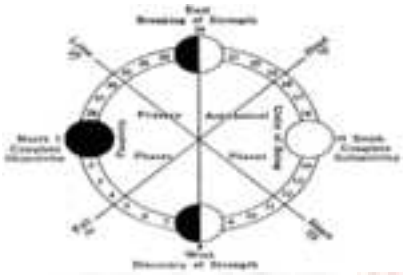
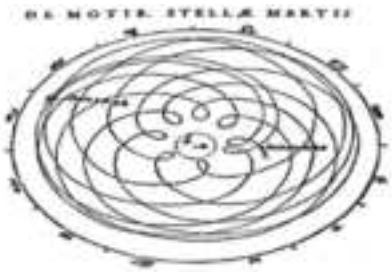




A Project Proposal for CSE 6230

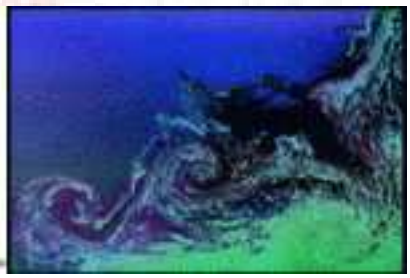
"High Performance Computing Tools and Applications"

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

Nick Black

Professor Rich Vuduc, Fall 2009





1 Forthright assertions. Questions in search of answers.

Why do UNIX servers not regularly support tens of thousands of concurrent clients[1]? Why are TCP starvation attacks like 2008's sockstress (courtesy Outpost24) and 2009's Slowloris (rsnake) effective? Given the general non-interference of transaction-based clients, one might expect linear or superlinear speedup across cores. Authorities no less magisterial than Patterson and Hennessy point out that, when shaped favorably for the architecture, web and transaction benchmarking suites (processes with inherent, heterogeneous functional parallelism and low-to-moderate communication needs) parallelize well on modern UNIX operating systems across widely disparate machines. Open-source libraries like libev, libevent, and liboop can support thousands of concurrent non-blocking clients per thread via a continuations model implemented atop Linux's `epoll`, FreeBSD's `kqueue`, or Solaris's `/dev/poll`, especially when combining carefully-written monads with today's large caches. Outside of the laboratory, and especially under attack conditions, real-world applications built atop these libraries fail to maintain this performance. The event libraries listed are oblivious to topology and architecture; I believe it impossible to effectively manage UNIX events without taking these parameters into account, and that it is unnecessary to foist these issues on application developers.

> "Thread scheduling provides a facility for juggling between clients without further programming; if it is too expensive, the application may benefit from doing the juggling itself. Effectively, the application must implement its own internal scheduler that juggles the state of each client."
>
> — George Varghese, *Network Algorithmics*

Advanced programming in the UNIX environment was defined in scope and idiom by Stevens's 1992 book of that name[2], and approaches adulthood. The POSIX 1003.1c threading standard enjoys (finally!) complete and reasonable support on all major platforms. FreeBSD and Linux's high-performance, non-standard extensions have emerged as active but predictable targets of sustainable opportunity. Huge page support will be by 2010 a universal reality, hardware sensors are coming into their own, power management's no longer quite so broken, and affinity/migration APIs are... well, this is still *le meilleur des mondes possibles*. The GNU Compiler Collection, forked for years along (at least) two lines, was reunited with the EGCS project in 1999. Competition from Intel's high-quality C++ compiler has seen the replacement of RTL with GENERIC and GIMPLE, full SSA integration, the RABLET register allocator, and proper templating/exception support. Major industrial support, especially from Intel, ensures top-quality drivers for enterprise hardware (XHCI was supported first on Linux, as were I/OAT and the E1000 chipset's TCP Offload Engine). Experimental API extensions are added when deemed necessary, and a strong technical argument remains the surest passport to change. It's a good time to be a UNIX systems programmer.

Multicore in the mainstream arrived on the x86 architecture only in 2006 (Conroe and Merom Core 2 Duos and the Yonah Core Duo, all on 65nm processes, and AMD's Denmark Opteron, on 90nm process). Scalability efforts too numerous to footnote, performed with an eye toward polyslot unicore ("Nineties SMP"), HyperThreading ("Nineties SMP, now with SMT for a New Millennium") and some mysterious IBM NUMA machines kept under tarps at DoE[3] left FreeBSD 7.x and Linux 2.6.x kernels well-positioned to take advantage of a brave new multicore world. Subtle hardware serialization points (eg, globally-shared TLBs) have been tamed via hardware-software combinations (large pages and transparent OS support thereof, distributed directory memories, IO-APIC and interrupt distribution, interrupt coalescing, on-die IOMMUs, DCA, write-combining in cache and on I/O buses, lock elision...). NPTL, the Linux threading implementation of record since 2004, benchmarked 200,000 threads being spun up and destroyed in seconds.

All the pieces are there[4]. It's high time UNIX application developers had a robust, parallel, architecture-sensitive unification of all their various event sources, engineered with explicit consideration for multilevel and non-uniform memories[5]. I intend to provide it; enter *libtorque*.

[1] See Dan Kegel's excellent "The C10K Problem.", and my own farrago.
[2] Competently revised and extended by Stephen Rago in 2005's second edition!
[3] "-mgb, Martin J. Bligh's Patchset."
[4] For an example open source application doing things the hard way, look at nginx
[5] Distributed systems are explicitly beyond the scope of this effort.

2 Goals.

This project's goal is to initiate, and advance to a credible state, an open source (LGPL- or BSD-licensed) library offering high-level, portable, robust use of multiple operating systems' high-performance extensions, particularly those related to:

- low-level system discovery,
- system resource acquisition (execution units, store partitioning),
- minimal-copy data movement (zero-copy networking[6], `splice(2)`),
- I/O events (e.g. files, devices, filesystems, sockets and networks),
- and scheduling.

Cases where the existing libraries break down must be synthesized, along with those optimistic situations which can already be handled (the DoS attacks listed in Section 1 provide a convenient starting point). Our library ought rectify breakdown cases, while maintaining the high laboratory performances available with current solutions. Furthermore, our API ought make self-evident inroads in terms of resulting program organizations, scalability, and robustness (especially in the face of hardware failures).

The differences between (and implications of) level- vs. edge-triggered event models will be investigated and precisely defined. Classic POSIX functions such as `select(2)` and `poll(2)` are level-triggered; while (arguably) simpler to use, I conjecture that concurrent, stateful, robust I/O systems are expressed more naturally in an edge-triggered context. This question is a passionate and pertinent one in especially the Linux community, where the value of edge-triggered I/O, if it indeed exists, has yet to be definitively or even quantitatively articulated.

As example programs and test sets, and also to provide valuable components to future users of the library, multithreaded DNS and SSLv3/TLS capabilities will also be implemented (likely as wrappers around GNU adns and either OpenSSL or GnuTLS).

3 Targets.

The abstract primitives required will include:

- Memory hierarchy and execution topology discovery
- Fast POSIX threading (Linux's NPTL, FreeBSD's libthr)
- Edge-triggered, stateful I/O reporting

while the following are recommended:

- Memory-mapped I/O and remapping of kernel VMAs
- Access to performance-monitoring counters

Linux 2.6.31[7] and FreeBSD 7.2 implementations, with support for x86 architectures from the Pentium Pro to the present day, are included in this project's scope.

[6] Van Jacobson channels and userspace networking stacks are explicitly outside this project's scope.

[7] Debian Unstable, using eglibc 2.10

4 Timeline.

I assume 20 hours' *good*, accountable work per week, and another 20 to 30 "unbillable" hours atop that. With seven weeks left, it'll be quite a whirlwind:

1. 2009-10-23 — Break ground on public-readable, private-write `git` repository with autobuilding and reporting capabilities. Determine and include licensing terms, copyright, and this documentation. Erect unit and regression testing as an essential part of the build.
2. 2009-10-30 — DNS reference client and server, SSL transaction reference client and server implemented using libev or libevent (both?).
3. 2009-11-13 — Networking and userspace event infrastructure of *libtorque* (`accept(2)`, non-blocking socket I/O, asynchronous I/O, condition variables, semaphores, RCU). Ported DNS and SSL components. *libtorque* is managing threads, but dispatching only on connection initiation. This ought equal the performance of existing solutions, with perhaps a small constant increase.
4. 2009-11-20 — **Checkpoint**. *libtorque* is moving event sources between event queues, dispatching work across processors, reclaiming and spawning threads in response to CPUSET changes, and balancing event notification against handling. Memory allocation is integrated (though primarily per-CPU) with centralized resource management and topology awareness. DNS and SSL component ports ought be performing better than original implementations, all other things being held equal, across the evaluation space. Robust concurrent handling of the full TCP[8] event matrix is demonstrated via formal analysis at the level of predicate transformations and Petri nets, making use of natural sequencing points provided by the operating system (eg recycling of file descriptors). **Risk-handling strategy:** if we are not outperforming with this functionality set, dial back the project's "extras"; exploring this space is the most critical research objective. **Risk:** A dialed-back library will not likely suffice for serious UNIX application development.
5. 2009-11-27 — Have answered: "why bother with edge-triggered I/O?" Develop abstractions for buffering around `splice(2)`-style system calls and `mremap(2)`-like capabilities. Event interaction APIs finalized, based off previous results. Watch GT take UGA to the woodshed. This ends the theoretical portion of the project.
6. 2009-12 — Polish and extend. Integrate filesystem and network notification interfaces, disk I/O, signals. API for job control of tracked children and spawned daemons. Integrate performance monitoring via libpfm, `rusage(2)`, watermarks. Begin moving to demultiplexing via protocol discovery, rather than port, and thus to my masters thesis...

5 Evaluation.

The libev/libevent reference implementations ought attain linear augmentation of steady-state throughput, or close to it, for each additional core, atop "reasonable" per-core service provision (in terms of both latency and throughput). It is critical that latency not be impacted negatively by adding cores.

libtorque-based implementations ought perform as well or better than this baseline for the simple test cases, and address issues that affect baseline performance (bursty clients, long fat pipes, mixed-throughput connections, denial-of-service attacks, task migration vs data and code locality). A noticeable, quantitative improvement should be seen, especially against denial-of-service attacks.

Furthermore, the UNIX multithreaded networking paradigm must be qualitatively improved. How will this be measured, save "we'll know it when we see it?" I am unsure; perhaps only shared experience can tell. If we're punting evaluation to the tired, the poor, the huddled masses of UNIX application programmers yearning to breathe free, the library must be professionally usable. Thus, some effort to publicize the library, and gain feedback, ought be made following the 2009-11-20 checkpoint.

[8] SCTP?

6 Addendum A - AMPED [1999], LAIO [2004]

Addendum regarding prior art

```
Date: Wed, 21 Oct 2009 03:39:21 -0400
Subject: [cse6230] project-relevant prior art found

Prof. Vuduc,

I followed a link to http://www.cs.rice.edu/~kdiaa/laio/ earlier today, and
just got around to reading it. I'm not sure how I missed this paper for five
years (especially given Alan Cox as an author), but it's highly relevant to my
project proposal.

On the positive side, my core advocacy of a dynamic continuations-based
model preferring asynchronous (as opposed to non-blocking) I/O seems
validated. On the negative side, that part's no longer research. The research
focus is thus shifted to:

 - that asynchronous i/o being delivered to a subset of execution units
    "clockwise-forward-intraNUMA from the queueing execution unit" (need
    a good term yesterday) can be "swirled" into a Pareto optimum (this is
    really the heart of my proposal, and the most aggressive theoretical work)
 - centralized buffer management tied deeply into the detected node topology,
    memory hierarchy, and event pattern leads to big wins (keep per-core I/O
    buffers from aliasing the core-shared fdesc tables, don't break VMA's up
    across cores, provide non-temporal (non-cache-polluting) areas/methods,
    prefetch/prime or lock pages onto a core as events are rotated into them
    functional decomposition across cores for gyral dataflow + static IC etc)
 - threading being embedded in the event system rather than multiple app
    threads calling into the event system (the whole point, it seems to me,
    of a monad-based continuations scheme)
```

7 About this document.

Images, clockwise from upper-left: Edmund Dulac's *Great Wheel*. Woodcut, 1937, accompanying William Butler Yeats's *A Vision, B.* Public domain. V8 flywheel and clutch conversion kit, copyright UUC Motorworks. Used with permission. *Orbit of Mars, 1580–1586*, from Kepler's *Astronomia Nova* (1609). Public domain. World ocean currents and major gyres atop Mercator projection courtesy of ITA, Inc., used with permission. Flywheel compulsator, from the 1998 UPEI Physics 261 project of Mr. Gerry Sheppard, permission pending.

Center image: *The Diamond and the Hourglass*, an illustration of Yeats's "Double-Gyre"/"Dual-Vortex" from Neil Mann's masters thesis, *W. B. Years and "A Vision"*. Used with permission. Lightened by Robert Stallworth for this author.

8 Undirected musings best ignored.

Does viscous flow not seem to have a deep connection to the optimal movement of I/O across current architectural models? I think so... *Wovon man nicht sprechen kann, darüber muß man schweigen...*