epicycles, flywheels, and (widening) gyres:

UNIX Network Programming in a Manycore NUMA World

It's high time UNIX application developers had a robust, parallel, architecture-sensitive unification of their various event sources, engineered with explicit consideration for manycore processing environments and non-uniform memory access.

I intend to provide it. Enter *libtorque*.

nick black <nickblack@linux.com> for gt arch-whiskey 2009-11-13

copyright © nick black 2009. information wants to be free: licensed under the creative commons attribution 3.0 license (unported).

Hmm. I was told there would be computer architecture research?

- I/O is indeed the red-headed stepchild of Architecture and HPC
  - Dependencies on (woeful) buses, (divergent) devices
  - Irregular, control and dataflow-wise
    - Branch predictors need not apply
    - Hardware prefetching's worst nightmare
  - Largely untamed by the locality principles
- “Simulator economy” is built atop SPEC CPU
  - SPECWeb doesn't drive architecture publishing
    - P&H4 mentions it once, P&H2 not a single time.
- Penryn's SSE4.2 “networking instructions”: Fast string operations
  - Sun's Rock, T1, T2 are doing something a bit more interesting
  - (MOVNTDQA ends up the network-relevant SSE4 instruction)
- How do the other half live...?

copyright © 2009 nick black <nickblack@linux.com>

Network programmers returned the warm feelings

- UNP3, APIUE2: no mention of caches
  - These are great, canonical books. Still, not a single word!
  - Very minimal coverage of `mmap()`
  - Dirtying cache and the memory wall weren't yet so relevant!
    - Linux was developed on 16-33MHz 80386's
- Threads introduced primarily as a control mechanism in UNP2, UNP3.
- Described from an API perspective in APIUE2, Butenhof
  - Neither go into performance details, especially architectural
- No standard interfaces export architectural details to userspace
- Protocols designed without architectural (alignment, cache) concerns
- The game was all about saving system calls and writing fast `poll()` loops.
- There is a spectre haunting the Internet!
  - Underabundant architectural wherewithal hobbles UNIX programming

copyright © 2009 nick black <nickblack@linux.com>

Internetworking enters the mainstream, with a target of “C10K”

- The Internet's explosion bears three great networking legacies:
  - Wave-division multiplexing and coastal abundances of SMF
  - Utter dominance by Ethernet and the Inverted Hourglass
  - Shift from `poll()` to asynchronous I/O and stateful event queues
- What university/company can provide the most FTP service?
  - cdrom.com: 10,000 concurrent connections on 1Gbps NIC (1998)
- A golden age of radical and competitive network server design
  - User-space networking (Mach, vJ channels, CSPF, BPF, PathFinder DPF)
  - Powerful, flexible open source networking stacks. ACE. Erlang.
  - User/kernelspace blend (Clark upcalls, Tuxhttpd, ADC/VIA)
  - Scheduler activations, AMPED, LAIO, threads in pools and on demand
  - LinuxThreads/NGPT/NPTL, libc_r/libkse/libthr, GNUPth, oh my!
- No clear winners (look at Apache). Machines got faster, and cheaper.
  - Dan Kegel (CT) collected state-of-the-art at “The C10K Problem“.

copyright © 2009 nick black <nickblack@linux.com>

Digression: Stateful event queues on Linux and FreeBSD

- Linux introduced epoll() in 2.5.44 (2002-10)
- FreeBSD introduced kqueue() in 4.1 (2006-04)
- Both return a file descriptor (int epoll_create(int), int kqueue(void))
- Notification masks are kept in the kernel:
  - epoll_ctl() performs EPOLL_CTL_ADD, EPOLL_CTL_DEL (1 op per invoke)
  - kevent() takes both a changeset and eventset (N ops per invoke)
- Unification of (some) event sources:
  - EVFILT_READ, EVFILT_WRITE, _AIO, _SIGNAL, _TIMER, _VNODE, _NETDEV
  - epoll is fd-only; use 2.6.25's signal- and timerfds (or epoll_pwait())
- Both level- and edge-triggered modes:
  - EPOLLET with epoll_ctl(), or EV_CLEAR with kevent()
- On Solaris: /dev/poll, and all Event Ports.
  - Ummm, honk if you ♥ Solaris.

copyright © 2009 nick black <nickblack@linux.com>

Digression: POSIX Asynchronous I/O

- Please do not confuse with F_GETOWN, F_GETSIG args to `fcntl()`!
- Defined in POSIX.1b. Only recently (circa 2007) well-supported on Linux.
- struct aiocb provides the user/kernel data definition
- aio_write(), aio_read(), aio_fsync()
- User notification: SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD
- aio_suspend, aio_cancel, aio_error, aio_return, scary aio_init

- Asynchronous I/O would always be more effective in a perfect world
- Our world is not perfect, and includes:
  - Very short I/O transactions, able to be completed “immediately”
  - Heavy instruction footprints in the kernel, significant signal overhead

copyright © 2009 nick black <nickblack@linux.com>

Digression: Threading considerations regarding I/O primitives

- Epoll and kqueue file descriptors (henceforth “efds”) are themselves pollable, returning a read indication if they have events ready.
- Efds may be added to each other's event notification masks, but not their own (this will result in EINVAL).
- Closing a file descriptor removes it from all efds' event notification masks, and purges outstanding events, ***iff*** there are no dup(2)d copies open. It is safe to do this while a thread is waiting on the efd.
- Threads may make concurrent use of efds, even with inexclusive event notification masks; all are reported any events for shared fds (in O(n)).
- epoll_wait() is a cancellation point. kevent() is not.
- A thread may manipulate another efd's event notification mask, even if some thread is blocking on that efd.
- It is not specified whether concurrent uses of the same efd are safe.

copyright © 2009 nick black <nickblack@linux.com>

The story as of 2006

- Linux's networking stack has become a multithreaded beauty of a beast
  - NAPI, IRQ distribution/coalescing, TOE, RCU, universal jhashes, tc
- Zero-copy networking via sendfile() and/or COW is developing
  - fbufs, IO-Lite, TCOW, scatter-gather (readv() and writev())
- Kernel development moves to throughput, wireless, power saving
  - Low-latency, scalable userspace primarily a focus of financial sector(!)
  - Dynamic page generation chews up the cycles of HTTP servers
  - Responsive event-based networking is easy. Cost effectiveness is hard.
    - Launch a thread per conn, let 'em fight it out in the scheduler.
  - Laboratory loads are easily served. Attack loads are hard.
    - Distribute event sources statically among per-CPU threads.
  - High-throughput event-based networking is easy. Low latency is hard.
    - Sub-ms I/O requires computation techniques from HPC.
- George Varghese publishes the inimitable *Network Algorithmics*
- Drepper's "What Every Programmer Should Know About Memory"
- Around the world, programmers find out everything they know is wrong.

copyright © 2009 nick black <nickblack@linux.com>

Moore's Law sounds a barbaric YAWP over the rooftops of Santa Clara

- AMD keeps the good times rolling with dual-core 90nm Denmark Opteron
- Intel comes back with 65nm Yonah (CD), Conroe and Merom (C2D)
- Martin J. Bligh's mjb- patchset, SGI's userspace tools, FreeBSD 7:
  - Linux by now scaling to thousands of processors (hierarchal RCU etc)
  - NUMA support for Linux and FreeBSD
  - Ingo Molnar's 4G/4G patch for systems with ~64G of RAM
  - FreeBSD 7's new ULE scheduler brings it back into the 16+ game
- CPUSet (thread and process affinity masks) are added to Linux, FreeBSD
- L2 and giant L3 caches, streamlined coherence, HT and QPI, DDR2 and 3
- Gradual replacement of ferromag+mech storage with MLC FM / NROM
- Multicore + Vtx (Vanderpool) / SVM (Pacifica) → Virtualization everywhere (hear that? the sound of more packet-handling latency?).
- IOMMU's hit the mainstream, clearing the path for DCA

copyright © 2009 nick black <nickblack@linux.com>

Open source networking multicore-readiness (as of 2009-11)

- Apache 2.2's can serve from a naïve worker pool. **FAIL?**
  - If you're not using an MT-unsafe module.
  - Like say, PHP. **FAIL.**
- OpenSSH 5.1 uses threads only during authentication. **FAIL.**
  - Take a look at Pittsburgh SCC's HPN-SSH
- OpenVPN had threading in 1.0, for authentication only. **FAIL.**
  - It was purged for 2.0. **DIRECTION FAIL.**
- Each can spawn or entask processes per connection, but (as always) this wastes cycles and memory, neutralizes cache, and binds to CPUs only at the cost of responsiveness in asymmetric loads.
- nginx... is unthreaded, **FAIL**! nginx must be spawned; this is actually a more reasonable model.
  - If you can't thread intelligently, please, don't thread at all. I'm looking at you, Apache's MinSpareThreads and MaxSpareThreads!
- libev, libevent, liboop, coronet, GUASI: Not one manages threading.
- One project is attempting something similar...Tcl-CORE, of all things(!)

copyright © 2009 nick black <nickblack@linux.com>

Why insist on internal scheduling?

- Why the emphasis on controlling scheduling in this lowest layer of code (the event handler)? Everyone else is getting away with ignoring it.
  - We're not controlling scheduling as a first-order objective. We assume that the CPUSet provided on entry is to be utilized completely, pin ourselves to each CPU in turn, and deposit a thread there. They don't compete.
  - The actual “scheduling”, in terms of what processes run where, when, is a secondary effect of I/O availability. We schedule the handling of reported events; if there are events to handle, threads will run.
- What's wrong with an event interface per process, a process per core?
  - We must be able to redistribute either event sources or event instances, or gross asymmetries can exist no matter the scheduler. These processes themselves would need bifurcate; just do it and be done.
- What about adding all event sources to a universal notification mask?
  - All listening threads would receive event notifications. Think thundering herds, a need to lock on almost any event, and increased latency.
- Only one process could block on the efd at a time, perhaps?
  - We could share the efd, sure, but we'd need also share all referenced file descriptors. That's an awful lot of SCM_RIGHTS passed over PF_UNIX! FreeBSD can't do this at all without `rfork(RFFDG)`.

copyright © 2009 nick black <nickblack@linux.com>

Multiprocessing effects: grokked. What about µ-architecture and NUMA?

- NUMA is relatively new to both Linux and FreeBSD, and essentially unheard of in the open source application space.
- We're still waiting on a comfortably-scalable GNU Libc `malloc()` (no, replacing Lea's with Gloger's `ptmalloc` only got to ~8), while FreeBSD's jemalloc has no NUMA support. Since we're already largely per-core, Hoard or tcmalloc are a bit heavyweight; we generally oughtn't lock.
- Our event distribution scheme is parameterized almost completely by system topology, which is improved by knowledge of NUMA properties.
- µ-architecture will play a major role in our allocation API; buffers especially will be shaped and placed in consideration of cache and TLB parameters, as will data known to be shared or unshared between various types of connections.
- Cache and TLB parameters will shape the frequency of callbacks when processing large amounts of data.
- Read-only data shared among connections will result in strong localization for those types of connections. Efforts will be taken to ensure unshared data is not falsely shared.
- Instruction cache and TLB parameters will shape event distribution.

copyright © 2009 nick black <nickblack@linux.com>

Digression: POSIX Multiprocessing APIs are Decadent and Depraved

- I'll let the screenshot speak for itself:
- Pretty underwhelming, team.
- Stay classy, x86.

copyright © 2009 nick black <nickblack@linux.com>

Programmatic, deterministic detection and enumeration of topology

- From `libtorque/hardware/topology.h`:

```
// The scheduling universe is defined by an ordered set of $N > 1$ levels
// $L_0..L_n$. A scheduling group is a structural isomorphism, a counter $C$ of
// instances, and the $C$ affinity masks of corresponding processing elements.
// A level contains $N > 0$ scheduling groups and a description of hardware
// unique to that level. A unique surjection from classes of usable hardware to
// levels (no hardware class is described at two levels, and all usable
// hardware is described by the scheduling universe) is defined by via our
// discovery algorithm.

// The reality compactifies things a bit, but in theory:
// Level L_0 consists of scheduling groups over threads and processor types.
// Each successive level $L_{n+1}, n >= 0$ extends $L_n$'s elements. For each
// element $E \in L_n$, extend $E$ through those paths reachable at equal cost
// from all elements in $E$.
//
// Examples:
// A uniprocessor, no matter its memories, is a single topology level.
// 2 unicore, unithread SMP processors with distinct L1 and shared L2 are two
// topology levels: 2x{proc + L1} and {L2 + memory}. Split the L2, and they
// remain two topology levels: 2x{proc + L1 + L2} and {memory}. Combine both
// caches for a single level: {2proc + L1 + L2 + memory}. Sum over a level's
// processors' threads for a thread count.
//
// Note that SMT does not come into play in shaping the topology hierarchy,
// only in calculating the number of threads in a topological group.
```

- We do not yet consider distributed systems, though they ought work in this model. One real problem is scaling failure with a complex base level.

copyright © 2009 nick black <nickblack@linux.com>

Affinities, APICs, and Process Identifiers, Oh My!

- Four meaningful maps exist on our allocated processing elements:
- Affinity ID, used with the CPU affinity subsystem (aid < CPU_SETSIZE)
  - We probe our inherited affinity mask on initialization to determine allocated processors (the only ones we care about).
  - We pin in succession, spawning a thread on each. This thread detects the CPU, and begins blocking on its efd.
- APIC (Advanced Programmable Interrupt Controller) ID. 8 or 32 bits.
  - This is how IPI's are addressed, and how the BMP and Aps are chosen in the Intel Multiprocessor Specification (current as of v1.4). 8 bits.
  - Since Nehalem Core i7's, x2APIC support is provided. This yields a 32-bit EAPIC (EAPIC % 0xffu == APIC).
  - x86 topology is determined via bitslicing of APIC's
  - AMD supports the same values, but with different names and CPUID methodologies. Thanks bunches, AMD!
- Our topology structure (libtorque_topt) knows each schedulable processor, and maps them to libtorque_cput's.
- After initialization, a bijection exists between processors and tIDs.

copyright © 2009 nick black <nickblack@linux.com>

Epicycloidal movement around the scheduling group isomorphism

- Unshared memory? Space it out:

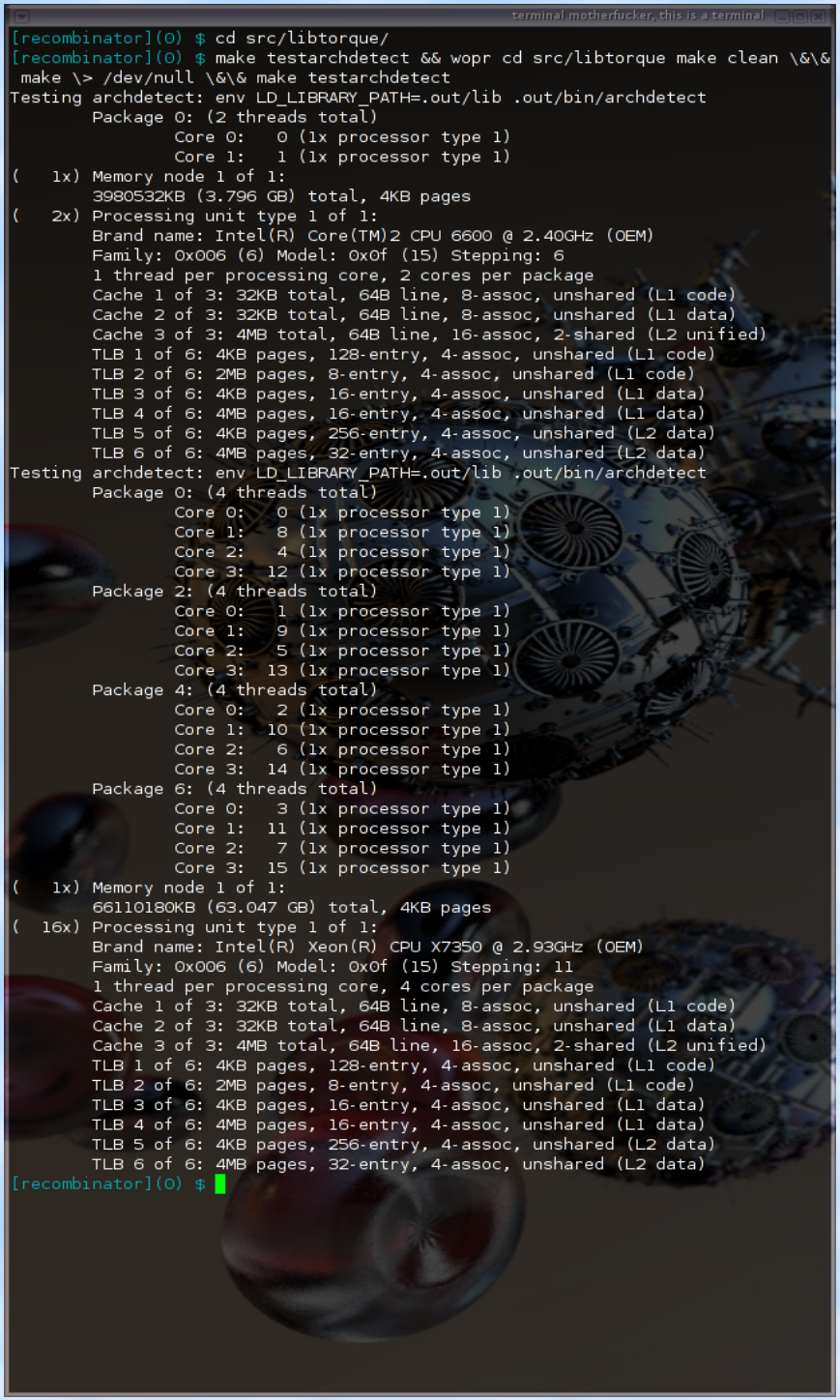
Split up the resource...	...for valuable results.
NUMA nodes	Balance memory bandwidth, use
Physical packages	Effectively utilize different caches
Physical cores	Engage all execution units
Logical cores (if data won't conflict)	Hide latency

- Sharing memory? Huddle together

Logical cores (if sufficient cache)	Minimize trips off-core
Physical cores (if sufficient cache)	Minimize trips off-die, save power
Physical packages	Minimize interconnect bw consumed, save power
NUMA nodes	Minimize memory bw consumed Minimize memory consumption

- Decide about exploiting logical siblings based on distribution of shared accesses (how?)
  - Shared writes are important to keep close (minimize true coherence)
  - Unshared even more important (minimize false coherence)

copyright © 2009 nick black <nickblack@linux.com>



Let's see one of these topologies!

recombinator:
4GB / 4MB unified / 32K each,
Core 2 Duo 6600 @2.40GHz
1 package, 2 cores, 2 PE's

dumbledore:
128GB / 8MB unified /
2x Core i7 E5520 @ 2.27GHz,
256K unified / 32K each
2 packages, 4 cores, 16 PE's

WOPR:
64G / 4MB unified /
4x Xeon X7350 @2.93GHz,
32K each
4 packages, 4 cores, 16 PE's

(output from `archdetect`, commit id dc45b508fa937ce77bd58985ed84b9e63b1c376e)

copyright © 2009 nick black <nickblack@linux.com>

Can't the operating system or libc make these decisions better?

- They probably could, if they provided the necessary interfaces. They don't; OS/libc interfaces are hard to change once issued, and this isn't general enough for either.
- We introduce the idea of *connection coloring*, where the location (modulo appropriate cache block sizes) and size of known per-connection allocations provide a jigsaw-like method for avoiding cache conflicts.
- Userspace networking won't come to libtorque until 2.0-series development. Until that time, we're resigned to at least one memory→memory copy during RX. Perform this mem→reg with minimal cache.
- Working out our instruction footprint is trivial with `dl_iterate_phdr()` and libelf, especially with DWARF debugging info (see `pfunct -s`). We can determine what's likely to step on one another, and distance them; we can determine what's likely to exploit locality, and bind them.
- Remember: Under a 3G/1G, 2G/2G, or 1G/3G split, the kernel lives in its own slice of the (shared) address space. We can still trash one another's TLBs and caches, though; detect this, or walk the kernel binary.
- If that turns out to be the case, we just don't include it! Libtorque is all about staggered improvement with each step while maintaining scale.

copyright © 2009 nick black <nickblack@linux.com>

What all can we exploit on a modern server?

Resource	What we can do	Done yet? Bugs entry?
Differentiated x86 functionality (SIMD, etc)	Expose the best code for large buffer copies, etc	PARTIAL. Detect it, but neither feedback nor choices to make.
Logical cores (SMT)	Run more threads of our (memory, branch, syscall)-intensive code on one core, hiding latency.	YES. We fully detect topology, map it to affinity ID's, and schedule on all SMT cores while breaking up unrelated sources.
	Share L1 cache, TLBs.	NO. Tie-breaker for logical core dispatch needs dep on sharing.
Multiple cores	Run more threads.	YES. We dispatch to distinct cores before overloading logical cores.
	Share L2 cache, NUMA.	NO. See SMT-2.
Cache line lengths	Align our arena allocator.	NO. We detect cache topology and properties, but there's no feedback.
Hard-affinity APIs	Prevent context switches, some migrations.	YES. We pin the main thread to each CPU in turn, then spawn.
Multisize TLBs	Minimize TLB cost	NO. See “Cache line lengths”.
Direct cache access	Avoid cache conflicts	PARTIAL. Detect it, but no feedback.
DIMMs	Minimize precharge delay	NO. SPD→I²C→SMBIOS→DMI.
Atomicity of aligned ops	RCU	N/A. All hotpaths are atm lockless.

copyright © 2009 nick black <nickblack@linux.com>

Continuation-Passing Style

- Term coined by Sussman and Steele (1975) in AI Note 349
  - “Continuation” itself is due van Wijngaarden (1964) regarding Algol 60
- A *continuation* sums up control state, to which one will later return
  - Compare and contrast: GNU Pth, C exception libraries, `setjmp()`
  - Compare and contrast: Conway and Knuth's *coroutines*
- The continuation-passing style consumes a function representing computation not yet done, aborting the computation by returning a value.
  - C doesn't deal well with saguaro stacks; we return a value other than 0 to abort. We exploit edge-triggered events to achieve lockless handling, though, so a fresh event mask must be reestablished each call.
  - Nota bene: a transaction-based protocol like HTTP can still churn us right off the stack without explicit limiting of C-Passings.
    - Nota bene: We need an event cache anyway (see `epoll(7)`), so this can be cleanly addressed there.
- The standard approach to extendable event machines; see Alan Cox et al's “Lightweight Asynchronous I/O” (2004), Flash's “Asymmetric Multiprocess Event-Driven” (AMPED) architecture (1999), or any of the major unthreaded event libraries.

copyright © 2009 nick black <nickblack@linux.com>

Flywheel Economics

- Consider a design space **S** with *n* parameters of *m* criteria. Map criteria vectors {Y} via *f*: $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$. **y** $\subset$ {Y} strictly dominates **y*** $\subset$ {Y} (**y**$\succ$**y***) iff:
  - $(\forall i, i < dim(y) \rightarrow y_i \leq \mathbf{y}^*_i)$ and $(\exists i, i < dim(y) \rightarrow y_i < \mathbf{y}^*_i)$
- The *Pareto Frontier* consists of tuples in {Y} not strongly dominated.
  - These are the Pareto efficient choices for a round in **S**
- Pareto efficient choices can only benefit a player to no player's detriment
- If the Pareto frontier is empty, the design is **Pareto efficient**.
- We distribute events through the system, defined between and within the *n* scheduling groups of each topology level (*recipients* {R}), using:
  - m_0: outstanding events in **r** $\subset$ {R}
  - m_1: number of rounds since we gave **r** events
- What is a round? A return from `kevent()` or `epoll()`, of course...

copyright © 2009 nick black <nickblack@linux.com>

και είπε ο Θεός γενηθήτω φως και εγένετο φως

- Each thread has an efd, prepared with one event mask: a RXfxn on some signal used internally by libtorque. The system's initially quiescent.
  - We do not consider this wake-up signal an event below
- Scheduling groups are sorted, descending, on $M = \mathrm{MST}_{\mathrm{Controlled}} / \mathrm{MST}$. For successive tiebreakers on S(M), choose the largest Wiener index relative to already-sorted processors.
- Each neighborhood initializes a (tiny) shared scoreboard (scoreboarding is lock-free (actually wait-free), but we can't get around basic sharing). Uniformly distribute initial indices into a common, large event vector.
- The application, at any time, adds external event sources. Sources might be configured with allocation triggers, and/or classified as one of self-collaborative or self-divergent (self-divergent is the default). *Lux fiat.*
- Eventually, an event source is triggered. The associated thread stops blocking and considers its e events. If $e == 1$, handle the event directly.
  - Otherwise, calculate the Pareto frontier of p possible moves in our n-neighborhood. Select one at random, and offload $e/2$.
  - If a slacker existed, it is signaled. Some thread picks up the efd.
- All internally-created event sources are Pareto-distributed (globally).

copyright © 2009 nick black <nickblack@linux.com>

hack on, hack on.

- Ground was broken on libtorque almost exactly three weeks ago.
  - 365 commits (git log | grep ^commit | wc -l)
  - 2232 ChangeLog lines (git log | wc -l)
  - 51 bugs filed, 30 outstanding
  - Bugs found, reported in: x86info, libcpuset, eglibc, gcc's ipa
    - Patches provided for the first three
- Full x86 support, more complete than any other open tool (AFAIK)
- Linux and FreeBSD support
- Just got access to a Sun Niagara 2
  - Expect SPARC and OpenSolaris support by next week
- Aggressively open source `(git clone git://github.com/dankamongmen/libtorque.git)`
  - GitHub hosting: http://github.com/dankamongmen/libtorque
  - Bugzilla: http://dank.qemfd.net/bugzilla/buglist.cgi?product=libtorque
  - Wiki: http://dank.qemfd.net/dankwiki/index.php/Libtorque
  - Mailing list: http://groups.google.com/group/libtorque-devel

copyright © 2009 nick black <nickblack@linux.com>

Thank you, you've been great!

come write code with me:

let's change the world through high-performance computing.

go jackets, sting 'em!

copyright © 2009 nick black <nickblack@linux.com>