What your mother didn't teach you about fast memcpy

Dick Sites
April 2022

Talk outline

Goal: Move 16 bytes per cycle
Copy-in-RAM fail, DMA engine fail
What the caches have to do
Load/Store partial
Prefetching
Memory organization for fast row access
Cache pollution
Summary

Move 16 bytes per cycle

Gedankenexperiment: Move 16 bytes per cycle

Factoid: at Google, about 25-35% of all active CPU time is spent moving bytes.

Here is a thought experiment,
to reveal some insights about microarchitecture

The goal: Move 16 bytes per cycle on a load/store machine with 64-bit addresses and some 16-byte data registers (e.g. ARM, x86).

What are the **consequences** of this goal in a chip design?

Move 16 bytes per cycle

```
memcpy(dest, src, len);
```

Some moves are short, ~10 bytes: words, tags, headers, compress/decompress

Some moves are medium, ~100 bytes: paragraphs, control blocks, structs

Some moves are long, ~1000+ bytes: web pages, images, video frames, packets

We want **all three** categories to be close to 16 bytes per cycle.
Memcpy does not know ahead of time which **`len`** is involved.

Copy-in-RAM fail
DMA engine fail

Fails

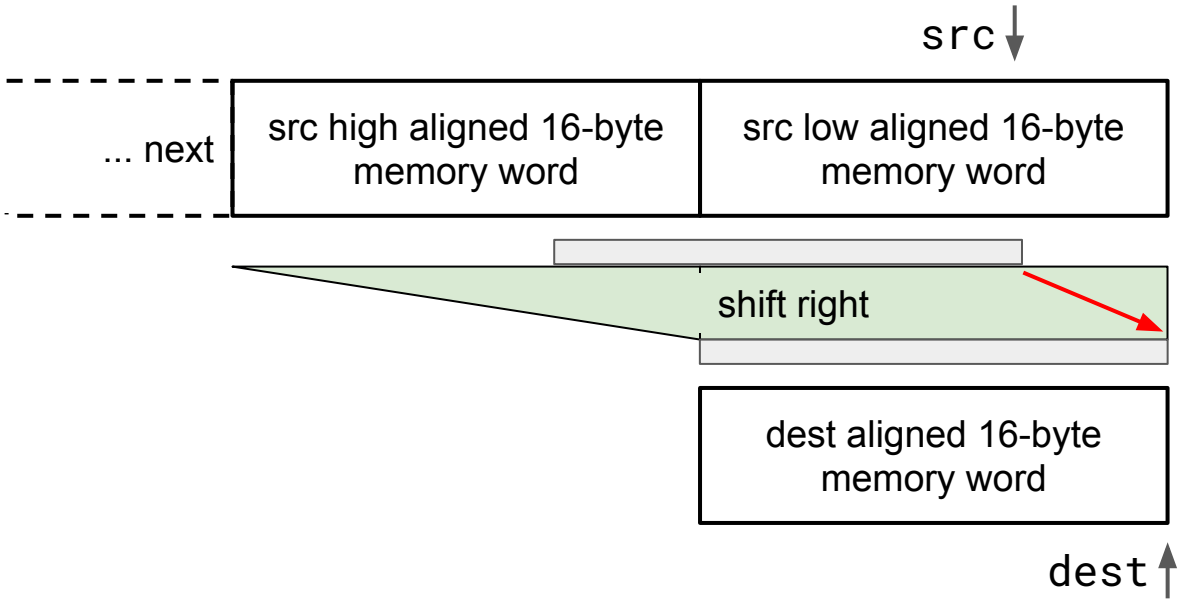
Who does the shift?

After copying the <html> tag to a destination buffer, the next dest starts at 6 (mod 16) but the next src may well be at 0 (mod 16).

Copy-in-RAM does not do the alignment shift

A separate "mover" DMA engine could, but for an N-core chip with just *one* DMA engine, software must take out a lock to use it, too slow for a 6-byte move.
If there are multiple users, the others wait, making them uncontrollably too slow.

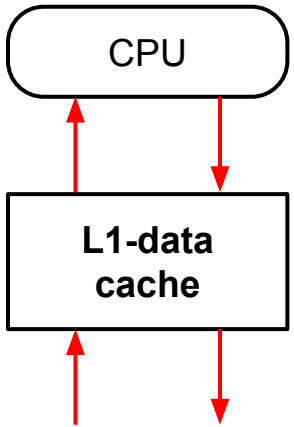
Generic alignment shift (little-endian picture)



What the caches have to do

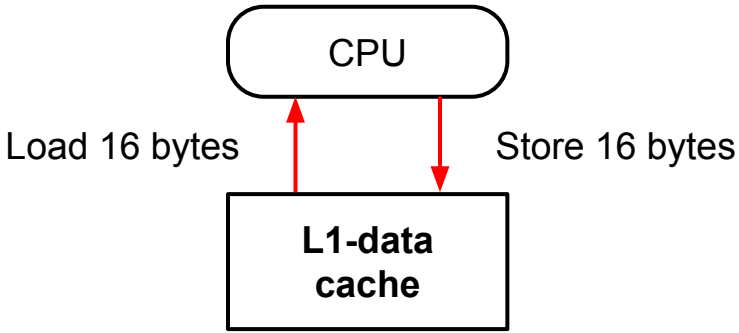
L1 data cache

What does the L1 data cache have to do every cycle?



L1 data cache

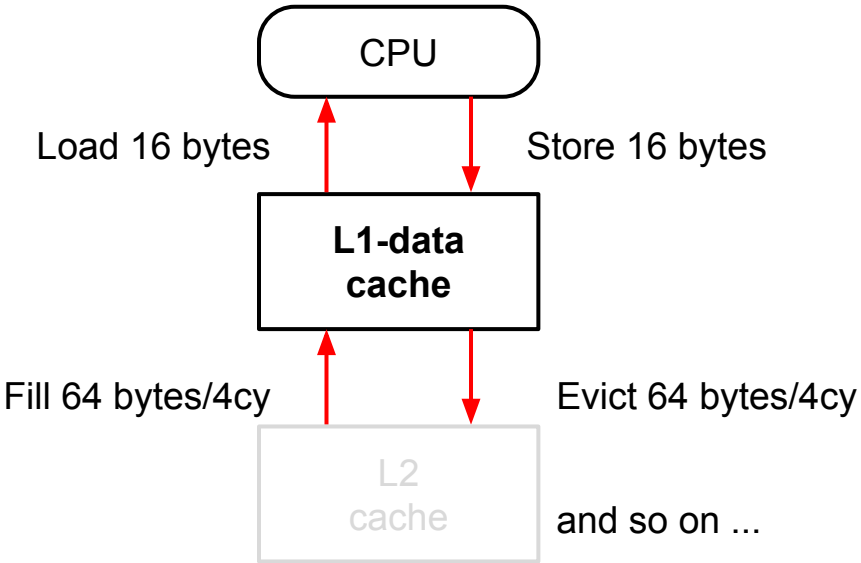
What does the L1 data cache have to do **every cycle**?



What else?

L1 data cache

What does the L1 data cache have to do **every cycle**?

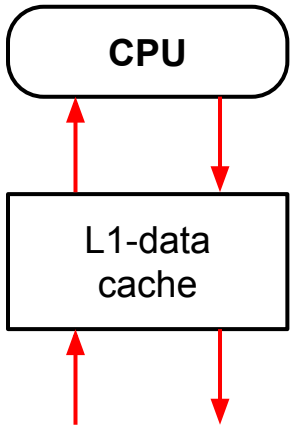


Your microarchitecture needs to support all this data movement.

CPU instruction stream

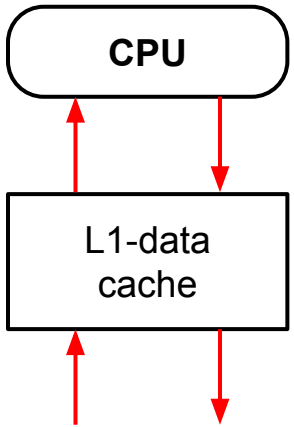
CPU instruction stream

What does the CPU have to do every cycle?



CPU instruction stream

What does the CPU have to do every cycle?

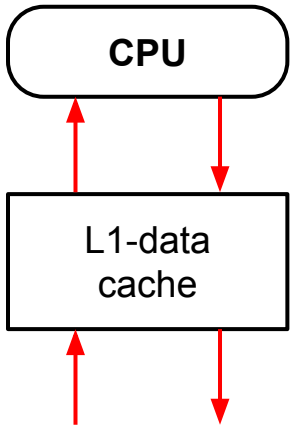


load 16 bytes
store 16 bytes
increment src += 16
increment dest += 16
decrement len -= 16
test
loop: branch if not done

Seven instructions per cycle?

CPU instruction stream

What does the CPU have to do every cycle, with loop unrolling and better test?



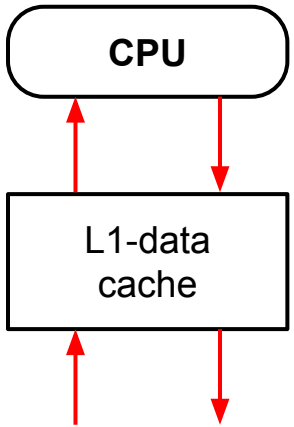
load 16 bytes
store 16 bytes
increment src += 32
increment dest += 32

load 16 bytes
store 16 bytes
test (against ending src addr)
loop: branch if not done

Your microarchitecture needs to issue 4 instructions per cycle, including **two** cache accesses

CPU instruction stream

Oops, sorry. Also have to do the shift...

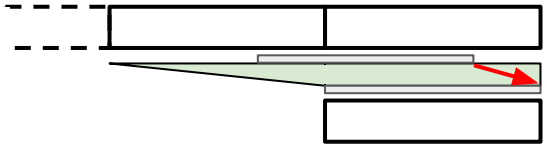


load next 16 bytes
byte shift 32 bytes to align
store 16 bytes
increment src += 64

load 16 bytes
store 16 bytes
byte shift 32 bytes to align
increment dest += 64

load 16 bytes
store 16 bytes
byte shift 32 bytes to align
test

load 16 bytes
store 16 bytes
byte shift 32 bytes to align
loop: branch if not done



Your microarchitecture needs to have a byte shifter that takes two 16-byte inputs

New instruction, 32-byte shift

Byte Shift Double

```
    SHRBD  R1, R2, R3
```

Shift the register pair R1,R2 right by 0..15 bytes, specified by R3<3:0>.
Place the result in R1.

Better design:
Shift the register pair **R2,R2^1** right by 0..15 bytes, specified by R3<3:0>.
Place the result in R1.

Short moves

Short moves, ~10 bytes

Standard code, byte at a time (software or microcode):

```
while (0 < len--) {*dest++ = *src++;}

loop:
  compare len to 0
  decrement len -= 1
  branch if equal
  load byte 0(src)
  increment src += 1
  store byte 0(dest)
  increment dest += 1
  branch to loop
```

Too slow, at least two cycles per byte, 20 cycles for 10 bytes

New instructions

Load Partial

```
LDP  R1, R2, R3
```

Load and zero extend 0..15 bytes from 0(R2) into R1, length specified by R3<3:0>. Length 0 does no load and takes no traps.

Store Partial

```
STP  R1, R2, R3
```

Store 0..15 bytes from R1 to 0(R2), length specified by R3<3:0>.
Length 0 does no store and takes no traps.

Chips with normal unaligned loads have all the hardware to do this.

Short moves

With load/store partial, short memcpy becomes simply:

```
memcpy:
    LDP  R1, src, len      ; move 0..15 bytes
    STP  R1, dst, len      ; move 0..15 bytes
    ANDNOT R1, len, 15     ; mask out low four length bits
    BNZ  medium            ; if not zero, more to move
    RET                    ; if zero, we are DONE
medium:
    ...
```

Just **one cycle** plus call/ret. Could inline and only do a call/ret if medium or larger

Medium moves

Medium moves

After LDP/STP for 0..15 bytes, the remaining length is a multiple of 16.

```
medium:
     AND  Rx, len, 15    ; length (mod 16)
     ADD  src, Rx        ; increment past the 0..15 bytes
     ADD  dest, Rx       ; increment past the 0..15 bytes

     AND  Rx, len, 0x10  ; does length have a 16 bit?
     BZ   1f
     LD                  ; yes, move 16 bytes
     ST                  ; yes, move 16 bytes
     ADD  src, src, 16   ; increment past the 16 bytes
     ADD  dest, dest, 16 ; increment past the 16 bytes

1:   <same pattern for bit 0x20 and move 32 bytes: LD/ST LD/ST ADD ADD>
2:   <same pattern for bit 0x40 and move 64 bytes: LD/ST LD/ST LD/ST LD/ST ... >
4:   <same pattern for bit 0x80 and move 128 bytes: 8x LD/ST ... >
```

Very close to 16 bytes per cycle, plus extra for hardware unaligned accesses.

Long moves

Long moves

If more than 255 bytes, we end up here with a multiple of 256 in len<63:8>.

Time to take a few cycles to switch strategy and **align** both src and dest on 16-byte boundaries. (If desired, test and branch to special-case perfectly aligned code.)

Use LDP/STP to advance `dest` to the next higher multiple of 16 . Also back up `src` to the next lower multiple of 16, and calculate as `shift` the right-shift needed to align `src` to `dest`.

Load R2 with the aligned 16 bytes containing the low `src` bytes that will go to `dest`. Increment `src`. Adjust remaining `len`. The high `src` goes to R3 next.

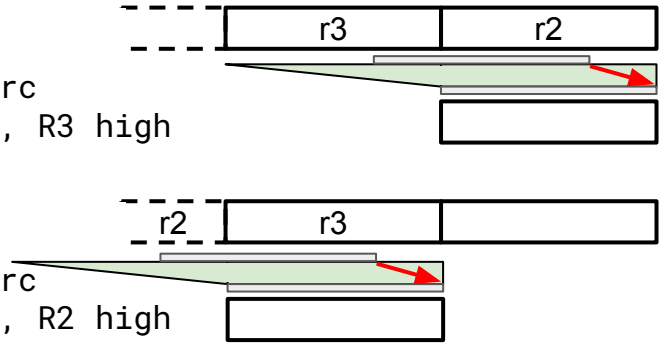
Long moves

This loop moves 64 bytes at a time. We saw it earlier. Note alternating R2,R3 use to avoid register-register moves.

```
longloop:
     LD     R3, 0(src)        ; next aligned 16 bytes of src
     SHRBD  R1, R2, shift     ; The R2,R2^1 design, R2 low, R3 high
     ST     R1, 0(dest)
     xx

     LD     R2, 16(src)       ; next aligned 16 bytes of src
     SHRBD  R1, R3, shift     ; The R3,R3^1 design, R3 low, R2 high
     ST     R1, 16(dest)
     xx

     <again, incrementing src and dest by 64, and then loop>
```



16 bytes per cycle exactly !

Long moves, tail end

After moving multiples of 64 bytes to aligned dest addresses, there will be a tail of 0..63 bytes remaining to move. Use the medium then short patterns to finish up.

The overall design comes very close to moving 16 bytes per cycle over the entire range of short, medium, and long memcpy.

But we aren't done yet ...

Prefetching

Prefetching

With L3 cache about 40 cycles away from the CPU and main memory 200-500 cycles away, prefetching becomes important for long moves. We can't do much about getting the initial bytes quickly, but we can prefetch subsequent bytes.

Moving 1KB at 16 bytes/cycle takes 64 cycles. This is enough time while moving 1KB to prefetch the next 1KB from L3 cache. Similarly, moving 4KB takes about 256 cycles, enough time to prefetch the next 4KB from a 200cy main memory.

But today's computers only prefetch single cache lines, often 64 bytes each.

New instructions

Prefetch_r, Prefetch_w

```
    PRE_R  --, R2, R3 (possibly also PRE_I I-cache prefetching)
    PRE_W  --, R2, R3
```

Prefetch data for reading from 0(R2), length min(R3, 4096).
Prefetch data for writing from 0(R2), length min(R3, 4096).

The 4KB upper bound on the length is important. It prevents describing a prefetch of megabytes, and it guarantees that the prefetch will need no more than two TLB lookups for 4KB or bigger pages. It is big enough to give code time to start subsequent prefetches as needed, pacing prefetching to data use.

New instructions

Prefetch_r, Prefetch_w

```
    PRE_R  --, R2, R3
    PRE_W  --, R2, R3
```

The desired implementation of `PRE_W` does allocation of exclusive cache lines but **defers** filling them with any data, except any partial first and last cache lines. If the entire `PRE_W` range is then written, avoiding the unnecessary fills cuts the needed memory bandwidth in half for pure writes and by 1/3 for memcpy.

The long move loop can be built as a pair of nested loops, the inner one issuing read and write prefetches and then moving 4KB at a time.

DRAM row access

DRAM row access

DRAMs internally copy bits out of very weak transistors to a row buffer and then serve bytes to a CPU chip from there. Row buffers are typically 1KB. Accessing bytes from an already-filled row buffer is three times faster than starting from scratch, approximately 15ns vs. 45ns. This hasn't changed much in 40 years.

If an implementation gets a **prefetch address and length** to the memory controller logic at the beginning of a long prefetch, the memory controller has enough information to optimize doing row-buffer fetches, even in the presence of competing memory requests.

Cache pollution

Cache pollution

Quickly moving many kilobytes of data through the caches normally has the downside of evicting other data belonging to other programs. Cache isolation remains an unsolved problem in the datacenter part of our industry.

Preventing cache pollution could be done by assigning a few bits of "ownership" to each cache line fetched and using that to keep track of how many lines each CPU/etc. owns in the cache. For an L1 cache in a hyperthreaded chip, each logical CPU is an owner. In a cache shared across many cores, each physical core might be an owner. To prevent kernel code from polluting user data while doing disk and network bulk moves, "kernel" could by an owner unto itself.

Cache pollution

Giving owners a **limit** on how many cache lines they can use allows an implementation to switch allocation strategies for owners that are over their limit, preferentially replacing their *own* lines or placing their new fills near the replacement end of a pseudo-LRU replacement list. A loose limit of 60% of all cache lines might be good enough.

This approach is superior to fixed *way partitioning* of an N-way associative cache because it does not leave as many resources stranded.

An alternate approach requiring no extra ownership bits is to essentially rate-limit each owner's fills. Those over their limit get the alternate allocation strategy.

Summary

Summary

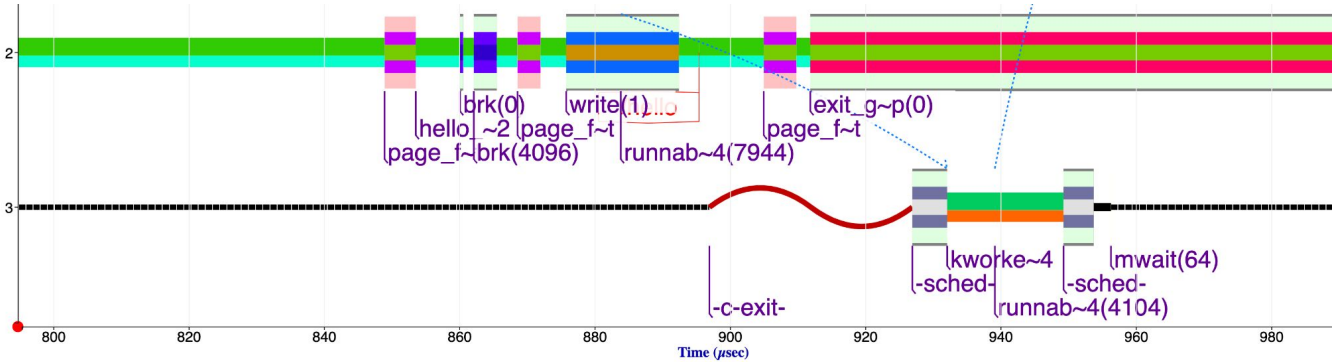
Our simple "Move 16 bytes per cycle" quest for `memcpy` and its ilk reveals a nuanced set of instruction-set and microarchitecture issues:

- Fail: Copy-in-RAM and DMA engines
- Pay attention to both the CPU side and the memory side of caches
- Double-width shift and Load/Store Partial significantly improve memcpy
- Prefetching can make a big difference
- Letting the memory controller know about prefetch length can be 3x faster
- Invent a way to control cache pollution

There are surely some thesis topics buried here...

References

Shameless plug: **KUtrace** every core, every nanosecond, 1% overhead



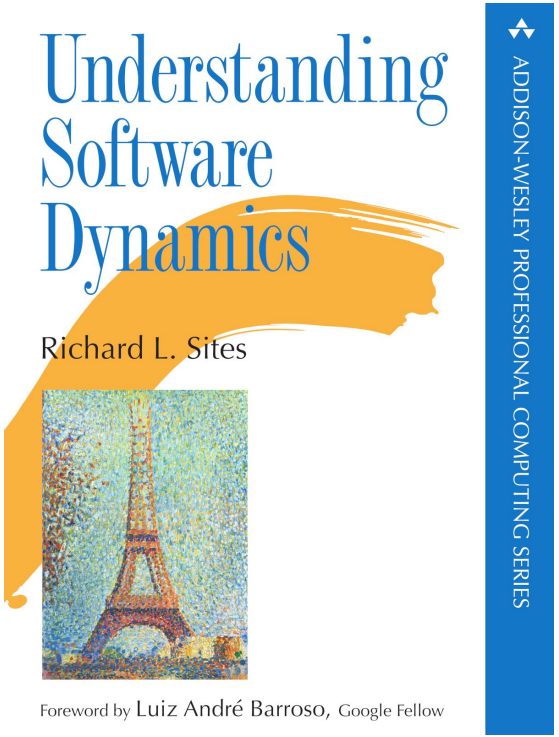
180 microsecond partial execution trace of hello_world
CPU 2: hello_world
CPU 3: kernel_worker (to ssh)
Half-high color: user mode, full: kernel mode
Black line: idle, red curve: exit powersave
Arc: wakeup from syswrite

References

Richard L. Sites, "Understanding Software Dynamics."
Addison-Wesley, December 2021.

Matching code and HTML at
https://www.informit.com/store/understanding-software-dynamics-9780137589739
(Downloads tab partway down)

p.s. I would like some help finding long address traces



"What your mother didn't teach you about fast memcpy"

I'll go through ideas on helpful instructions and implementation details to make memcpy and other memory movement really fast on a machine with a few 16-byte registers (e.g. x86 or ARM). For short lengths ~10 bytes, and medium ~100 bytes, and long ~1000+ bytes.

- copy-in-RAM design fail, DMA engine design fail
- Load/store partial instructions to do 0..15 bytes without branching and without byte at a time loop. For short, and for leaving remaining length a multiple of 16 for medium and long
- Hardware prefetch prediction saturating 2-bit counters per load/store PC (similar to branch prediction per cond. branch PC) -- on miss fetch 1/2/4/8 cache lines; counter increment/decrement details
- Prefetch for read/write up to 4KB for long
- Cache pollution considerations for long
- Memory organization to take advantage of multiple row accesses in DRAM (CAS only is 3x faster than precharge, RAS, CAS) for long and for hardware prefetch of 4 and 8 cache lines.

A student with access to good memory-address traces could probably make a master's thesis out of one of these ideas (I don't know which one).