Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

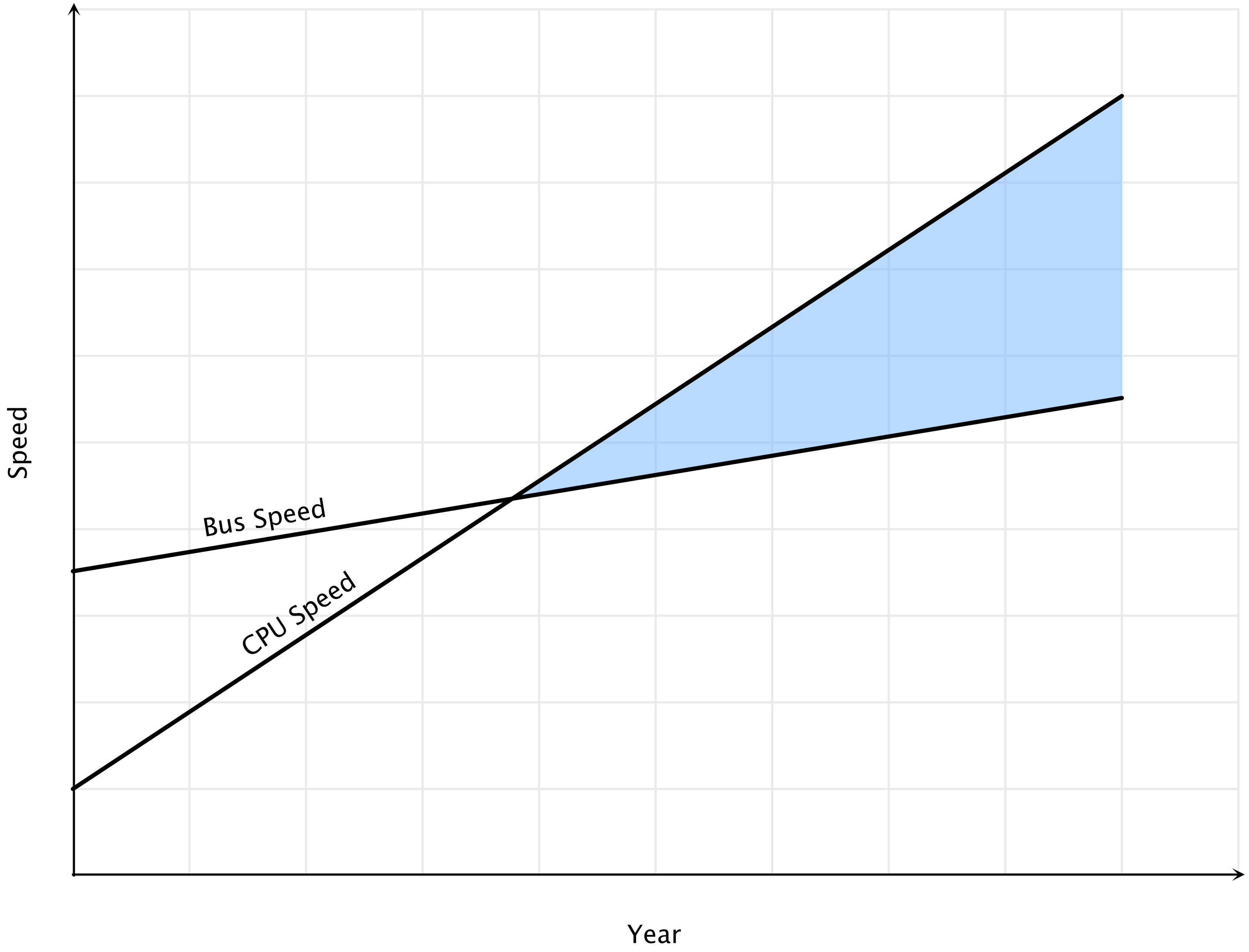
Memory Hierarchies && The New Bottleneck == Cache Conscious Data Access

Martin Grund

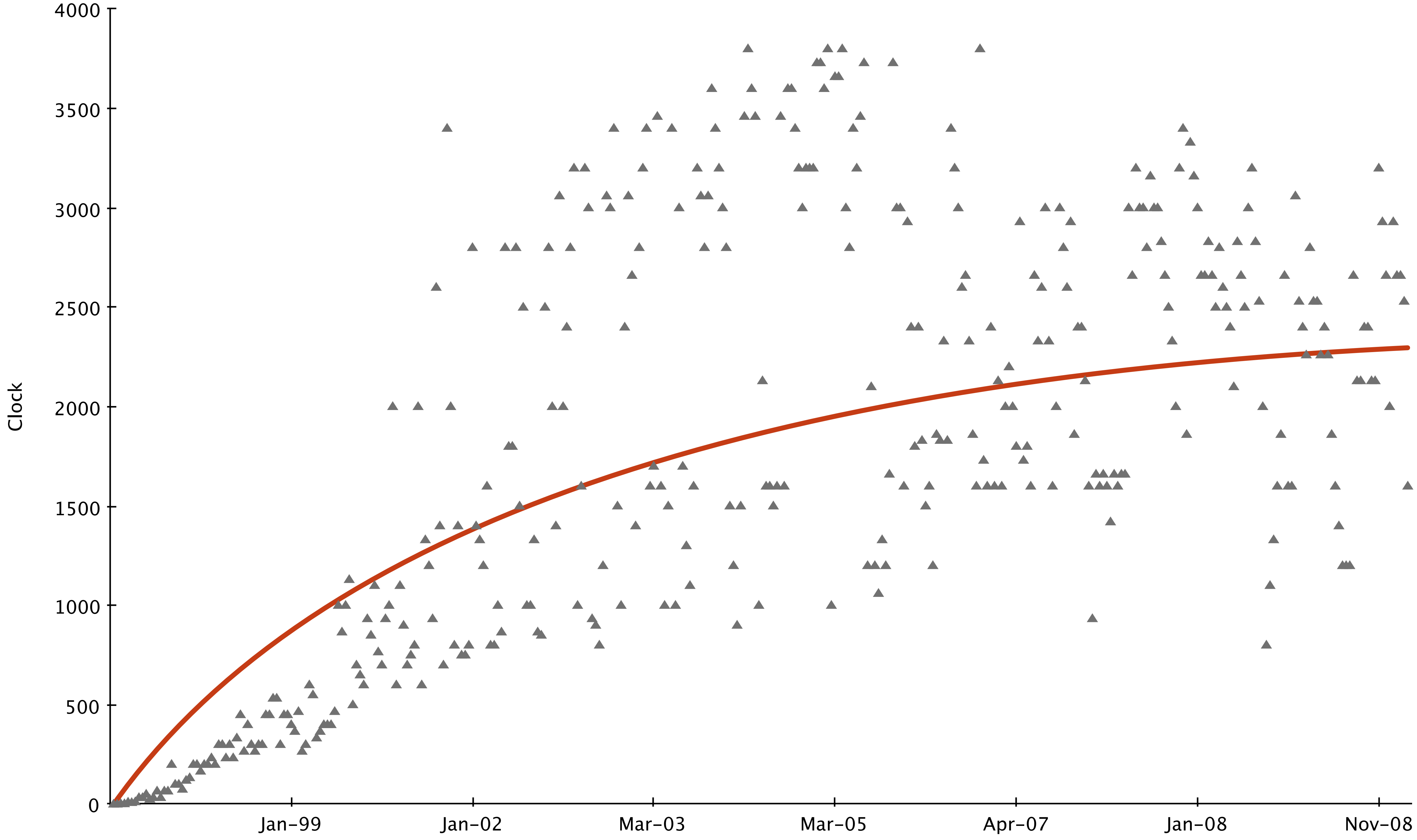
Agenda

- **Key Question:** *What is the memory hierarchy and how to exploit it?*
- **What to take home**
  - How is computer memory organized.
  - What should be considered when working with main memory.
  - How might future computer architectures look like.

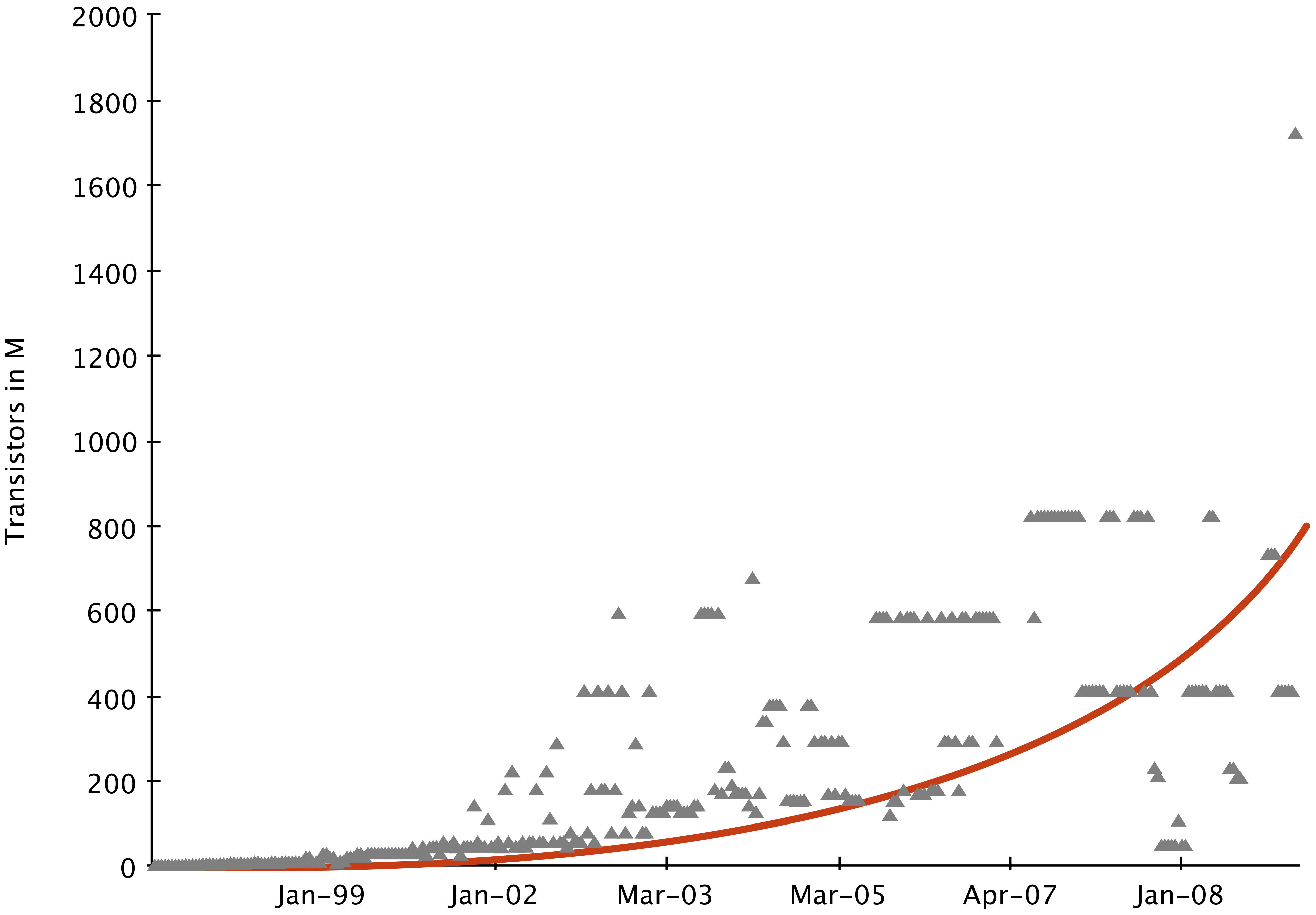
Frequency + Bandwidth



Frequency


Clock
4000
3500
3000
2500
2000
1500
1000
500
0
Jan–99
Jan–02
Mar–03
Mar–05
Apr–07
Jan–08
Nov–08

Transistors



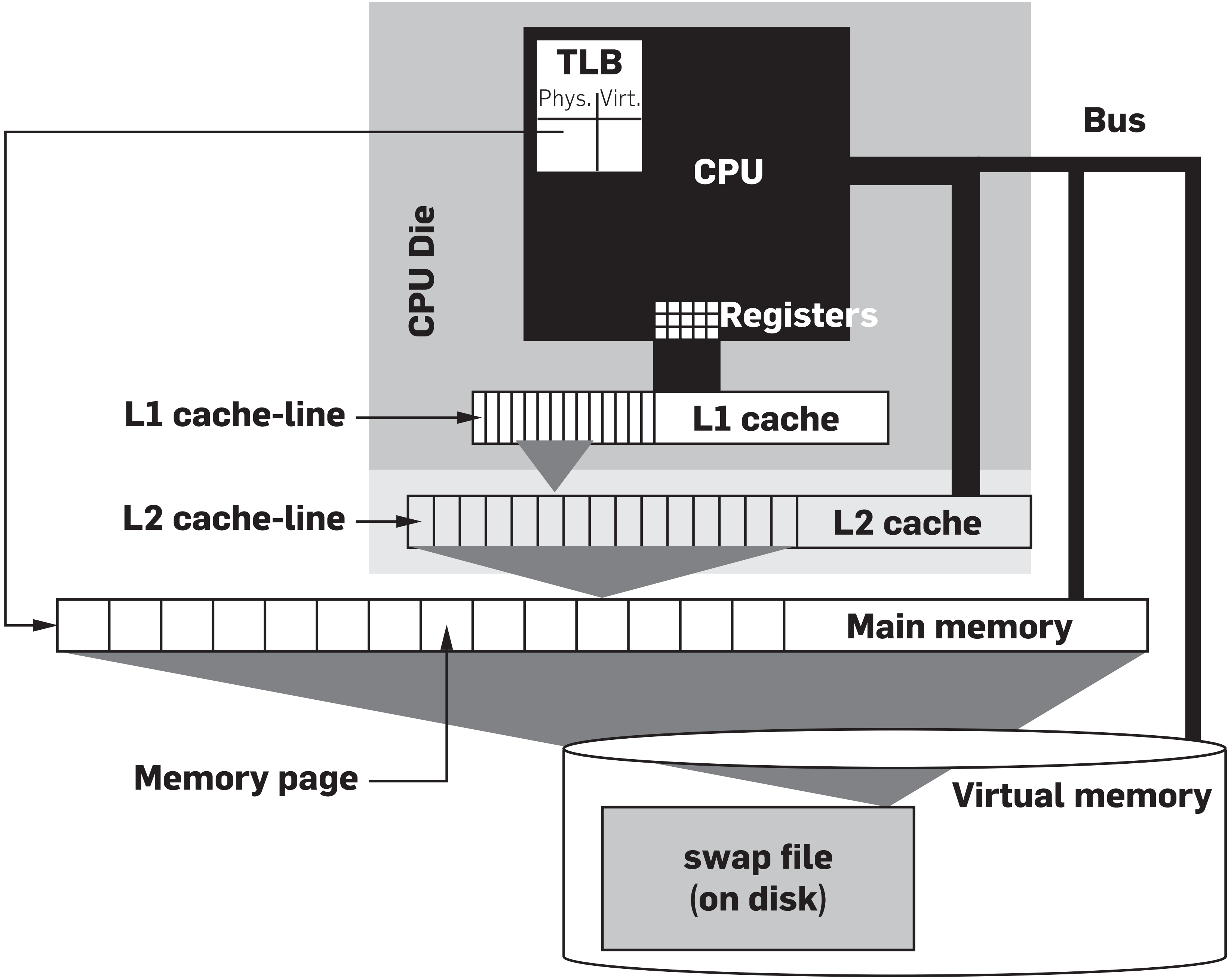
What does this mean?

What does this mean?

- Memory gets slower for the CPU
- More memory channel, stalling latency
- Frequency is stalling, while # transistors increases
- Degree of parallelism increases

Back to Main Memory

Memory Hierarchy



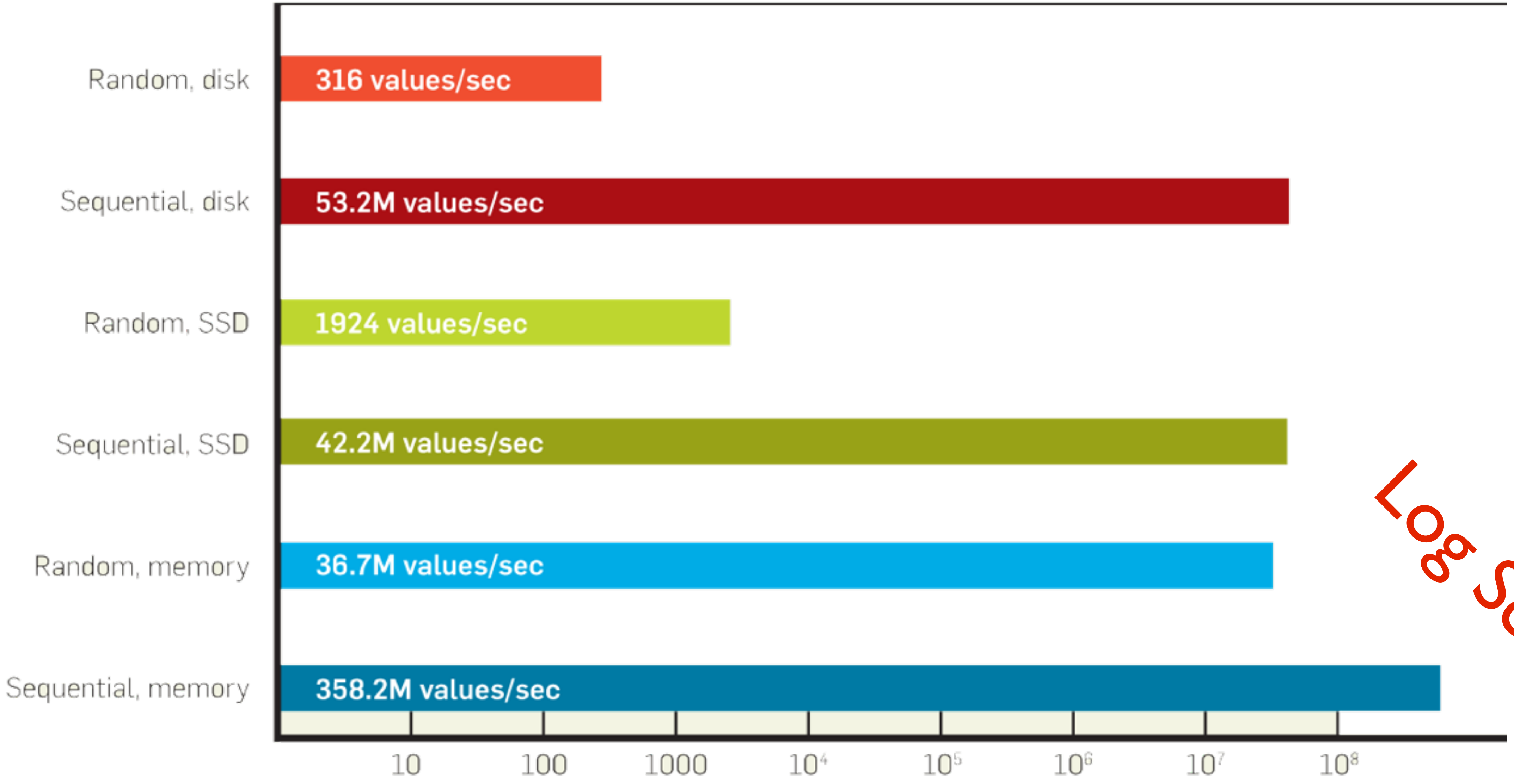
Memory Hierarchy

Why not make larger caches?

- SRAM - complicated structure, bigger, more expensive, very fast
- DRAM - simpler structure, slower, constant refresh needed. But, can be read in parallel and exploited by applying sequential access patterns.

Organize memory in hierarchies where faster memories are used as caches for slower memory.

Memory Access



Log Scale!

* Disk tests were carried out on a freshly booted machine (a Windows 2003 server with 64GB RAM and eight 15,000RPM SAS disks in RAID5 configuration) to eliminate the effect of operating-system disk caching. SSD test used a latest generation Intel high-performance SATA SSD.

A. Jacobs. The Pathologies of Big Data. Comm. of the ACM, 52(8), Aug. 2009

Cost of Memory Access

- Each memory access incurs a latency due to the organization of DRAM.
- The different caches try to hide this latency
  - Multi-level data caches
  - Instruction caches
  - Translation lookaside buffer (TLB) for virtual address translation
- Caches resemble features known from database systems (e.g. buffer pool) but are controlled by hardware.

Locality

- Caches exploit locality in programs
  - **Spatial Locality** - related data is often spatially close
  - **Temporal Locality** - programs tend to re-use data frequently

Writing I

- Processor caches are supposed to be inherent and should be completely transparent to user level code.
- Different write policies
  - **Write-through** - direct write to memory
  - **Write-back** - dirty flag per cache line, as soon as the cache line is evicted, data is written
  - **Write Combining** - combine multiple write operations per cache line and write then
  - **Uncacheable** - cache line is not stored in cache before write

Writing II

- For multi-processor systems cache coherency becomes increasingly complex
- MESI protocol (modified, exclusive, shared, invalid) - 4 states to implement write-back with concurrent read-only access
- With multiple threads, prefetching and write-back may saturate the bus very early

Cache Conscious

Cache Conscious - optimizing the program's performance by changing the organization and layout of its data with additional knowledge of the cache properties ("Cache-conscious structure definition", Chilimbi et al., ACM SIGPLAN 1999).

Cache Oblivious - optimizing the program's performance by changing the algorithms to adopt the underlying hardware properties without additional knowledge.

How to exploit this?

Allocation && Loading

Memory allocation can be crucial to program performance and it is important to understand its implementation.

- **Avoid cache line splits** - Padding
- **Avoid memory page splits** - Alignment / Padding
- **Heap contention** - multiple threads try to allocate from the same allocator with exclusive access

ftp://g.oswego.edu/pub/misc/malloc.c

Padding

Modify the data structure to match integer denominators of the size of a cache line.

```
#include <stdio.h>

typedef struct _bad
{
   unsigned first;
   unsigned b;
   unsigned c;
   char second;
} bad;

int main(int argc, char* argv)
{
   printf("%ld\n", sizeof(bad));
   return 0;
}
```

sizeof(bad) == ??

Padding

Modify the data structure to match integer denominators of the size of a cache line.

```
#include <stdio.h>

typedef struct _bad
{
   unsigned first;
   unsigned b;
   unsigned c;
   char second;
} bad;

int main(int argc, char* argv)
{
   printf("%ld\n", sizeof(bad));
   return 0;
}
```

sizeof(bad) == ??

13

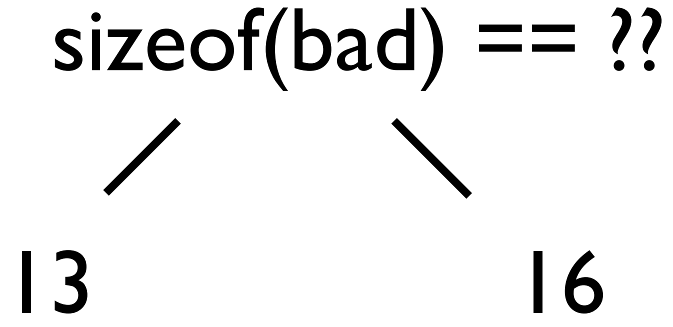
Padding

Modify the data structure to match integer denominators of the size of a cache line.

```
#include <stdio.h>

typedef struct _bad
{
   unsigned first;
   unsigned b;
   unsigned c;
   char second;
} bad;

int main(int argc, char* argv)
{
   printf("%ld\n", sizeof(bad));
   return 0;
}
```



sizeof(bad) == ??

13

16

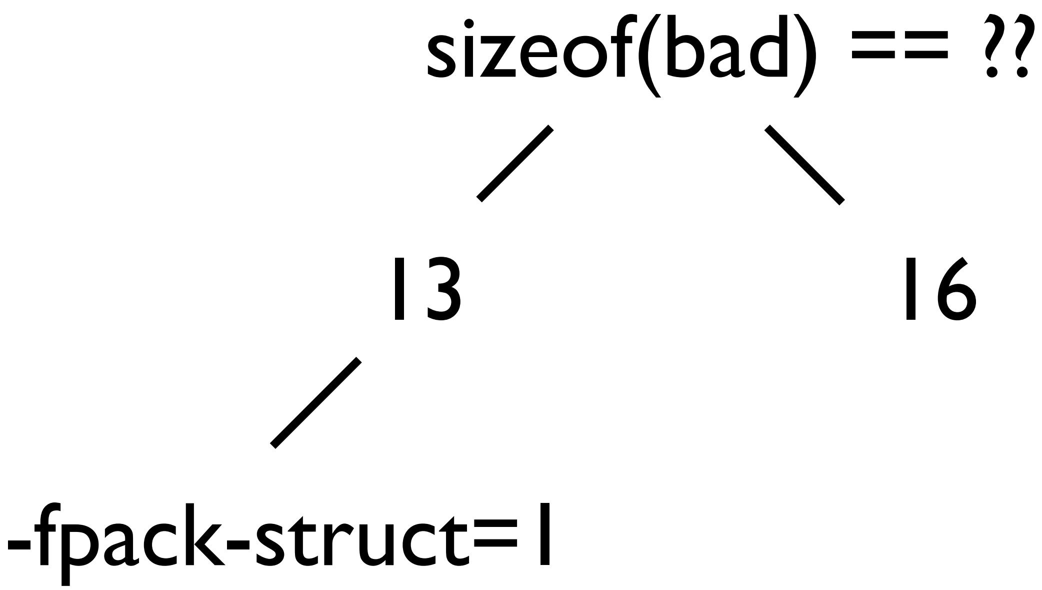
Padding

Modify the data structure to match integer denominators of the size of a cache line.

```
#include <stdio.h>

typedef struct _bad
{
   unsigned first;
   unsigned b;
   unsigned c;
   char second;
} bad;

int main(int argc, char* argv)
{
   printf("%ld\n", sizeof(bad));
   return 0;
}
```



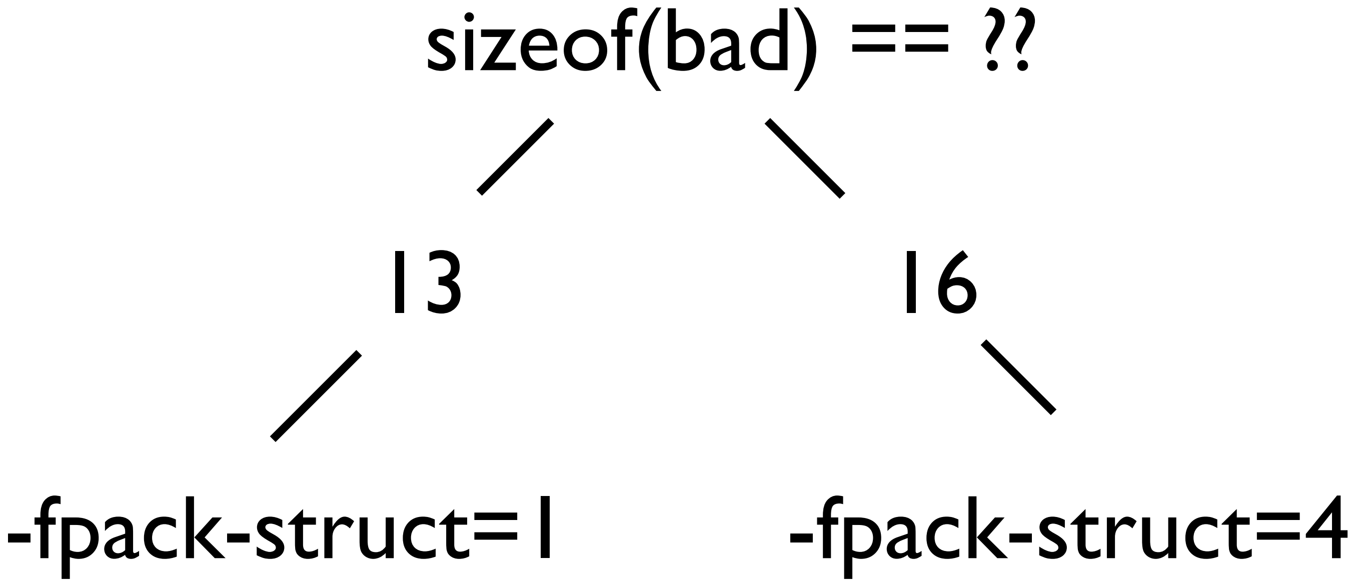
Padding

Modify the data structure to match integer denominators of the size of a cache line.

```
#include <stdio.h>

typedef struct _bad
{
   unsigned first;
   unsigned b;
   unsigned c;
   char second;
} bad;

int main(int argc, char* argv)
{
   printf("%ld\n", sizeof(bad));
   return 0;
}
```


sizeof(bad) == ??

13

16

-fpack-struct=1

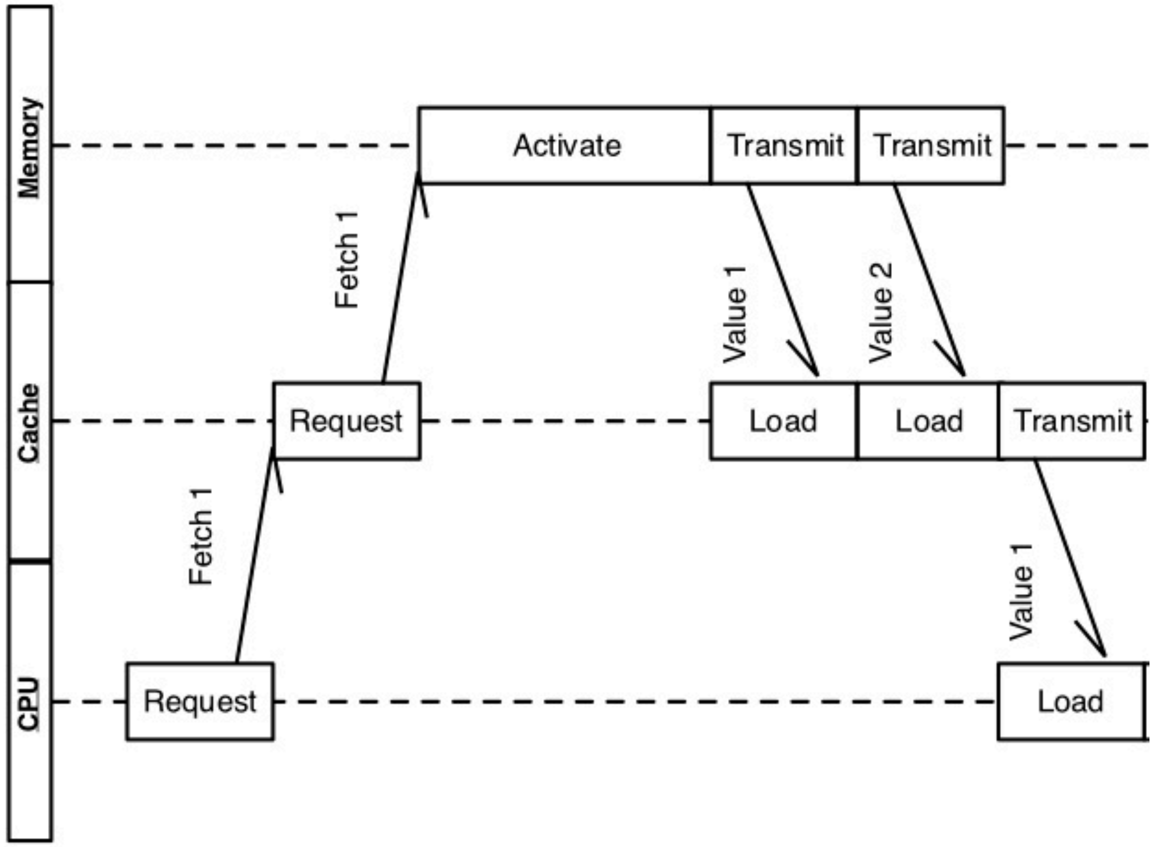
-fpack-struct=4

Prefetching

Prefetching is used to asynchronously read co-located data (see Locality).

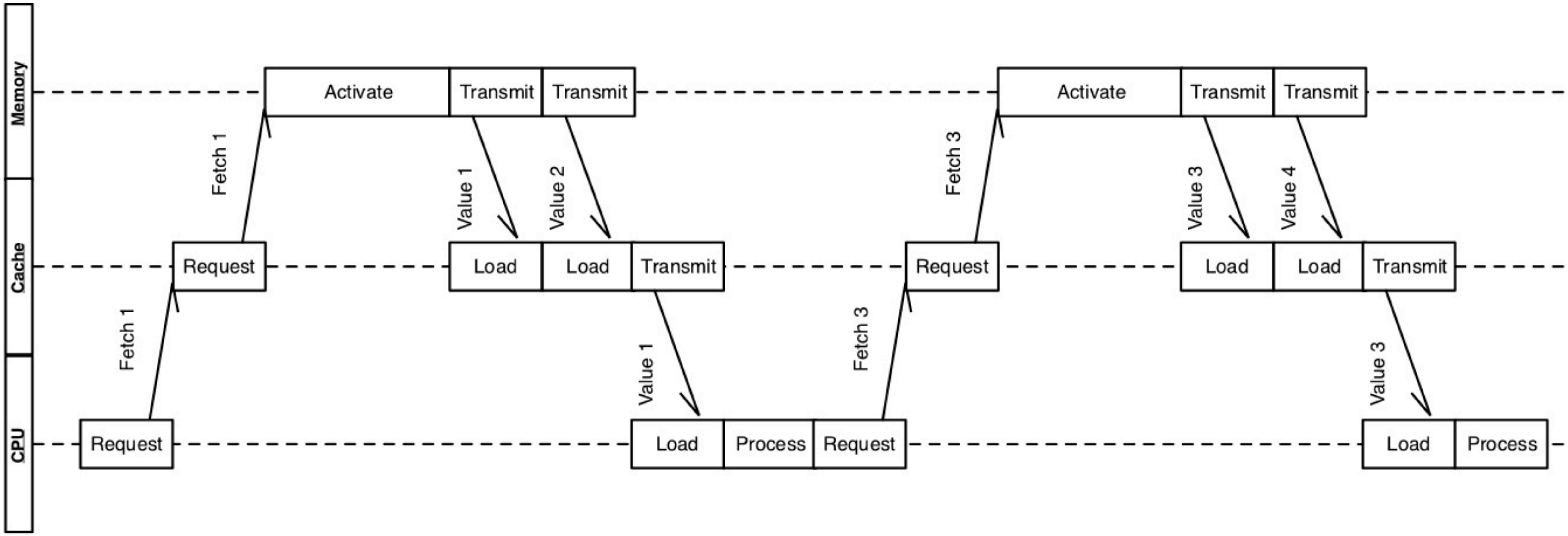
- Load adjacent cache line
- Pattern detection with stride based loading

No Prefetching



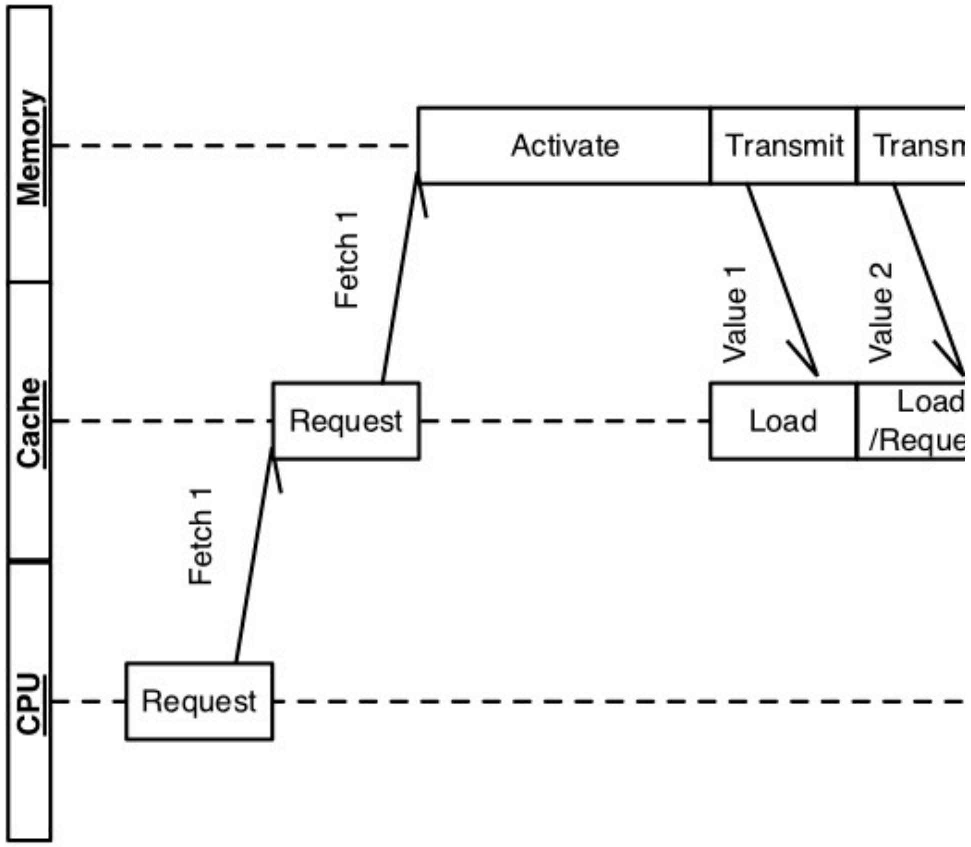
2 values per cache line

No Prefetching

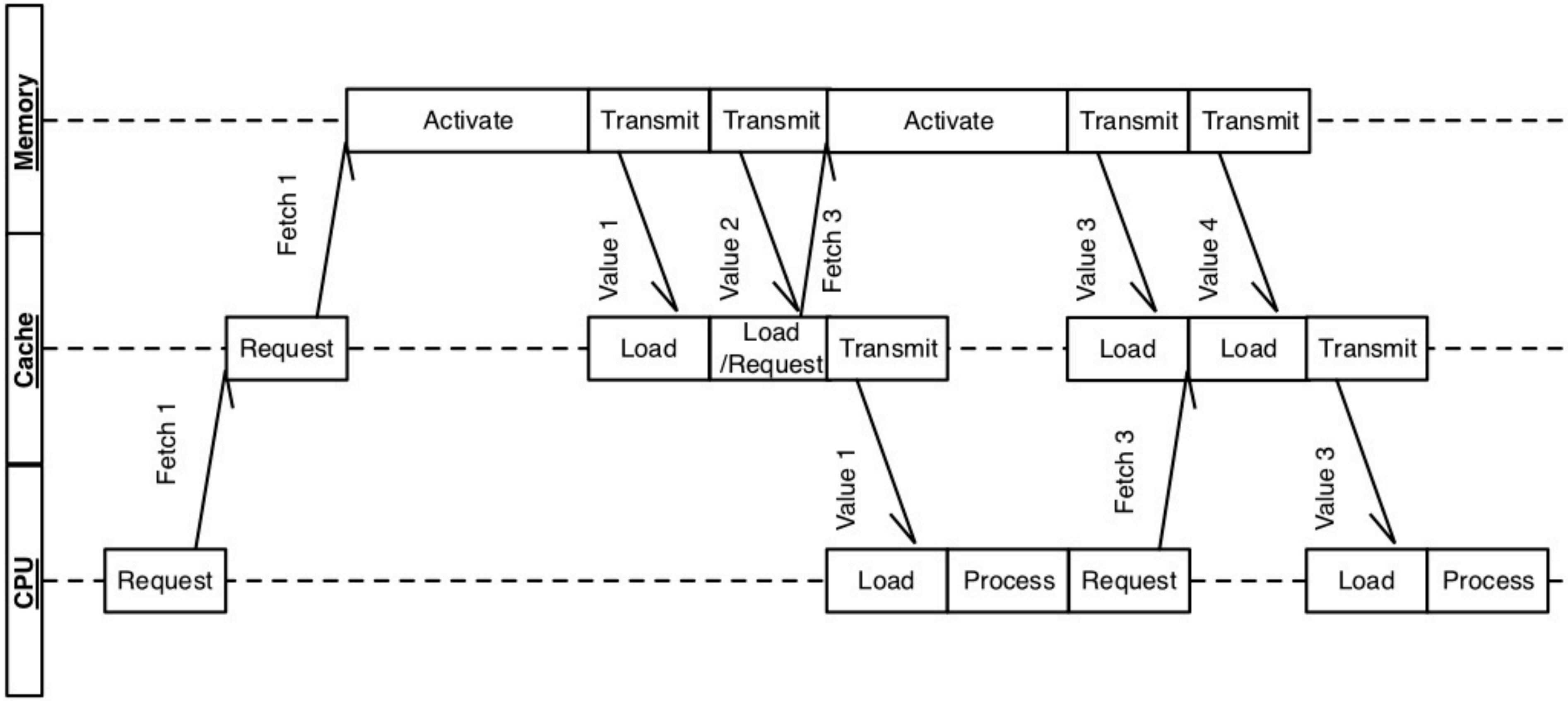


2 values per cache line

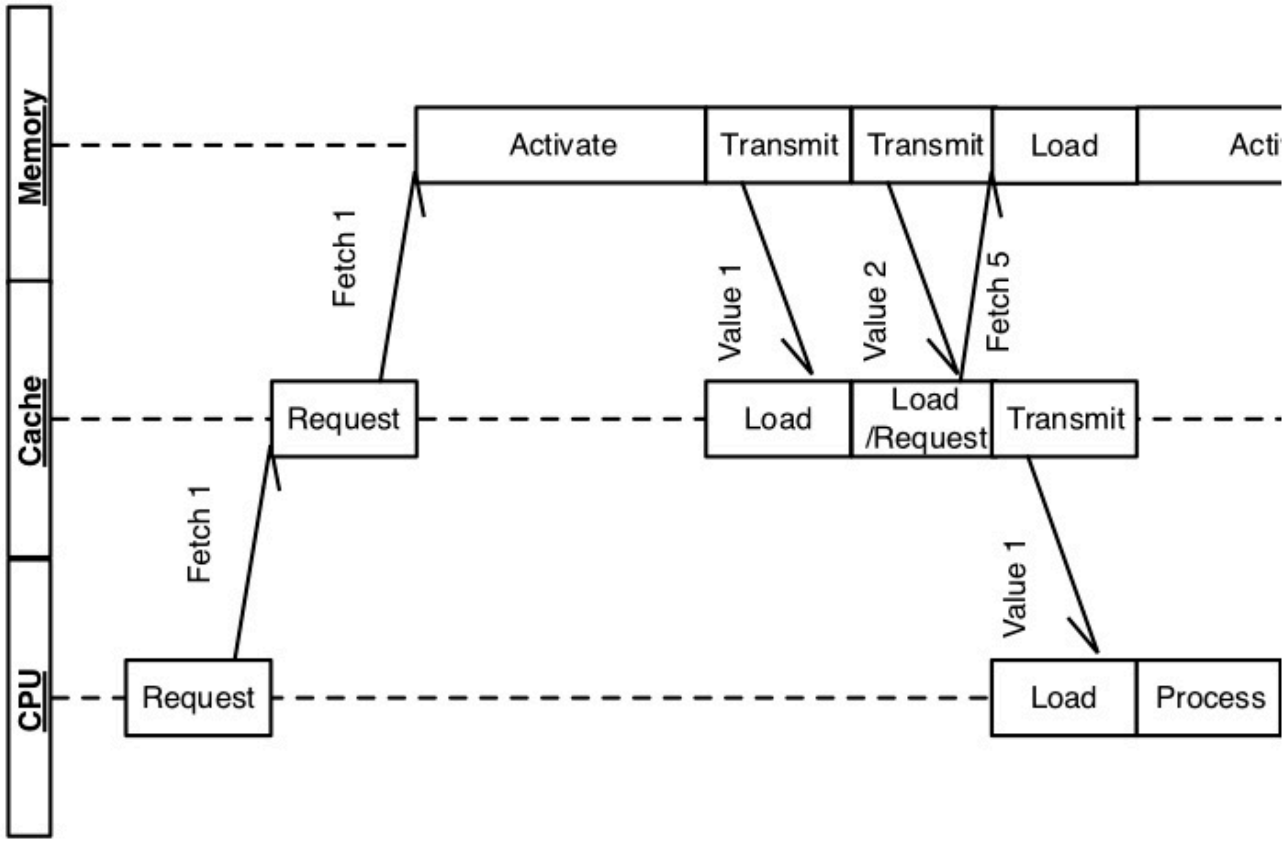
Correct Prefetching



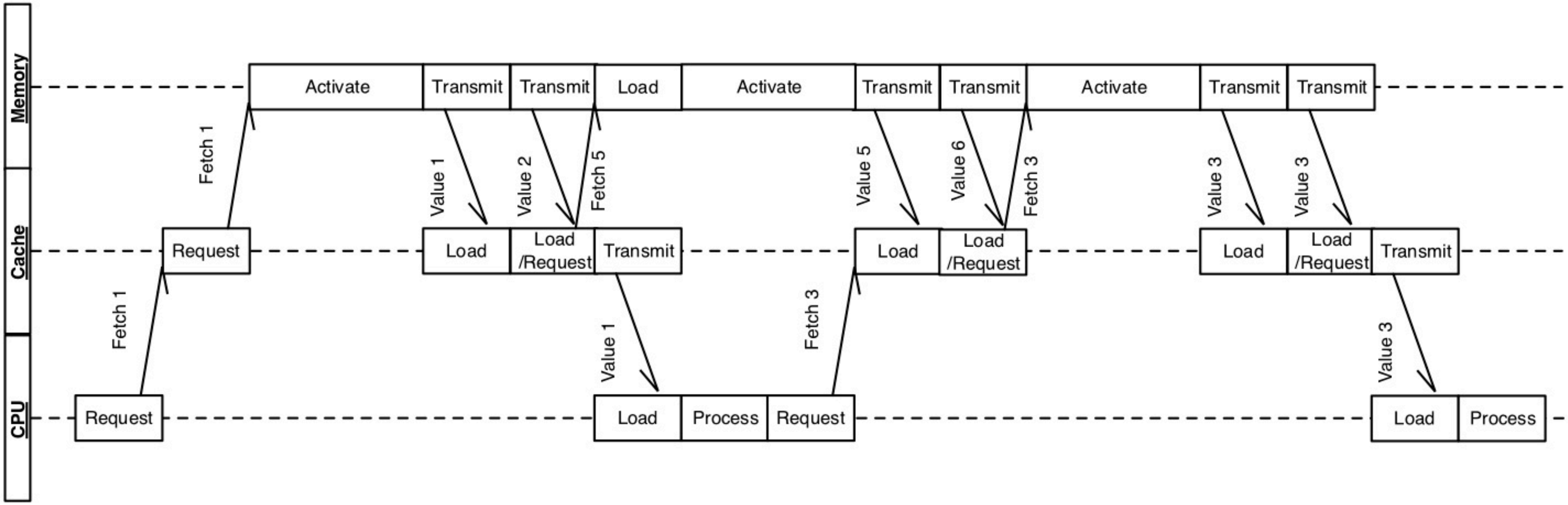
Correct Prefetching



Incorrect Prefetching



Incorrect Prefetching



Virtualized Access

Virtualization of resources becomes more and more important even for main memory databases:

- System consolidation,
- Administrative consolidation,
- Better provisioning and better scaling.

Virtualized Access

Description	Physical System	Virtualized System
L1 Miss Latency	10	10
L1 Replace Time	12	12
L2 Miss Latency	197	196
L2 Replace Time	334	333
TLB Miss Latency	-	23

Virtualized Access

Description	Physical System	Virtualized System
L1 Miss Latency	10	10
L1 Replace Time	12	12
L2 Miss Latency	197	196
L2 Replace Time	334	333
TLB Miss Latency	-	23

Virtualized Access

Description	Physical System	Virtualized System
L1 Miss Latency	10	10
L1 Replace Time	12	12
L2 Miss Latency	197	196
L2 Replace Time	334	333
TLB Miss Latency	-	23

First Conclusion: Memory Access **does not** incur a dedicated access latency!

Memory page change requires **additional** handling and thus incurs latency!

Measure Performance

To **understand** the system’s performance it is necessary to correctly **observe** its behavior.

- Identify relevant measures (CPU cycles, cache misses, resource stalls)
- Collect profiling data using **sampling** based or “real” **counting**
    - **oprofile** - provides sample-based performance evaluation for time based profiling.
    - **PAPI** - measure program performance based on hardware counters. Each CPU provides special registers to count profiling information based on special hardware events (CPU cycles, cache misses).

Measure Performance (PAPI Example)

```c
#include <stdio.h>
#include <papi.h>

int main()
{
  // PAPI events can be identified by name or const value
  char* event = "PAPI_TOT_CYC";
  // Events and results are identified
  // as array
  int events[1];
  long long result[1];

  PAPI_library_init(PAPI_VER_CURRENT);
  PAPI_event_name_to_code((char *) papi, &events[0]);
  PAPI_start_counters(events, 1);
  long long sum = 0;
  // Do something intensive here
  for (unsigned i=0; i < 1000; ++i)
     sum += i;
  PAPI_stop_counters(result, 1)
  printf("%ld\n", result[0]);
  return 0;
}
```

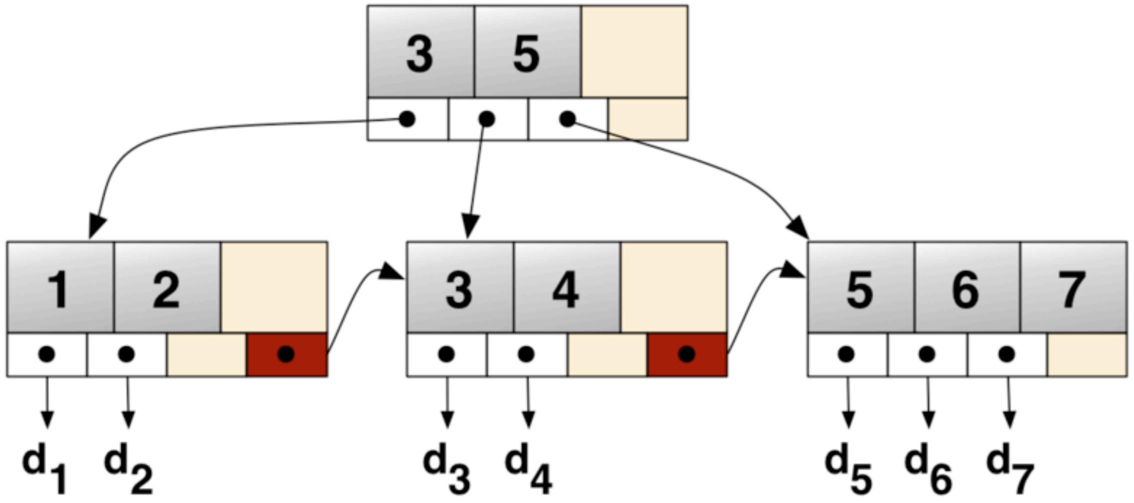
Example

- “Making B+-Trees Cache Conscious in Main Memory” SIGMOD 2000, J. Rao and K. A. Ross

Cache Sensitive B+ Tree

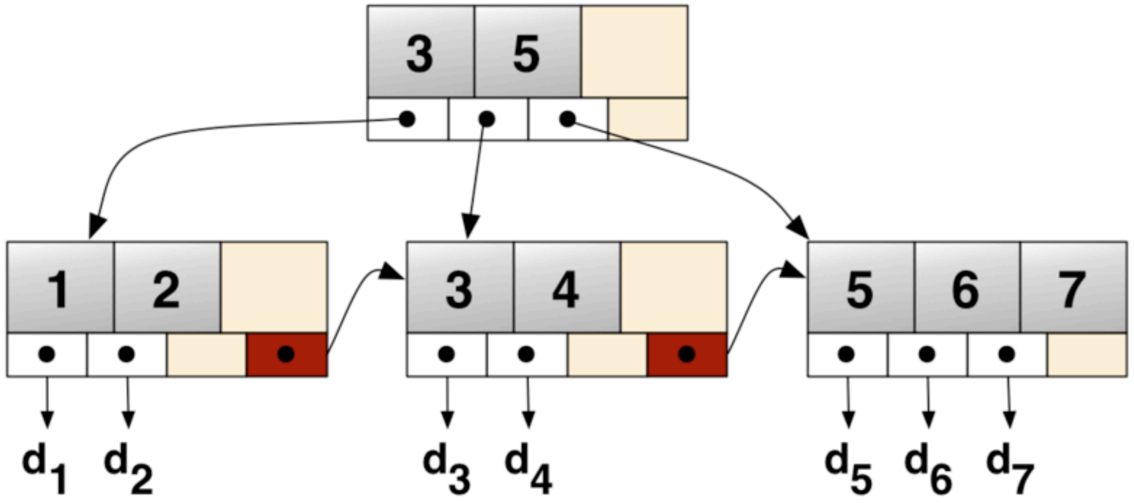
- B+ Tree - optimized search tree, typically used for indices, originally used to persist indexed data on disk!
- Cache Sensitive B+ Trees optimize cache performance by applying cache conscious techniques
  - Pointer elimination
  - Block structures

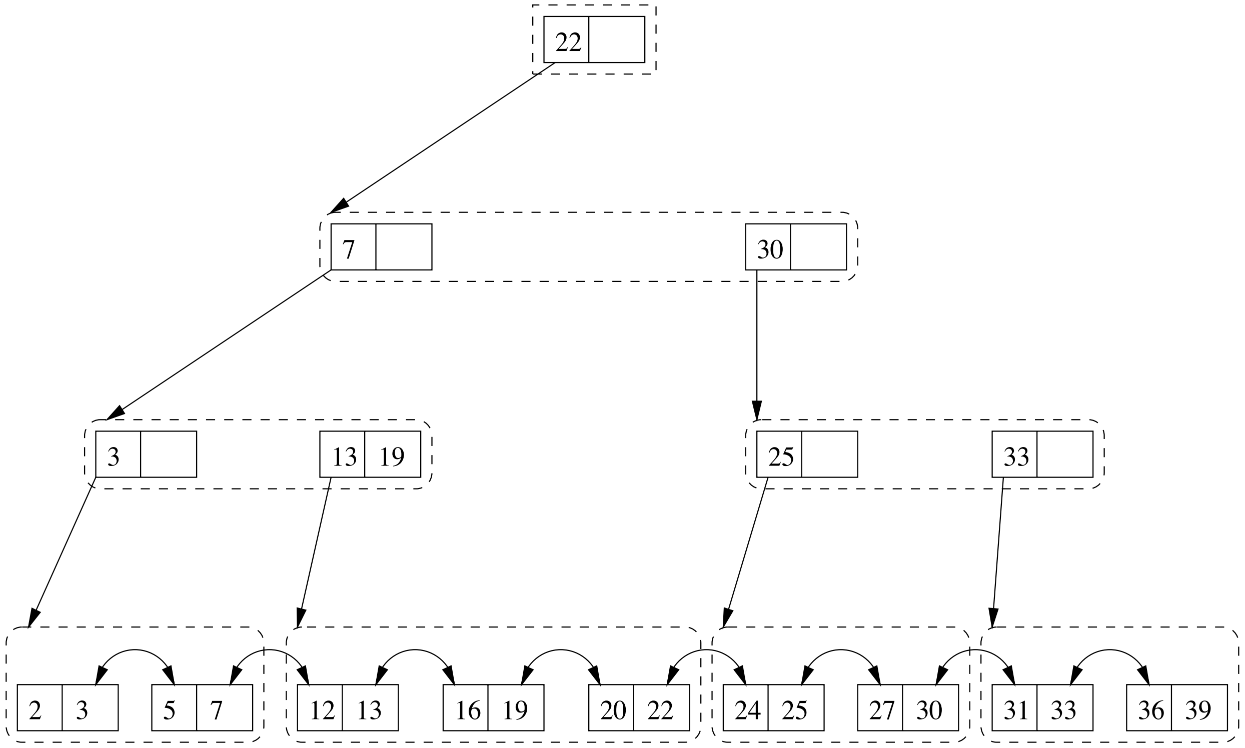
Tree Comparison



B+ Tree

Tree Comparison

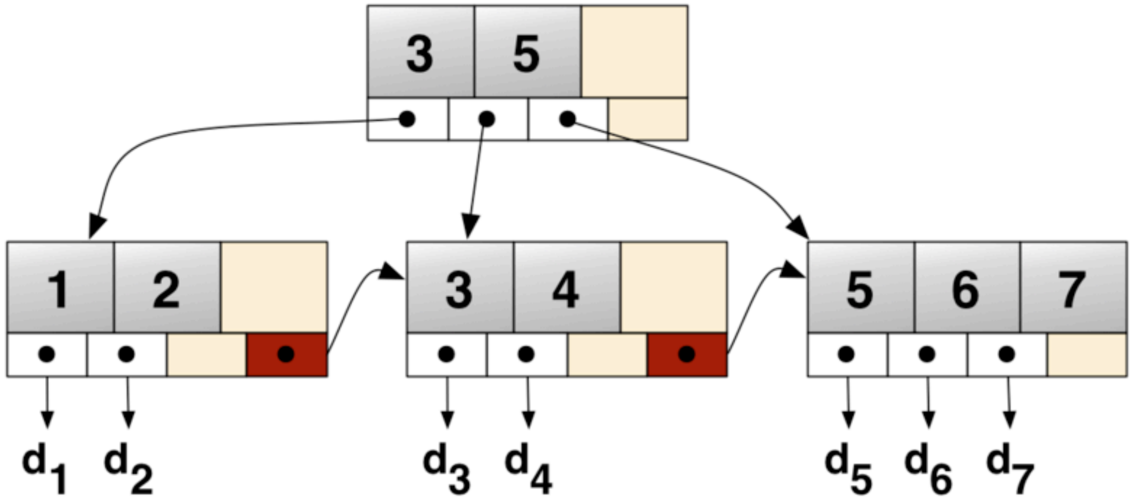




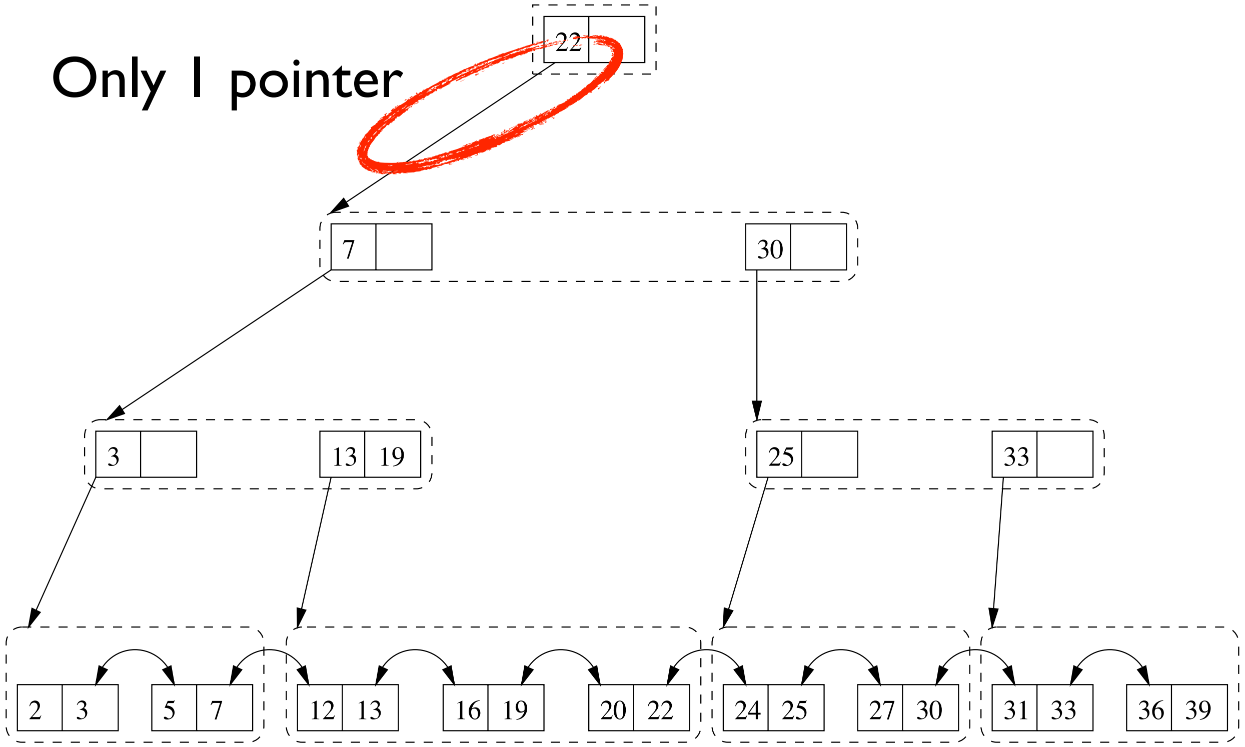
B+ Tree

CSB+ Tree

Tree Comparison

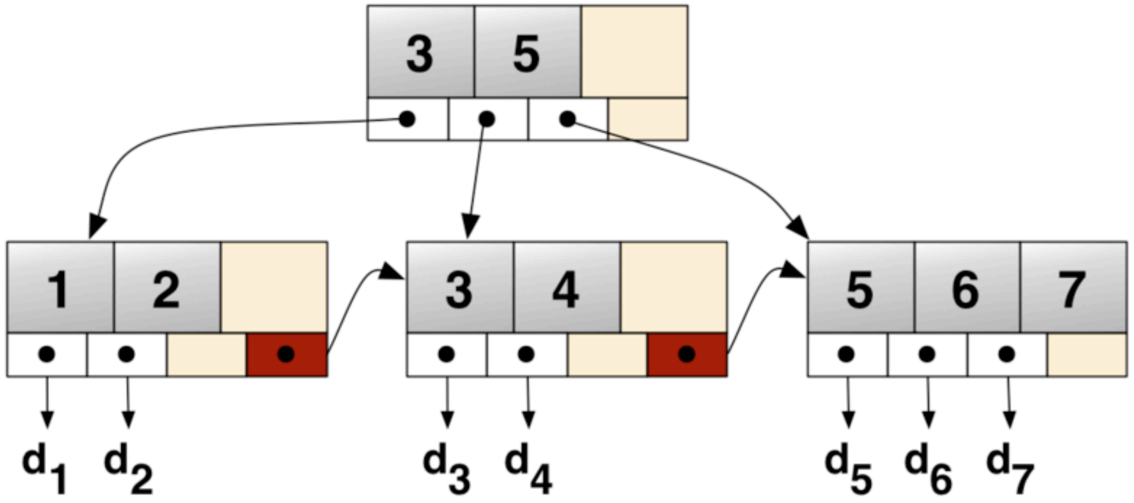


B+ Tree

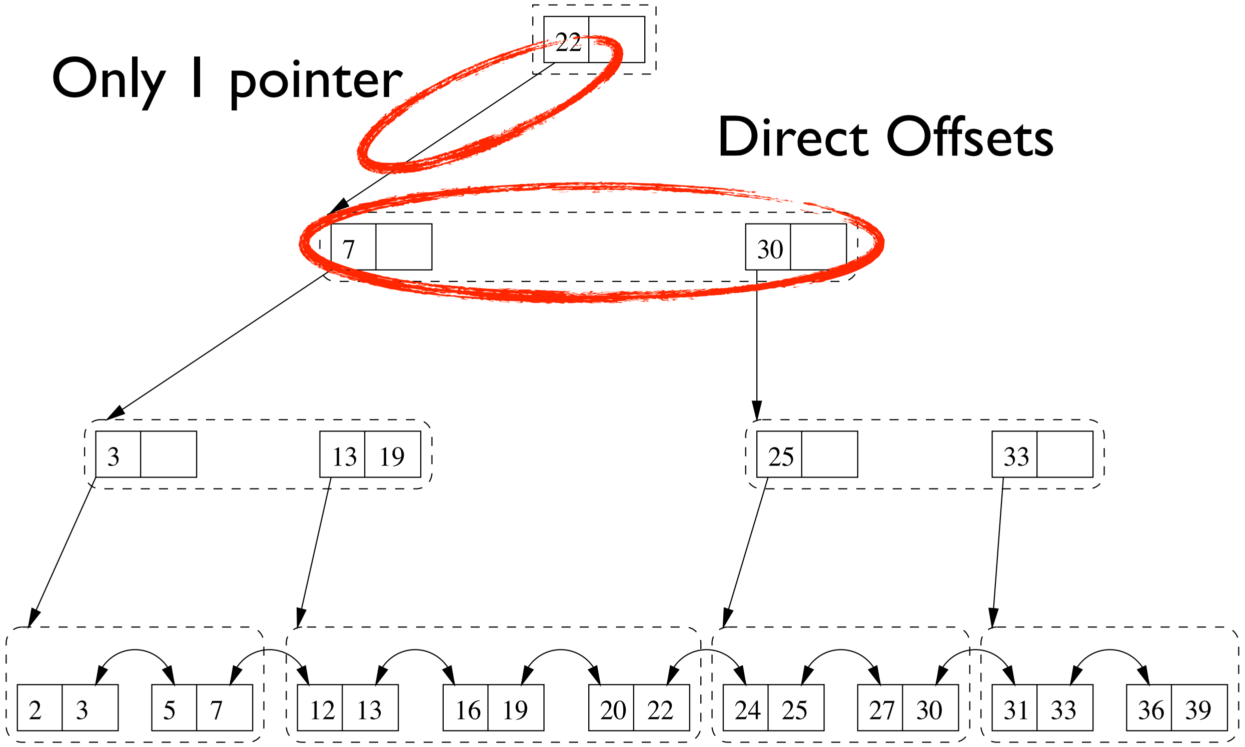

Only 1 pointer

CSB+ Tree

Tree Comparison



B+ Tree

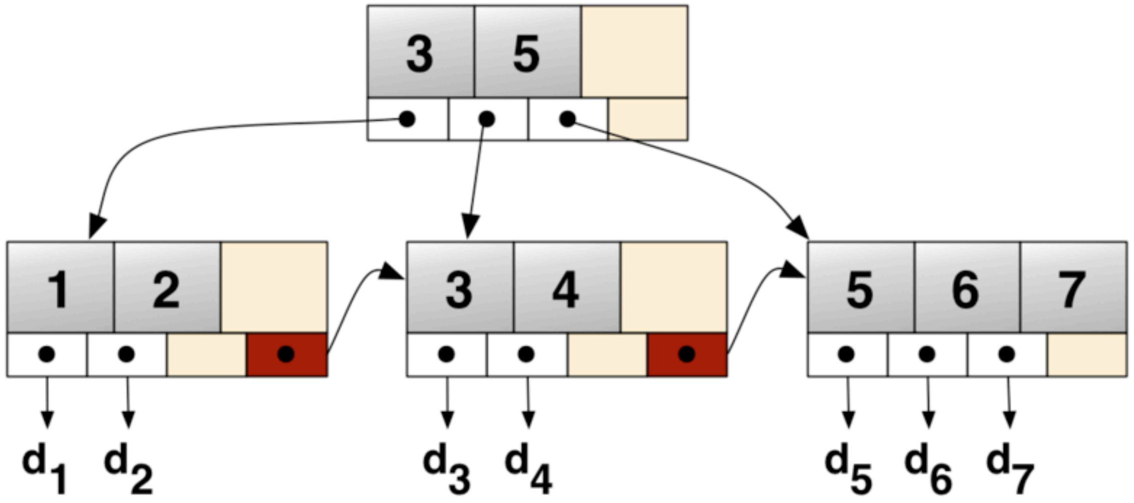


Only 1 pointer

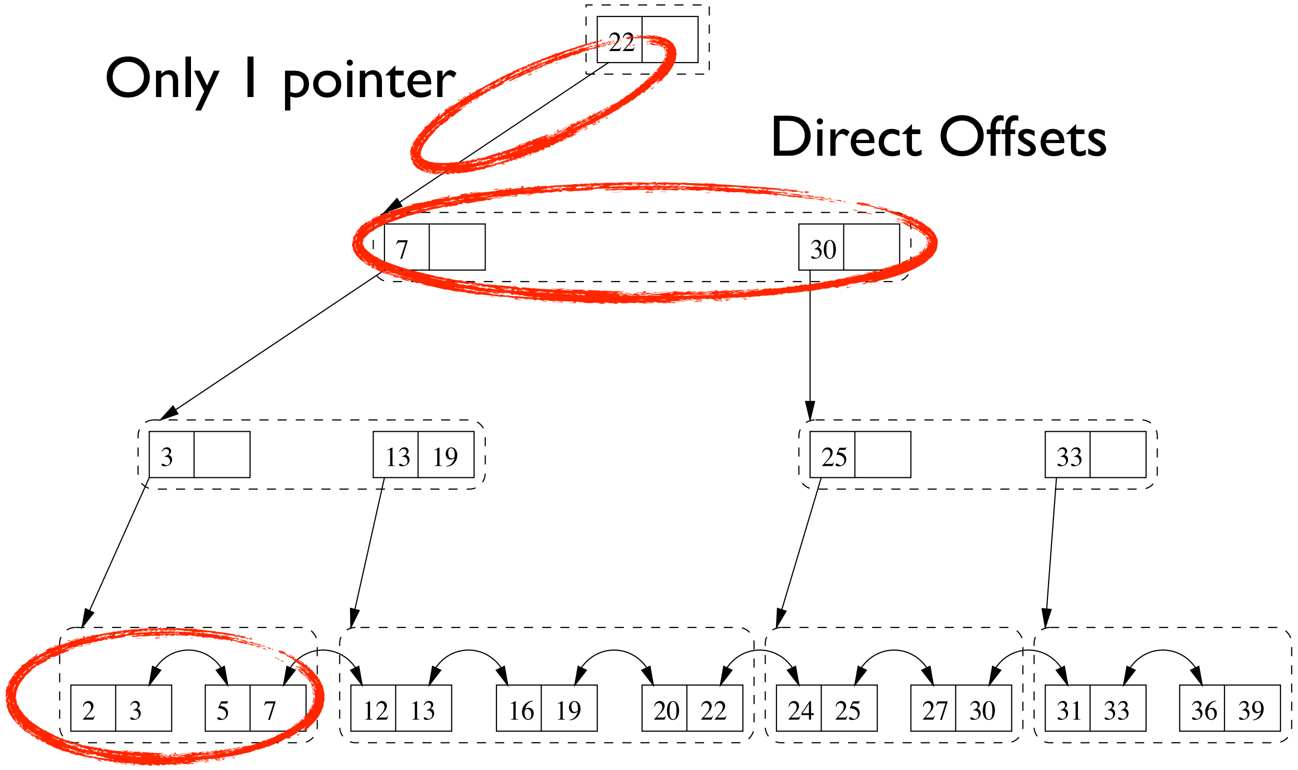
Direct Offsets

CSB+ Tree

Tree Comparison



B+ Tree



Only I pointer

Direct Offsets

Segmented Structure with pointer traversal

CSB+ Tree

Conclusion

- Observe system’s behavior
- Understand system’s performance
- Apply applicable optimization techniques.

The Future

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Many-Core

- From single-core to multi-core to many-core!
- Frequency ~ Power Consumption ~ Moores Law [1]
- Underclocking a single core by 20 percent saves half the power while sacrificing just 13 percent of the performance.

[1] http://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled

Multiple Memory Channels

- Instead of increasing the memory bandwidth, increase the number of peers
- NUMA - Non Unified Memory Access
  - Locality revisited - each CPU has local and remote memory, remote memory incurs additional loading latency.
  - Intel Quick Path Interconnect - point to point communication protocol for memory access allows to connect multiple memory channels to the CPU.

Intel SCC

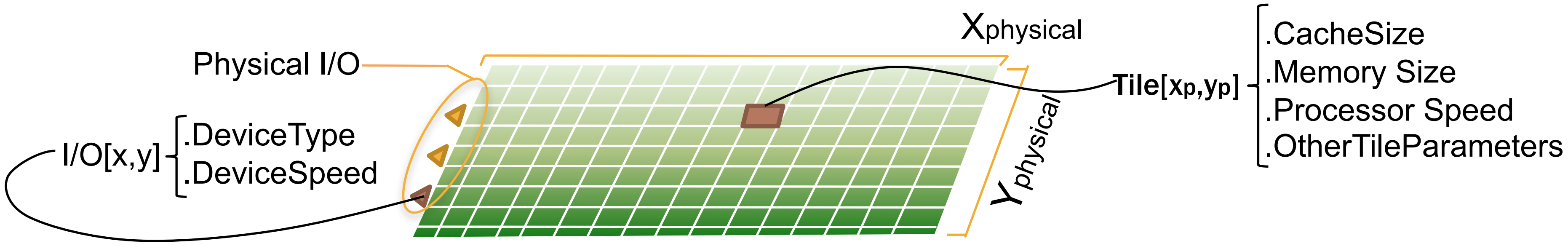
- Experimental research platform for new concepts to evaluate the evolution from multi-core to many-core.
- 48 pentium style cores arranged in a two-dimensional array of 6 x 4 tiles with 2 cores each
- Each core has 16kB L1, and 256kB L2 cache private to the core
- All cores are connected to each other using an on-chip mesh network with 256 GB/s bisectional bandwidth

Intel SCC

- **Real NUMA**, measurable latency between memory access of cores
- **Dynamic lookup tables** - each core has a dynamical mapping of main memory to its visible, no cache coherency
- **Direct Message Passing** - new communication strategy for work partitioning and coherency protocols

The Angstrom Multi-Core Computing Project

- 1000 cores by 2014, the core is the logic gate of the 21st century
- **Spatial Problem** - huge near-neighbor bandwidth, low long distance bandwidth. Limited per-core on-chip memories, off-chip memory bandwidth is a big issue. Energy is new constraint.



Conclusion and Motivation

- For optimal performance it is crucial to **understand** the system and **observe** its behavior.
- Main memory based applications need to exploit this:
  - Sequential reading
  - Block sizes of the different caches

Recommended Readings

- **Latency lags bandwitdh** - http://portal.acm.org/citation.cfm?id=1022594.1022596
- **Database architecture optimized for the new bottleneck: Memory access** - http://ece.ut.ac.ir/classpages/f84/advanceddatabase/paper/db_paper/boncz99database.pdf
- **DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing** - http://portal.acm.org/citation.cfm?id=1457150.1457160
- **Making B+-Trees Cache Conscious in Main Memory** - http://portal.acm.org/citation.cfm?id=335191.335449
- **Breaking the memory wall in MonetDB** - http://portal.acm.org/beta/citation.cfm?id=1409360.1409380
- **Generic database cost models for hierarchical memory systems** - http://portal.acm.org/beta/citation.cfm?id=1287369.1287387&coll=DL&dl=ACM&CFID=93162465&CFTOKEN=94738359
- **PAPI Performance counter** - http://icl.cs.utk.edu/papi/index.html
- **Memory system support for irregular applications** - http://www.springerlink.com/index/TQY3BCP1AEL3AQH6.pdf
- **The pathologies of big data** - http://portal.acm.org/beta/citation.cfm?id=1536616.1536632&coll=DL&dl=ACM&CFID=93162465&CFTOKEN=94738359
- **Intel Tera Scale Research** - http://techresearch.intel.com/articles/Tera-Scale/1421.htm