DragonFlyBSD

WWW.DRAGONFLYBSD.ORG

The Theory behind DragonFly's MP Implementation

- Develop algorithms which are naturally cpu-localized and non-cache-contended.
- Remove non-critical elements from critical paths.
- Avoid cache contention whenever possible.
- Implies naturally lockless algorithms for the most part.
- Sometimes locks are necessary, but we don't embrace them.
- Cpu-localized algorithms tend to exercise the same code paths on both UP and SMP systems, making them easier to test.

Testing conditions and use of the Kernel Trace Facility

- AMD Athlon X2 3800+ (dual core), 2 Ghz clock
- Shuttle SN95G5V3, 1G ram
- Intention to test algorithmic performance and overheads, not network throughput (can't test network throughput anyway).
- Limitations of KTR – measurement effects the result

0	1.601uS	testlog_test1
0	0.337uS	testlog_test2
0	0.073uS	testlog_test3
0	0.074uS	testlog_test4
0	0.044uS	testlog_test5
0	0.184uS	testlog_test6

0	3.348uS	testlog_test1
0	0.178uS	testlog_test2
0	0.058uS	testlog_test3
0	0.071uS	testlog_test4
0	0.039uS	testlog_test5
0	0.044uS	testlog_test6

0	3.340uS	testlog_test1
0	0.173uS	testlog_test2
0	0.063uS	testlog_test3

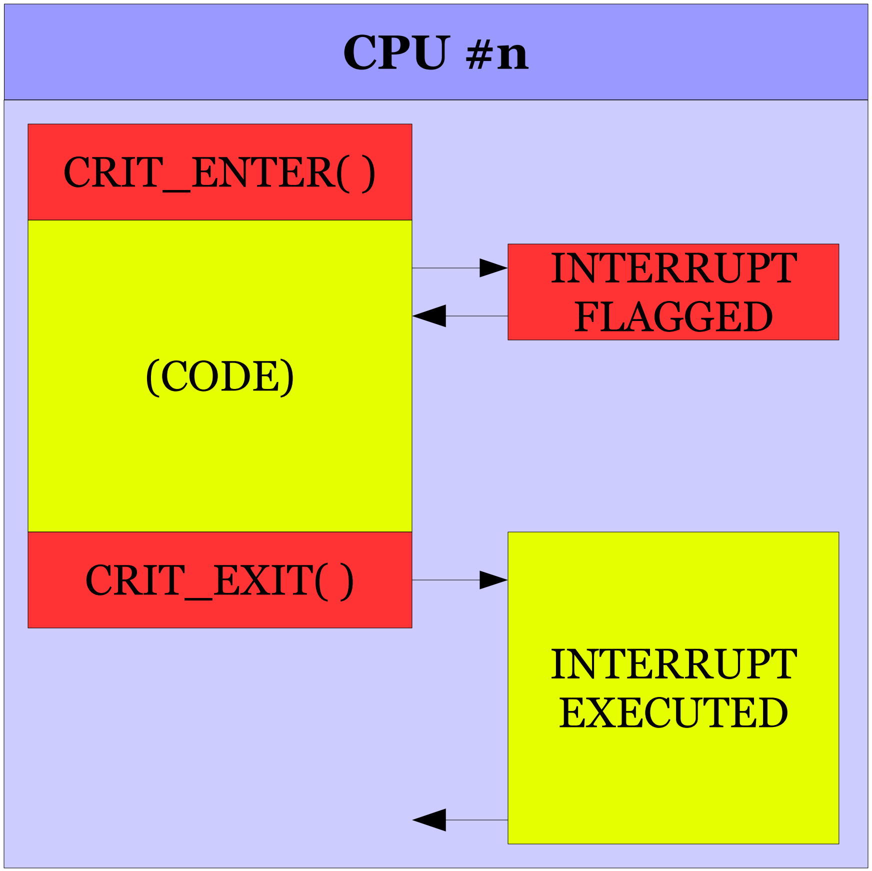
484	0	0.039uS	testlog_test5
485	0	0.040uS	testlog_test6

- Time delta from previous event
- Significant cache effects
- Assume ~40ns for remaining tests
- 1-3 w/args, 4-6 wo/args

Subsystems Characterized

- Critical Sections
- IPIQ Messaging
- Tsleep/Wakeup
- SLAB Allocator's free() path – use of passive IPIQ messages
- Single and multiple packet interrupt overheads
- Single and multiple packet TCP processing overheads
- Interrupt moderation and message aggregation

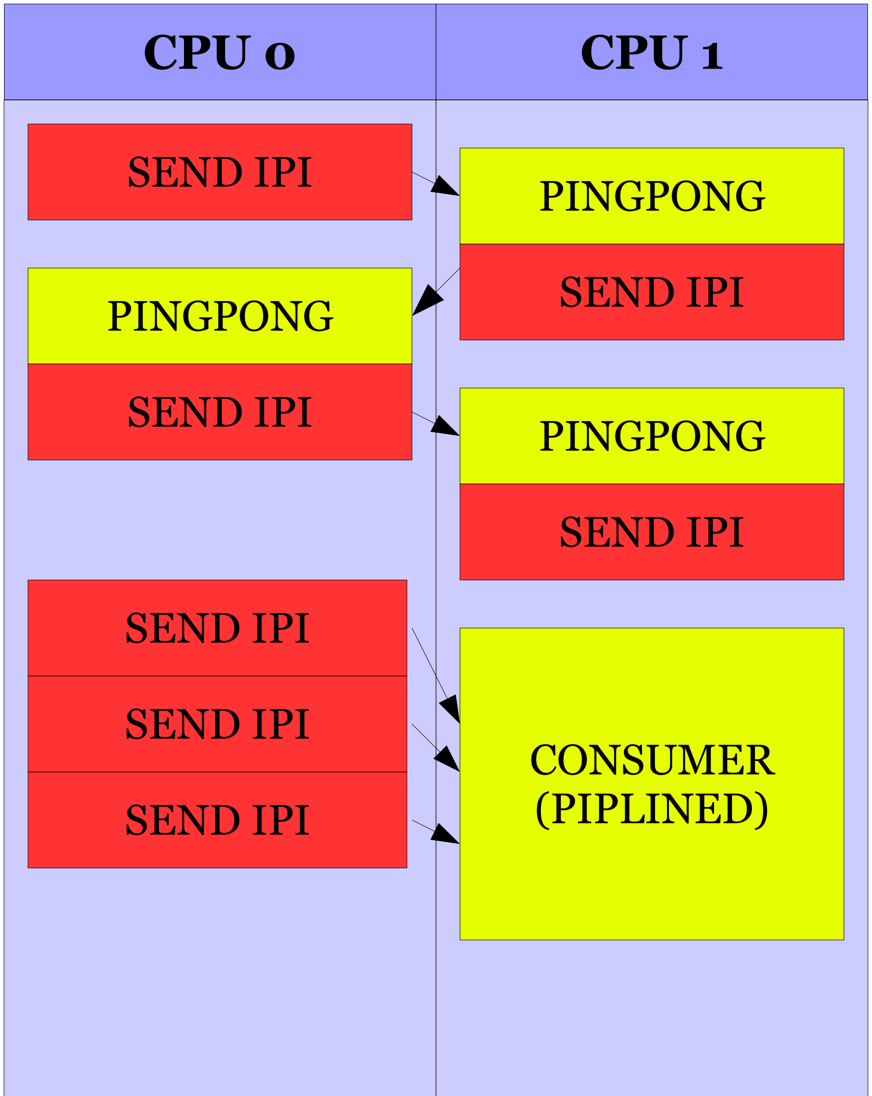
Critical Sections



61	0		testlog_crit_be
62	0	0.902uS	testlog_crit_en
732	0		testlog_crit_be
733	0	0.970uS	testlog_crit_en
758	0		testlog_crit_be
759	0	0.902uS	testlog_crit_en
443	0		testlog_crit_be
444	0	1.008uS	testlog_crit_en

- 100 Iterations per test
- ~9.5 ns enter+exit
- Per-thread nesting count
- Does not CLI/STI
- Cpu-localized

IPIQ Messaging

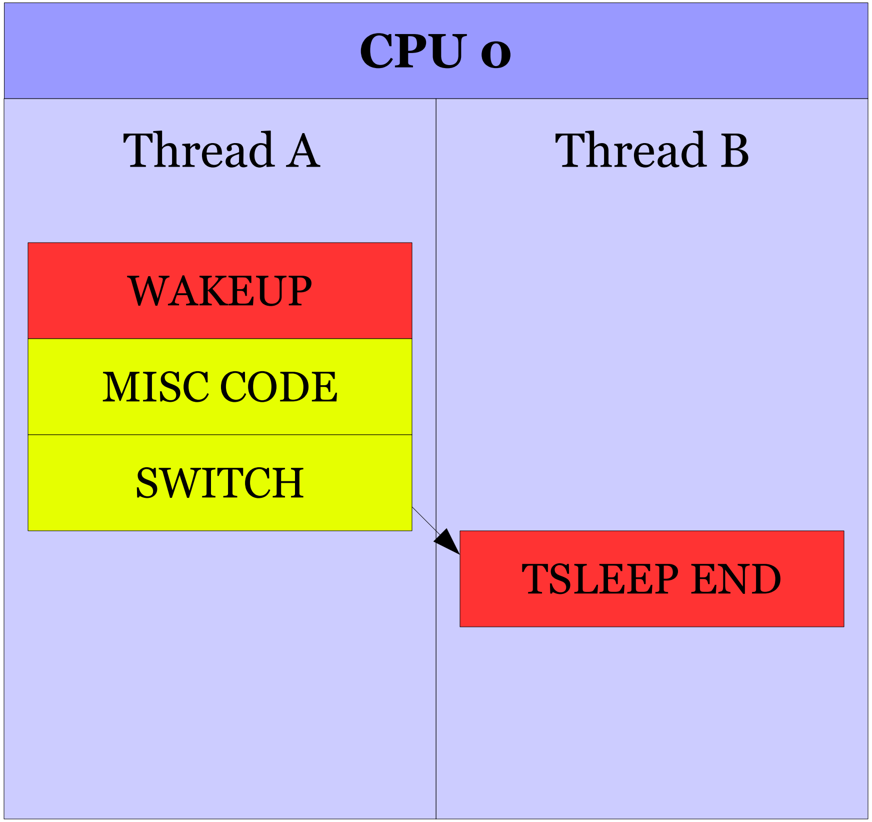


652	1	1.040 uS	PING
794	0	1.171 uS	PONG
653	1	1.008 uS	PING
795	0	1.164 uS	PONG
654	1	0.995 uS	PING
796	0	1.169 uS	PONG
655	1	1.025 uS	PING
797	0	1.174 uS	PONG

813	0	1.159 uS	PIPELINE
814	0	0.141 uS	PIPELINE
815	0	0.044 uS	PIPELINE
816	0	0.079 uS	PIPELINE
817	0	0.015 uS	PIPELINE

- Software FIFO crossbar
- Note TSC drift ~122ns error
- Latency != Performance
- ~1.1 uS latency between cpus
- ~50 nS pipelined

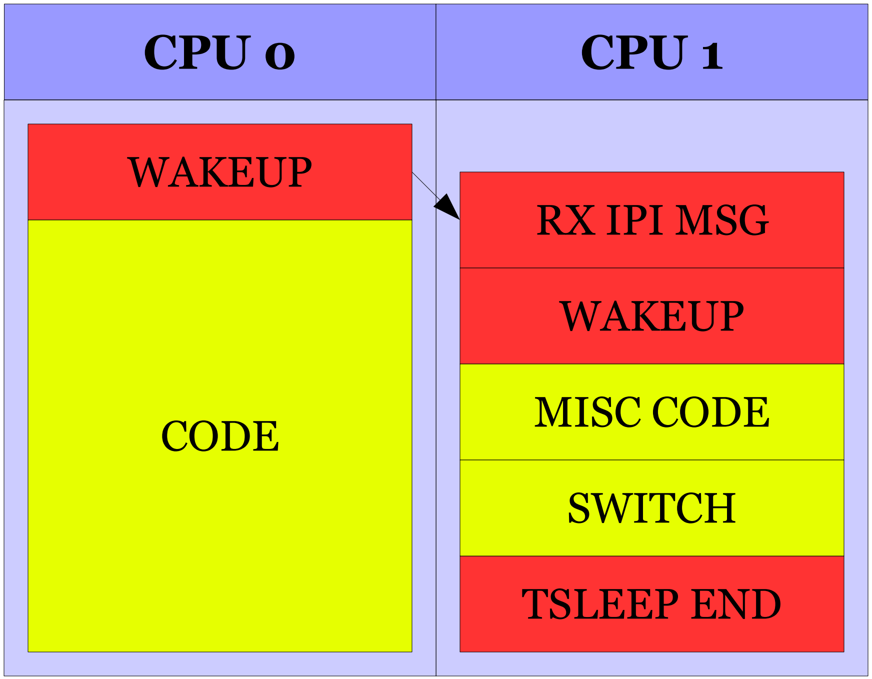
LWKT Scheduler – Same CPU



812	**0**		**WAKEUP_BEG**
813	**0**	**0.291 uS**	**WAKEUP_END**
814	**0**	**1.102 uS**	**TSLEEP_END**
119	**1**		**WAKEUP_BEG**
120	**1**	**0.297 uS**	**WAKEUP_END**
121	**1**	**1.000 uS**	**TSLEEP_END**
760	**0**		**WAKEUP_BEG**
761	**0**	**0.276 uS**	**WAKEUP_END**
762	**0**	**0.999 uS**	**TSLEEP_END**

- ~288 nS wakeup overhead
- ~1 uS switching overhead

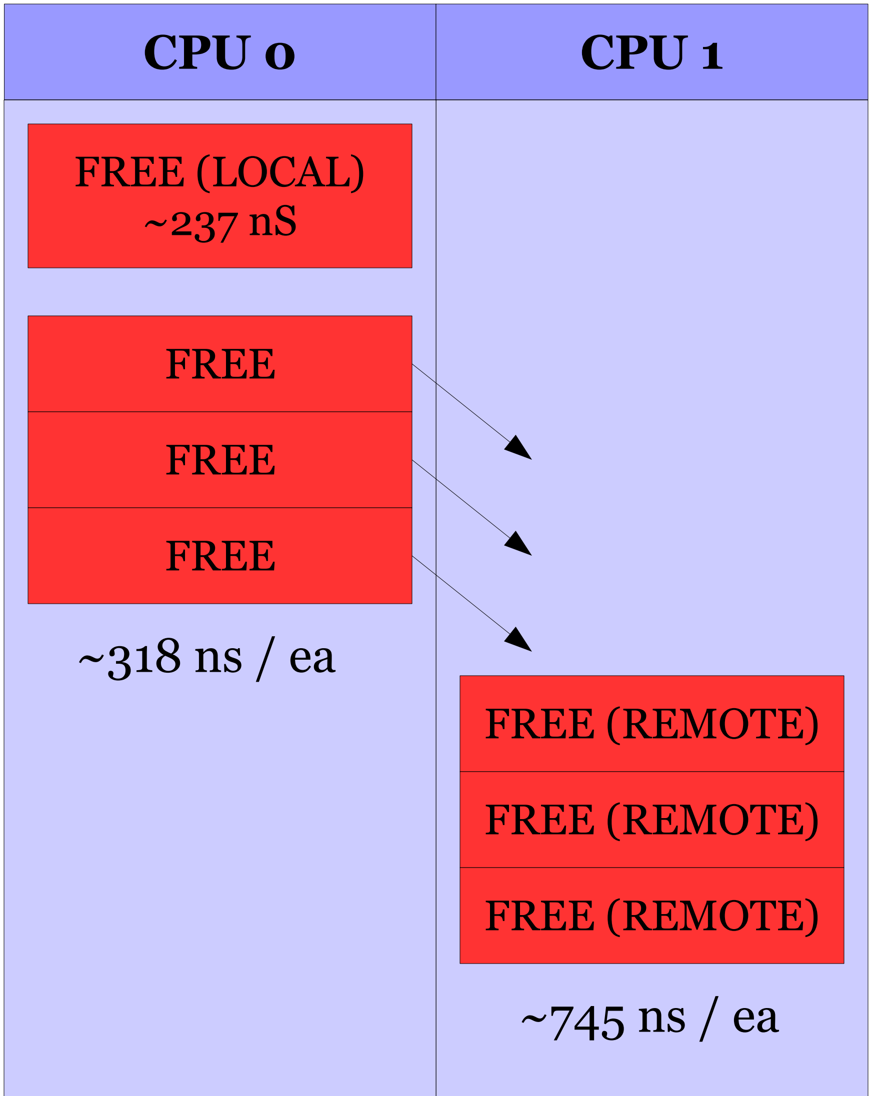
LWKT Scheduler – Different CPU



627	1		WAKEUP_BEG
628	1	0.016 uS	WAKEUP_END
230	0	0.953 uS	TSLEEP_END
570	1		WAKEUP_BEG
571	1	0.016 uS	WAKEUP_END
161	0	1.180 uS	TSLEEP_END
406	0		WAKEUP_BEG
407	0	0.168 uS	WAKEUP_END
768	1	1.295 uS	TSLEEP_END
372	0		WAKEUP_BEG
373	0	0.117 uS	WAKEUP_END
754	1	3.412 uS	TSLEEP_END

- Note TSC drift ~122ns error
- Latency != Performance
- Overhead on originating cpu ~75ns
- Overhead on target ~1.7 uS
- IPI + Switching load on target

SLAB Allocator's free( ) path – Take 1



615	1		FREE_BEG
616	1	0.211 uS	(misc)
617	1	0.097 uS	SEND IPI
618	1	0.119 uS	FREE_END
619	1		FREE_BEG
620	1	0.135 uS	(misc)
621	1	0.034 uS	SEND IPI
622	1	0.040 uS	FREE END
5	0		RECV IPI
6	0	0.036 uS	FREE_REMOT
7	0	0.142 uS	FREE_BEG
8	0	0.156 uS	(misc)
9	0	0.398 uS	FREE_END
10	0		RECV IPI
11	0	0.030 uS	FREE_REMOT
12	0	0.098 uS	FREE_BEG
13	0	0.124 uS	(misc)
14	0	0.507 uS	FREE_END

SLAB Allocator's free( ) path – Take 2

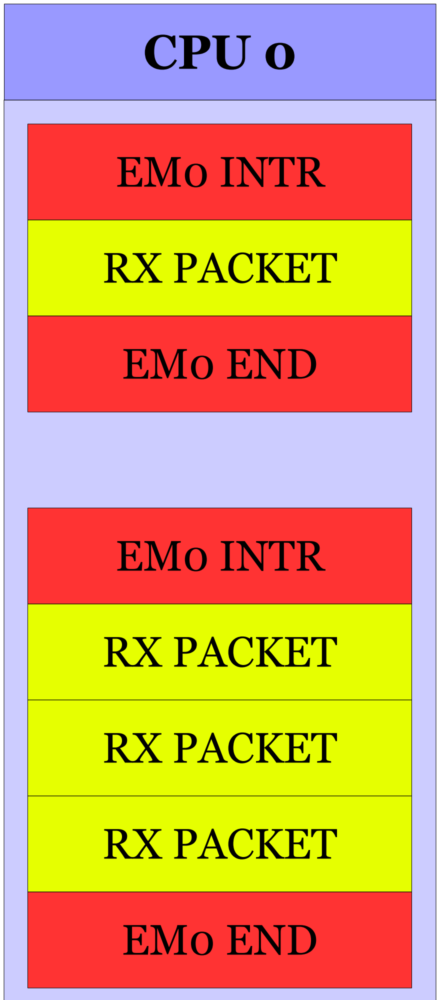
542	0		FREE BEGIN
543	0	0.285 uS	FREE END
938	1		FREE BEGIN
939	1	0.476 uS	FREE END
940	1		FREE BEGIN
941	1	0.214 uS	FREE END
236	0		FREE BEGIN
237	0	0.154 uS	FREE END

89	1		FREE BEGIN
90	1	0.879 uS	PASSIVE IPI
91	1	0.248 uS	FREE END
92	1		FREE BEGIN
93	1	0.178 uS	PASSIVE IPI
94	1	0.062 uS	FREE END
95	1		FREE BEGIN
96	1	0.147 uS	PASSIVE IPI
97	1	0.059 uS	FREE END

367	0		IPIQ RECEIVE
368	0	0.421 uS	FREE REMOTE
369	0	0.148 uS	FREE BEGIN
370	0	1.584 uS	FREE END
371	0		IPIQ RECEIVE
372	0	0.032 uS	FREE REMOTE
373	0	0.084 uS	FREE BEGIN
374	0	0.541 uS	FREE END
375	0		IPIQ RECEIVE
376	0	0.032 uS	FREE REMOTE
377	0	0.075 uS	FREE BEGIN
378	0	0.318 uS	FREE END

- SAME-CPU CASE – 282 nS
- MP CASE LOCAL SIDE – 262 nS
- MP CASE REMOTE SIDE – 359 nS
- Note cache effects @939, 90, 368-370
- Note cache effects on target cpu

Network Interrupt
Single and Muliple frames

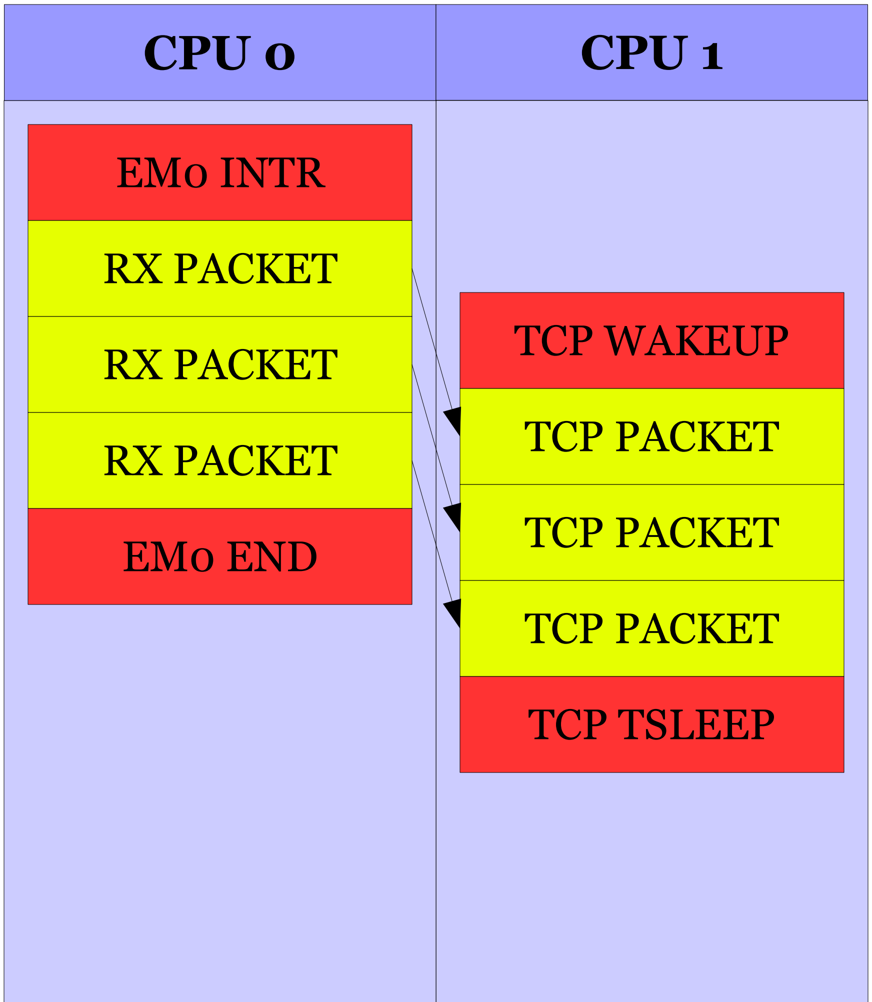


310	0		INTR_BEG
311	0	0.864 uS	RX PACKET
312	0	1.841 uS	INTR_END
502	0		INTR_BEG
503	0	0.856 uS	RX PACKET
504	0	1.662 uS	INTR_END
498	0		INTR_BEG
499	0	0.955 uS	RX PACKET
500	0	1.575 uS	RX PACKET
501	0	1.093 uS	RX PACKET
502	0	1.089 uS	RX PACKET
503	0	1.254 uS	RX PACKET
504	0	1.166 uS	RX PACKET
505	0	1.825 uS	INTR_END

- Single packet overhead is ~2.5 uS.
- Multi-packet overhead is ~1.26 uS per packet.
- Can be improved.
- Additional unmeasured interrupt overheads.
- Does not include protocol stack.

- Similar results from polling
- No-work overhead was ~300 ns when polling.

TCP Protocol Stack – Pipelined case



651	1		TCP_WAIT
862	0		EM0 RX PKT
863	0	2.234 uS	EM0 RX PKT
652	1	0.637 uS	TCP RX PKT
864	0	0.940 uS	EM0 RX PKT
865	0	1.668 uS	EM0 RX PKT
866	0	2.684 uS	EM0 RX PKT
653	1	0.812 uS	TCP RX PKT
867	0	0.666 uS	EM0 RX PKT
654	1	1.474 uS	TCP RX PKT
868	0	0.948 uS	EM0 END INTR
655	1	0.074 uS	TCP RX PKT
656	1	1.114 uS	TCP RX PKT
657	1	0.962 uS	TCP RX PKT
658	1	0.952 uS	TCP DELAYED
659	1	6.081 uS	TCP XMIT ACK
669	1	1.662 uS	TCP_WAIT

- MPSAFE enabled on interrupt
- IPIQ clearly pipelined
- Some unidentified overheads

Conclusions

- Medium-complexity procedures, such as processing a TCP packet, still run in very short (~1uS) periods of time. Due to the short running time, any 'regular' cache contention between cpus will seriously effect performance. Even spinlocks and locked bus-cycle instructions may be too expensive in this context.

- Work aggregation reduces the relevancy of thread switching overheads and improves cache characteristics at the same time. This is a natural result of interrupt moderation, polling, passive IPI messaging, non-preemptive message passing, etc.

- Code paths designed to directly support MP operations, such as the IPIQ messaging subsystem, must use contentionless algorithms to reap their full benefit. e.g. Consider pipelining, passive free().

- Low (~50ns) overhead pipelining of operations between cpus is possible.

Conclusions

- Preemption will decrease cache performance by elongating code paths and reducing work aggregation so only use it when we actually need to use it. There is no point preempting something that might only take 2 uS to run anyway. The concept of preemption is not bad but I would conclude that any preemption which destroys our ability to aggregate work is very bad and to be avoided at all costs. Should the TCP protocol thread run preemptively? Probably not.

- Preservation of cache locality of reference is important, but one must also consider the time frame. Instruction caches appear to get blown out very quickly (< 100 uS), making work aggregation all that much more important.

- Thread switching times are quite consistently in the ~1 uS range and it does not appear to take much work aggregation to make up for the overhead (witness the speed of a fully cached free()).

The DragonFly BSD Project
April 2005

Joe Angerson
David Xu
Matthew Dillon
Craig Dooley
Liam J. Foy
Robert Garrett
Jeffrey Hsu
Douwe Kiela
Sascha Wildner
Emiel Kollof
Kip Macy
Andre Nathan
Erik Nygaard
Max Okumoto
Hiten Pandya
Chris Pressey
David Rhodus
Galen Sampson
YONETANI Tomokazu
Hiroki Sato
Simon Schubert
Joerg Sonnenberger
Justin Sherrill
Scott Ullrich
Jeroen Ruigrok van der Werven
Todd Willey

and Fred -->

DragonFlyBSD

WWW.DRAGONFLYBSD.ORG