsignal(struct libtorque_ctx *,const sigset_t *,libtorquercb,void *)
        __attribute__ ((visibility("default")))
        __attribute__ ((warn_unused_result))
        __attribute__ ((nonnull(1,2,3)));

// After a minimum time interval, invoke the callback as soon as possible.
int libtorque_addtimer(struct libtorque_ctx *,const struct itimerspec *,libtorquercb,void *)
        __attribute__ ((visibility("default")))
        __attribute__ ((warn_unused_result))
        __attribute__ ((nonnull(1,2,3)));

// Watch for events on the specified file descriptor, and invoke the callbacks.
int libtorque_addfd(struct libtorque_ctx *,int,libtorquercb,libtorquewcb,void *)
        __attribute__ ((visibility("default")))
        __attribute__ ((warn_unused_result))
        __attribute__ ((nonnull(1)));

// The same as libtorque_addfd_unbuffered, but manage buffering in the
// application, calling back immediately on all events. This is (currently) the
// preferred methodology for accept(2)ing sockets.
int libtorque_addfd_unbuffered(struct libtorque_ctx *,int,libtorquercb,
                                        libtorquewcb,void *)
        __attribute__ ((visibility("default")))
        __attribute__ ((warn_unused_result))
        __attribute__ ((nonnull(1)));

// Watch for events on the specified path, and invoke the callback.
int libtorque_addpath(struct libtorque_ctx *,const char *,libtorquercb,void *)
        __attribute__ ((visibility("default")))
        __attribute__ ((warn_unused_result))
        __attribute__ ((nonnull(1,2,3)));

#ifndef LIBTORQUE_WITHOUT_SSL
#include <openssl/ssl.h>
// The SSL_CTX should be set up with the desired authentication parameters etc
// already (utility functions are provided to do this).
int libtorque_addssl(struct libtorque_ctx *,int,SSL_CTX *,libtorquercb,
                                                libtorquewcb,void *)
        __attribute__ ((visibility("default")))
        __attribute__ ((warn_unused_result))
        __attribute__ ((nonnull(1,3)));
#endif

// Wait until the libtorque threads exit via pthread_join(), but don't send
// them the termination signal ourselves. Rather, we're waiting for either an
// intentional or freak exit of the threads. This version is slightly more
// robust than calling libtorque_stop() from an external control thread, in
// that the threads' exit will result in immediate program progression. With
// the other method, the threads could die, but your control threads is still
// running; it's in a sigwait() or something, not a pthread_join() (which would
// succeed immediately). The context, and all of its data, are destroyed.
int libtorque_block(struct libtorque_ctx *)
        __attribute__ ((visibility("default")));

// Signal and reap the running threads, and free the context.
int libtorque_stop(struct libtorque_ctx *)
        __attribute__ ((visibility("default")));

```

```
#ifdef __cplusplus
}
#endif

#endif
```

NAME

libtorque − High−performance I/O and primitives

SYNOPSIS

int libtorque_init(*void***);**

int libtorque_stop(*void***);**

DESCRIPTION

libtorque provides the tools necessary for cross−platform high−performance computing and I/O via continuations, scaling from single processors to manycore NUMA architectures. This includes discovery of processing elements, memories, and topology, multithreaded use of edge−triggered, scalable event notification, unification of event sources, advanced scheduling based on architecture−aware allocator, and sophisticated buffering.

BUGS

Search **http://dank.qemfd.net/bugzilla/buglist.cgi?product=libtorque**. Mail bug reports and/or patches to the authors.

SEE ALSO

madvise(2), **mincore**(2), **mmap**(2), **mprotect**(2), **posix_memalign**(3), **sendfile**(2), **sigaction**(2)

On Linux: **aio**(3), **CPU_SET**(3), **epoll**(4), **libcpuset**(3), **numa**(3), **numa**(7), **pthreads**(7), **sched_getaffinity**(2), **sched_setaffinity**(2), **signalfd**(2), **splice**(2), **timerfd_create**(2)

On FreeBSD: **aio**(4), **cpuset_getaffinity**(2), **cpuset_setaffinity**(2), **kqueue**(2), **pthread**(3)

GitHub: **http://dank.qemfd.net/dankwiki/index.php/Libtorque**

Project wiki: **http://github.com/dankamongmen/libtorque**

AUTHOR

Nick Black <dank@qemfd.net>
Design and implementation.

COPYRIGHT