

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

Source: <http://unix.derkeiler.com/Mailing-Lists/FreeBSD/current/2007-01/msg00395.html>

- *From:* Bruce Evans <bde@xxxxxxxxxxxx>
 - *Date:* Fri, 19 Jan 2007 16:59:46 +1100 (EST)
-

I'll try to keep this shorter :-).

On Thu, 18 Jan 2007, Matthew Dillon wrote:

```
:Fully cached case:
:% copy0: 16907105855 B/s ( 242265 us) (movsq)
:% ...
:% copyQ: 13658478027 B/s ( 299887 us) (movdqa)
:% ...
:% copyT: 16456408196 B/s ( 248900 us) (movq (i))
:
:This doesn't contradict your claim since main memory is not really involved.
:Everything should be limited by instruction and cache bandwidth, but for
:some reason movdqa is quite slow despite or because of my attempts to
:schedule it perfectly.
```

Well, that's one of the problems. Actually two of the problems. First, prefetching tends to add more confusion than it solves when used in a copy loop, because for all intents and purposes you are already keeping the memory pipeline full simply by the fact that the writes are buffered.

The above is for the non-prefetched case (except for the target of nontemporal writes). Handling and interpreting prefetching is certainly confusing in other cases.

I'd better describe what my benchmark does in a bit more detail: to get reasonably accurate, it is necessary to iterate many times, but that gives the same not-very-real-world cache state for all iterations except the first. I just run it with different block sizes. 4K gives the "fully (L1) cached case", somewhere between 40K and 400K gives the "fully (L2) cached case", and 4M-200M gives the "fully uncached case". I don't test lower levels except by accidentally starting swapping.

There is an option to pre-cache the target before starting the iterations.

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

This can only make a difference if the block size is small enough for the target to remain cached (or there number of iterations is too small), and the copying used nontemporal stores so that the copying itself doesn't pre-cache the target. I didn't use this option in the benchmark results quoted in this thread.

By the time the loop gets back around to the read it has to stall anyway because the writes are still being pushed out to memory (read-before-write ordering doesn't help because all that accomplishes is to force the write buffer to stay perpetually full (possibly in a non-optimal fashion), and the cpu stalls anyway.

I had mostly forgotten about read/write ordering. It is going to limit execution reordering in a MD way. I only have good/handy documentation of it for A64. On A64, write ordering is fairly strict, but almost any order is possible for a combination of reads and writes provided the writes can be held in the write buffer and the reads and writes are to different locations. Read ordering is fairly non-strict. Reads can be reordered ahead of writes.

This doesn't quite agree with what you said. It allows enough reordering to prevent stalls provided the CPU is clever enough and the copy is not overlapped. The CPU can take almost any ordering of sequential reads and writes and turn it into:

```
setup
for (;;) {
read next cache line
write previous cache line from write buffer
(write buffer size must be >= cache line size)
}
```

I don't know which CPUs are clever enough to do this or exactly which static instruction order makes it easiest for them, but have noticed reasonable orders which give mysterious slowdowns that are probably related to this.

I've tried every combination of prefetching and non-temporal instructions imaginable and it has reduced performance more often than it has improved it. I can write an optimal copy loop for particular situations but the problem is that the same loop becomes extremely inefficient under any *OTHER* conditions.

Same here, except I wouldn't call most of the non-optimal cases `_extremely_` inefficient. Do you have real-world tests?

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

The behavior of any given copy algorithm is radically different if either the source or target are present in the L1 or L2 caches, verses if they aren't. The algorithm one uses to do tiny mostly in-cache copies is totally different from the algorithm one uses to do large bulk copies. The algorithm one uses where the source data is partially or fully cached is totally different from the one used if the target data is partially or fully cached, or if neither is cached, or if both are cached.

And it's hard to know whether/where the data is cached. The only simplification is that if an FPU switch is required, then very small copy sizes need not be considered.

```
:Fully uncached case:
:% copy0: 829571300 B/s ( 493749 us) (movsq)
:% ...
:% copyQ: 835822845 B/s ( 490056 us) (movdqa)
: ...
:% copyW: 1350397932 B/s ( 303318 us) (movnti)
: ...
:Now movdqa beats movsq by a bit, at least with prefetch, but has the
:same speed as integer moves of half the size (or MMX moves of half the
:size (benchmark data not shown)). It is necessary to use movnt to get
:anywhere near memory bandwidth, and any form of movnt can be used for
: ...
```

But you can't safely use *ANY* non-temporal instruction unless you know, specifically, that the data is uncached. If you use a non-temporal

I think you mean "optimally", not "safely". sfence is supposed to give coherent data.

instruction in a situation where the data is cached or can be easily cached, you destroy the performance for that case by causing the cpu not to cache data that it really should cache. Use of non-temporal instructions results in a non-self-healing algorithm... it is possible to get into a situation the is permanently non-optimal. That's why I don't use those instructions.

Not caching the target is part of the point of using movnt. It can win not only in copy speed but also by not thrashing useful things out of the cache. Unfortunately we don't really know when this happens (except for idle pagezero). You may be right that it happens so rarely that never doing it is best.

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

Sometimes I wish the non-temporal instructions had a random component. That is, that they randomly operates as normal moves instead of nt moves for a small percentage of executions (~1–25%) in order to create a self-healing (performance wise) algorithm.

Wouldn't normal (non-copy) accesses cache the data reasonably if it is actually used? Multiple copying of the same data doesn't seem to be an important case. I just remembered on case where nontemporal should win: for writes of data that will be DMAed to disks but not read soon by anything except the DMA. Even if disk drivers read it, the disk writes should be delayed by a few seconds on average so the data wouldn't remain in caches except on idle systems.

Use of non-temporal instructions for non performance-critical operations is a clear win when executed from the idle loop, I agree. Use of NT instructions in any other case is difficult because it is virtually impossible to predict the conditions under which other cases occur.

Do you use it for copying pages?

:Which modern CPUs can't keep up with L2?

The instruction pipeline is the problem, not the L2 cache bandwidth. Things get a bit weird when every single instruction is reading or writing memory. I try to design algorithms that work reasonably well across a large swath of cpus. It turns out that algorithms which do block loads followed by block stores tend to maintain consistent and good performance across a large swath of cpus. I outline the reasons why later on.

I think it's not mainly the instruction pipeline throughput (except on very old CPUs), but latency issues caused by interactions with the memory accesses.

:On amd64 for the fully-L2-cached case:

:% ...

:% copyQ: 4369052686 B/s (937503 us) (movdqa)

:% copyR: 2785886655 B/s (1470268 us) (movdqa with prefetchnta)

:% copyS: 4553271385 B/s (899573 us) (movdqa with block prefetch)

:% ...

:

:So here is a case where prefetchnta is bad. OTOH, movnti is quite fast

:provided prefetchnta is not used with it. movnti is even more competitive

:with L2 if main memory is DDR-2.

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

It also depends *WHERE* in the loop you put the prefetchnta. I was able to get fairly close to block prefetching speeds using prefetchnta by carefully locating the instruction and adjusting the read-ahead point. For example, by locating the instruction before the write side and indexing the prefetched memory address 128 bytes ahead of the curve. The placement of the instruction is sensitive down to the instruction slot... moving it one forward or one back can result in crazy differences in performance (due to forced RAS cycles I believe, in particular with prefetchnta).

I've only tried that for block prefetch. It was too MD to maintain. I hoped prefetchnta was easier to use. I tried several things, and just noticed that I missed a point in the AMD docs -- block prefetch should be in reverse order to confuse the hardware into not competing with the explicit prefetch.

:Load/store-unit:
:
: (lots of good information about load-store units)

Well, but these are for very specific flavor-of-the-day hardware tweaks. The designers change these parameters literally every other month.

All that can really be said here is: Avoid 32 bit accesses if you can, 64 bit accesses seem to be the most dependable, and 128 bit accesses are the up-and-coming thing.

I think the using the FPU and 128-bit accesses are also flavor-of-the-day (yesterday for the FPU). Let the cache and write buffer handle combining of accesses.

In particular I remember reading that AMD had been spending a lot of time optimizing XMM register loads and stores. I presume that Intel is doing the same thing since both are going face forward into gaming.

Bah, I wish the would spend more time optimizing important things like FPU latency and wider integer registers :-).

:The asymmetric load/store capabilities on the AXP can't be good for using
:128-bit XMM registers, especially using 8 loads followed by 8 stores in
:the instruction stream. They make 128 stores want to take twice as long
...

The advantages of doing N loads followed by N stores (where $N \geq 4$)

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

and operates on 64 or 128 bit entities) are as follows:

- * There is no instruction or register reordering (or very little) because the data loaded by any given instruction is not used until e.g. 8 instructions later.

But I think I want to maximize reordering possibilities.

- * The write buffer is ordered (either through to the L2 cache or to main memory, it doesn't matter). This enforces a degree of resynchronization of the instruction stream. That is, it creates a self-healing situation that makes it possible to write code that works very well under all sorts of conditions.

Ordered relative to reads? BTW, do you order the instruction stream so that the reader is 1 cache line ahead of the writer? I only do this for old methods optimized for P1's. It saves about 1 read access time per loop unless this time can be hidden by reordering or loop overhead. The cache line size is MD so it is better for this to happen automatically, but perhaps less confusing to program it explicitly for a preferred CPU.

- * The read instructions should theoretically tend not to be reordered simply because the memory pipeline imposes a stall due to handling previously buffered writes and because of the burst nature of linear reads from main memory. Even if the cpu DID reorder the reads,

...

- * I don't think the cpu could reorder reads or reorder reads and writes under these conditions even if it wanted to, and we don't want it to.

- * Write combining is overrated. It serves an important purpose, but it is a bad idea to actually depend on it unless your algorithm is self-healing from the point of view of resynchronizing the pipeline and/or memory read and write ordering. Instruction

Er, don't you depend on it even with 128-bit registers? 16 bytes isn't very large. Even an AthlonXP has a 64-byte write buffer to combine 4 of these.

...

It should be possible to use MMX/XMM registers in the kernel simply as call-saved registers by issuing a FWAIT or equivalent at the user-kernel boundary to take care of any exceptions (instead of saving the state

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

and calling `clts ; fnclex`). Presumably by the time the cpu actually gets the integer and cpu state context pushed any FP operations will have long since completed. But I haven't timed it to find out whether that actually works.

I thought about making some XMM registers call-used (switched on every boundary crossing). Copying only needs 1 and switching just 1 doesn't have much overhead. Call-saved has different tradeoffs. Both methods should drop the semi-lazy FPU context switching and do a full switch at context switch time so that DNA traps never occur and the FPU/XMM can just be used. This also hopefully doesn't have much additional overhead, since the FPU instructions run in an otherwise-idle pipeline and the extra memory traffic is either small enough or can be scheduled better. MMX would have complications for the tag word. I think new code shouldn't support MMX. The `fnclex` is unnecessary (see below).

I don't quite understand what your `fwait` does or how call-saved XMM context switching could work well for preemptible kernels. Doesn't it give all the old problems with the state saved where the general context switcher can't see it? Call-used XMM context switching works using normal stack discipline.

The nice thing about most MMX/XMM media instructions is that they don't actually care about what state the FP unit is in because they aren't actually doing any FP math. It is just a matter of synchronizing the free FP registers from some prior user FP operation which may not have posted results yet.

XMM doesn't even need the synchronization. It runs independently of the i387 state, and if it is not used for FP math then it also runs independently of `mxcsr`. Thus `fnclex` and `fwait` have no effect on it, but if it is used for FP math then something would have to be done to ignore (user) exceptions and mode bits in `mxcsr`.

If this IS possible then it removes most of the `NPXTHREAD` junk from the kernel bcopy path for the case where `NPXTHREAD != NULL`, giving you the ability to use the FP optimally whether `NPXTHREAD == NULL` or `NPXTHREAD != NULL`. And, in such a case, it might be more beneficial for small copies to only save two or four XMM registers instead of all eight.

What do you think?

One XMM register is enough for even large copies :-).

Sorry, didn't succeed in keeping this shorter.

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

Re: [PATCH] Maintaining turnstile aligned to 128 bytes in i386 CPUs

Bruce

freebsd-current@xxxxxxxxxxx mailing list

<http://lists.freebsd.org/mailman/listinfo/freebsd-current>

To unsubscribe, send any mail to "freebsd-current-unsubscribe@xxxxxxxxxxx"