Efficient Memory Management

Vincenzo Innocente - CERN

Original lectures by Giulio Eulisse - CERN and Lassi Tuura (FNAL, now Google)

About These Lectures

These lectures will address memory use and management in large scale scientific computing applications, with Linux/C++ focus.

I will introduce general concepts mainly through specific concrete examples common to everyday developer work. I will focus on common aspects on commodity hardware, in areas I am personally experienced in – this is not a tour of absolutely everything there is to know about memory management.

http://infn-esc.github.io/esc19/memory

All the exercise material for these lectures

Additional Reading

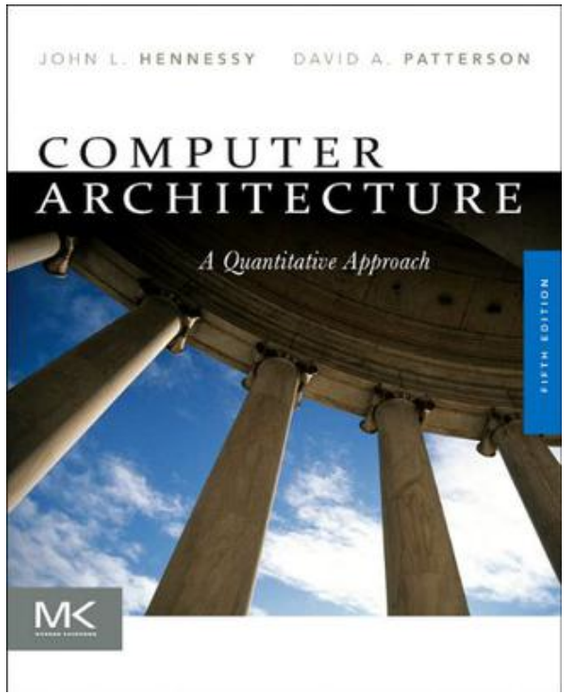
J. Hennessy, D. Patterson,
Computer Architecture: A Quantitative Approach,
5th edition (2011), ISBN 978-0-12-383872-8

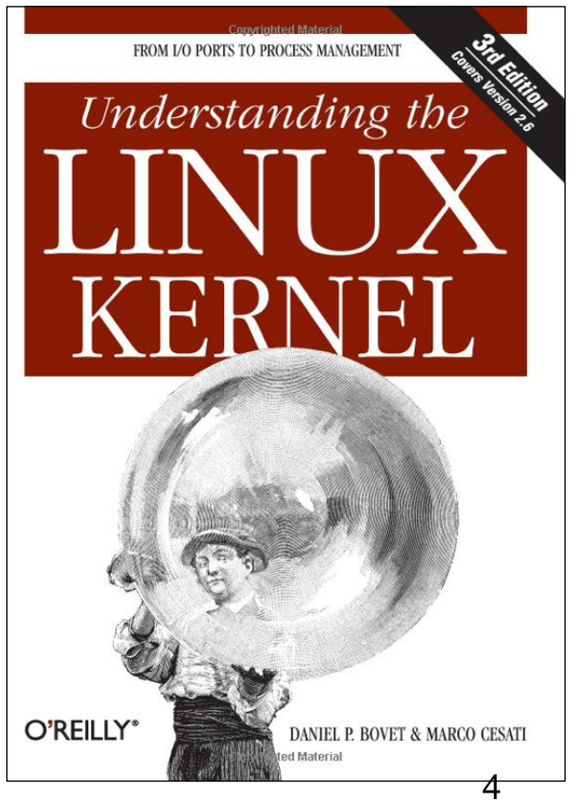
U. Drepper,
What Every Programmer Should Know About Memory,
http://people.redhat.com/drepper/cpumemory.pdf

D. Bovet, M. Cesati,
Understanding the Linux Kernel,
3rd Edition, O'Reilly 2005, ISBN 0-596-00565-2

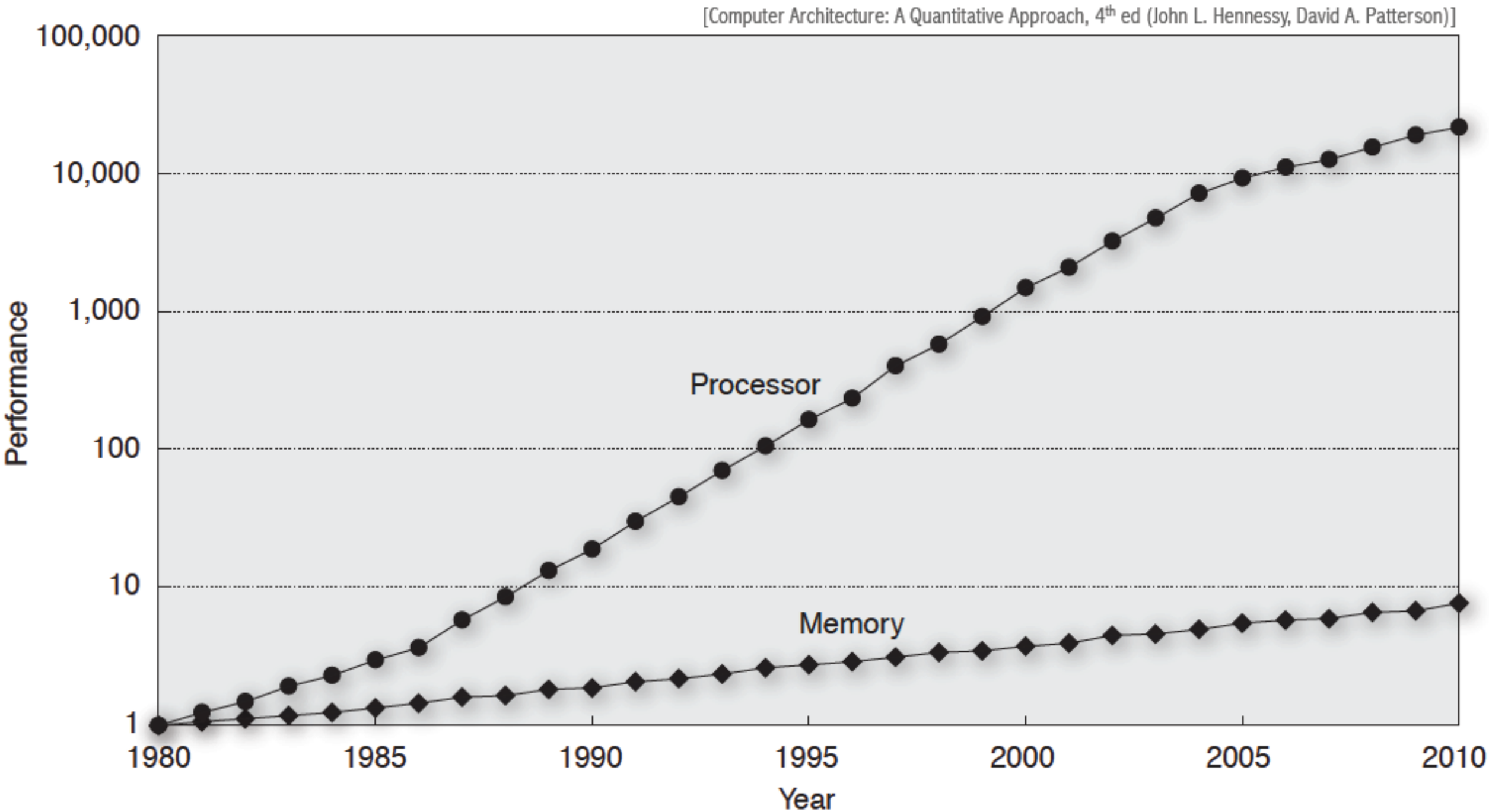
http://techreport.com, reviews with technical detail

http://jemalloc.net one of the best memory manager





Why Memory Matters: The Performance Gap



Memory performance evolution compared with processor performance

Memory Management at 10’000ft

Physical hardware

CPU pipelines and out-of-order execution; memory management unit [MMU] and physical memory banks and access properties; interconnect – front-side bus [FSB] vs. direct path [AMD: HT, Intel: QPI]; cache coherence and atomic operations; memory access non-uniformity [NUMA].

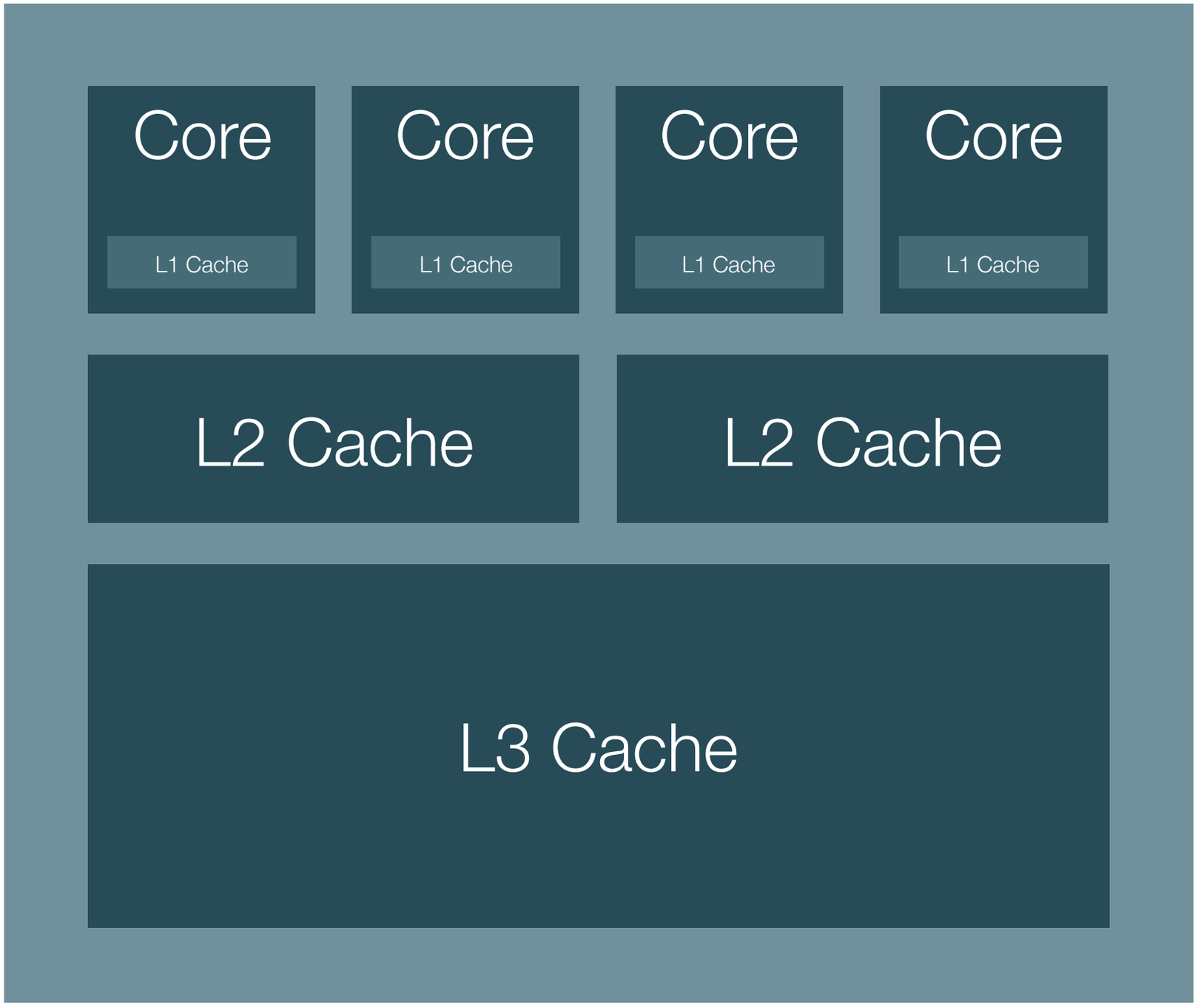
Operating system kernel

Per-process linear virtual address space; virtual memory translation from logical pages to physical page frames; page allocation and swapping; file and other caching; shared memory.

Run time

Code, data, heap, thread stacks; acquiring memory [sbrk/mmap]; sharing memory [shmget/mmap/fork]; C/C++ libraries and containers; application memory management.

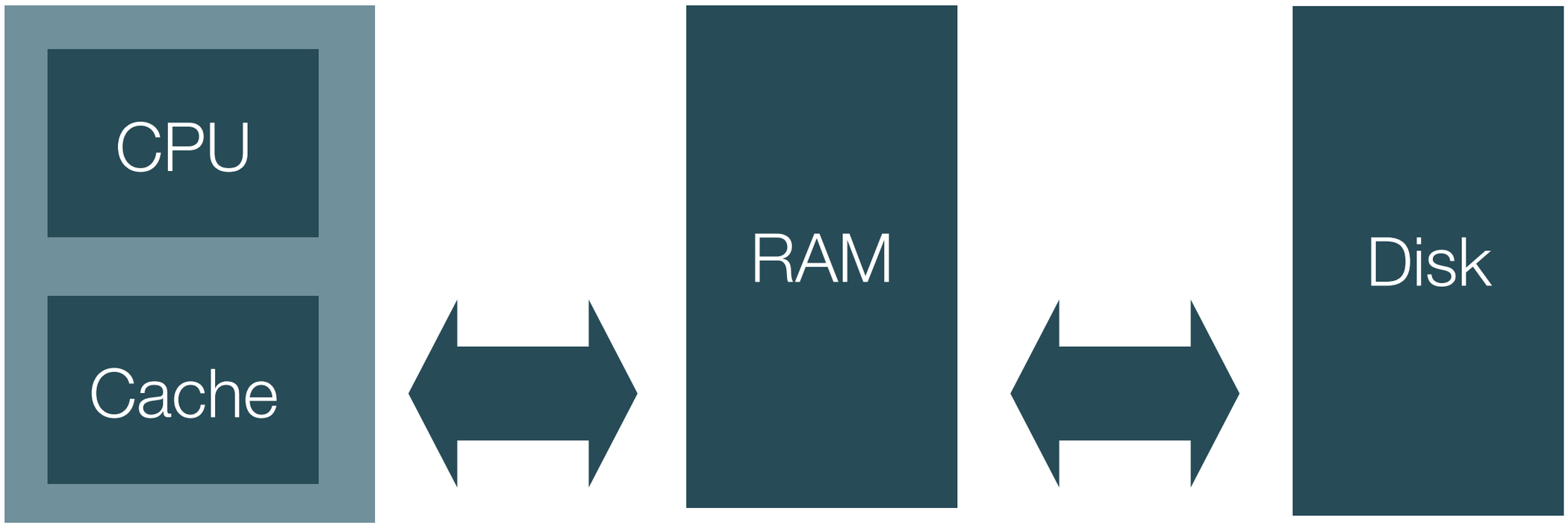
Memory hierarchy



2 - 64 cores per die, 1 - 2 dies per package,1-N packages per system.
3 levels of cache

- Small [32kB] separate L1 I+D caches for each core.
- Medium [256kB - 6MB] combined L2 cache, perhaps shared among some cores.
- Large [4 - 20MB] combined L3 cache shared between all cores on die.
- Can have even more exotic setups, especially when on cpu GPU is present.

Memory hierarchy



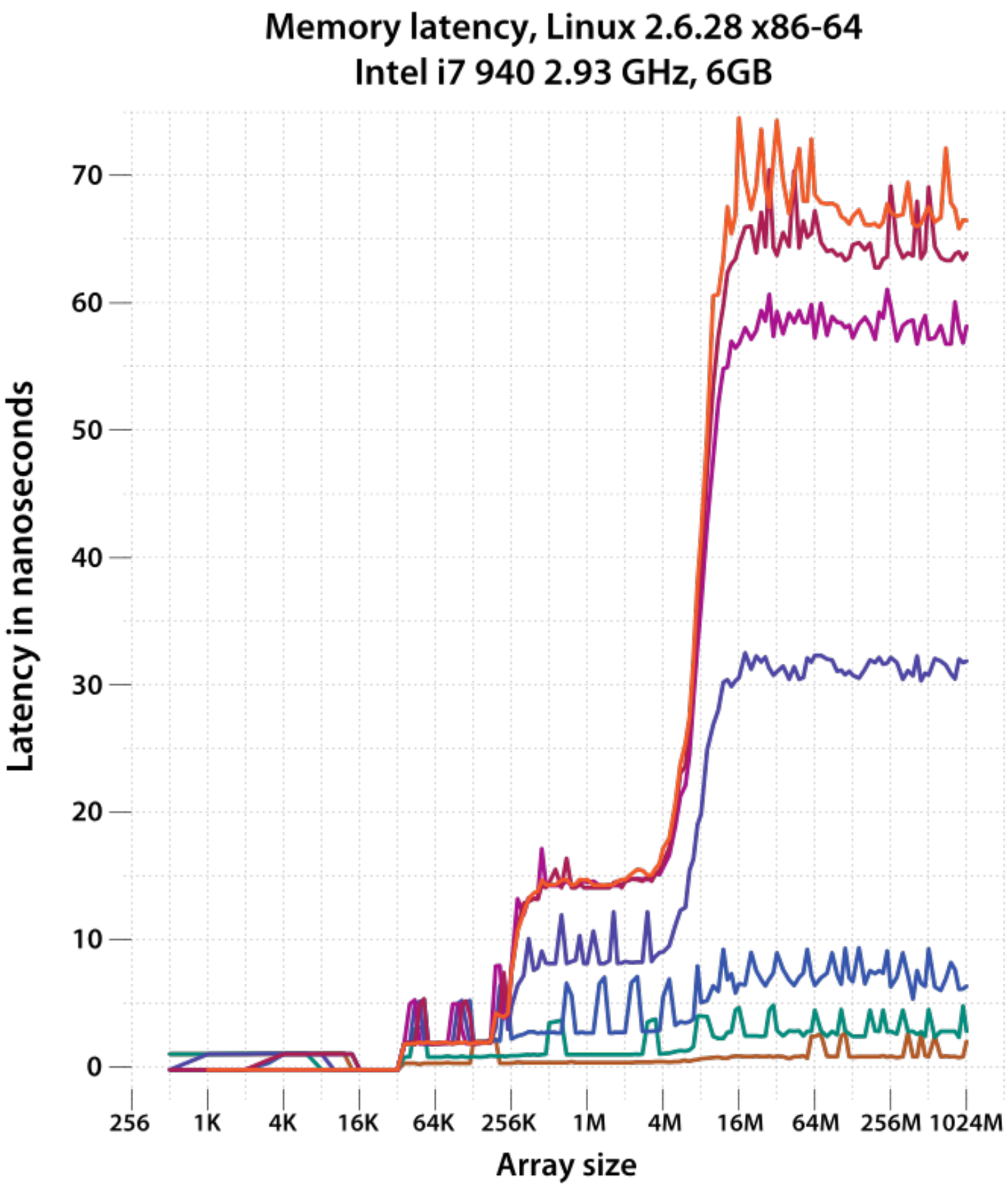
Exchange cache-lines: 64 bytes*, aligned.

Exchange pages: 4096 bytes**, aligned.

*: on most architectures

**: larger pages are available under certain cases

The Memory Wall



[LMBENCH 2.5 results for array strides 16, 32, 64, 256, 512, 1024B]

Average memory access time
= Hit time + Miss rate × Miss penalty.

I/D$: L1 hit = 2-3 clock cycles.

I/D$: L1 miss, L2 hit =~ 10-15 cycles.

TLB: L1 miss, L2 hit =~ 8-10 cycles.

TLB: L1 miss, L2 miss =~ 30+ cycles.

What happens when you drop to memory?

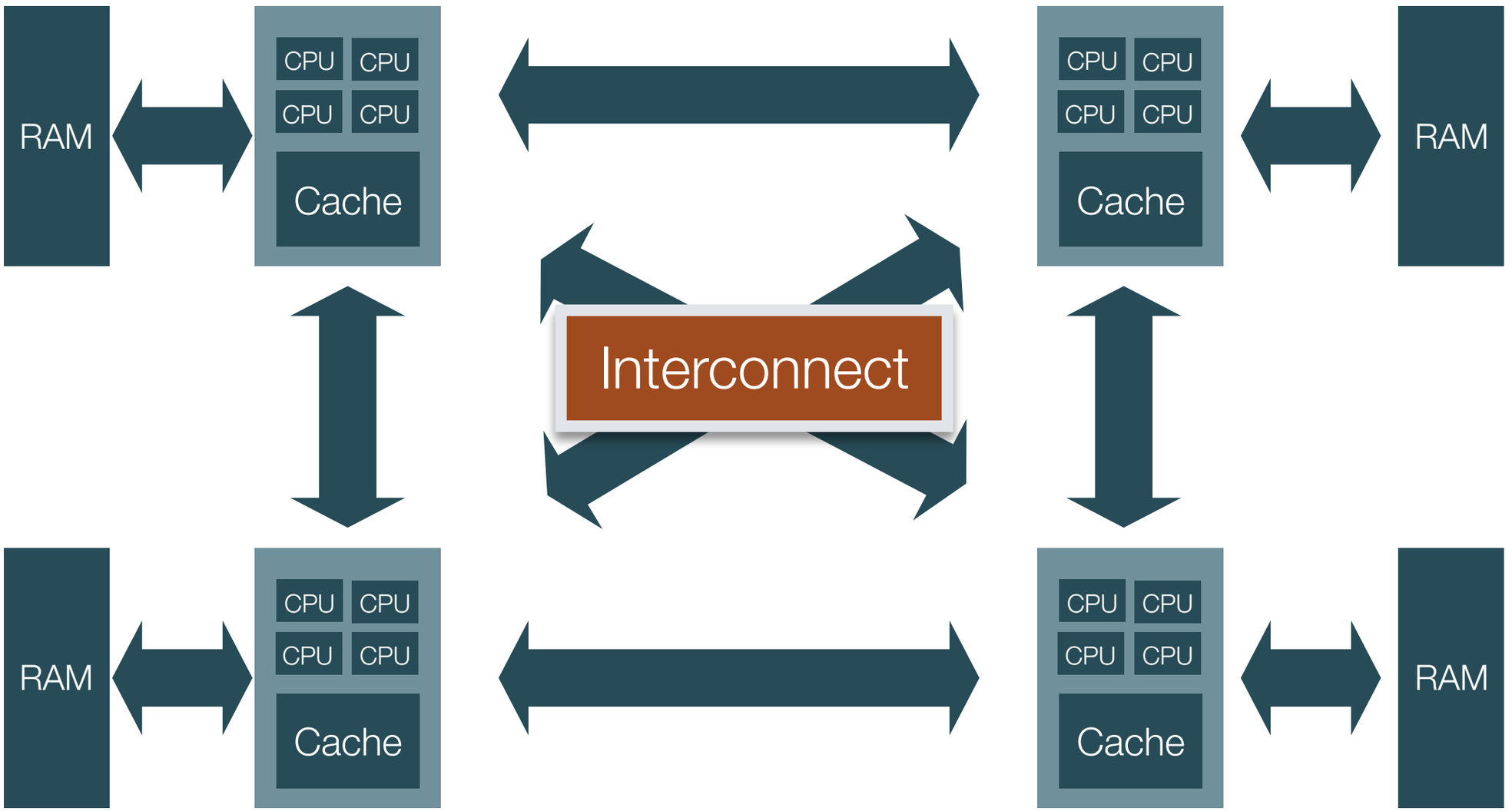
Intel Netburst Xeon (Pentium-era) memory latency was 400-700 clock cycles depending on access pattern and architecture.

AMD Opteron, Intel Core 2 and later CPU memory latency is ~100 cycles (times any NUMA overhead if crossing interconnect).

Good cache efficiency matters.

Non-Uniform memory access

RAM is not necessarily local anymore



Operating System and Memory

The operating system manages processes and their **address spaces.**

Each process has a virtual linear address space to itself, isolated from other address spaces and the kernel itself. Each process has **one or more threads,** which share the address space but have a separate stack and execution state.

The operating system manages **memory allocation and sharing.**

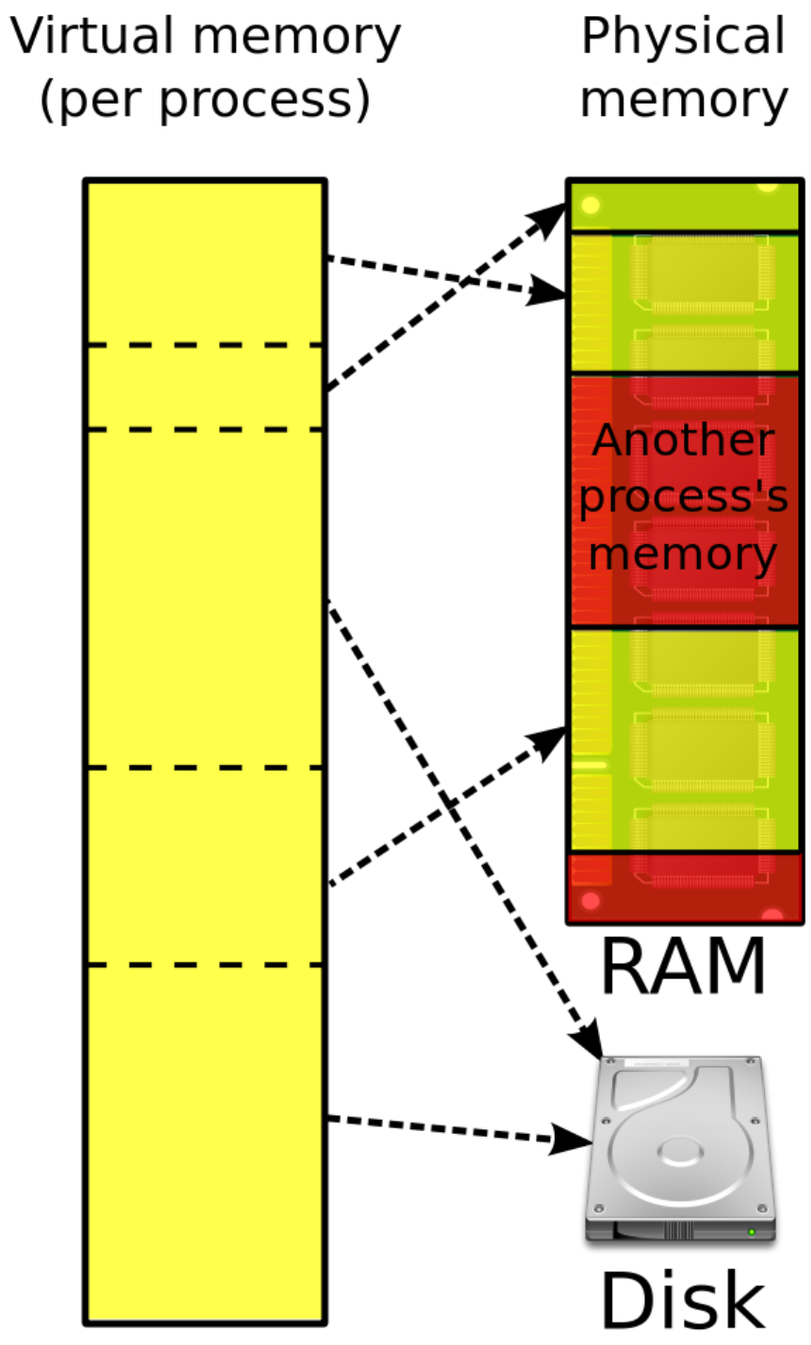
On **NUMA** systems the OS also manages process-to-physical memory mapping. In practice **application affinity hinting** is necessary (cf. numactl).

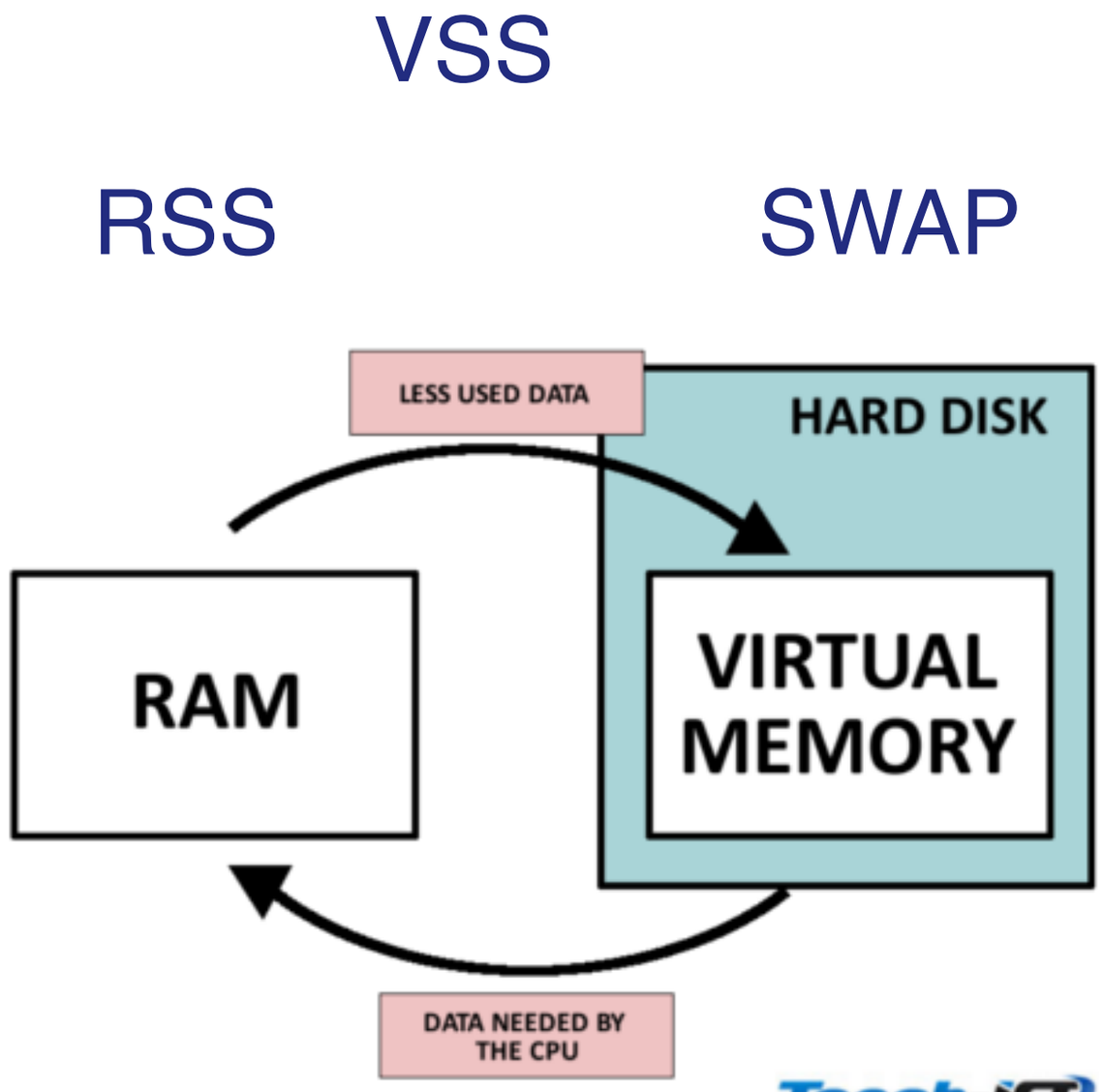
http://man7.org/linux/man-pages/man1/top.1.html#OVERVIEW

http://man7.org/linux/man-pages/man5/proc.5.html

https://www.howtoforge.com/linux-pmap-command/

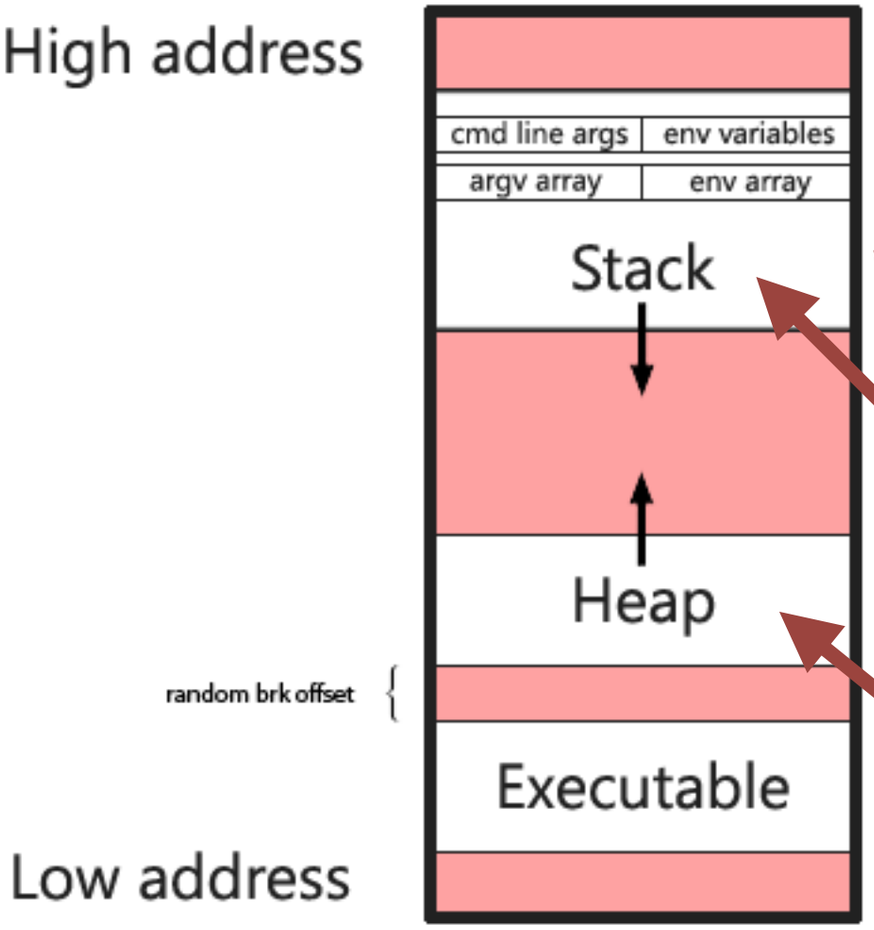
Virtual Memory

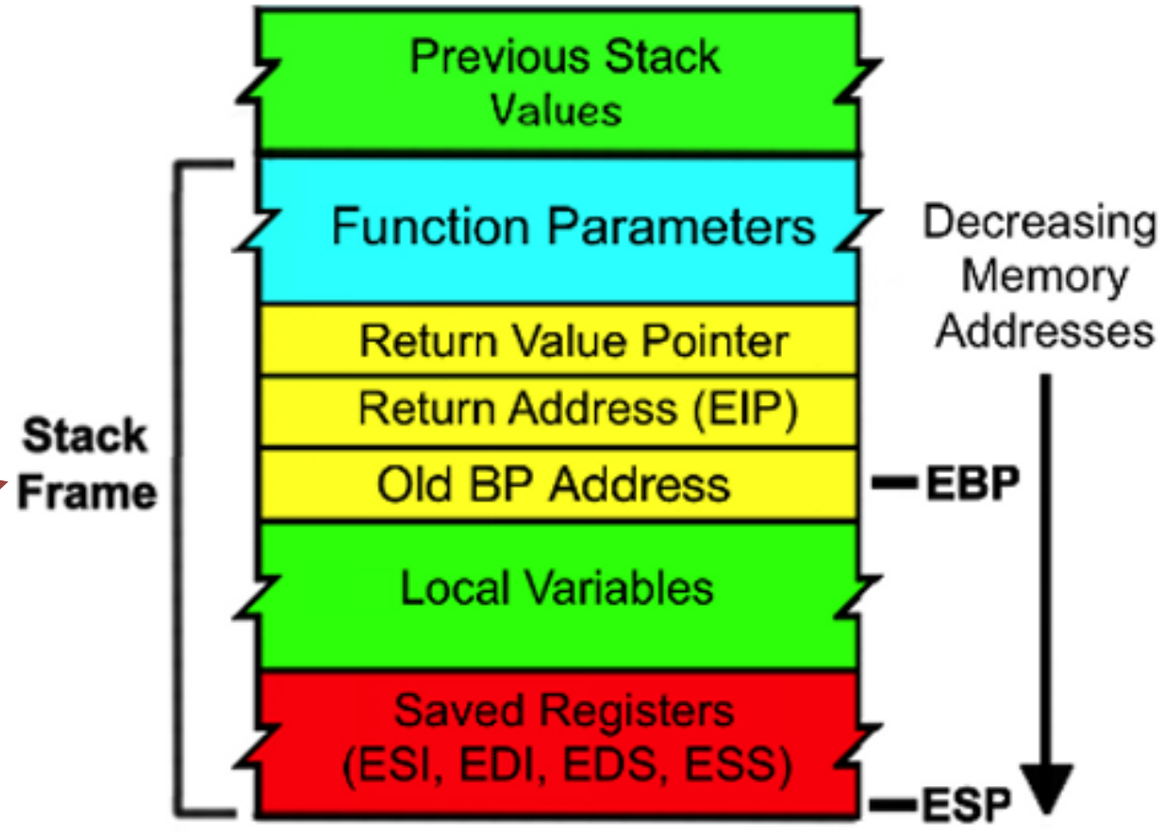




By Ehamberg - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=8352077

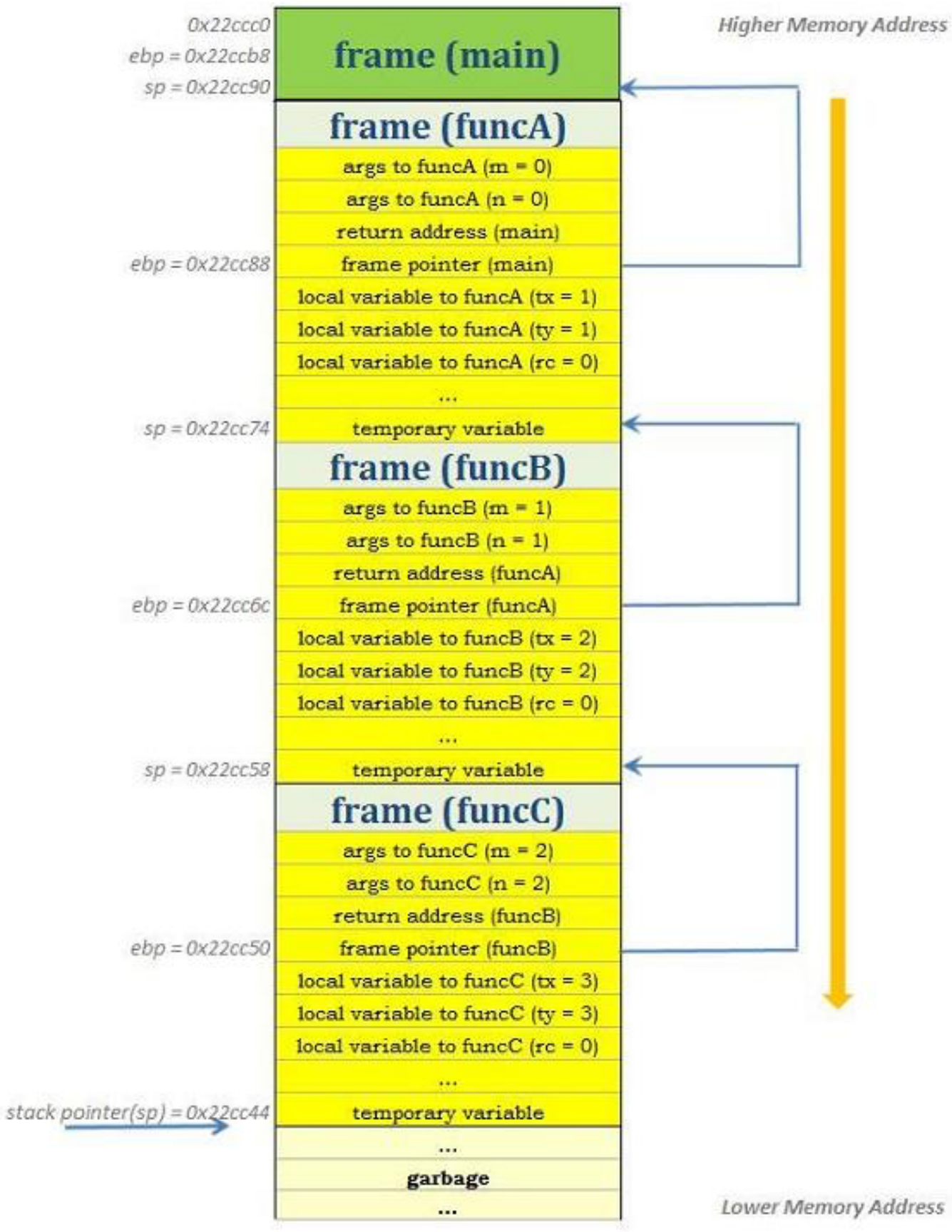
THE VIRTUAL MEMORY






```
void func(int n) {

  int k[n];

  auto * p = new int[n];
}
```

Stack frames a.k.a Activation records



Perry: ~/stack_frames.c

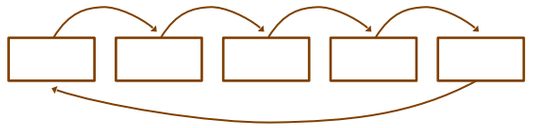
```c
int funcA(int, int);
int funcB(int, int);
int funcC(int, int);

int main () {
    int tx = 0;
    int ty = 0;
    int rc = funcA (tx, ty);
    return rc;
}

int funcC (int m, int n) {
    int tx = 3;
    int ty = 3;
    int rc = 0;
    rc = tx + ty;
    return rc;
}

int funcB (int m, int n) {
    int tx = 2;
    int ty = 2;
    int rc = 0;
    funcC (tx, ty);
    rc = tx + ty;
    return rc;
}

int funcA (int m, int n) {
    int tx = 1;
    int ty = 1;
    int rc = 0;
    funcB (tx, ty);
    rc = tx + ty;
    return rc;
}
```

memory (Heap) management Runtime

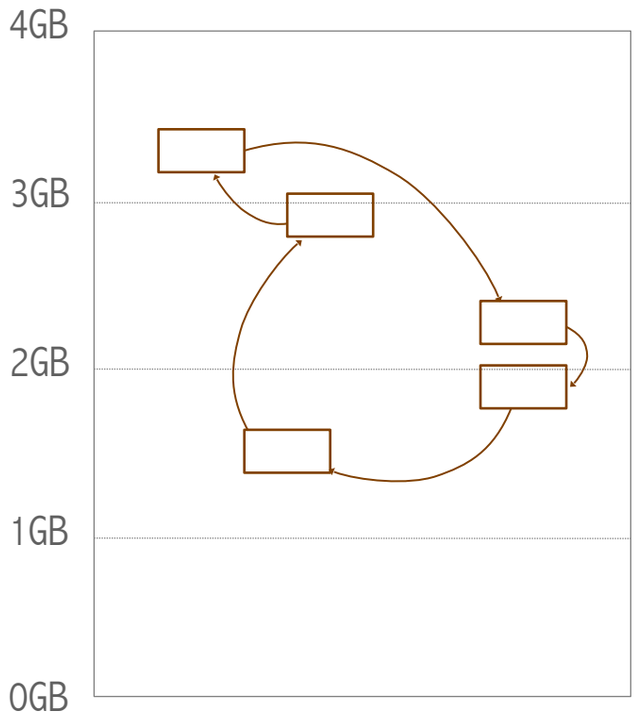
C++ / std	Posix Libs	KERNEL/OS
new, delete		brk, sbrk
aligned_storage	malloc, calloc	mmap, munmap
std::allocator	realloc	
make_unique	free	madvice
make_shared	posix_memalign	
vector, list	aligned_alloc	
map,		
unordered_map		

Logical vs. Real Data Structures

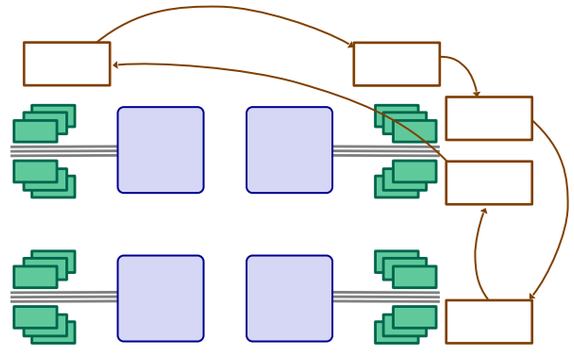
This logical linked list…



Could be scattered in virtual address space like this…



And in physical memory like this…

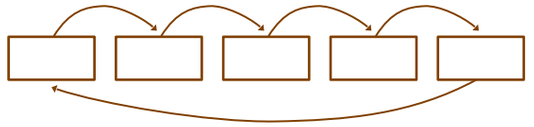


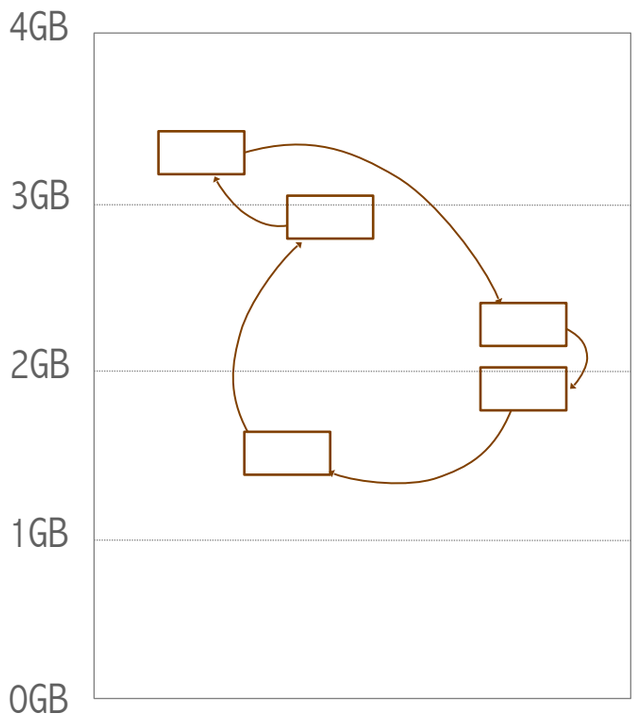
Logical vs. Real Data Structures

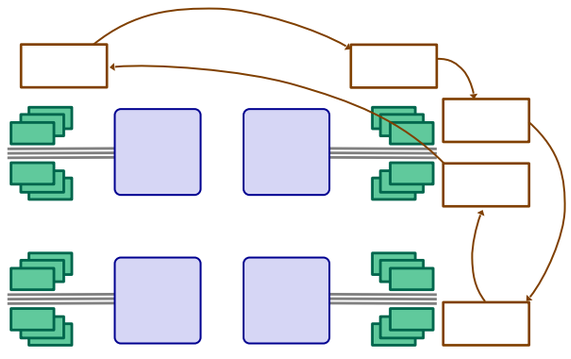
The scatter is unimportant as long as Ln and TLB caches hide all latencies. Otherwise you must explicitly arrange for a better memory ordering.

There is no silver bullet to make this problem go away.

Custom application-aware memory managers, such as pool / slab / arena allocators, other data structure changes, and affinity hints are the tools.







Key Memory Management Factors

Many factors at different levels: physical hardware, operating system, in-process run-time, language run-time, and application level.

#1: Correctness matters.

- If your results are incorrect, buggy, or unreliable, none of the rest matters.

#2: Memory overhead, alignment & churn matter.

- Badly coded good algorithm ≈ bad algorithm. If you spend all the time in the memory allocator, your algorithms may not matter at all.

#3: Locality matters, courtesy of the memory wall.

- Cache locality – stay on the fast hardware, away from the memory wall.
- Virtual address locality – address translation capacity is limited.
- Kernel memory locality – share memory across processes.
- Physical memory locality – non-uniform memory access issues.

Key Memory Management Factors

Many factors at different levels: physical hardware, operating system, in-process run-time, language run-time, and application level.

#1: Correctness matters.

- If your results are incorrect, buggy, or unreliable, none of the rest matters.

#2: Memory overhead, alignment & churn matter

- Badly coded good algorithm ≈ bad algorithm. If you spend all the time in the memory allocator, your algorithms may not matter at all.

#3: Locality matters, courtesy of the memory wall.

- Cache locality – stay on the fast hardware, away from the memory wall.
- Virtual address locality – address translation capacity is limited.
- Kernel memory locality – share memory across processes.
- Physical memory locality – non-uniform memory access issues.

Key Memory Management Factors

Many factors at different levels: physical hardware, operating system, in-process run-time, language run-time, and application level.

#1: Correctness matters.

- If your results are incorrect, buggy, or unreliable, none of the rest matters.

#2: Memory overhead, alignment & churn matter.

- Badly coded good algorithm ≈ bad algorithm. If you spend all the time in the memory allocator, your algorithms may not matter at all.

#3: Locality matters, courtesy of the memory wall.

- Cache locality – stay on the fast hardware, away from the memory wall.
- Virtual address locality – address translation capacity is limited.
- Kernel memory locality – share memory across processes.
- Physical memory locality – non-uniform memory access issues.

Memory Overheads

- Virtual Memory
  - Size (VSZ): not a real issue
  - Fragmentation: can become a real issue in particular for long running jobs
    - reboot machine time to time?
- Resident memory
  - Size (RSS): IS an issue: swapping is not an option
  - Churn: is an issue in particular if triggers system-calls
    - cpu overhead, fragmentation

Memory Monitoring: @System level

- /proc/meminfo : stat at node level
  - cat /proc/meminfo l grep -i anon
- ps (top): stat at process level
  - ps -eo pid,command,rss,vsz l grep a.out
- /proc/[pid]/smaps: details at process level
  - pmap -X yourpid (l tail -n 1)
  - parse it with a small C++/python program…
- strace : real-time or summary for system calls
  - strace (-c/C) —e trace=memory ./a.out

Memory Monitoring: @malloc level

- for jemalloc *mallctl* function provides a general interface for introspecting the memory allocator
  - http://jemalloc.net/jemalloc.3.html
- see *memory_usage.cc* for a simple, robust wrapper
- cpu overhead
  - std::chrono
  - perf record/report

Key Memory Management Factors

Many factors at different levels: physical hardware, operating system, in-process run-time, language run-time, and application level.

#1: Correctness matters.

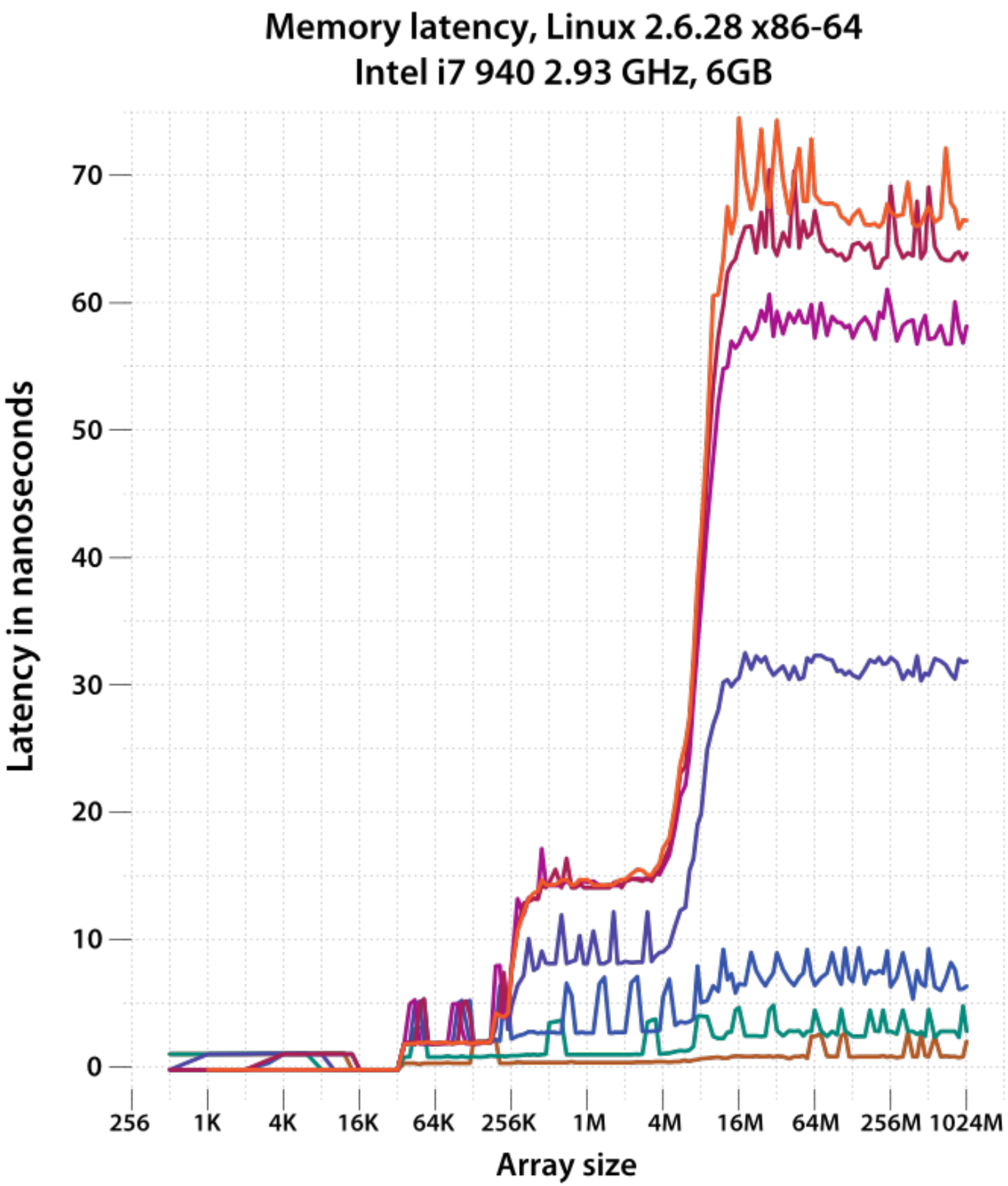
- If your results are incorrect, buggy, or unreliable, none of the rest matters.

#2: Memory overhead, alignment & churn matter

- Badly coded good algorithm ≈ bad algorithm. If you spend all the time in the memory allocator, your algorithms may not matter at all.

#3: Locality matters, courtesy of the memory wall.

- Cache locality – stay on the fast hardware, away from the memory wall.
- Virtual address locality – address translation capacity is limited.
- Kernel memory locality – share memory across processes.
- Physical memory locality – non-uniform memory access issues.



[LMBENCH 2.5 results for array strides 16, 32, 64, 256, 512, 1024B]

The Memory Wall

Average memory access time
= Hit time + Miss rate × Miss penalty.

I/D$: L1 hit = 2-3 clock cycles.

I/D$: L1 miss, L2 hit =~ 10-15 cycles.

TLB: L1 miss, L2 hit =~ 8-10 cycles.

TLB: L1 miss, L2 miss =~ 30+ cycles.

What happens when you drop to memory?

Intel Netburst Xeon (Pentium-era) memory latency was 400-700 clock cycles depending on access pattern and architecture.

AMD Opteron, Intel Core 2 and later CPU memory latency is ~100 cycles (times any NUMA overhead if crossing interconnect).

Good cache efficiency matters.

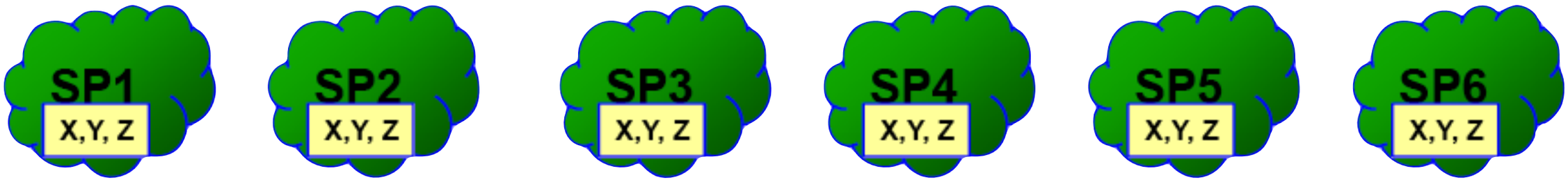
https://github.com/CppCon/CppCon2014/tree/master/Presentations/Data-Oriented%20Design%20and%20C%2B%2B

http://aras-p.info/texts/files/2018Academy%20-%20ECS-DoD.pdf

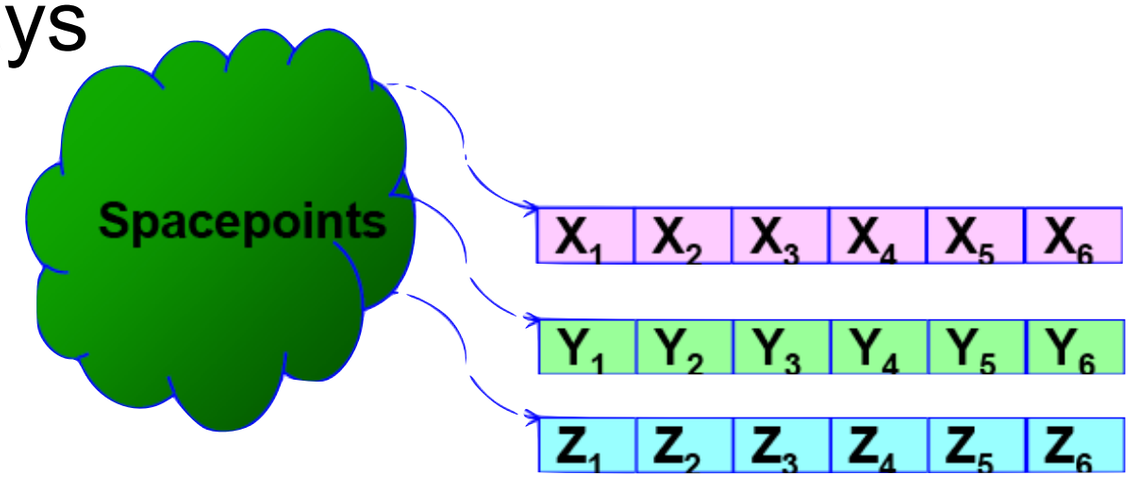
DATA ORGANIZATION

Data Organization: AoS vs SoA

- Traditional Object organization is an Array of Structures
  - Abstraction often used to hide implementation details at object level



- Difficult to fit SIMT/SIMD computing
- Better to use a Structure of Arrays
  - (column-wise storage)
- OO can wrap SoA as the AoS
  - Move abstraction higher
  - Expose data layout to the compiler



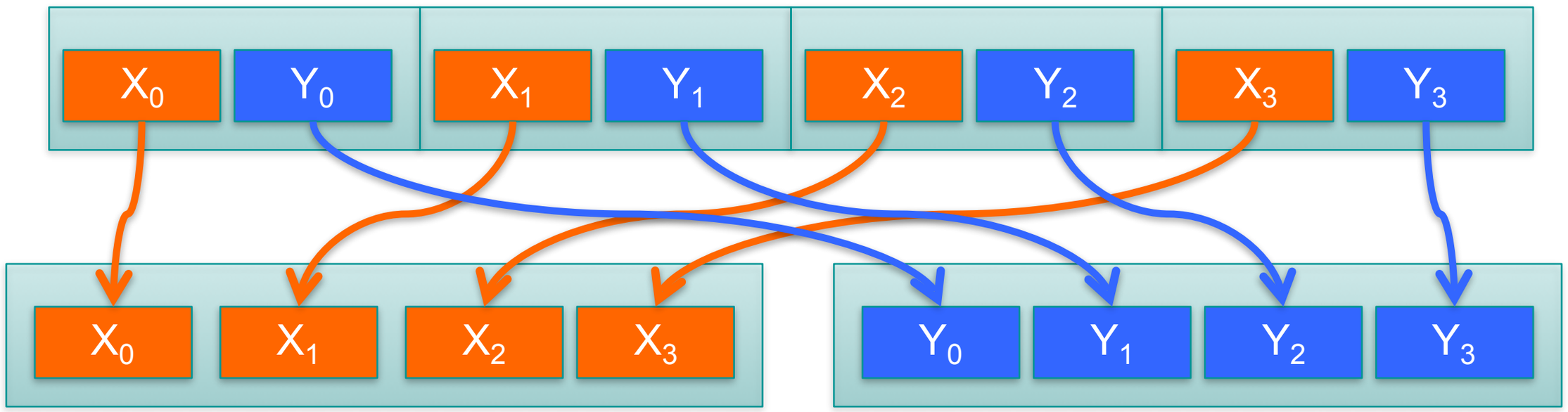
- Explicit copy in many cases more efficient
  - (notebooks vs whiteboard)

AoS vs SoA

```
struct Point {
  float x,y,z;
};
struct Points {
  std::vector<Point> p;
};
```

```
struct Points {
  std::vector<float> x,y,z;
};
```

The choice of AoS vs SoA depends on the access patterns in the application
Operations on AoS can be vectorized at the cost of *shuffles* or *permutations*
For large structures this becomes not profitable pretty fast.
In general SoA shall be preferred to fully exploit vectorization



Wrapping Up

The CPU – memory performance difference has profound impact.

Operating systems create illusion of one flat virtual address space. In reality the virtual memory is divided into pages, and pages are mapped to physical memory. Performance critical application must account for this in their design for both data and code management.

A process =~ file-backed page mappings for code and read-only data plus anonymous page mappings for stack, heap and global data. Creating many memory regions, for example by loading many shared libraries, harms performance because good performance requires static page working set which fits in TLB. Frequent page table changes are costly, some operations require a system-wide stall to synchronise the memory views of all the processors.

Shared memory is created by pointing pages tables of several processes to the same physical memory pages. Shared memory is common place, and there are numerous convenient ways to create sharing.

Exotic Efficiency Issues

Applications may need to become NUMA aware.

> May have to if on NUMA hardware, and either make significant use of concurrency and shared memory (multi-threading or multi-processing); or need more memory than a single physical node has. Read up on numactl.

Poor cache use, not getting enough out of prefetching hardware.

> Make sure you use SoA/AoS data structures, then see the other sessions this week on cache awareness, proper strides, alignment, collision avoidance, SIMD, and which tools to use identify problems and possible solutions.
>
> Multi-threaded systems may suffer from cache line contention for heavily accessed data (e.g. locks). Lots of research out there; typical solution is finer grained locks, or eliminating locking using e.g. read-copy-update (RCU). Use multithread aware allocators (like jemalloc , TCmalloc).

Killed by large page tables or TLBs? Look into using huge pages.

Summary

Memory management is expensive

Real-world limitations of CPUs and programming languages make memory management a significant factor in overall performance. The solution will vary with technical evolution. If you missed everything else, remember this: get the latency down. May mean you have to design to use hardware-aware AoS/SoA data structures.

No silver bullet

There’s no silver bullet for making your applications scream. For top performance you have to invest in real understanding and custom application-specific solutions. Beware memory churn in particular.

Know your tools

There are tools out there which will reduce the mysteries a lot. Now we will combine several of them for more serious exercises!

1. ssh

```
end

-bash-4.2$ ./psloop
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472  31284
24952 ./a.out                    7472 3938868
24952 ./a.out                    7472 3938868
24952 ./a.out                    7472 3938868
24952 ./a.out                    7472 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
24952 ./a.out                    9520 3938868
```

3. ssh

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
   10 root      20   0       0      0      0 S   6.2  0.0   4:12.63 rcu_sched
    1 root      20   0   53876   6104   2516 S   0.0  0.0   0:38.95 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.22 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.06 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    8 root      rt   0       0      0      0 S   0.0  0.0   0:00.32 migration/0
    9 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
   11 root      rt   0       0      0      0 S   0.0  0.0   0:03.87 watchdog/0
   12 root      rt   0       0      0      0 S   0.0  0.0   0:03.66 watchdog/1
   13 root      rt   0       0      0      0 S   0.0  0.0   0:00.26 migration/1
   14 root      20   0       0      0      0 S   0.0  0.0   0:00.01 ksoftirqd/1
   16 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/1:0H
   17 root      rt   0       0      0      0 S   0.0  0.0   0:04.09 watchdog/2
   18 root      rt   0       0      0      0 S   0.0  0.0   0:00.40 migration/2
-bash-4.2$ cat /proc/meminfo | grep -i anon
Active(anon):     153892 kB
Inactive(anon):   127144 kB
AnonPages:       8540008 kB
AnonHugePages:   8200192 kB
-bash-4.2$
```

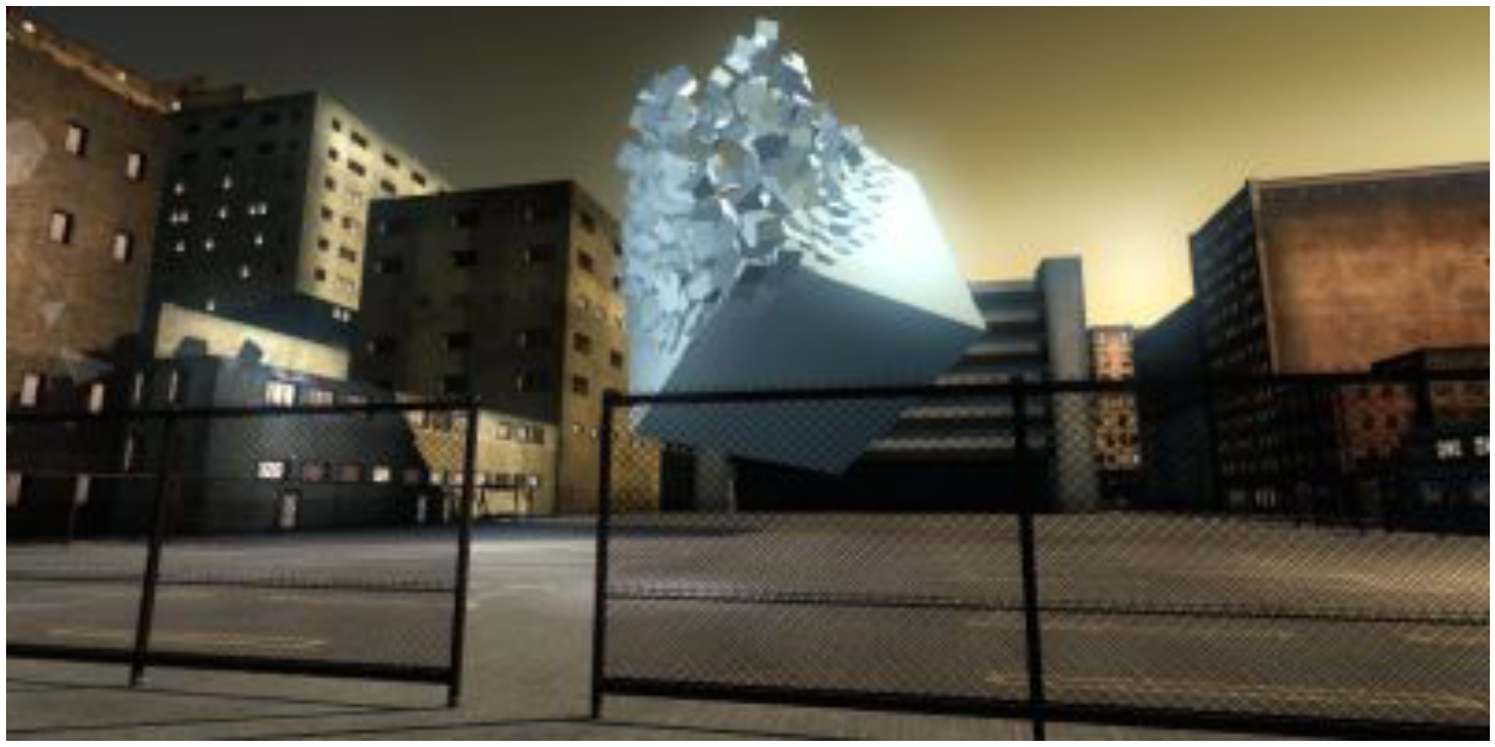
2. ssh

```
bash-4.2$
bash-4.2$ head memoryTest.cpp
//  cat /proc/meminfo | grep -i anon
//  ps -eo pid,command,rss,vsz | grep a.out
// strace -e trace=memory ./a.out
#include<iostream>
#include<cstdint>
#include<vector>
#include<memory>
#include<chrono>

#include "memory_usage.h"
bash-4.2$ ./a.out
jemalloc counters are available
start elapsted time 1856370
 allocated so far 32 deallocated so far 0
total live 36864
continue?c
carray: after create elapsted time 29865
 allocated so far 4001366048 deallocated so far 0
total live 4001402880
continue?c
carray: after assign 0 elapsted time 816302
 allocated so far 4001366048 deallocated so far 0
total live 4001402880
continue?
```

For the child nerd in all of us...

Old "arcade" games did not have enough raw CPU power to copy memory around, nor enough memory to store whole levels as big images images. They relied on the ability of the (graphics) hardware to "compose" scan-lines from predefined tiles, superimposing the result with sprites(e.g. the player) images. Tiles and sprites were actually sitting at fixed locations.

For the teenage nerd inside all of us...



https://www.youtube.com/watch?v=mxfmxi-boyo

The video is generated (in realtime) with a 177KB executable on 2007 hardware