COMS4040A & COMS7045A: High Performance Computing & Scientific Data Management Modern Processors

Hairong Wang

School of Computer Science & Applied Mathematics
University of the Witwatersrand, Johannesburg

Contents

Outline

Outline

General purpose cache-based microprocessor architecture

- Microprocessors implement the stored-program digital computer concept —
  - Instructions are stored as data in memory;
  - Instructions are read and executed by a control unit;
  - A separate arithmetic/logic unit is responsible for the actual computations
- A limiting factor — the speed of memory interface poses a limitation on compute performance.
- The architecture is inherently sequential – processing a single instruction with possibly a single operand or a group of operands from memory. The term SISD has been used for this concept.

Cache-based microprocessor

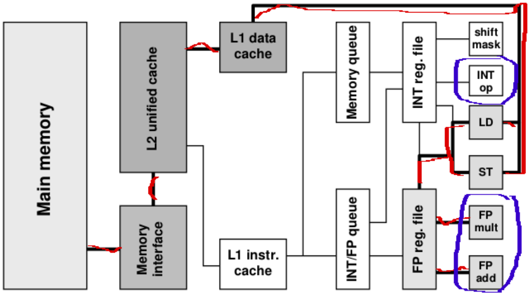
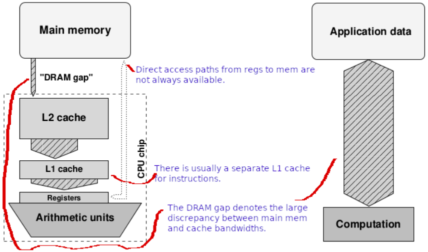


Figure: Simplified block diagram of a typical cache-based single-core microprocessor.

- Take note of the function blocks and data paths that are most relevant to performance issues in scientific computing.

Cache-based microprocessor cont.



Main memory
"DRAM gap"
L2 cache
L1 cache
Registers
Arithmetic units
CPU chip
Direct access paths from regs to mem are not always available.
There is usually a separate L1 cache for instructions.
The DRAM gap denotes the large discrepancy between main mem and cache bandwidths.
Application data
Computation

Figure: Data-centric view of cache-based microprocessor

Cache-based microprocessor cont.

- Basic metrics that can quantify the speed of a CPU: GFlops/sec and GBytes/sec metrics usually suffice for explaining most relevant performance features of microprocessors.
- Floating point operations per second (Flops/sec): The performance at which the FP units generate results for multiply and add operations.
- The most important data paths from the programmer's point of view are those to and from the caches and main memory. The performance, or bandwidths of those paths is quantified in GBytes/sec.

Advanced techniques for microprocessor design

- Increasing chip transistor counts and clock speed; and
- A multitude of concepts have been developed, including
    - **Pipelined functional units**
    - **Superscalar architecture**
    - **Data parallelism through SIMD instructions**
    - Out-of-order execution
    - Larger caches
    - Simplified instruction set

Pipelining

The idea of pipelining is illustrated in the following diagram for floating point multiplication:

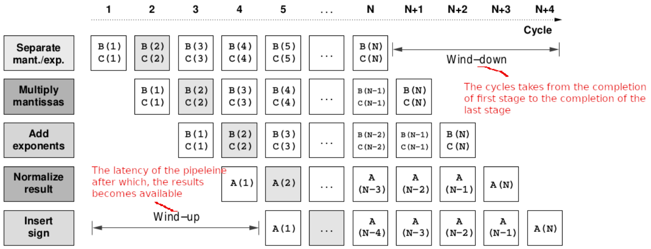


Figure: As an example, the multiplication operation is divided into five subtasks:

Pipelining cont.

- 1st cycle: separation of mantissa and exponent – 4 functional units are idle;
- 2nd cycle: multiply the mantissas, the first stage started working on `B[2]` and `C[2]` – 3 functional units are idle;
- after 4th cycle, the pipeline has finished its wind-up phase – from then on, all functional units are busy;
- after 5 cycles, the results become available – after which one result will be generated per cycle;
- when the 1st stage finished working on the last element, the wind-down phase starts.
- After 4 cycles, the loop is finished, and all results have been produced.

Pipelining cont.

- In general, for a pipeline of depth m, executing N independent operations takes $N + m - 1$ steps.
- The expected speedup over a non-pipelined operations

$$\frac{T_{seq}}{T_{pipe}} = \frac{mN}{N + m - 1}. \tag{1}$$

Pipelining cont.

Example 1

```
for(i = 0; i<N; i++)
  A[i] = s*A[i];
```

```
loop:
  load A[i]
  mult A[i]=A[i] * s
  store A[i]
  i = i+1
  branch -> loop
```

Assume it takes 1 cycle to issue an instruction; `load`: 4 cycles; `mult`: 2 cycles; `store`: 2 cycles; and the remaining comes for free. Then the above code execution could be inefficient.
How to optimize the code?

Superscalarity

Superscalarity: a special form of parallel execution, and a variant of instruction-level parallelism (ILP). The goal of superscalarity is to execute more than one instruction per cycle. It is reflected in many design details:

- multiple instructions can be fetched and decoded concurrently
- address and other integer calculations are performed in multiple integer units
- multiple floating point pipelines can run in parallel
- caches are fast enough to sustain more than one load or store operation per cycle.

Out-of-order execution and simplified instruction set

- Instructions that appear later in the instruction stream, and have their operands available can be executed before those that appear before them and still waiting for their operands.
- Compilers use reorder buffers that stores instructions until they are ready for execution.
- Simplified instruction set refers to the general move from the CISC (complex instruction set computer) to RISC (reduced instruction set computer).
- In a CISC, a processor executes very complex, powerful instructions, requiring a large hardware effort for decoding but keeping programs small and compact.
- A RISC features a very simple instruction set that can be executed rapidly - few clock cycles per instruction; in the extreme case each instruction takes only a single cycle.

SIMD

SIMD: SIMD operations allow the concurrent execution of arithmetic operations on a ‘wide’ register that can hold, e.g., two double precision (DP) or four single precision (SP), floating point words. A single instruction can initiate multiple additions at once.

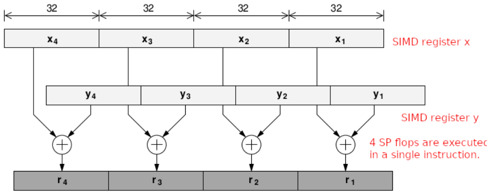


Figure: An example of SIMD.

Outline

Memory hierarchies

- Cache: low-capacity, high-speed memories that are commonly integrated on the CPU die.
- The need for cache:
  - The data transfer rates to main memory are slow compared to CPU arithmetic performance;
  - Memory latency is usually of the order of several hundred CPU cycles.
  - Cache can alleviate the DRAM gap in many cases.
- There are 2 levels of caches usually, L1 and L2 cache. L1 cache is normally split into two parts, instruction cache and data cache.
- L2 cache is unified, storing both instructions and data

Cache cont.

- The closer a cache is to CPU's registers, the higher its bandwidth and the lower its latency.
- Whenever the CPU issues a read request for transferring a data item to a register, first level cache logic checks whether this item already resides in cache. If it does, it is called a **cache hit** and request can be met immediately, with low latency.
- In a case of **cache miss**, data must be fetched from outer level cache levels or, in the worst case, from main memory.
- I-cache misses are rare events compared to D-cache misses.
- Caches do not help to increase performance always, they can only have positive effect on performance if the data access pattern of an application shows some locality of reference.
- **Temporal locality**: data items that have been loaded into a cache are to be used again soon enough to not have been evicted in the meantime; **Spatial locality**: the probability of successive accesses to neighboring items is high.

WITS UNIVERSITY

Cache cont.

- Example of a simple model. Assume T_m is the access time to main memory; cache access time is reduced to $T_c = T_m/\tau$; and β is the cache reuse ratio. Then the average access time will be $T_{av} = \beta T_c + (1-\beta) T_m$, and the performance gain can be calculated as

$$G(\tau, \beta) = \frac{T_m}{T_{av}} = \frac{\tau T_c}{\beta T_c + (1-\beta)\tau T_c} = \frac{\tau}{\beta + \tau(1-\beta)}. \qquad (2)$$

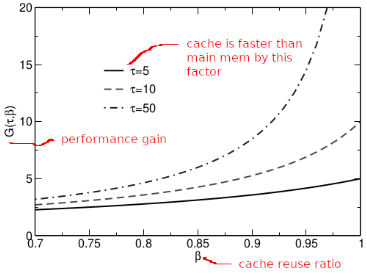


Figure: A simple model of the cache effect on perfromance.

Cache cont.

- The content of a cache is organized as cache lines. (A cache line has space for multiple data items.)
- Data transfers between main memory and caches happen on the cache line level.
- In this way, the penalty occurs only on the first miss on an item belonging to a line. The line is fetched from memory as a whole; neighbouring items can then be loaded from cache with much lower latency, increasing the cache hit ratio γ.
- So, if a code has good spatial locality, the latency problem can be significantly reduced.
- The downside of this is that erratic data access patterns are not supported.

Cache mapping

- Fully associative: there is no restriction on which cache line can be associated with which memory locations.
  - Hard to build large, fast, fully associative caches because of large bookkeeping overhead – for each cache line, the cache logic must store its location in the CPU's address space, and each memory access must be checked against the list of all those addresses.
  - Cache replacement issue: Which cache line to replace next if the caches are full: Least recently used (LRU) – the oldest items are evicted first;

Cache mapping cont.

- Directly-mapped cache: Memory locations that lie a multiple of the cache size apart are always mapped to the same cache line, and the cache line that corresponds to some address can be obtained very quickly by masking out the most significant bits.
- Downside: Cache thrashing: Cache lines are loaded into and evicted from the cache in rapid succession.
- For example, thrashing happens when an application uses many memory locations that get mapped to the same cache line. A simple example would be a 'strided' vector triad code for DP data.

  ```
  for (i=0; i<N; i+= CACHE_SIZE_IN_BYTES/8)
      A[i] = B[i]+C[i];
  ```

  By using the cache size in units of DP words as a stride, successive loop iterations hit the same cache line so that every memory access generates a cache miss, even though a whole line is loaded every time. In principle there is plenty of room left in the cache, so this kind of situation is called a conflict miss.

Cache mapping cont.

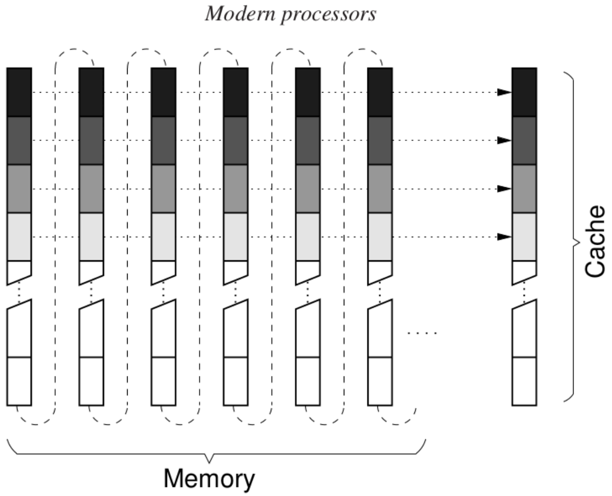


Figure: In a direct-mapped cache, memory locations which lie a multiple of the cache size apart are mapped to the same cache line.

Cache mapping cont.

- Set-associative cache: To keep administrative overhead low and still reduce the danger of conflict misses and cache thrashing, set-associative cache is divided into *m* (*m*-way) direct-mapped caches equal in size. *m* is the number of different cache lines a memory address can be mapped to.
- Nowadays, set-associativity varies between two- and 48-way.

Cache mapping cont.

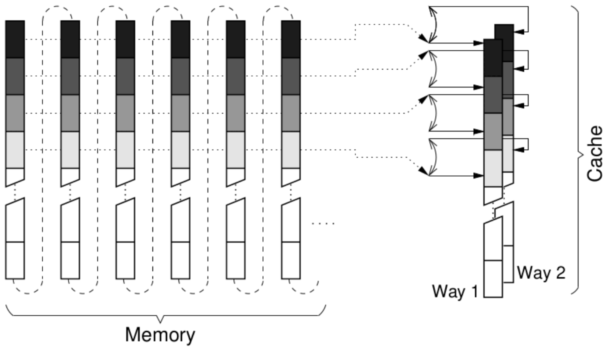


Figure: A two-way ($m = 2$) set-associative cache, memory locations which are located a multiple of $1/2$th the cache size apart can be mapped to either of m (=2) cache lines.

Prefetch

- Cache can improve the performance in the case of spatial (or temporal) locality. But there is still the problem of latency on the first miss. See the figure below.
- Making cache line longer slows down erratic data access patterns more. Cache line length is typically between 64 and 128 bytes (8 - 16 DP words).
- Prefetch helps to address latency problem.

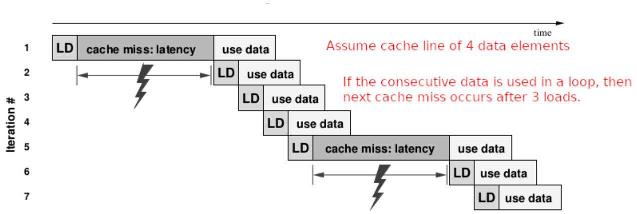
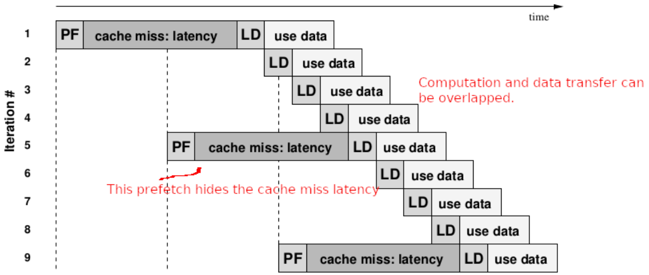

time
Assume cache line of 4 data elements
If the consecutive data is used in a loop, then next cache miss occurs after 3 loads.
Iteration #
1
2
3
4
5
6
7
LD
cache miss: latency
use data

Figure: The penalty of cache misses for a vector norm loop

Prefetch cont.

- Prefetching supplies the cache with data ahead of the actual requirements of an application.
- Prefetch can be implemented by compiler, or hardware prefetcher. In any case, it requires additional resources to be implemented.



time
Iteration #
1
2
3
4
5
6
7
8
9
PF
cache miss: latency
LD
use data
Computation and data transfer can be overlapped.
This prefetch hides the cache miss latency

Figure:

Prefetch cont.

- The memory subsystem must be able to sustain a certain number of outstanding prefetch operations.
- We can estimate the number of outstanding prefetches required for hiding the latency.
- Let T_ℓ is the latency; B the bandwidth, then the transfer time a cache line of length L_c (Bytes) takes $T = T_\ell + \frac{L_c}{B}$. One prefetch must be initiated per cache line transfer, and the number of cache lines that can be transferred during time T is the number of prefetches P that the processor must be able to sustain:

$$P = \frac{T}{L_c/B} = 1 + \frac{T_\ell}{L_c/B}. \tag{3}$$

- For example, $L_c = 128$B (bytes), $B = 10$GB/sec, and $T_\ell = 50$ns, then we get $P \approx 5$ outstanding prefetches. If this requirement is not met, the latency cannot be hidden completely.

Outline

Multicore processors

- Multicore design comes in as a solution to the power-performance dilemma.
- The technology behind microprocessor is based on reducing the clock frequency of a single core, and placing more than one CPU core on the same die while keeping the same power envelope as before.

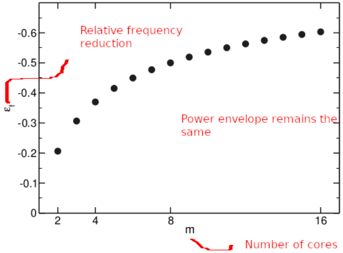


Figure: Required relative frequecny reduction to stay within a given power envelope on a given process technology, vs number of cores on a multicore chip.

Multicore processors cont.

- Due to the multicore transition from single core, it is necessary to put those resources to efficient use by parallel programming.
- An apparent challenge resulting from multicore system is that the gradual reduction in main memory bandwidth and cache size available per core.
- Programming techniques for traffic reduction and efficient bandwidth utilization are hence important.

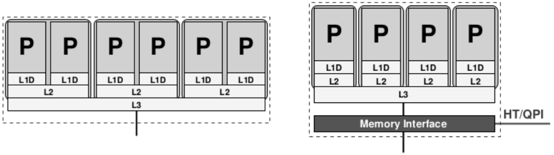


Figure: Hexa-core processor chip (Intel) and quad-core processor chip (AMD and Intel). HyperTransfer and QuickPath, a buil-in memory interface allows to attach memory and other sockets without a chipset.

Multicore processors cont.

- Intersocket networks: HyperTransfer or QuickPath – reduces main memory latency.
- The cores on one die can either have separate caches or share certain levels.
- Sharing a cache enables communication between cores without reverting to main memory, reducing latency and improving bandwidth by about an order of magnitude. A drawback is possible cache bandwidth bottlenecks.

Summary

- Various features of a single-core processor architectural design that are closely relevant to performance properties of programs, be it serial or parallel.
- Understand benefits and drawbacks of these processor design features, the predominant multicore feature and trend of computers nowadays, the issues come with such trends that include utilizing the resources efficiently through programming.

References

- The contents in this lecture are based on book [1].

[1] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., USA, 1st edition, 2010