Performance of the Cache and Memory Hierarchy

June, 2014

Presenter:
Adrian Loteanu
adrian.loteanu@intel.com

Disclaimer

- The views expressed in this tutorial are those of the people delivering the tutorial.
- We are not speaking for our employers.
- We are not speaking for any third party.

Legal Notice

- Intel and the Intel logo, are trademarks of Intel Corporation in the U.S. and/or other countries.

- Other names and brands may be claimed as the property of others.

- THE MATERIALS ARE PROVIDES “AS IS”, WITHOUT WARRANTY O ANY KIND, EXPRESS OR IMPLIED, INCLUDED BUT NOT LIMITED TO THE WARRANTIES OF MECHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN ACTION OF CONTRACT, TORT, OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE MATERIALS OR THE USE OR OTHER DEALINGS IN THE MATERIALS

- Creative Commons License: Attribution-NonCommercial-ShareAlike 4.0 International
  http://creativecommons.org/licenses/by-nc-sa/4.0/

Disclaimer

Some numbers have been made up for educational purposes.

Information in this presentation, especially regarding specific numbers, is merely made up for educational/learning purposes and should not be used for any other purpose. There is no guarantee that any information contained in the presentation is correct or accurate.

Table of Contents

Why do we need caches?

- Main memory has big latency compared to cpu clock times.
- Modern multi-core processors can require in excess of **500GB/sec** of bandwidth.
- Main memory (RAM) on the same system would have a peak bandwidth of **~30GB/sec.**
- The solution: Add a memory hierarchy of increasingly faster and smaller caches that hide the latency to main memory.

How do caches work?

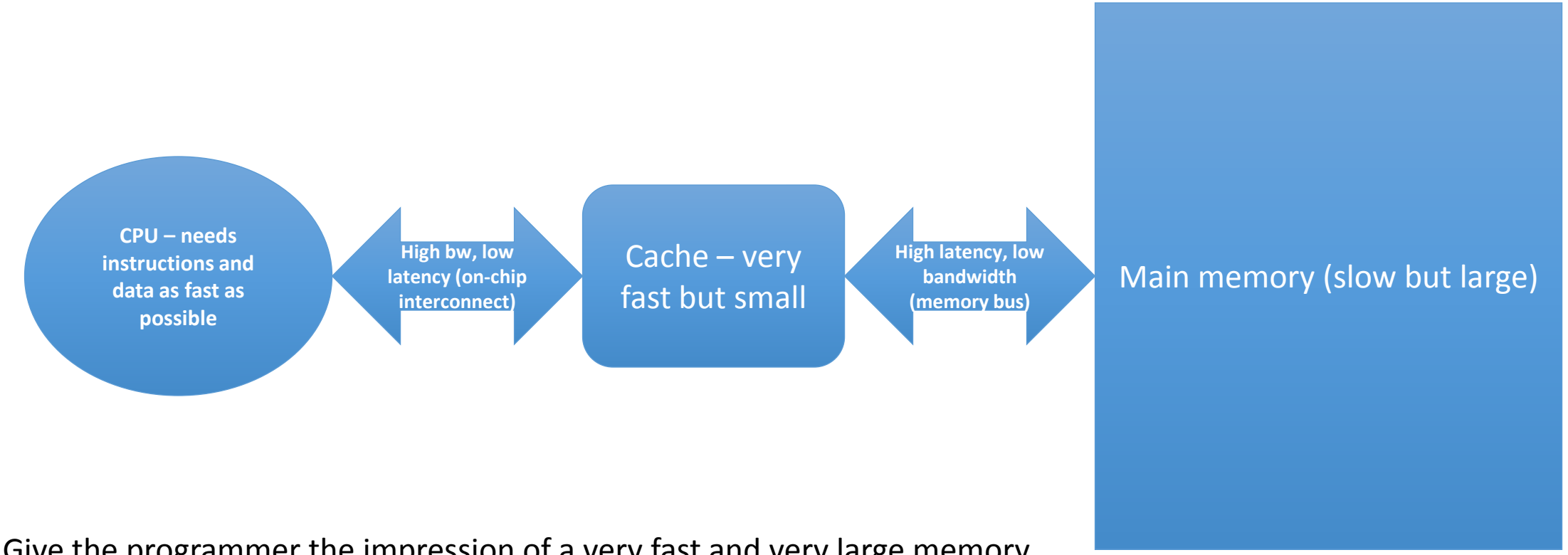
- Ideally a programmer would want a very large and very fast memory that is also cheap and low power.

- DRAM is large, cheap and low power and SRAM (caches) are high performance but consume more power and are more expensive in terms of $/byte.

How do caches work?

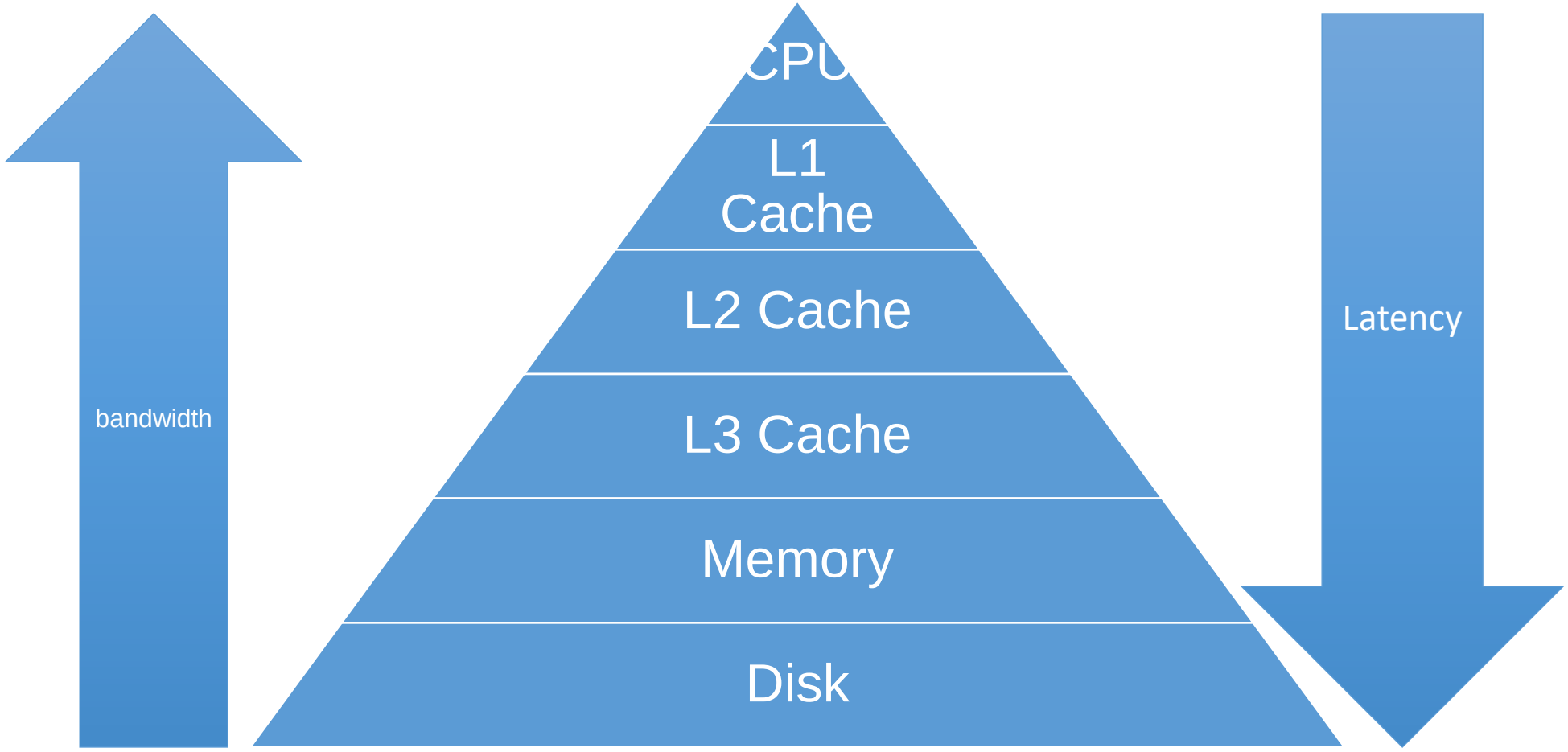
- **Spatial locality** – if the software accesses an area of memory it is likely to access adjacent areas in the near future.
- **Temporal locality** – if the software accesses an area of memory it is likely to need to access the same area in the near future.
- So moving these areas to a faster memory will speed up future accesses to that area.
- Give the programmer an impression of a very large very fast memory.

How do caches work?



Give the programmer the impression of a very fast and very large memory by providing as much data as possible from the cache.

How do caches work?



Fictional Example of Memory Access Times

Memory level	Access time scaled to human timescale	Event of comparable duration
CPU Registers	1 sec	Reading the next word from a sentence in a newspaper.
L1 $	4 sec	Finding an article on the same page of the newspaper.
L2 $	12 sec	Finding an article on another page.
L3 $	24 sec	Finding an article in a newspaper that is in another room.
DRAM	2 m 13 sec	Commercial break.
Spinning Disk I/O	~1 year 2 months	Getting half of a Master's degree.

Numbers are not measured in any way, they are fictional examples and merely for educational purposes.

Important terms

- **Cache HIT** – If you find the data you were looking for in your cache, this is called a HIT.
- **Cache MISS** – Similarly, if the data you were looking for is not in your cache this is called a MISS. If you miss you have to go looking at a lower cache level and ultimately in RAM.
- **HIT Rate** – The percentage of cache accesses that result in HITs.
- **MISS Rate** – The percentage of cache accesses that result in MISSes.
- **HIT Time** – The time it takes to access a piece of data if you find it in the cache.
- **MISS penalty** – The time it takes to look for the data at lower cache levels/in memory if you do not find it in the cache.
- **MPI (Misses Per Instruction)** – The number of cache accesses that result in MISSes divided by the total number of instructions executed.

How do caches work?

- How does cache reduce your latency?
  (Assuming no out of order processing and blocking misses)

Average memory access time = Hit_timeL1 + Miss_rateL1 x Miss_penaltyL1

Miss_penaltyL1 = Hit_timeL2 + Miss_rateL2 x Miss_penaltyL2

Miss_penaltyL2 = Hit_timeL3 + Miss_rateL3 x Miss_penaltyL3

Miss_penaltyL3 = average DRAM access time

Causes of cache misses

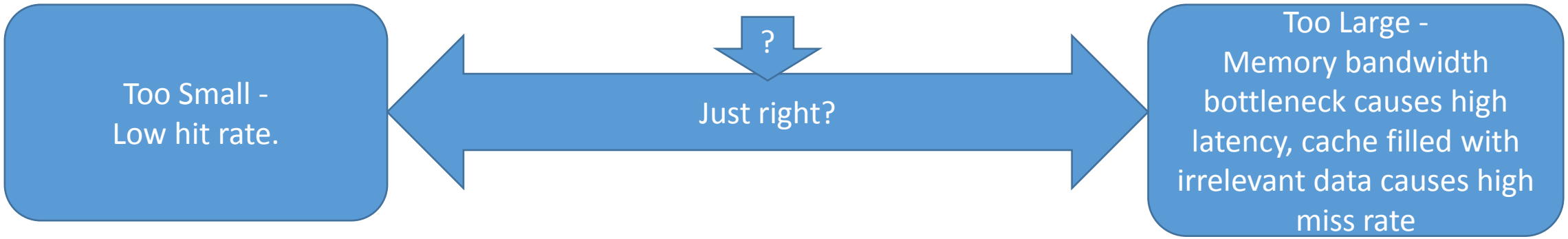
- **Compulsory** – The first access to a block cannot result in a hit. These misses will occur even with infinite cache.
- **Capacity** – If the cache is not large enough to contain all blocks needed during execution, capacity misses will occur because some blocks have been evicted.
- **Conflict** – If the block placement policy is not fully associative, conflict misses will occur.
- **Coherency** – Misses due to cache flushes to keep caches coherent in multiprocessor environments.

Designing a cache – a game of tradeoffs

- There are a lot of tradeoffs involved in engineering, especially when it comes to performance.
- What speeds up something may cause a bottleneck elsewhere.
- Choosing the perfect balance involves a lot of measuring, research and some intuition. There may be more than 1 similarly good solutions.
- The perfect balance for one software application may not be so good for another – general purpose computers have to run everything well while special purpose ones may only run one thing very well.

Tradeoff #1 - How much data to bring into the cache at once? – Cache line.

- If the cache line is too small, you may get more misses. Bringing in larger cache lines means you bring in more bytes and are more likely to have data you will need in your cache on farther accesses – spatial locality.
- If the cache line is too big, you generate a lot of memory traffic and may increase hit time. Beyond a certain size you will also increase your miss rate because you have fewer cache lines for a cache of a given size.



Tradeoff #2 - So, where do we put in and get data from the cache?

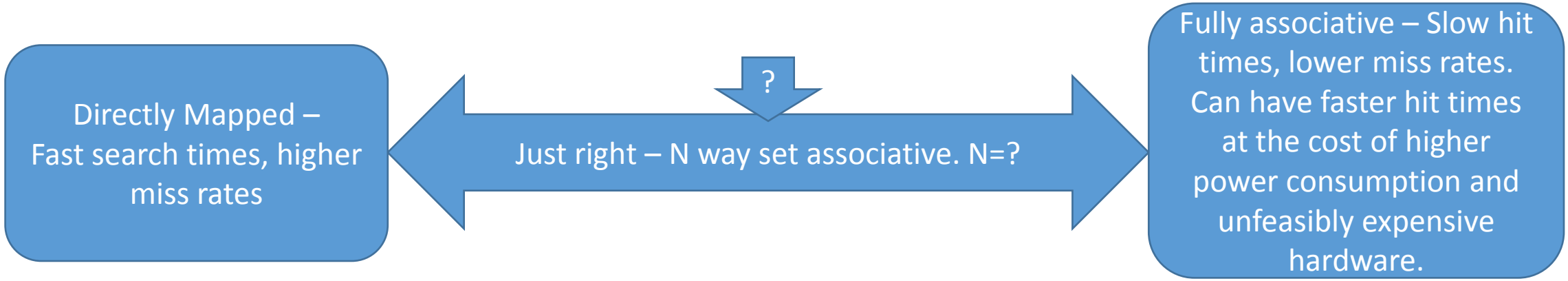
- Directly mapped – a every memory address can be placed in only 1 cache location.
- Pro: You can find if your data is in the cache very fast (fast hit time) because it can only be in 1 place.
- Con: You can get misses because of conflicts – a piece of data removes another that you may still need because it is only mapped to that area.

Tradeoff #2 - So, where do we put in and get data from the cache?

- Associative – data can be placed anywhere in the cache.
- PRO: You do not have conflict misses as you can put your incoming data anywhere.
- CON: It takes a long time OR very complex hardware to search the data - search can be done in parallel with higher power consumption and higher manufacturing cost because of extra circuits. This is yet another tradeoff.

Tradeoff #2 - So, where do we put in and get data from the cache?

- (N Way) Set Associative – tradeoff between directly mapped and associative. Data can only go into a certain set but can be placed anywhere within the set.
- You have a low number of conflict misses as you can put your incoming data anywhere within a set.
- It is feasible to do parallel data searching within that set.



Tradeoff #3 - Why multilevel caches?

- The larger the cache, the more time it takes to find something in it.
- However, the smaller it is, the fewer things you can put in it so the higher the miss rate.
- So you want to balance hit time and miss rate to get the best average access time possible.

Tradeoff #4 – Exclusive vs Inclusive?

- On a multilevel hierarchy, is the data in a level of cache contained in the level below it? Example: is the data from L1 also duplicated in L2 and L3 caches?

- Yes – Inclusive cache policy.
- No – Exclusive.

- An Exclusive policy eliminates redundant duplication of data and thus an Exclusive hierarchy can hold more data in it.
- Can you think of an advantage of the Inclusive policy? Hint: Think Multi-cores & coherency.

Tradeoff #5 - Private or shared cache?

- **Private cache** – only a single core has access to this cache, it is “private” to that core.
- **Shared cache** – More cores share the same piece of cache
- Shared caches allow more flexibility – if only a single thread is running it has access to the whole cache and if more are running they share it.
- A private cache can be generally made faster but it adds the overhead of having to synchronize the data from each core to ensure coherency.

Cache coherency

- When multiple cores have individual caches you need to make sure that any modification they make to local data is known by the other cores in the system.
- The same happens between L3 Caches on multi processor systems.
- The MESIF protocol ensures this coherency. Each cache line can be in one of the states: Modified, Exclusive, Shared, Invalid, Forward. In order to write to a cache line you need to have it in the Exclusive state. You request this from other cores/processors. After writing it becomes Modified.

Virtual Memory and the TLB

- In order to give applications the impression that they have a large address space available (larger than the total amount of RAM) and to give them the impression of a contiguous memory space and also control their access permissions and protect them from each other, virtual memory was developed.
- Currently it is implemented using a system called paging. Programs see virtual addresses and the OS generates a page table in RAM which holds the correlation between real and physical addresses. Some pages may even be moved to the disk if they are not used so as to free main memory.
- Because searching the page table on each memory access would take too long, the TLB was implemented.

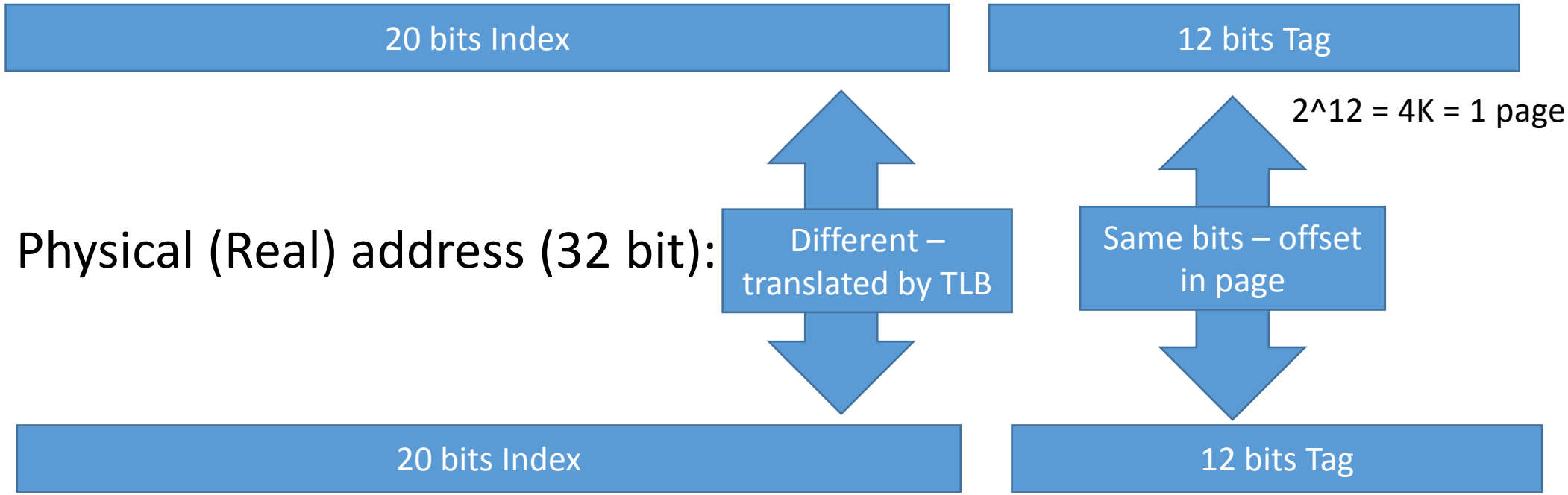
Virtual Memory and the TLB

- TLB = Translation Lookaside Buffer. It is a small piece of cache memory, which holds the most recent translations between real and physical addresses. These days it is usually implemented in multiple levels for the same reason for which caches are also on multiple levels.

Virtual Address of page	Real (Physical Address) of page
0xFF	0x1233
0xABC	0x6553
…	…

Virtual Memory and the TLB

- Virtual address example (32 bit system):



- Physical (Real) address (32 bit):

About indexing and tagging

- Virtually Indexed, Virtually Tagged (VIVT) Caches
- Physically Indexed, Physically Tagged (PIPT) Caches
- Virtually Indexed, Physically Tagged (VIPT) Caches
- Physically Indexed, Virtually Tagged Caches

Assignments

- You are hired at a new company called Slow Software Inc. One of the company's products is a very popular image manipulation tool used by professionals around the world.
- One of the features of this tool is an image rotation algorithm – it transposes an input image stored as a matrix of pixels – RGB. The software engineer that wrote that routine, Mr. Hertz Segfault didn't spend a lot of time optimizing it so, although correct, it is rather slow.

- Use any means necessary to make the image rotation algorithm work as fast as possible!
- Hint: Use performance tools to profile the cache behavior and try to improve the code locality.
- Hint: Since users upload their image onto a server before transposing it, it may not make sense to store the image on the disk.
- Hint: Compiler tricks are allowed. In fact, anything goes as long as the output is correct.

Assignment - #2

- You are given a black box application. It implements a sorting algorithm written by the competition of Slow Soft Inc so you do not have access to the code.
- You must profile the application using your given tools and try to identify what the algorithm is and what it’s bottlenecks are. You must also compare this app to Slow Soft’s own implementation of sorting and determine which is better. Answer the following questions:
- Is it CPU intensive? Is it memory intensive? Are there a lot of cache misses? Does it scale well to multiple threads? What is the routine it accesses most of the time? What is the most called system function? Which app is better and why?
- Attempt to get a trace of the memory accesses (addresses) of the application.