Computer Design and Organization

Cache Performance (Uniprocessors)

Jean-Loup Baer

The first caches were introduced in 1968 in the IBM 360/85. In 30 years there have been hundreds of implemented and proposed improvements on cache design in single and multiprocessor environments. Yet, new features for caches and research on caches, or rather memory hierarchies, is still of actuality because memory latency remains the main bottleneck in high-performance computer systems.

Consider this back of the envelope calculation of the CPI contributed by the memory hierarchy. Assume that an L1 hit takes 1 cycle, an L1 miss - L2 hit takes 10 cycles and an L1 miss - L2 miss (i.e., a main memory access) takes 100 cycles. Let's further assume that the hit ratio for L1 is 97% and that one half of the misses in L1 are L2 hits. The CPI contributed by the cache, CPI_c is:

$$CPI_c = 0.03 \times 10 + 0.015 \times 100 = 1.8$$

If by some clever technique I can reduce the miss rates in L1 by 10% and insure that now 2/3 of L1 misses are L2 hits, then

$$CPI_c^1 = 0.027 \times 10 + 0.009 \times 100 = 1.17$$

a very significant improvement of about 35%.

In these notes, I summarize some of the techniques that have been proposed to improve cache performance. This is by no means an exhaustive survey. Some of these techniques are also described in your text.

1 Increasing write hits; decreasing write latency

Loads (reads) occur two times more frequently than stores (writes) but the miss ratios due to respectively loads and stores are approximately the same. Thus enhancing write behavior is important but not as critical as improving on load performance.

The primary goal for cache enhancements wrt to writes is to insure that write traffic to the next level of the memory hierarchy will not slow down the CPU. In other words, means to reduce the traffic argues for a *write-back* policy in that regard since there will be fewer writes. On the other hand, recall that *write-through* has advantages in terms of reliability (no need for ECC in the cache, next level consistent with the write-through cache) and ease of implementation. High-performance systems use a write-back cache at the interface off-board cache / main memory. For the on-chip /off-chip cache hierarchy, both write-back and write-through have been advocated. There are memory subsystems that allow a run-time decision on which write policy to follow on a page per page basis (the information to do that is recorded in the page table entry, PTE).

The write miss policy (allocate vs. non-allocate, fetch-on-write vs. no-fetch-on-write) dictates latency more than bandwidth. Another possibility for write-through caches is write-before-hit, i.e., the line in the cache is written without a tag check so that writes can proceed in the pipeline as fast as reads [1]. This appears to be feasible only for direct-mapped caches. Thus, we have possibly four useful combinations for on-chip **write-through** caches, namely:

[1] For reads you can "assume" that you have a hit and proceed with the check in parallel; if you had a miss you cancel sending the data to the register file. For a write-back cache you cannot "assume" a hit. Why?

1. *fetch-on-write and write-allocate*
   Proceed like on a read-miss (fetch the line from memory) and perform a write-through after that. This policy can also be used for **write-back** caches.
2. *no-fetch-on-write and write-allocate* called *write validate*
   This policy requires a valid bit per entity being written rather than a valid bit for the whole line; the portion of the line being written is valid in the cache. So if your ISA allows byte storing, this is expensive (it was proposed for the Alpha ISA that does not have load/store byte instructions).
3. *no-write-before-hit, no-fetch-on-write, no write-allocate* called *write around*
   The cache is not changed (on write misses). This policy can also be used for **write-back** caches.
4. *write-before-hit, no-fetch-on-write, no write-allocate* called *write invalidate*
   Possible for direct-mapped caches only. The line is invalidated on a miss except for the data being written since "erroneous" data has been written into it. It requires a valid bit per entity being written.

Overall, it has been shown that *write validate* performs best for write-through caches but it is also more costly in terms of valid bits. The fetch-on-write and write-allocate policy is the one preferred for write-back caches.

Simple *write buffers* are used to store, in FIFO order, data that has to be written back to memory. They are particularly useful for write-through caches. Note that with a write buffer, checks must be made on read misses to see whether the block is in the write buffer. In the case of multiprocessors, cache coherency checks must also include the write buffers. Another problem with write buffers is that writes have a tendency of being successive, thus the buffer needs to be deep.

A *coalescing write buffer* is one in which writes to an address already in the write buffer are combined within the buffer. Note that there is a tension between writing entries in the coalescing buffer to memory at a fast rate (little coalescing takes place) and allowing much coalescing to happen by keeping the buffer as full as possible (but then the processor will stall on stores with full buffers and no coalescing).

To overcome part of the dilemma of the last paragraph, coalescing write buffers can be extended to become *write caches*. Write caches are small caches with a write miss allocate, write-back, no allocate on read miss strategy. They have a dirty/valid bit per byte (if byte stores are allowed). In single processor environments they are most useful when added to a write-through cache. A couple of studies have provided slightly different guidelines for the design of write caches. The first advocates a 4 line (8 or 16 bytes) direct-mapped write cache while the second is in favor of a fully associative write cache with 5 entries of 8 bytes each. As far as I know, no current microprocessor uses a write cache.

Write buffers can also be used in conjunction with write-back caches to buffer the dirty lines that need to be written back.

2 Increasing "associativity"

As set-associativity increases, miss ratios decrease. However, the larger the set-associativity the more comparators and levels of multiplexing are needed. Hence cache access time can become the critical path in the pipeline and increasing it would be detrimental to overall performance. Several techniques have been proposed to make direct-mapped caches or caches with small associativity behave better, i.e., reduce the number of conflict misses.

2.1 Victim caches

A way to reduce conflict misses is to place a small fully associative buffer "behind" the first level cache and "before" the next level in the memory hierarchy. The combination of the regular and victim cache functions as follows (we restrict ourselves to reads).

1. Check the regular cache. On a hit proceed normally.
2. If there is a regular cache miss, check the victim cache.
   (a) If there is a hit in the victim cache, send the info to the register and swap the entry you would replace in the regular cache (the *"victim"* with the one you hit in the victim cache).
   (b) If there is a miss in the victim cache, the miss needs to be resolved at the next level of the memory hierarchy. The missing line replaces the victim in the regular cache. The victim is placed in the victim cache. If the latter is full, one of its lines is evicted to the next level in the memory hierarchy.

Experiments show that small victim caches (say 4 to 8 entries) are often quite effective in removing conflict misses for small (say less than 32K) direct-mapped caches. Larger victim caches have been proposed to assist L2 caches (in one of the recent Alpha's). In general accessing (swapping) the victim cache takes one extra cycle but it is well worth the extra cycle if misses can be substantially reduced.

2.2 Increasing (small) direct-mapped cache "associativity"

Direct-mapped caches have the advantages of simplicity of design and (slightly) faster access times. They use to be the preferred design for small caches (first level). However, the smaller the cache, the more important conflict misses become (from 20 to 40% of misses are conflict misses in direct-mapped caches). Several designs have been proposed to improve on the "associativity" of direct-mapped caches with the goal of disturbing as little as possible the fast access time of a standard direct-mapped cache.

The first idea is to split a direct-mapped cache in two. The first half is accessed using the standard hashing function (index in the address field). In case of a hit we are done (1 cycle). In case of a miss, we rehash and access the second half (one more cycle). In case of a hit in the second half, the locations accessed in the first half and in the second are swapped (1 extra cycle) and the data is forwarded to the CPU. In case of a miss, we go to memory and swap. Care must be taken to have "easy" hashing functions. For example, for the second hashing one can flip the highest bit in the index, yielding *column-associative caches*. This requires that this highest bit be stored in the tag. One more improvement is needed, namely a rehash bit/block, to prevent thrashing for reference strings of the form *abcbcbcbc*... (of course thrashing will occur for *abcabcabc*... as it would for a regular 2-way set associative cache) where *a, b,* and *c* are all mapping to the same block in the cache. Without this rehash bit, *a* would remain in the cache and *b* and *c* would ping-pong in and out. Thus the hardware cost is 2 tag bits/block plus the swapping logic. Simulations show that the miss rates of *column associative caches* are similar to those of two-way set associative caches and therefore smaller than those of direct-mapped caches, and that the average memory access times are also significantly smaller than those of direct-mapped caches. There were rumors that Intel would use caches with a similar idea but recent on-chip caches in the Pentium family are 2-way set-associative.

The *MRU scheme* is based on the fact that in a set-associative cache, a very high-percentage of hits occurs on the most recently used (MRU) line in a cache. Some experiments have shown a hit ratio of 0.85 to the MRU partition in a 32-way set-associative cache and 0.95 in an 8-way if the MRU partition capacity is significant enough, e.g., 4K bytes. The overall miss ratio of an MRU replacement algorithm, where on a miss the non-MRU line is selected randomly for replacement, shows a degradation of less than 10% over pure LRU in an 8-way set-associative cache. The conclusions are therefore obvious; partition the cache in two parts: the MRU one (on-chip for example) and the other one with the remaining capacity. The advantage of such a partition is that the two parts do not have to be of the same capacity. The disadvantage is that we'll have swapping/thrashing for reference strings of the form *abababab a*.... The MRU scheme was proposed for some IBM mainframes. It might have been implemented in some versions of the PowerPC.

While fast access times are of prime importance for first level caches, they are not as critical for second level caches. Moreover, other criteria such as multilevel inclusion and the importance of a high hit ratio since a memory request is very expensive, play in favor of set-associative caches with rather large associativity. Techniques to implement cheap and not too slow searches for these caches range from a naive approach (look at each member of a set serially), to MRU for the first compare, to LRU with a prediction table for the next use (MIPS R10000) and then look naively, to *partial compares*, i.e., a method where in an x-way set associative cache, x partial tags are first compared (using a comparator of the same width as if the cache were direct-mapped) and those sets with partial matches are then selected for a second serial comparison. Experiments show that MRU and partial compares require approximately the same number of probes.

2.3 Increasing large caches "associativity": Page coloring

In large caches, where large means that the size of the cache is larger than the page size, bits of the virtual address are needed to select the cache line. Therefore conflict misses can be reduced by a careful mapping of virtual pages to physical frames.

A number of static page allocation techniques have been proposed. The meaning of "static" is that the allocation policy is not based on dynamic referencing information (some research has been done on dynamic remapping but it won't be covered in these notes). Rather, on a page fault, a page is allocated from the pool of available pages in a non-random (or non-FIFO) fashion. Let's call a *bin* the group of cache lines (or sets) selected by the index bits (bin-index) of the address that are not part of the page offset.

The first technique is *page coloring*, a variant of what has been implemented on MIPS. The mapping is very simple, trying to emulate a virtually addressed cache, i.e., the system tries to select a frame from the pool whose bin-index bits match those of the virtual address of the faulting page. If it does not succeed, then a frame is selected in FIFO fashion from the pool. This form of page coloring avoids cache conflicts between sequential virtual pages of the same process but does not avoid inter-address space conflicts since it might often give the same mapping to, say, stack pages of different processes. And since we are looking at large L2 caches shared by several processes, the inter-address space conflicts might dominate. This latter drawback can be avoided in refining the technique by hashing on PID bits. The next technique, called *bin hopping*, uses a roving pointer to point to the last bin selected (with wrap around). It then allocates the next frame from the bin pointed to by the roving pointer and advances the latter. If the selected bin is full, the pointer goes to the next bin. The advantages are similar to that of page coloring with the addition that, in case of a full bin, the next selection is less random. Note that the above two techniques are very simple but do not utilize the knowledge of the occupancy of bins. This knowledge can be acquired and used in the decision process. In the *best bin* strategy, the number of (per-process) pages in a given bin is kept in a *used* counter and the number of pages available system-wide for this bin is kept in a *free* counter. On a page fault, a frame in the bin with smallest *used* and then largest *free* values is selected.

Experiments show that the *best bin* is consistently better than random (10% to 20% reduction in misses) and that naive page coloring (i.e., without hashing on PID bits) is not good.

2.4 Skewed-associative caches

Yet another design is to have different mapping functions for the two halves of a two-way associative cache. Note that the two hashing functions are not independent. They must be able to "disperse" in the second half those addresses that conflict in the first. Experiments show that a 2-way skewed cache performs (in terms of miss ratios) as well as a 4-way set associative cache. As far as I know, skewed caches have not been implemented in current micros.

3 Tolerating memory latency

There are several ways of reducing or tolerating memory latency. The first one is prefetching, an alternative to resolving cache misses only on demand. A second one is to allow several cache misses to be in progress via lock-up free caches.

3.1 Prefetching

Prefetching can be used to reduce compulsory and conflict misses. It can be hardware-based, compiler-directed, or a combination of both.

Any prefetching scheme has for goal to reduce the processor stall time by bringing data into the cache before its use so that it can be accessed without delay. Several factors have to be considered:

- *when* to prefetch (too early, we risk cache pollution; too late we mask only part of the memory latency);
- *what* to prefetch (ideally the semantic object that will be referenced; practically –for hardware schemes – a cache line);
- *where* to prefetch (in general prefetching directly in the cache is superior to using and managing a dedicated buffer).

Hardware-based approaches can be spatial, where access to the current block is the basis for the prefetch decision, and temporal, where lookahead decoding of the instruction stream is implied (this is the rationale behind so-called *decoupled architectures* where there are two instruction streams: one for load-stores and one for the rest).

In the spatial schemes, prefetches occur when there is a miss on a cache block. An example is the one block lookahead (OBL) policy where when block i is referenced, block $i+1$ is prefetched (variations on OBL include learning techniques similar to those for naive branch prediction). An extension to OBL where several consecutive data streams are prefetched in FIFO *stream buffers* has been proposed. In OBL and extensions, miss rates can be reduced, mostly for direct-mapped caches, at the expense of some increase in memory traffic. Stream buffers can be very effective if they are augmented with a stride detection mechanism as the one briefly described in the next paragraph. In some scientific programs, stream buffers with stride detectors can replace advantageously large L2 caches.

A more sophisticated approach to hardware-based prefetching for D-caches, is to predict the instruction execution stream and the data access patterns far enough in advance. The instruction stream is predicted with the usual Branch Prediction Table. The data access patterns are predicted with the help of a Reference Prediction Table (RPT) that holds data access patterns of load instructions. The RPT is organized as an instruction cache. Minimally, each entry in the table will contain a tag related to the instruction address, fields to record the memory operand address and its stride (i.e., the difference between the operand addresses for the last two loads at that PC address), and a state transition field. When the status of the state transition field says "prefetch", the operand at the current memory address + the stride is prefetched. In a *look-ahead* scheme, a Look-Ahead Program Counter (LA-PC), that will remain about one memory latency ahead of the regular PC is used to access the RPT to generate prefetches.

Another approach is to consider loading lines or super-lines (a multiple, say 4, number of consecutive lines). The decision on whether to load or superload depends on the pattern of utilization of the (super)lines. A similar approach is to look at lines and sublines (like in sector caches).

3.2 Lock-up free caches

Until recently, the CPU used to stall on a cache read miss. The newest generation of microprocessors include *lock-up free* D-caches, i.e., caches that allow several outstanding read miss requests. The concept applies as well for read and write misses but write buffers or write caches can take care of most of the write latency. The first description of lock-up free caches is almost 20 years old but the implementation complexity has slowed down a wide use of lock-up free caches in their full generality. The coming of age of out-of-order processors has obliged designers to include lock-up free caches in the most aggressive designs. Note however that a restricted form of non-blocking loads has existed for some time e.g., "hit under miss" in the HP PA7100, i.e., one outstanding miss is possible.

To make a regular cache lock-up free, a set of MSHR's (Miss Status Holding Register) is added, with one such MSHR per outstanding read miss that will be allowed. Each MSHR must hold a valid (busy) bit (if all MSHR's are busy, the CPU will stall), the address of the requested cache block, and a comparator so that further misses to the same block do not acquire another MSHR. The cache index can also be kept although it can be recomputed when the data comes back from the next level in the memory hierarchy. If the request is to be forwarded directly to the CPU, then for each possible word in a cache block, the MSHR must contain a valid bit (in case of writes), the destination address (a register) as well as the type of load. Note that with this organization, there cannot be two misses for the same word in a block, a situation that can happen in two different ways such as loads of different bytes of the same word or multiple loads of the same word to different registers. Also, when data comes back from memory it must either have a tag showing which MSHR will handle the request or the request address so that the MSHR's can compare it with the one it has stored.

Variations on this design comprise MSHR's where several generic word fields are used instead of one per position in the block (this allows to circumvent the two drawbacks aboveat the cost of extra hardware), using the cache block where the contents of the miss should be stored to hold the MSHR information (the problem might be one of bandwidth there), and inverted MSHR's where each destination (register) has its own MSHR. Results of extensive simulations show that the simplest technique (one MSHR) is all that is needed for integer type programs. However for numeric programs sophisticated implementations (e.g., in cache MSHR's) will pay off.

It is also clear that there is a direct interaction between code scheduling and the value of lock-up free caches. The code scheduler should try and schedule loads taking into account the load miss latency rather than the load hit latency as is done usually. Thus, loads should be scheduled as far away as possible from their use, e.g., as far away as the latency to the next level although there is little incentive to go beyond 10 instructions. Methods such as trace scheduling, loop unrolling, and/or enhancements to traditional list schedulers are to be used to take full advantage of lock-up free caches.

4 Virtually addressed caches

Vanilla caches are physically addressed. This requires that the address translation from virtual to physical address must be done (in the TLB) before the final tag checking occurs. In heavily pipelined architectures a stage of the pipeline can be devoted to this checking (cf. DEC Alpha 21064). Another way to make this process as parallel as possible is to use a set-associative cache such that the page offset bits are sufficient to access data in the cache at the same time as tags are checked (cf. IBM Power PC 601: 4K page, 8-way set-associative = 32K cache). Note that this comparison is not simple. In fact the 3 stages devoted to cache access in the Alpha take less "real" time than the two cycles in Power PC. Note also that for a fixed page size, the possible associativity limits the cache size. Page coloring can come to the rescue by making some of the physical and virtual address bits be the same.

However, there are alternatives, namely *virtually addressed* caches and *virtually indexed* caches. Let's call a physically (Real) addressed cache an R/R cache (real address/real tag). A virtually addressed cache is V/V (indexed by virtual address/ virtual tag) and a virtually indexed cache is V/R (indexed by virtual address/ real tag).

The main advantage of a V/V cache is that address translation is not needed on cache hits. Note that it is needed though on cache misses and on writes in write-through caches. But the speed requirement is not the same and thus the TLB could be slower than the cache. However, there are several problems with V/V caches. First, on a context-switch the whole cache must be invalidated (and a bunch of write-backs needed if write-back is the write policy). The flush can be partially avoided by including the PID in the tag. Only when a PID is recycled does the part of the cache with lines with the same PID need to be flushed. Second, *synonyms* need to be taken care of. A synonym occurs when two virtual addresses map to the same physical address. Synonyms can be disallowed (via software) or can be taken care of with a reverse translation buffer in a cache hierarchy (outside the scope of these notes), with the constraint that a single copy of the synonym be present in the V/V cache. Note that synonyms are not rare; synonym frequency could be as high as the miss ratio of an R/R cache in large caches (256KB) or caches with large associativity. The frequency is however very small in small to medium caches (32K or less). A third problem has to do with I/O that usually works on physical address. Again, the reverse translation buffer technique can take care of it. Thus, since a V/V cache performance, in terms of hit ratio, is good only for small caches and since its implementation is much simpler if it is part of a cache hierarchy, they should be considered only as candidates for first-level on-chip caches where in fact they are most needed.

In V/R caches, the cache is indexed by the virtual address. Thus translation in the TLB and data access can be done in parallel without the restrictions imposed by R/R caches for the same goal. Note that we can still have synonyms in V/R caches. In the HP Precision architecture, a write-back V/R cache is used. Synonyms are avoided by software (two virtual pages are not allowed to map to the same physical page) and consistency for DMA I/O is taken care of by special instructions that can flush a line or the whole data cache. Less stringent consistency conditions are possible. The basic idea is to introduce a *stale* state for a line, i.e, a line whose contents are inconsistent with a more recently line at the same physical address either in memory (because of DMA) or another cache line (synonym). Thus a line can be in state empty (invalid), or present (clean), or dirty, or stale. State transitions for a line and its synonyms can be formally defined and the finite state machine implementation uses information in the TLB to maintain consistency among synonyms at the page level. For example this allows synonyms in the present (clean) state to be residing in the cache.

5 Cache hierarchies: Multi-level inclusion properties

Consider an L2 (parent blocks) shared by one or more L1's (children blocks; on one or more processors). If L2 is much larger than L1, e.g., L2 off-chip and L1 on-chip, it is advantageous to have L2 be a superset of the L1's. If the L1's are write-back caches, the superset property means that there is room in L2 to write-back the contents of dirty blocks in L1 without replacing an L2 block that covers an L1 block. The main reason is that L2 will shield the L1's from interferences due to I/O and cache coherence in multiprocessor systems. This *multi-level inclusion* (MLI) property imposes some restrictions on the design of the hierarchy. Note that if L2 has about the same capacity as L1, we want to have the contents of the two caches be exclusive.

In order to insure MLI we need a replacement algorithm that takes into account the number of valid children of a given parent. When a parent has no valid children, then it can be replaced. In the case of set-associative caches, under the replacement algorithm described above, the general result is that the degree of associativity of the parent cache must be at least as large as the product of the number of its children, their set associativity, and the ratio of block sizes. This has serious implications in multiprocessors where L2 can be shared by several L1's since the degree of associativity might become unduly large. A remedy to this problem is to use *partial inclusion* whereby the L2 cache controller can invalidate some of the children blocks in L1.