Review of Memory Hierarchy (Week#15a)

Six Basic Cache Optimizations

To show the benefit of associativity, conflict misses are divided into misses caused by each decrease in associativity. Here are the four divisions of conflict misses and how they are calculated:

- ✓ **Eight-way:** Conflict misses due to going from fully associative (no conflicts) to eight-way associative
- ✓ **Four-way:** Conflict misses due to going from eight-way associative to fourway associative
- ✓ **Two-way:** Conflict misses due to going from four-way associative to two-way associative
- ✓ **One-way:** Conflict misses due to going from two-way associative to one-way associative (direct mapped)

Six Basic Cache Optimizations

First Optimization: Larger Block Size to Reduce Miss Rate

- ✓ To reduce miss rate is to increase the block size. Larger block sizes will reduce misses.
- ✓ This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.
- ✓ Larger blocks increase the miss penalty because they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small.
- ✓ Figure B.10 shows the trade-off of block size versus miss rate for a set of programs and cache sizes.

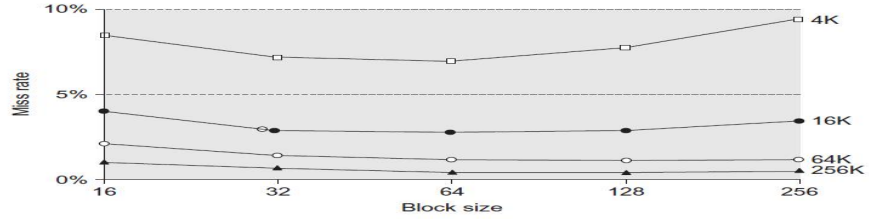

Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 (Gee et al. 1993).

Six Basic Cache Optimizations

Example Figure B.11 shows the actual miss rates plotted in Figure B.10. Assume the memory system takes 80 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 82 clock cycles, 32 bytes in 84 clock cycles, and so on. Which block size has the smallest average memory access time for each cache size in Figure B.11?

Answer Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

If we assume the hit time is 1 clock cycle independent of block size, then the access time for a 16-byte block in a 4 KiB cache is

$$\text{Average memory access time} = 1 + (8.57\% \times 82) = 8.027 \text{ clock cycles}$$

and for a 256-byte block in a 256 KiB cache the average memory access time is

$$\text{Average memory access time} = 1 + (0.49\% \times 112) = 1.549 \text{ clock cycles}$$

Figure B12

	Cache size			
Block size	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Figure B.11 Actual miss rate versus block size for the five different-sized caches in Figure B.10. Note that for a 4 KiB cache, 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KiB in order for a 256-byte block to decrease misses.

Six Basic Cache Optimizations

Figure B.12 shows the average memory access time for all block and cache sizes between those two extremes.

		Cache size			
Block size	Miss penalty	4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

Figure B.12 Average memory access time versus block size for five different-sized caches in Figure B.10. Block sizes of 32 and 64 bytes dominate. The smallest average time per cache size is boldfaced.

Six Basic Cache Optimizations

Second Optimization: Larger Caches to Reduce Miss Rate

To reduce capacity misses in Figures B.8 and B.9 is to increase capacity of the cache. Drawback is potentially longer hit time and higher cost and power. This technique has been especially popular in off-chip caches.

Third Optimization: Higher Associativity to Reduce Miss Rate

- ✓ Figures B.8 and B.9 show miss rates improve with higher associativity.
- ✓ Eight-way set associative is effective in reducing misses for these sized caches as fully associative.
- ✓ **2:1 cache rule of thumb:** a direct mapped cache of size N has about the same miss rate as a two-way set associative cache of size N/2.
- ✓ Increasing block size reduces miss rate while increasing miss penalty, and greater associativity can increase hit time.
- ✓ Fast processor clock cycle encourages simple cache designs, but the increasing miss penalty rewards associativity,

Six Basic Cache Optimizations

Example Assume that higher associativity would increase the clock cycle time as listed as follows:

$$\text{Clock cycle time}_{2\text{-way}} = 1.36 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{4\text{-way}} = 1.44 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{8\text{-way}} = 1.52 \times \text{Clock cycle time}_{1\text{-way}}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to a level 2 cache (see next subsection) that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure B.8 for miss rates, for which cache sizes are each of these three statements true?

$$\text{Average memory access time}_{8\text{-way}} < \text{Average memory access time}_{4\text{-way}}$$
$$\text{Average memory access time}_{4\text{-way}} < \text{Average memory access time}_{2\text{-way}}$$
$$\text{Average memory access time}_{2\text{-way}} < \text{Average memory access time}_{1\text{-way}}$$

Answer Average memory access time for each associativity is

$$\begin{aligned}\text{Average memory access time}_{8\text{-way}} &= \text{Hit time}_{8\text{-way}} + \text{Miss rate}_{8\text{-way}} \times \text{Miss penalty}_{8\text{-way}}\\ &= 1.52 + \text{Miss rate}_{8\text{-way}} \times 25\\ \text{Average memory access time}_{4\text{-way}} &= 1.44 + \text{Miss rate}_{4\text{-way}} \times 25\\ \text{Average memory access time}_{2\text{-way}} &= 1.36 + \text{Miss rate}_{2\text{-way}} \times 25\\ \text{Average memory access time}_{1\text{-way}} &= 1.00 + \text{Miss rate}_{1\text{-way}} \times 25\end{aligned}$$

The miss penalty is the same time in each case, so we leave it as 25 clock cycles. For example, the average memory access time for a 4 KiB direct-mapped cache is

$$\text{Average memory access time}_{1\text{-way}} = 1.00 + (0.098 \times 25) = 3.44$$

and the time for a 512 KiB, eight-way set associative cache is

$$\text{Average memory access time}_{8\text{-way}} = 1.52 + (0.006 \times 25) = 1.66$$

Figure B8

Six Basic Cache Optimizations

Figure B.13 shows the average memory access time for each cache and associativity.

	Associativity			
Cache size (KiB)	1-way	2-way	4-way	8-way
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. *Boldface* type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

Six Basic Cache Optimizations

Fourth Optimization: Multilevel Caches to Reduce

- ✓ Technology trends have improved the speed of processors faster than DRAMs, making the relative cost of miss penalties increase over time.
- ✓ First-level cache can be small enough to match the clock cycle time of the fast processor.
- ✓ second-level cache can be large enough to capture many accesses that would go to main memory, decreasing miss penalty.
- ✓ First-level and a second-level cache, the original formula is :

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

and

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

so

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

Six Basic Cache Optimizations

Local miss rate: number of misses in a cache divided by the total number of memory accesses to this cache. For first-level cache it is equal to Miss rate L1, and for the second-level cache Miss rate L2.

Global miss rate: number of misses in the cache divided by the total number of memory accesses generated by the processor.Global miss rate for the first-level cache is still just Miss rateL1, but for the second-level cache it is Miss rateL1 x Miss rateL2.

- ✓ Local miss rate is large for second-level caches
- ✓ global miss rate is the more useful measure: It indicates what fraction of the memory accesses that leave the processor go all the way to memory.

$$\text{Average memory stalls per instruction} = \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \times \text{Miss penalty}_{L2}$$

Six Basic Cache Optimizations

Example Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.

Answer The miss rate (either local or global) for the first-level cache is 40/1000 or 4%. The local miss rate for the second-level cache is 20/40 or 50%. The global miss rate of the second-level cache is 20/1000 or 2%. Then

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$
$$= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5.4 \text{ clock cycles}$$

To see how many misses we get per instruction, we divide 1000 memory references by 1.5 memory references per instruction, which yields 667 instructions. Thus, we need to multiply the misses by 1.5 to get the number of misses per 1000 instructions. We have 40×1.5 or 60 L1 misses, and 20×1.5 or 30 L2 misses, per 1000 instructions. For average memory stalls per instruction, assuming the misses are distributed uniformly between instructions and data:

$$\text{Average memory stalls per instruction} = \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \times \text{Miss penalty}_{L2}$$
$$= (60/1000) \times 10 + (30/1000) \times 200$$
$$= 0.060 \times 10 + 0.030 \times 200 = 6.6 \text{ clock cycles}$$

If we subtract the L1 hit time from the average memory access time (AMAT) and then multiply by the average number of memory references per instruction, we get the same average memory stalls per instruction:

$$(5.4 - 1.0) \times 1.5 = 4.4 \times 1.5 = 6.6 \text{ clock cycles}$$

As this example shows, there may be less confusion with multilevel caches when calculating using misses per instruction versus miss rates.

Six Basic Cache Optimizations

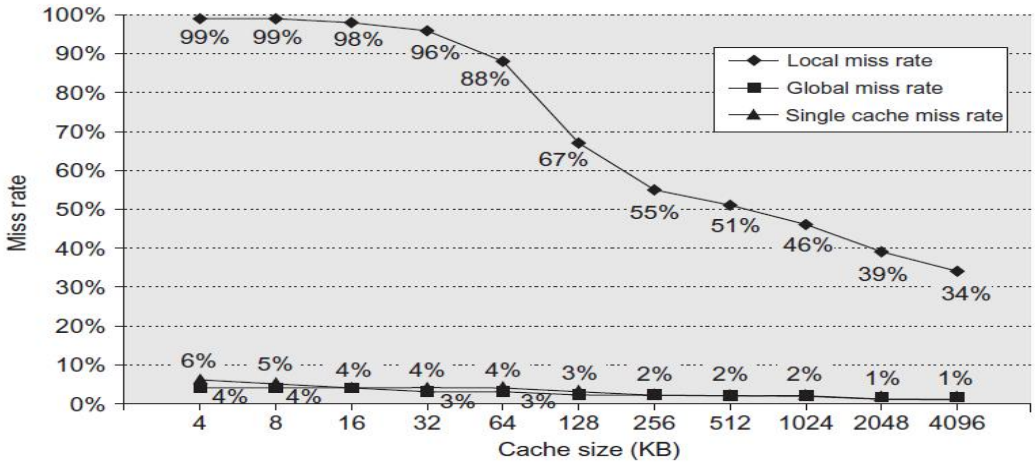

Figure B.14 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64 KiB first-level caches make little sense, as reflected in the high miss rates. After 256 KiB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32 KiB first-level cache. The L2 caches (unified) were two-way set associative with replacement. Each had split L1 instruction and data caches that were 64 KiB two-way set associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data were collected as in Figure B.4.

Six Basic Cache Optimizations

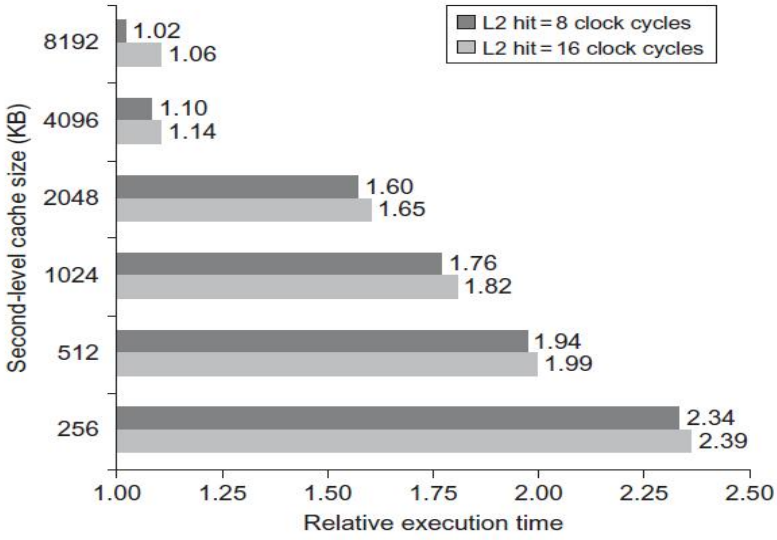

Figure B.15 Relative execution time by second-level cache size. The two bars are for different clock cycles for an L2 cache hit. The reference execution time of 1.00 is for an 8192 KiB second-level cache with a 1-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure B.14, using a simulator to imitate the Alpha 21264.

Six Basic Cache Optimizations

Example Given the following data, what is the impact of second-level cache associativity on its miss penalty?

- Hit time$_{L2}$ for direct mapped = 10 clock cycles.
- Two-way set associativity increases hit time by 0.1 clock cycle to 10.1 clock cycles.
- Local miss rate$_{L2}$ for direct mapped = 25%.
- Local miss rate$_{L2}$ for two-way set associative = 20%.
- Miss penalty$_{L2}$ = 200 clock cycles.

Answer For a direct-mapped second-level cache, the first-level cache miss penalty is

$$\text{Miss penalty}_{1\text{-way L2}} = 10 + 25\% \times 200 = 60.0 \text{ clock cycles}$$

Adding the cost of associativity increases the hit cost only 0.1 clock cycle, making the new first-level cache miss penalty:

$$\text{Miss penalty}_{2\text{-way L2}} = 10.1 + 20\% \times 200 = 50.1 \text{ clock cycles}$$

In reality, second-level caches are almost always synchronized with the first-level cache and processor. Accordingly, the second-level hit time must be an integral number of clock cycles. If we are lucky, we shave the second-level hit time to 10 cycles; if not, we round up to 11 cycles. Either choice is an improvement over the direct-mapped second-level cache:

$$\text{Miss penalty}_{2\text{-way L2}} = 10 + 20\% \times 200 = 50.0 \text{ clock cycles}$$
$$\text{Miss penalty}_{2\text{-way L2}} = 11 + 20\% \times 200 = 51.0 \text{ clock cycles}$$

Six Basic Cache Optimizations

- ✓ **Multilevel inclusion:** is the natural policy for memory hierarchies:
- ✓ L1 data are always present in L2.
- ✓ Inclusion is desirable because consistency between I/O and caches can be determined by checking the second-level cache.
- ✓ L1 data are always present in L2. Inclusion is desirable because consistency between I/O and caches can be determined by second-level cache.
- ✓ One drawback to inclusion is that measurements can smaller blocks for smaller first-level cache and larger blocks for larger second-level cache.
- ✓ Second-level cache must invalidate all first-level blocks that map onto the second-level block to be replaced causing higher first-level miss rate.

Six Basic Cache Optimizations

Fifth Optimization: Giving Priority to Read Misses over Writes to Reduce Miss Penalty

- ✓ With a write-through cache, improvement is a write buffer of proper size.
- ✓ Write buffers complicate memory accesses because they might hold the updated value of a location needed on a read miss.
- ✓ Cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block.
- ✓ Instead of writing dirty block to memory and then reading memory, copy dirty block to a buffer, then read and write memory.
- ✓ If a read miss occurs, processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.
- ✓ reduce cache miss penalties or miss rates, it is time to reduce final component of average memory access time.
- ✓ Hit time is critical because it can affect the clock rate of the processor;
- ✓ cache access time limits the clock cycle rate, even for processors that take multiple clock cycles to access the cache.

Six Basic Cache Optimizations

Sixth Optimization: Avoiding Address Translation During Indexing of the Cache to Reduce Hit Time

- ✓ Cache cope with the translation of virtual address to a physical address to access memory.
- ✓ Memory hierarchy and address of virtual memory exists on disk mapped to main memory.
- ✓ Virtual caches with physical cache used to identify traditional cache that uses physical addresses.
- ✓ it is important to distinguish two tasks: indexing the cache and comparing addresses.
- ✓ Issues are:
  - i. whether a virtual or physical address is used to index the cache
  - ii. whether a virtual or physical address is used in the tag comparison.
- ✓ Virtual addressing for both indices and tags eliminates address translation time from a cache hit.
- ✓ Page-level protection is checked as part of virtual to physical address translation.
- ✓ Every time a process is switched, virtual addresses refer to different physical addresses requiring the cache to be flushed.

Six Basic Cache Optimizations

Figure B.16 shows the impact on miss rates of this flushing

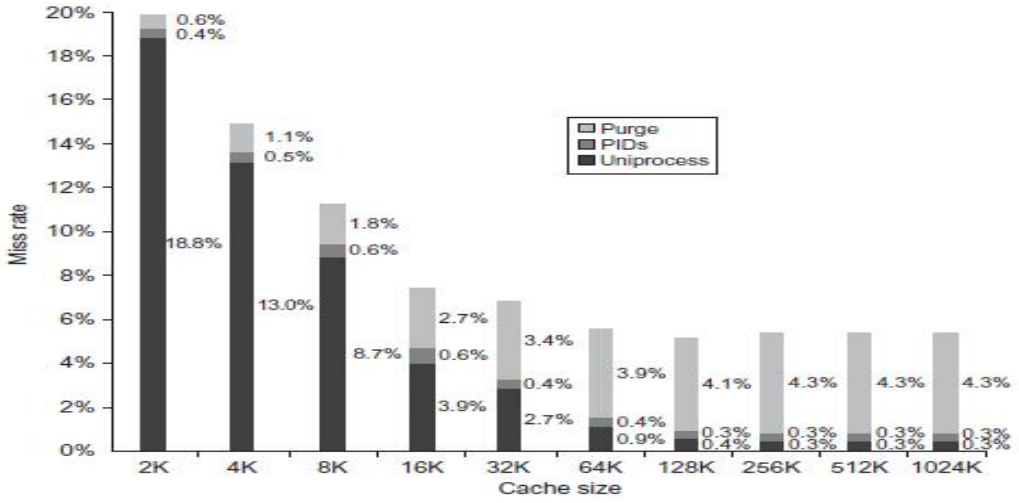

Figure B.16 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PID), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3%–0.6% and save 0.6%–4.3% over purging. Agarwal (1987) collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes. Note that the miss rate goes up from 128 to 256 K. Such nonintuitive behavior can occur in caches because changing size changes the mapping of memory blocks onto cache blocks, which can change the conflict miss rate.

Six Basic Cache Optimizations

- ✓ One solution is to increase the width of the cache address tag with a process-identifier tag (PID).
- ✓ If the operating system assigns these tags to processes, it flush the cache when a PID is recycled;
- ✓ **Synonyms Problem:**
- ✓ User programs may use two different virtual addresses for the same physical address.
- ✓ These duplicate addresses called synonyms or aliases, could result in two copies of the same data in a virtual cache; if one is modified, the other will have the wrong value.
- ✓ With a physical cache, accesses translated to same physical cache block.
- ✓ **Antialising:** Hardware solutions to the synonym problem, called antialiasing, guarantee every cache block a unique physical address.

Six Basic Cache Optimizations

Page Coloring:

- ✓ An older version of UNIX required all aliases to be identical in the last 18 bits of their addresses; this restriction is called page coloring.
- ✓ Page coloring is set associative mapping applied to virtual memory:
- ✓ 4 KiB (2^12) pages are mapped using 64 (2^6) sets to ensure that the physical and virtual addresses match in the last 18 bits.
- ✓ Direct-mapped cache that is 2^18 (256 K) bytes or smaller can never have duplicate physical addresses for blocks.
- ✓ Page coloring increases the page offset as software guarantees that the last few bits of the virtual and physical page address are identical.

Six Basic Cache Optimizations

Figure B.17 shows the organization of the caches, translation lookaside buffers (TLBs), and virtual memory.

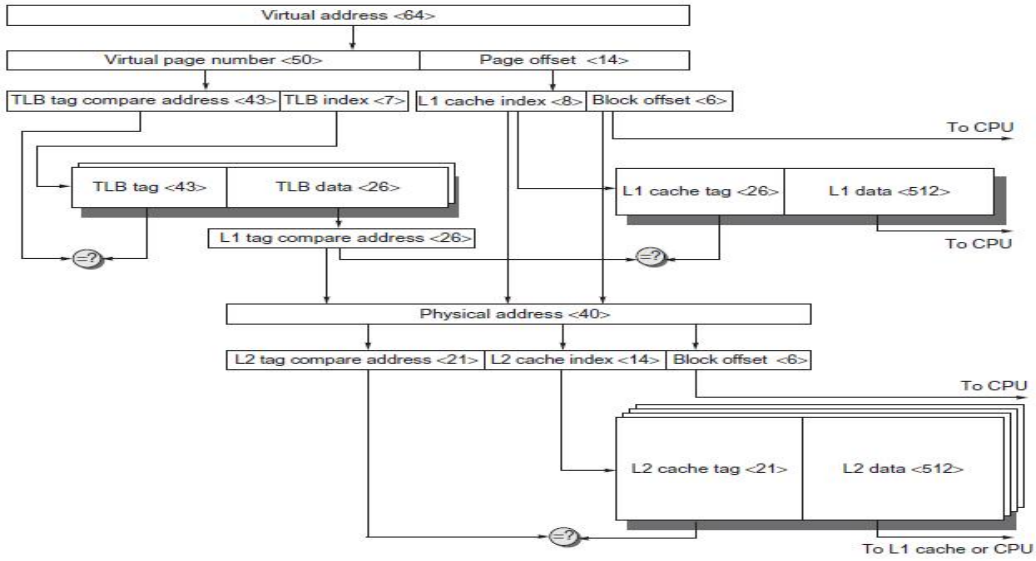

Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 16 KiB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KiB, and the L2 cache is a four-way set associative with a total of 4 MiB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

Six Basic Cache Optimizations

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used
Read priority over writes		+		1	Widely used
Avoiding address translation during cache indexing	+			1	Widely used

Figure B.18 Summary of basic cache optimizations showing impact on cache performance and complexity for the techniques in this appendix. Generally a technique helps only one factor. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

Virtual Memory

- ✓ Virtual memory divides physical memory into blocks and allocates them to different processes.
- ✓ virtual memory also reduce the time to start a program, because not all code and data need be in physical memory before a program can begin.

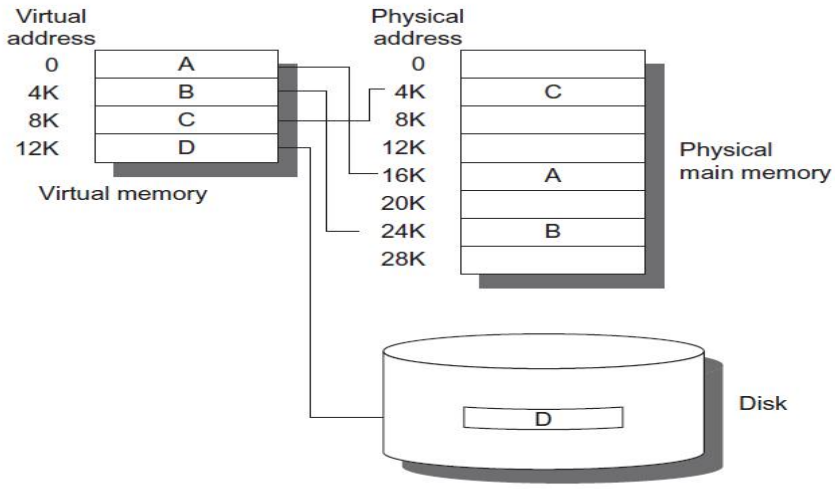

Figure B.19 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages, A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

Virtual Memory

Relocation: In addition to sharing protected memory space and automatically managing the memory hierarchy, virtual memory also simplifies loading the program for execution called relocation, this mechanism allows the same program to run in any location in physical memory.

Parameter	First-level cache	Virtual memory
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–3 clock cycles	100–200 clock cycles
Miss penalty	8–200 clock cycles	1,000,000–10,000,000 clock cycles
(access time)	(6–160 clock cycles)	(800,000–8,000,000 clock cycles)
(transfer time)	(2–40 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.1%–10%	0.00001%–0.001%
Address mapping	25–45-bit physical address to 14–20-bit cache address	32–64-bit virtual address to 25–45-bit physical address

Figure B.20 Typical ranges of parameters for caches and virtual memory. Virtual memory parameters represent increases of 10–1,000,000 times over cache parameters. Usually, first-level caches contain at most 1 MiB of data, whereas physical memory contains 256 MiB to 1 TB.

Virtual Memory

Differences between caches and virtual memory:

- ✓ Replacement on cache misses is controlled by hardware, while virtual memory replacement is controlled by the operating system.
- ✓ Size of the processor address determines the size of virtual memory, but the cache size is independent of the processor address size.

Virtual Memory

- ✓ Virtual memory systems can be categorized into two classes: those with fixed-size blocks, called pages, and those with variable-size blocks, called segments.
- ✓ Pages are fixed at 4096–8192 bytes, while segment size varies.
- ✓ Largest segment supported on any processor ranges from 216 bytes up to 232 bytes; the smallest segment is 1 byte.
- ✓ **Paged Segements:** Some computers use a hybrid approach called paged segments, in which a segment is an integral number of pages. This simplifies replacement because memory need not be contiguous, and the full segments need not be in main memory.
- ✓ hybrid is for a computer to offer multiple page sizes, with the larger sizes being powers of 2 times the smallest page size

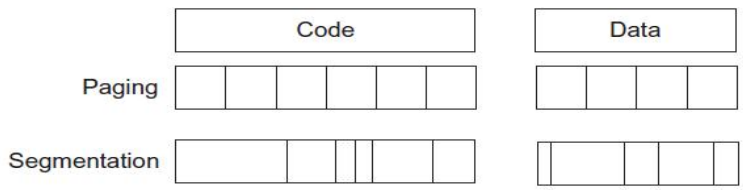

Figure B.21 Example of how paging and segmentation divide a program.

Virtual Memory

	Page	Segment
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Difficult (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

Figure B.22 Paging versus segmentation. Both can waste memory, depending on the block size and how well the segments fit together in main memory. Programming languages with unrestricted pointers require both the segment and the address to be passed. A hybrid approach, called *paged segments*, shoots for the best of both worlds: segments are composed of pages, so replacing a block is easy, yet a segment may be treated as a logical unit.

Four Memory Hierarchy Questions Revisited for Virtual Memory

Q1: Where Can a Block be Placed in Main Memory?

The miss penalty for virtual memory involves access to a rotating magnetic storage device. OS designers pick lower miss rates because of excessive miss penalty. OS allow blocks to be placed anywhere in main memory.

Q2: How Is a Block Found If It Is in Main Memory?

Both paging and segmentation rely page or segment number. It contains the physical address of the block. For segmentation, the offset is added to the segment's physical address to obtain the final physical address. For paging, the offset is simply concatenated to this physical page address (see Figure B.23).**h**

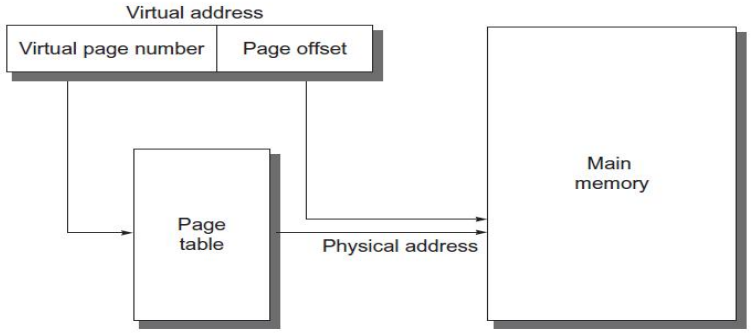

Figure B.23 The mapping of a virtual address to a physical address via a page table.

Four Memory Hierarchy Questions Revisited for Virtual Memory

Physical page addresses takes the form of a page table. Indexed by the virtual page number, size of table is the number of pages in the virtual address space.

Example # Given a 32-bit virtual address, 4 KiB pages, and 4 bytes per page table entry (PTE), the size of the page table would be (2^32/2^12)x2^2 = 2^22 or 4 MiB.

- ✓ To reduce the size of this data structure, some computers apply a hashing function to the virtual address.
- ✓ Hash allows the data structure to be the length of the number of physical pages in main memory. This number could be much smaller than the number of virtual pages. Such a structure is called an inverted page table.
- ✓ HP/Intel IA-64 covers both bases by offering both traditional pages tables and inverted page tables.
- ✓ To reduce address translation time, computers use a cache dedicated to these address translations, called a translation lookaside buffer.

Four Memory Hierarchy Questions Revisited for Virtual Memory

Q3: Which Block Should be Replaced on a Virtual Memory Miss?

- ✓ For minimizing page faults operating systems try to replace the least recently used (LRU) block because if the past predicts the future, that is the one less likely to be needed.
- ✓ Processors provide a reference bit or use bits, which is logically set whenever a page is accessed.
- ✓ OS periodically clears the use bits and records them so it can determine which pages were replaced during a particular time period.

Q4: What Happens on a Write?

- ✓ Level below main memory contains rotating magnetic disks that take millions of clock cycles to access.
- ✓ Due to discrepancy in access time, no one has yet built a virtual memory operating system that writes through main memory to disk on every store by the processor.
- ✓ the write strategy is always write-back.

Techniques for Fast Address Translation

- ✓ By keeping address translations in a special cache, a memory access rarely requires a second access to translate the data. This special address translation cache is referred to as a translation look aside buffer (TLB) or translation buffer (TB).
- ✓ A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page frame number, protection field, valid bit, and usually a use bit and dirty bit.
- ✓ To change the physical page frame number or protection of an entry in the page table, the operating system must make sure the old entry is not in the TLB; otherwise the system won't behave properly.
- ✓ OS resets dirty bits by changing the value in the page table and then invalidates the corresponding TLB entry.

Techniques for Fast Address Translation

Step 1 and 2:

- ✓ TLB uses fully associative placement; translation begins by sending the virtual address to all tags.
- ✓ Tag must be marked valid to allow a match.
- ✓ At the same time, the type of memory access is checked for a violation against protection information in the TLB.

Step 3:

- ✓ No need to include the 12 bits of the page offset in the TLB.
- ✓ Matching tag sends the corresponding physical address through a 40:1 multiplexor .

Step 4:

- ✓ Page offset is combined with physical page frame to form a full physical address. Address size is 40 bits

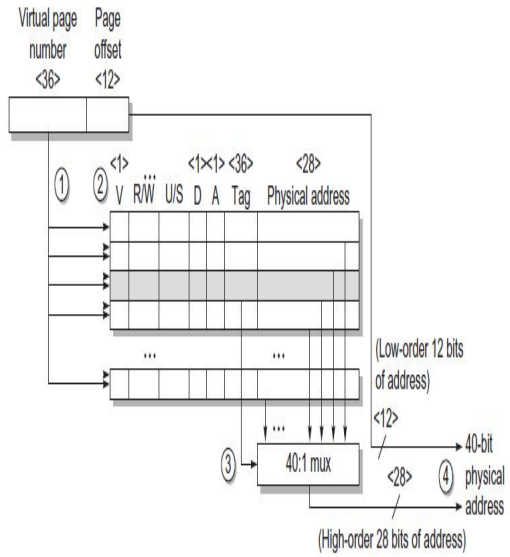

Figure B.24 Operation of the Opteron data TLB during address translation. The four steps of a TLB hit are shown as *circled numbers*. This TLB has 40 entries. Section B.5 describes the various protection and access fields of an Opteron page table entry.

Techniques for Fast Address Translation

Selecting a Page Size:

- ✓ Size of the page table is inversely proportional to the page size; memory (or other resources used for the memory map) can therefore be saved by making the pages bigger.
- ✓ larger page size can allow larger caches with fast cache hit times.
- ✓ Transferring larger pages to or from secondary storage, possibly over a network, is more efficient than transferring smaller pages.
- ✓ Number of TLB entries is restricted, so a larger page size means that more memory can be mapped efficiently, thereby reducing the number of TLB misses.

Recent microprocessors have decided to support multiple page sizes; TLB misses can be as significant on CPI as the cache misses.

Small page size will result in less wasted storage when a contiguous region of virtual memory is not equal in size to a multiple of the page size. The term for this unused memory in a page is internal fragmentation.

Summary of Virtual Memory and Caches

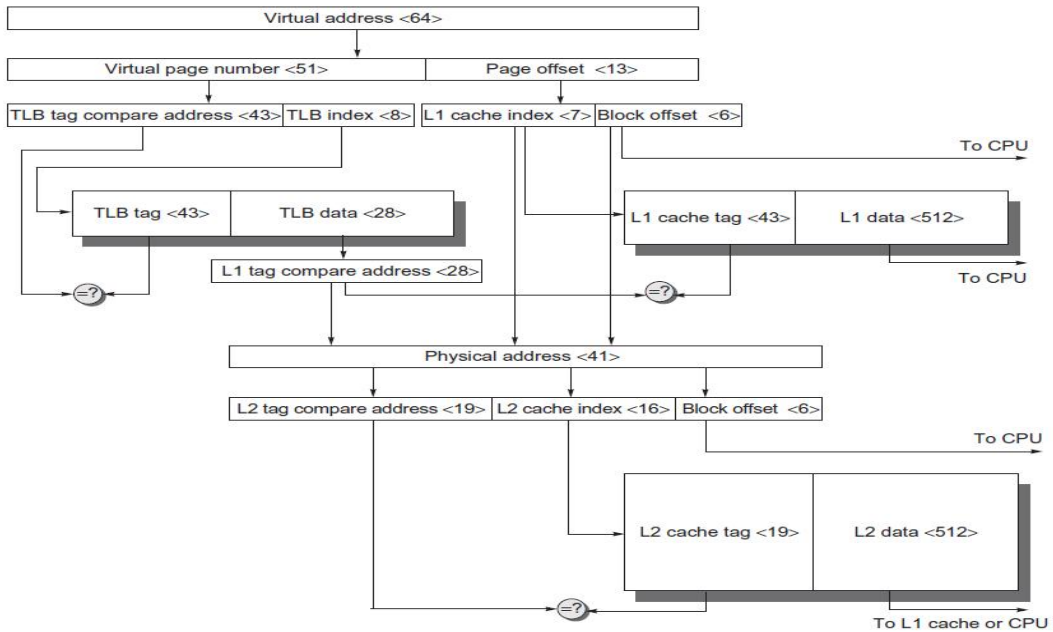

Figure B.25 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KiB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KiB, and the L2 cache is a direct-mapped 4 MiB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache is replication of pieces of this figure.

Summary of Virtual Memory and Caches

- ✓ Figure B.25 shows 64-bit virtual address to a 41-bit physical address with two levels of cache.
- ✓ L1 cache is virtually indexed and physically tagged because both cache size and page size are 8 KiB. L2 cache is 4 MiB. Block size for both is 64 bytes.
- ✓ 64-bit virtual address is divided into a virtual page number and page offset.
- ✓ Virtual page number is sent to TLB to be translated into a physical address and the high bit of the latter is sent to L1 cache as an index.
- ✓ If TLB match is a hit, then physical page number is sent to L1 cache tag to check for a match. If it matches, it's an L1 cache hit.
- ✓ If the L1 cache result is a miss, physical address is tried to use L2 cache.
- ✓ Middle portion of physical address is used as an index to 4 MiB L2 cache.
- ✓ Resulting L2 cache tag is compared with upper part of physical address to check for a match.
- ✓ If it matches, it is an L2 cache hit, and the data are sent to the processor.
- ✓ On an L2 miss, the physical address is then used to get the block from memory.

Protection and Examples of Virtual Memory

Protecting Processes:

- ✓ Processes can be protected from one another by having their own page tables, each pointing to distinct pages of memory.
- ✓ User programs must be prevented from modifying their page tables.
- ✓ Rings added to the processor protection structure expand memory access protection from two levels (user and kernel) to many more.
- ✓ Intel 80x86 protection structure uses rings
- ✓ Analogy of this system is to keys and locks: a program can't unlock access to the data unless it has the key.
- ✓ For these keys, or capabilities, to be useful, the hardware and operating system must be able to explicitly pass them from one program to another without allowing a program itself to forge them. Such checking requires a great deal of hardware support if time for checking keys is to be kept low.

Protection and Examples of Virtual Memory

A Segmented Virtual Memory Example: Protection in the Intel Pentium

- ✓ First enhancement is to double the traditional two-level protection model: the IA-32 has four levels of protection. The innermost level (0) corresponds to kernel mode and outermost level (3) is least privileged mode.
- ✓ IA-32 has separate stacks for each level to avoid security breaches between levels.
- ✓ IA-32 divides the address space, allowing both operating system and user access to full space.
- ✓ IA-32 allows OS to maintain protection level of called routine for the parameters that are passed to it.
- ✓ loophole in protection is prevented by not allowing user process to access itself. (Such security loopholes are called Trojan horses.)

Protection and Examples of Virtual Memory

Adding Sharing and Protection

- ✓ half of the address space is shared by all processes and half is unique to each process, called global address space and local address space.
- ✓ A descriptor pointing to a shared segment is placed in global descriptor table, while a descriptor for a private segment is placed in local descriptor table.
- ✓ A program loads an IA-32 segment register with an index to the table and a bit saying which table it desires. The operation is checked according to the attributes in the descriptor, the physical address being formed by adding the offset in the processor to the base in the descriptor, provided the offset is less than the limit field.
- ✓ Every segment descriptor has a separate 2-bit field to give the legal access level of this segment. A violation occurs only if the program tries to use a segment with a lower protection level in the segment descriptor.
- ✓ Program could be given a descriptor for information that has writable field clear, meaning it can read but not write the data. A trusted program can then be supplied that will only write the year-to-date information. It is given a descriptor with the writable field set (Figure B.26).

Protection and Examples of Virtual Memory

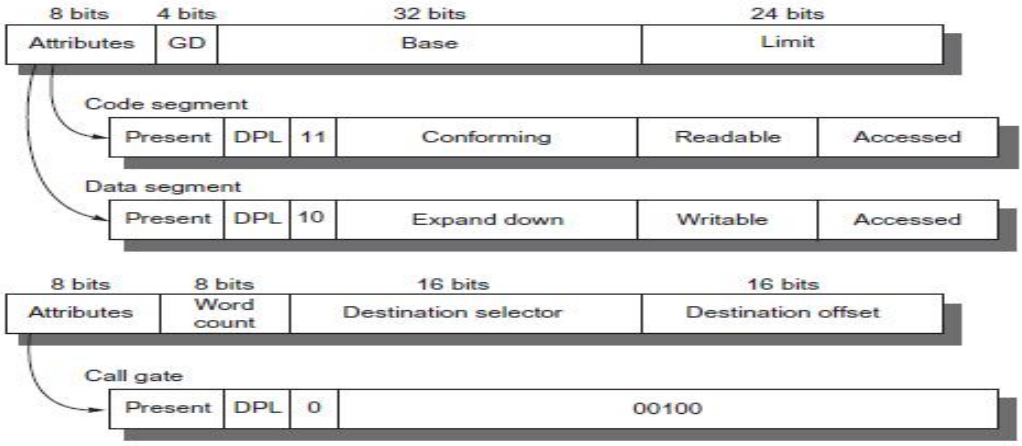

Figure B.26 The IA-32 segment descriptors are distinguished by bits in the attributes field. *Base, limit, present, readable,* and *writable* are all self-explanatory. D gives the default addressing size of the instructions: 16 bits or 32 bits. G gives the granularity of the segment limit: 0 means in bytes and 1 means in 4 KiB pages. G is set to 1 when paging is turned on to set the size of the page tables. DPL means *descriptor privilege level*—this is checked against the code privilege level to see if the access will be allowed. *Conforming* says the code takes on the privilege level of the code being called rather than the privilege level of the caller; it is used for library routines. The *expand-down field* flips the check to let the base field be the high-water mark and the limit field be the low-water mark. As you might expect, this is used for stack segments that grow down. *Word count* controls the number of words copied from the current stack to the new stack on a call gate. The other two fields of the call gate descriptor, *destination selector* and *destination offset*, select the descriptor of the destination of the call and the offset into it, respectively. There are many more than these three segment descriptors in the IA-32 protection model.

Protection and Examples of Virtual Memory

Adding Safe Calls from User to OS Gates and Inheriting Protection Level for Parameters

- ✓ IA-32 approach is to restrict where the user can enter a piece of code, to safely place parameters on the proper stack, and to make sure the user parameters don't get the protection level of the called code.
- ✓ To restrict entry into others' code, IA-32 provides a special segment descriptor, or call gate, identified by a bit in the attributes field.
- ✓ Call gates are full physical addresses of an object in memory;
- ✓ In FigureB.26, When a call instruction invokes a call gate descriptor, it copies number of words specified in the descriptor from local stack onto stack corresponding to this segment.
- ✓ This copying allows user to pass parameters by first pushing them onto local stack.
- ✓ Hardware safely transfers them onto the correct stack. A return from call gate will pop the parameters off both stacks and copy any return values to proper stack.
- ✓ IA-32 solves this problem by dedicating 2 bits in every processor segment register to the requested protection level.

Protection and Examples of Virtual Memory

- ✓ When an operating system routine is invoked, it can execute an instruction that sets this 2-bit field in all address parameters with protection level of user that called the routine.
- ✓ when these address parameters are loaded into the segment registers, they will set the requested protection level to the proper value.

A Paged Virtual Memory Example: The 64-Bit Opteron Memory Management

- ✓ 64-bit virtual address of AMD64 architecture is mapped onto 52-bit physical addresses, although implementations can implement fewer bits to simplify hardware.
- ✓ Opteron uses 48-bit virtual addresses and 40- bit physical addresses.
- ✓ AMD64 requires that the upper 16 bits of the virtual address be just the sign extension of the lower 48 bits.
- ✓ Size of page tables for the 64-bit address space is alarming.
- ✓ AMD64 uses a multilevel hierarchical page table to map the address space to keep the size reasonable. The number of levels depends on the size of the virtual address space.

Protection and Examples of Virtual Memory

Figure B.27 shows the four-level translation of the 48-bit virtual addresses of the Opteron.

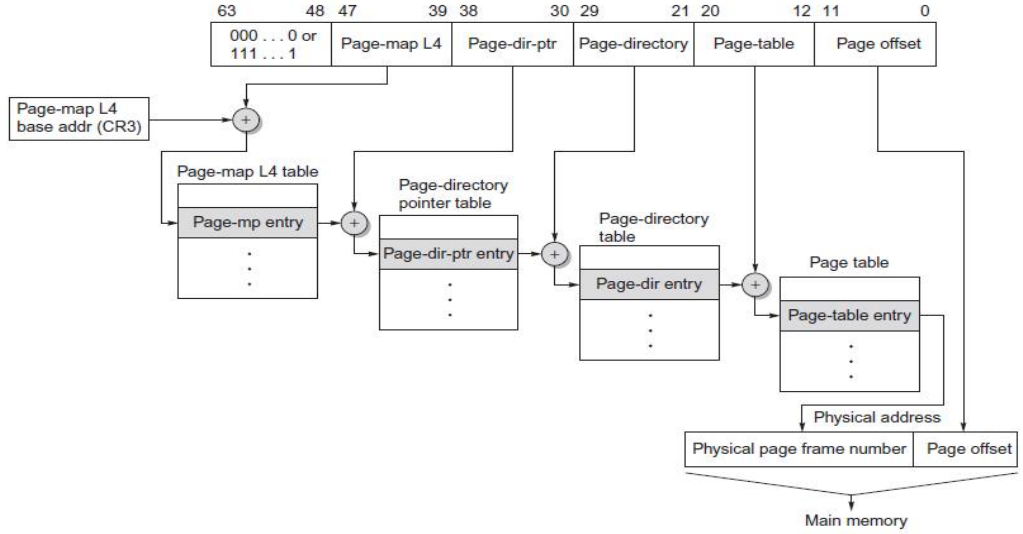

Figure B.27 The mapping of an Opteron virtual address. The Opteron virtual memory implementation with four page table levels supports an effective physical address size of 40 bits. Each page table has 512 entries, so each level field is 9 bits wide. The AMD64 architecture document allows the virtual address size to grow from the current 48 bits to 64 bits, and the physical address size to grow from the current 40 bits to 52 bits.

Protection and Examples of Virtual Memory

- ✓ Offsets for each of these page tables come from four 9-bit fields.
- ✓ Address translation starts with adding the first offset to page-map level 4 base register and then reading memory from this location to get the base of the next-level page table.
- ✓ Next address offset is added to newly fetched address and memory is accessed again to determine the base of the third page table.
- ✓ Last address field is added to this final base address and memory is read using this sum to get the physical address of page being referenced.
- ✓ This address is concatenated with 2-bit page offset to get the full physical address.
- ✓ Page table in the Opteron architecture fits within a single 4 KiB page.
- ✓ Opteron uses a 64-bit entry in each of these page tables. The first 12 bits are reserved for future use, next 52 bits contain the physical page frame number, and last 12 bits give the protection.

Protection and Examples of Virtual Memory

Fields vary between the page table levels:

Presence: page is present in memory.

Read/write: page is read-only or read-write.

User/supervisor: user can access the page or if it is limited to the upper three privilege levels.

Dirty: if page has been modified.

Accessed: if page has been read or written since the bit was last cleared.

Page size: Last level is for 4 KiB pages or 4 MiB pages; if it's the latter, then the Opteron only uses three instead of four levels of pages.

No execute: Not found in the 80386 protection scheme, this bit was added to prevent code from executing in some pages.

Page level cache disable: page can be cached or not.

Page level write through: page allows write back or write through for data caches.

Opteron employs four TLBs to reduce address translation time, two for instruction accesses and two for data accesses. Like multilevel caches, the Opteron reduces TLB misses by having two larger L2 TLBs: one for instructions and one for data.

Protection and Examples of Virtual Memory

Parameter	Description
Block size	1 PTE (8 bytes)
L1 hit time	1 clock cycle
L2 hit time	7 clock cycles
L1 TLB size	Same for instruction and data TLBs: 40 PTEs per TLBs, with 32 4 KiB pages and 8 for 2 MiB or 4 MiB pages
L2 TLB size	Same for instruction and data TLBs: 512 PTEs of 4 KiB pages
Block selection	LRU
Write strategy	(Not applicable)
L1 block placement	Fully associative
L2 block placement	4-way set associative

Figure B.28 Memory hierarchy parameters of the Opteron L1 and L2 instruction and data TLBs.

Thread Level Parallelism

Introduction

This increased importance of multiprocessing reflects several major factors:

- ✓ Lower efficiencies in silicon and energy use attempted to find and exploit more ILP, since power and silicon costs grew faster than performance.
- ✓ High-end servers as cloud computing and software-as-aservice become more important.
- ✓ Data-intensive applications driven by the availability of massive amounts of data on the Internet.
- ✓ Use multiprocessors effectively in server environments where there is significant inherent parallelism, arising from data parallelism or request-level parallelism.
- ✓ combination of Amdahl's Law effects and Dennard scaling mean that the future of multicore may be limited, at least as a method of scaling up the performance of single applications
- ✓ Execution of multiple, relatively independent processes may originate from one or more users, which is a form of request-level parallelism.
- ✓ Request-level parallelism may be exploited by a single application running on multiple processors, such as a database responding to queries or multiple applications running independently called multiprogramming

Multiprocessor Architecture: Issues and Approach

- ✓ **Grain Size:** Amount of computation assigned to a thread called the grain size.
- ✓ Thread-level parallelism is identified at a high level by the software system or programmer and threads consist of hundreds to millions of instructions that may be executed in parallel.
- ✓ Threads can also be used to exploit data-level parallelism although the overhead is higher than would be seen with an SIMD processor or with a GPU.
- ✓ Overhead means that grain size must be sufficiently large to exploit the parallelism efficiently.
- ✓ **Symmetric (shared-memory) multiprocessors (SMPs):** or centralized shared-memory multiprocessors consists of small to moderate numbers of cores typically 32 or fewer.
- ✓ **Symmetric Multiprocessing:** For multiprocessors with such small processor counts, it is possible for the processors to share a single centralized memory that all processors have equal access called symmetric multiprocessing.

Multiprocessor Architecture: Issues and Approach

- ✓ In multicore chips,memory is often shared in a centralized fashion among the cores; most existing multicores are SMPs
- ✓ **Non Uniform Cache Access:** Multicores have nonuniform access to outermost cache, a structure called NUCA for Nonuniform Cache Access and are are not SMPs even if they have a single main memory.

  IBM Power8 has distributed L3 caches with nonuniform access time to different addresses in L3.
- ✓ **Uniform Memory Access Processors:** SMP architectures are also sometimes called uniform memory access (UMA) multiprocessors, arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks.

Multiprocessor Architecture: Issues and Approach

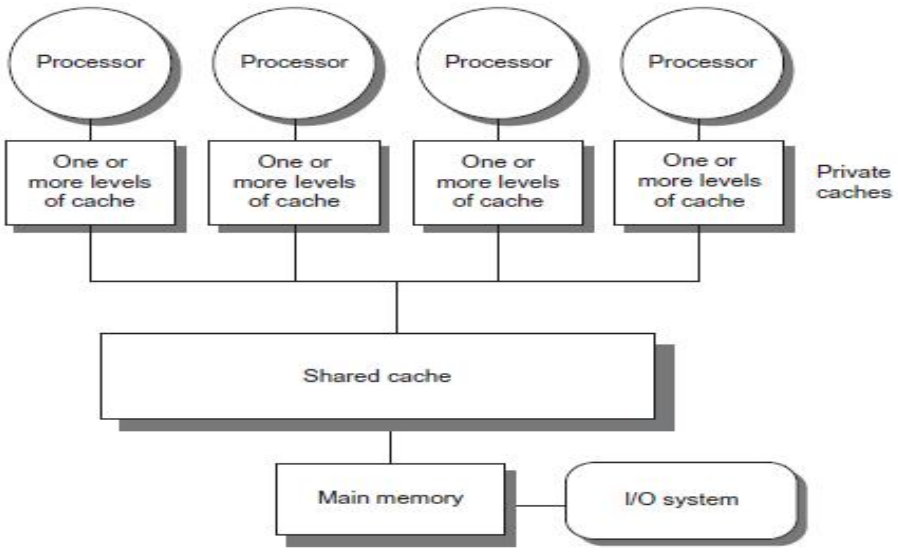

Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor-cache subsystems share the same physical memory, typically with one level of shared cache on the multicore, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip design, an interconnection network links the processors and the memory, which may be one or more banks. In a single-chip multicore, the interconnection network is simply the memory bus.