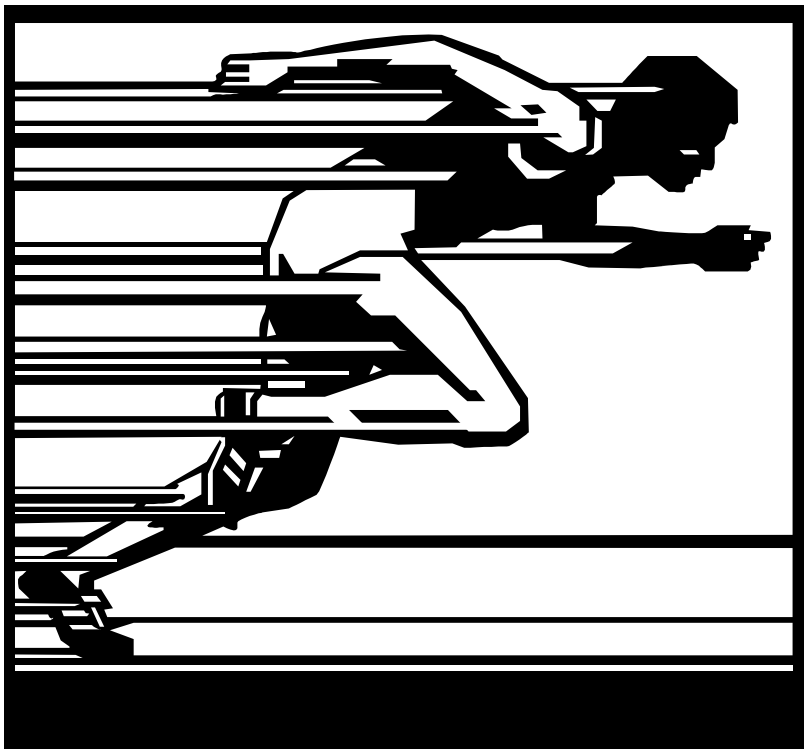


Efficient Memory Management

Lassi Tuura, Northeastern University

ESC09 – Bertinoro, Italy – 12-17 October 2009

Architectures, tools and methodologies for developing efficient large scale scientific computing applications

About These Lectures

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

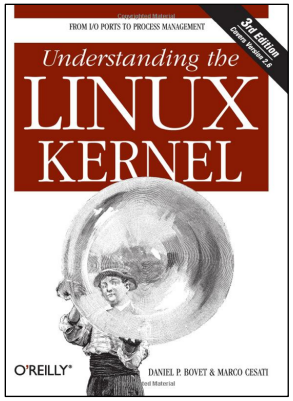
I will introduce general concepts mainly through specific concrete examples common to everyday developer work. I will focus on common aspects on commodity hardware, in areas I am personally experienced in – this is not a tour of absolutely everything there is to know about memory management.

The following are valuable additional reading:

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 good technical detail



Why Memory Management Matters?

So, you've got a problem to solve. You've designed an algorithm to solve it. Now all you need is it code it up and you are done, right?

Actually, you have just begun. Your algorithm will translate to *real machine code,* which will run on very *real physical systems,* which have very *real practical limitations.*

A complete design must account for the real world limitations. This means *"the solution" will vary over time* with technology evolution.

Why Memory Management Matters?

Different solutions to the same problem vary dramatically in real life performance.

Algorithmic and data structure changes can easily result in several orders of magnitude improvement and regression. Always research this option first.

In some cases, *changes in memory use and management* can also easily produce orders of magnitude performance wins and losses – even without major logical change to the underlying algorithms. Common critical factors include *memory churn*, *poor locality*, and in multi-processing, *memory contention*.

In other cases, simple, subtle changes can yield performance wins in the *1-10% range*. When % of your computing capacity is counted in rows of racks and days of processing, this still matters a great deal in practise! The small stuff still directly affects how much science you get out of your funding.

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 churn matters.

- 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 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.

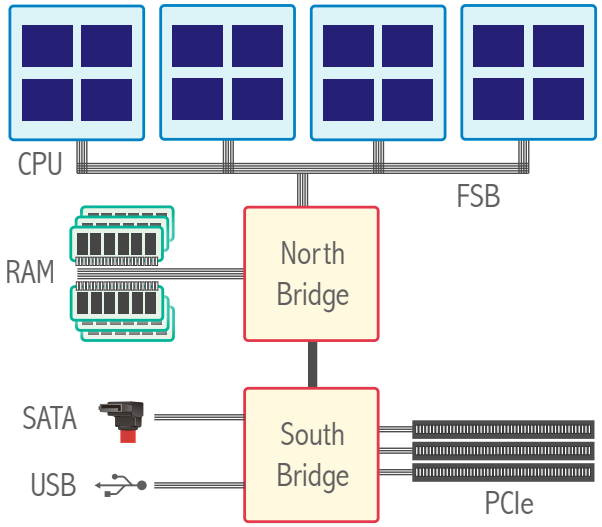
Run time

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

CPU, Physical Memory and Interconnect

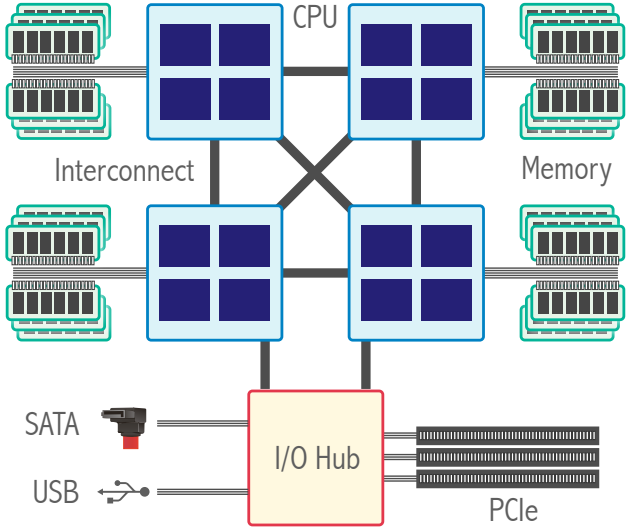
Old design: All CPUs linked via the *front side bus* (FSB) to the north bridge, which provides access to memory, and to south bridge which attaches to I/O.

Issues: FSB bottlenecks in SMP systems, device-to-memory bandwidth via north bridge, north bridge to RAM bandwidth limitations.

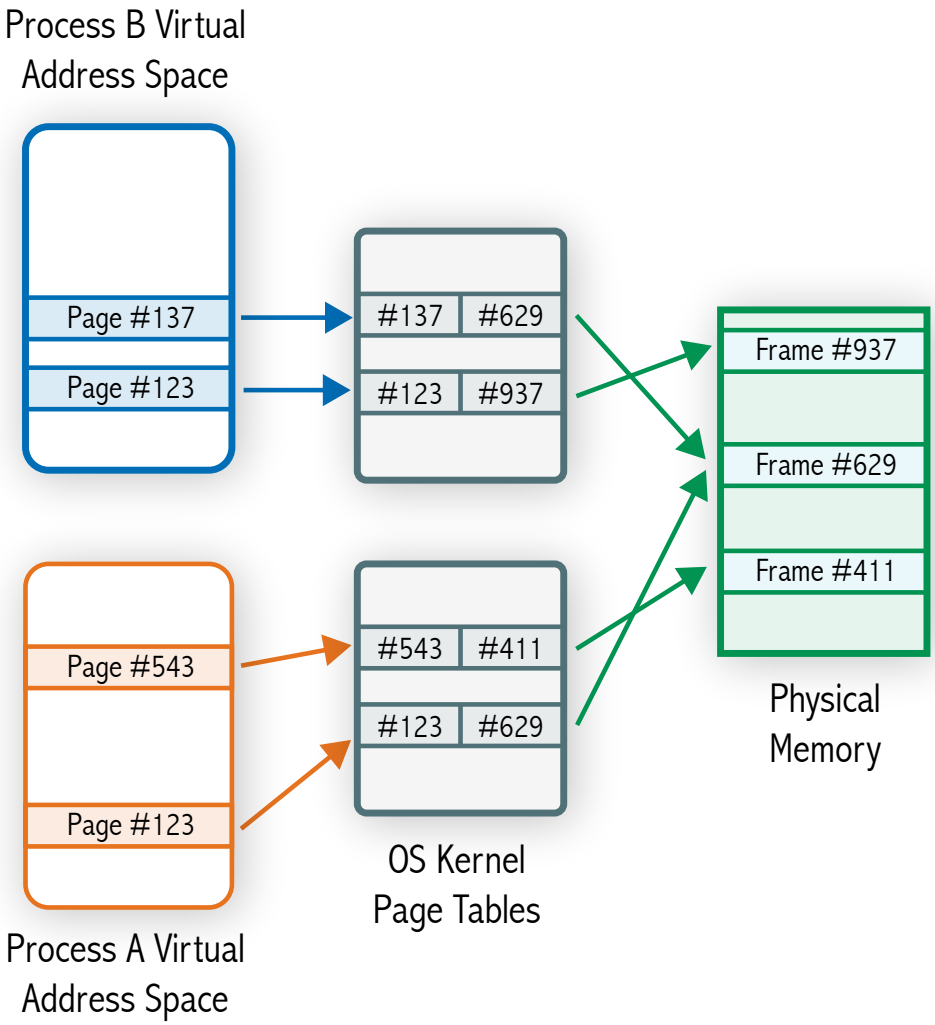


Current design: memory controller directly on each CPU, physical memory partitioned per CPU, fast interconnect to link CPUs to each other and to I/O.

Solves many issues in FSB design, but *memory is no longer uniform* – 30-50% overhead to accessing remote memory, up to 50x in obscure multi-hop systems.



Virtual Memory



Today's OSes give processes a flat* **linear virtual address space:** the same linear address in two different address spaces means two entirely different physical addresses.

Virtual and real physical memory is divided in **pages,** usually 4kB, but optionally 1-4MB. The OS provides the CPU per-process **page tables** to map a virtual address to a contiguous **physical page frame** plus offset, which in turn translates to memory bank, row and column.

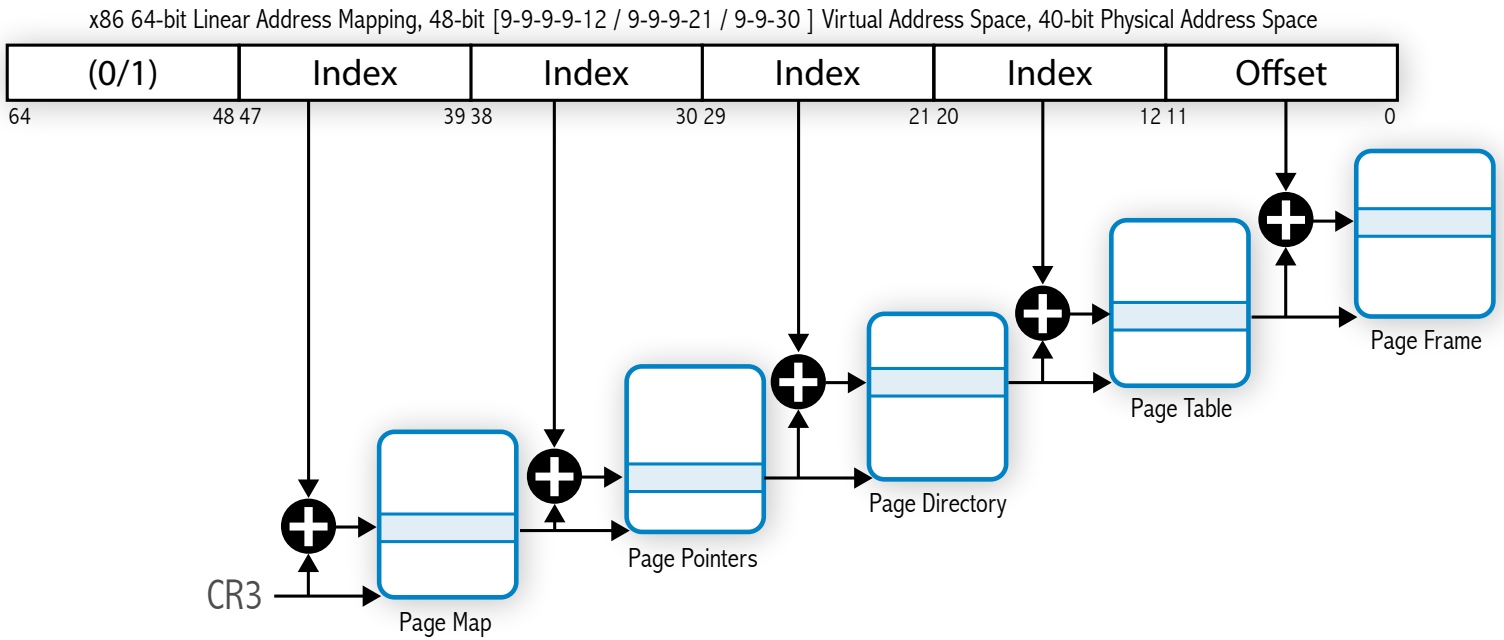
Page tables themselves **use memory, consume L2+ cache space,** and are never swapped out.

Even if processes share physical page frames, the **page tables are not shared.** With 4kB pages, large address spaces mean *big page tables,* even if the memory itself is shared: there's over 2MB of page tables for every 1GB of committed address space.+

+ 2GB VSIZE × 128 processes requires 0.5GB page tables.

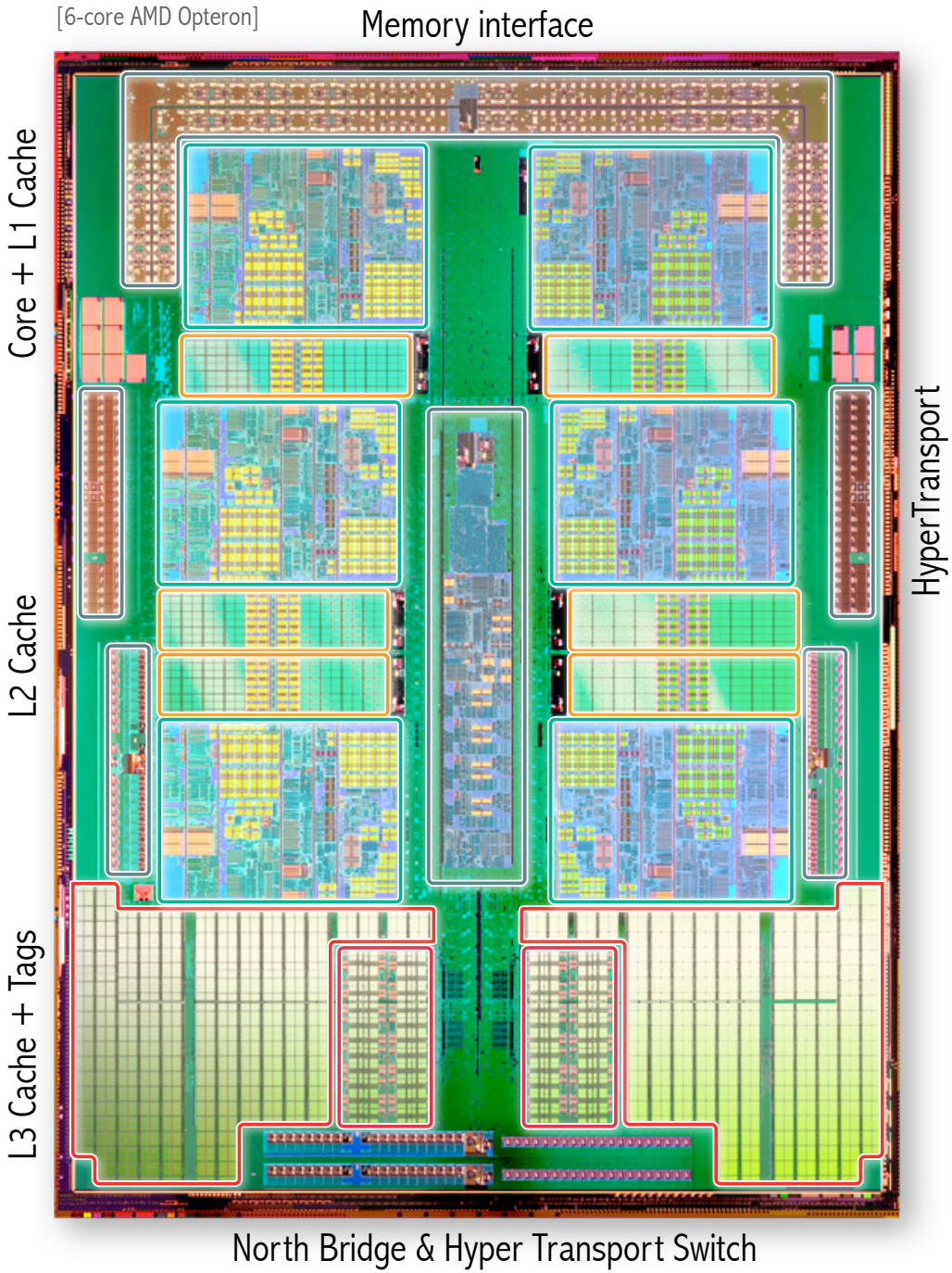
* CPUs also segment or otherwise divide memory in regions; details in the references. "Flat" does not mean "simple", the address space can be a fairly hairy object.

Virtual Address Translation



Special cache hardware called **TLB,** translation look-aside buffer, accelerates virtual-to-physical address mapping to avoid a full page table walk on every memory op. TLB fits only a **limited number of pages.**

A page which isn't present or valid causes a **page fault.** The OS handles these, e.g. code page is read in from a file on disk on first use. Some page table changes force a synchronous update on all processors ("TLB shootdown").



Typical CPU Architecture Today

2-6 **cores** per die, 1-2 dies per package, 1-N packages per system.

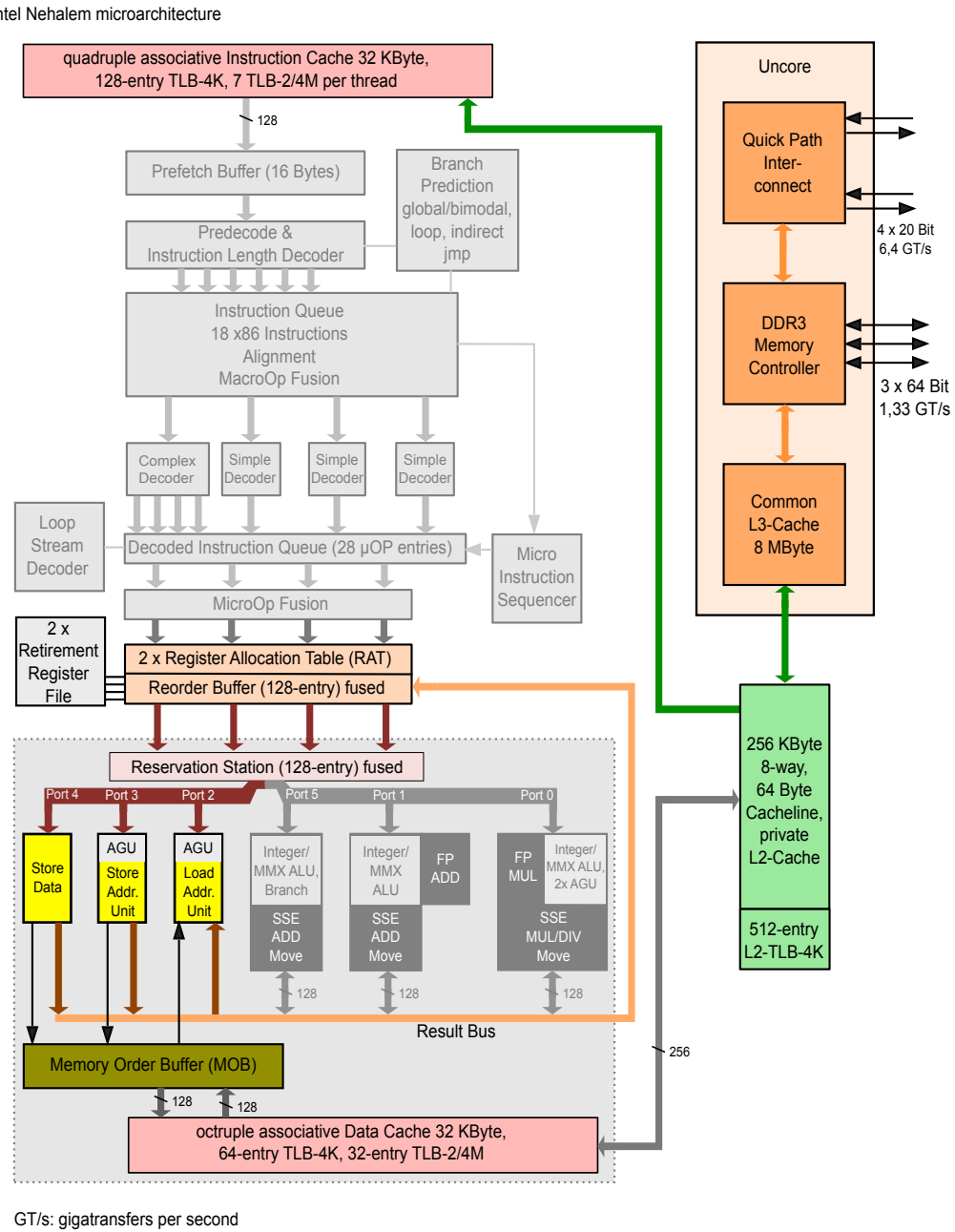
3 levels of **cache**

- Small [32kB] separate L1 I+D caches for each core.
- Medium [256kB-3MB] combined L2 cache, perhaps shared among some cores.
- Large [4-16MB] combined L3 cache shared between all cores on die.

2-3 **channels** to DDR2 or DDR3, 2-3 DIMMs per DDR channel, or up to 8 FB-DRAM DIMMs per channel; *practical performance varies **a lot** depending on how many DIMMs there are on the channels.*

High-speed **interconnect** to other CPUs: HyperTransport (AMD) or QPI (Intel). Cache snooping, cache tagging follow memory use in other packages.

[Wikipedia / Intel Nehalem / By "Appaloosa" / GFDL]



GT/s: gigatransfers per second

Typical Core Memory Architecture Today

Out-of-order, super-scalar, deep pipelines.

Significant capacity to reorder and buffer memory operations, will automatically prefetch several different access patterns.

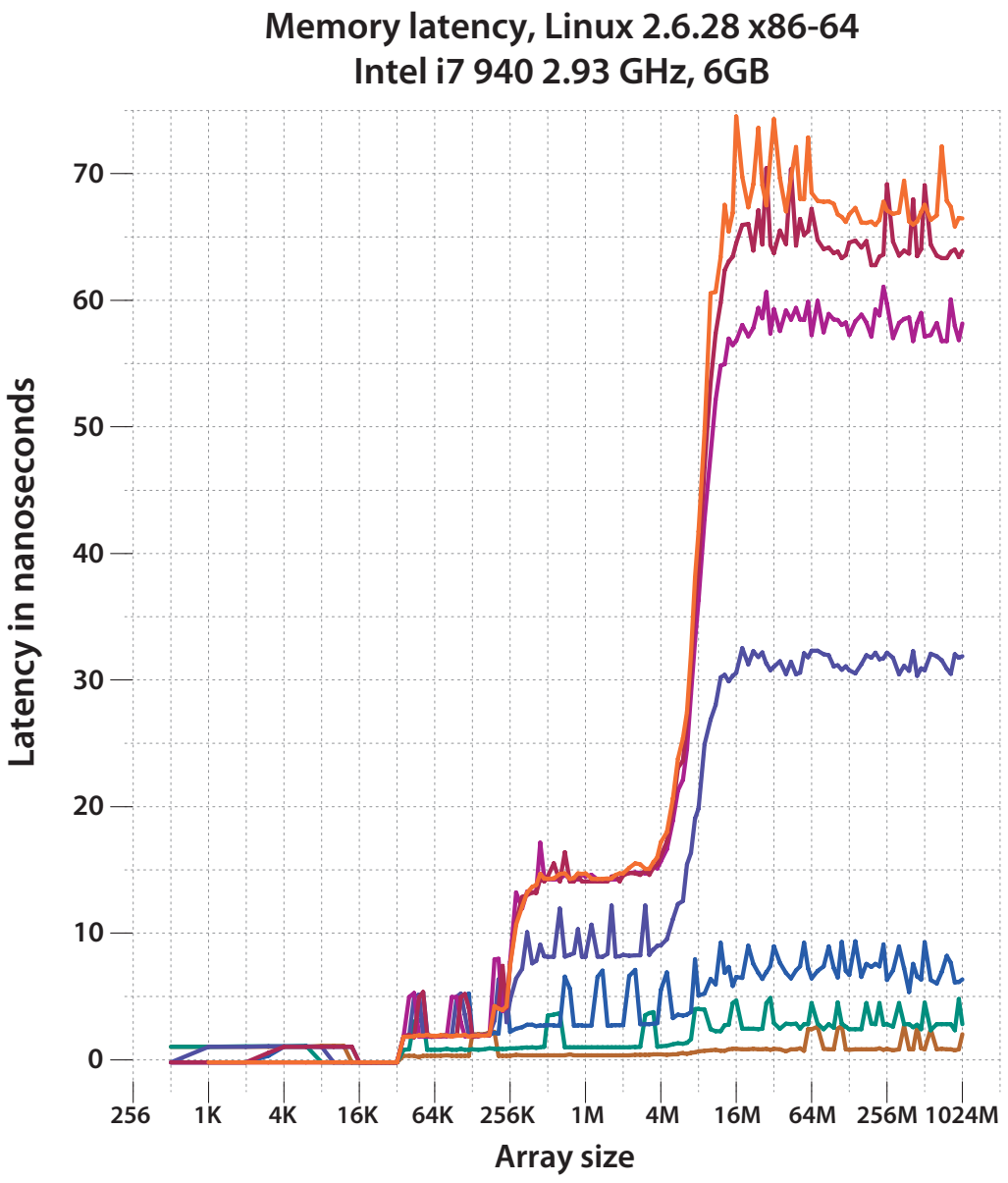
32kB L1I + L1D caches, 128-entry L1 ITLB, 64-entry L1 DTLB ≅ 512kB code, 256kB data addressing capacity.

512-entry L2 TLB ≅ 2MB code + data addressing capacity – less than fits in L3 cache, but more than one core share of L3.

All this exists to combat the **memory wall.**

BUT for all practical purposes a modern CPU performs well on **large data volumes** only if organised as **arrays-of-structures** (AoS) or **structures-of-arrays** (SoA) – pointer-rich "objects" *will* perform poorly.

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.

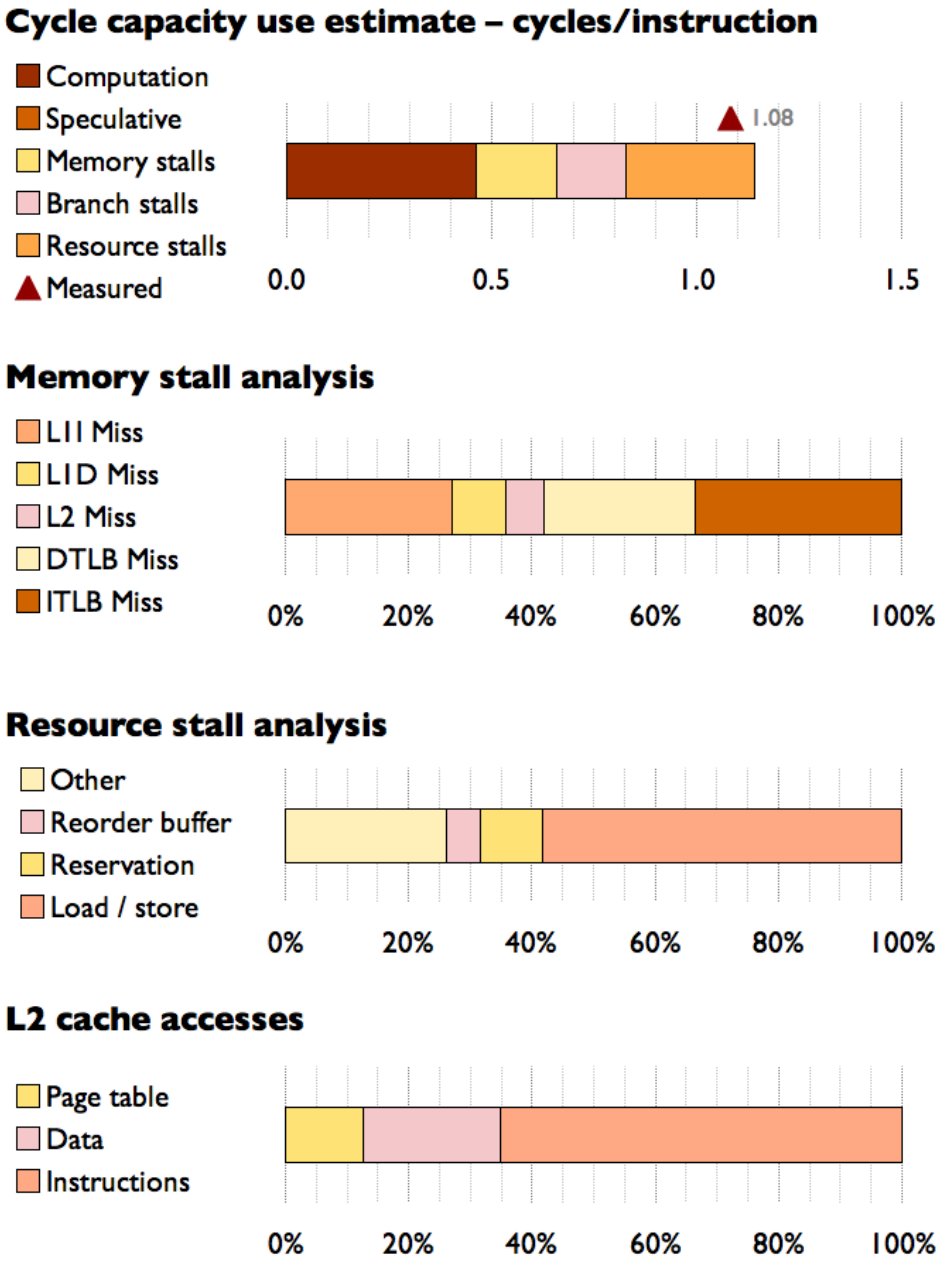
What happens when you drop to memory?

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 ~200 cycles (times any NUMA overhead if crossing interconnect).

Good cache efficiency matters.

[CMSSW on 4-core AMD Opteron 270, 2007]



A Typical Dilemma for Scientific C++ App

Relatively resource-rich CPU, 4-core AMD Opteron 270 from ~2007, but application is nowhere near compute bound.

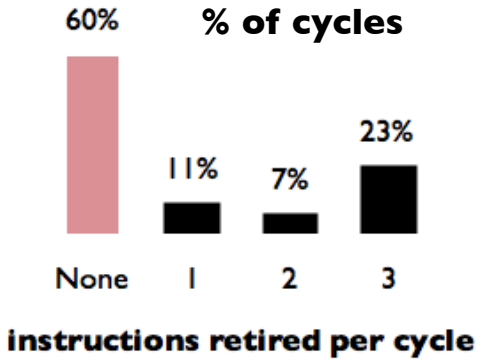
60% of clock cycles are completely stalled and do not retire a single instruction.

60% of memory stalls are for instructions.

60% of memory stalls are for page tables.

L2 cache accesses are dominated by ***code*** and ***page tables.***

Oops?



Starting Programs

$ cmsRun somecfg.py

OS creates a new process

- create and initialise a new address space, initial thread stack, command line args
- mmap code, data + other loadable segments from the main executable, dynamic linker (creating page tables)
- start thread in the dynamic linker

Dynamic linker finishes the start-up

- mmap code, data segments recursively from all shared library dependencies
- relocate position independent code, data
- invoke init sections, start executing

As process executes…

- page fault code, data in as needed

```
$ readelf -l cmsRun

Elf file type is EXEC (Executable file)
Entry point 0x80519f0
There are 8 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x1bbb3 0x1bbb3 R E 0x1000
  LOAD           0x01c000 0x08064000 0x08064000 0x00bdc 0x00c14 RW  0x1000
  DYNAMIC        0x01c01c 0x0806401c 0x0806401c 0x00208 0x00208 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00020 0x00020 R   0x4
  GNU_EH_FRAME   0x019360 0x08061360 0x08061360 0x002f4 0x002f4 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
```

```
$ readelf -d cmsRun

Dynamic section at offset 0x1c01c contains 60 entries:
  Tag        Type         Name/Value
  0x00000001 (NEEDED)     Shared library: [libFWCoreFramework.so]
  0x00000001 (NEEDED)     Shared library: [libFWCoreService...so]
  0x00000001 (NEEDED)     Shared library: [libFWCorePython...so]
  0x00000001 (NEEDED)     Shared library: [libDataFormatsCommon.so]
  0x00000001 (NEEDED)     Shared library: [libFWCoreParameter...so]
  0x00000001 (NEEDED)     Shared library: [libDataFormats...so]
  0x00000001 (NEEDED)     Shared library: [libFWCoreMessage...so]
  0x00000001 (NEEDED)     Shared library: [libFWCorePlugin...so]
  [...]
  0x0000000c (INIT)       0x8051278
  0x0000000d (FINI)       0x8060084
  0x00000004 (HASH)       0x8048168
  0x00000005 (STRTAB)     0x804a6d4
  0x00000006 (SYMTAB)     0x8048c34
  0x0000000a (STRSZ)      24813 (bytes)
  0x0000000b (SYMENT)     16 (bytes)
  0x00000015 (DEBUG)      0x0
  0x00000003 (PLTGOT)     0x806430c
  0x00000002 (PLTRELSZ)   936 (bytes)
  0x00000014 (PLTREL)     REL
  0x00000017 (JMPREL)     0x8050ed0
  0x00000011 (REL)        0x8050b88
  0x00000012 (RELSZ)      840 (bytes)
  0x00000013 (RELENT)     8 (bytes)
  0x6ffffffe (VERNEED)    0x8050b18
  0x6fffffff (VERNEEDNUM) 3
  0x6ffffff0 (VERSYM)     0x80507c2
  0x00000000 (NULL)       0x0
```

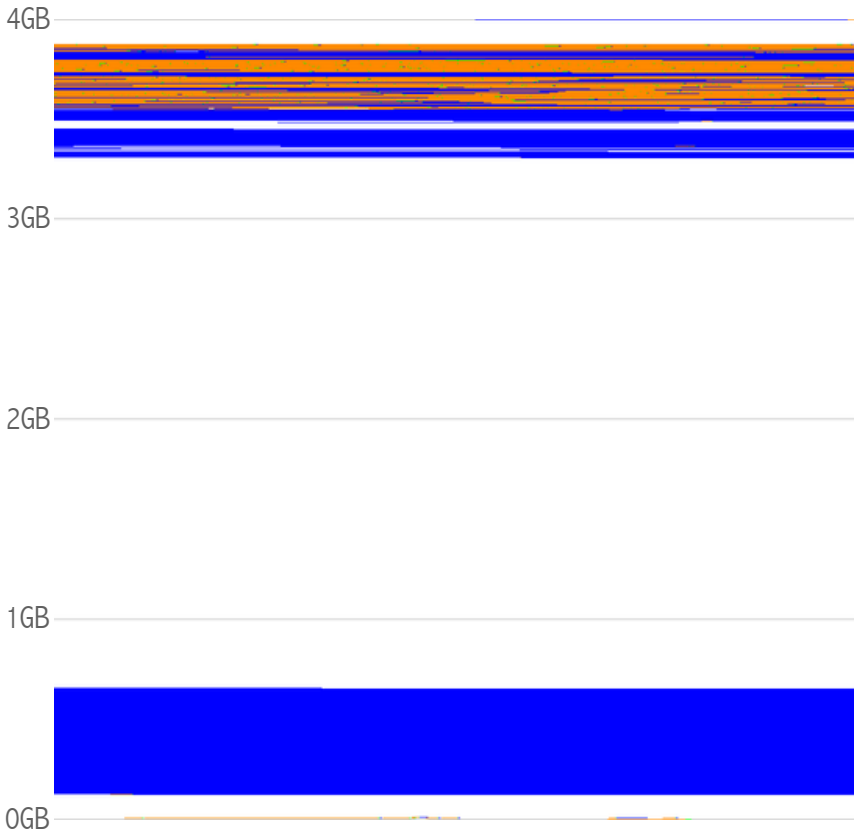
After a while…

Process has loaded even more code and has allocated quite a bit of heap space

- Invoked the dynamic linker to bring in even more shared libraries, each of which mmaped more code and data segments
- Called sbrk, mmap to acquire additional heap memory from the operating system

Result: 1060MB VSIZE, 850MB RSS, 600 libraries, 1370 memory regions

- Each shared library has separate code and data pages, which is **bad for virtual address space locality and stresses TLB**
- Random scatter of mapped library pages (a security feature) × lots of libraries = dense address map with many holes = **fragmented address space and heap**
- This produced 2.3MB new page tables
- **Definitely not smart – dwarfs the capacity of even the latest hardware**



1024x1024 pixel image map of the address space of a 32-bit cmsRun process. Every pixel is one 4096B page. Orange = code, green = data, blue = heap, stack. Total VSIZE 1060MB of which 230 MB is code(!)

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.

In 32-bit, the 4GB address space is usually split 3:1 and sometimes 4:4 between the user space and the kernel. In 64-bit the split does not matter.

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

Memory is used for kernel itself and files in the **buffer cache.** Applications can share memory by referring to shared physical pages: just memory blocks, buffer cache regions, or special objects such as pipe memory with vmsplice(). Methods to share memory include **fork()**, **mmap()** or **shmget()**.

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

About Shared Memory

Shared memory is not special – it is completely natural and widely used on modern systems, with many ways to initiate sharing:

Calling **mmap() on a file in multiple processes** can be used to create shared read-only or read-write mappings, on any file region. Example: shared library **position independent code**. One way to share static read-only data is to wrap and load it as a shared library. Suitable use of mmap() + {f,m}advise() can map windows of the OS buffer cache and provide hints on future use.

Calling **fork() without exec()** makes copy-on-write shared memory of the entire process address space; writing to a page after fork() creates a private copy. One of the simplest ways to create **writeable transient shared memory** without file association is to use anonymous mmap() and then call fork().

It's also possible to create **persistent named shared memory** with shmget().

Pages can be **shuffled around** with vmsplice(), tee() and remap_file_pages().

C, C++ Run Time Memory Management

"C++: The power, elegance and simplicity of a hand grenade."

C/POSIX provides some very basic memory allocation primitives

malloc(); free(); realloc(); calloc(); memalign(); valloc(); alloca()

Various libraries provide alternatives, or higher-level managers

Some of the best alternatives: Google TCMalloc, FreeBSD jemalloc;
Managers: Boost Pool, Sun SLAB allocator + derivatives, SAMBA talloc, GNU obstacks

C++ provides partially incompatible allocation technology

operator new/delete; object constructors, destructors and copy constructors; standard library containers and allocator objects; smart pointers, etc.; does map easily on top of malloc + free, somewhat painfully on anything else

I will not comment here on other languages, e.g. java, c# or scripting languages like python.

Getting Hands Dirty: Logical Data Structures

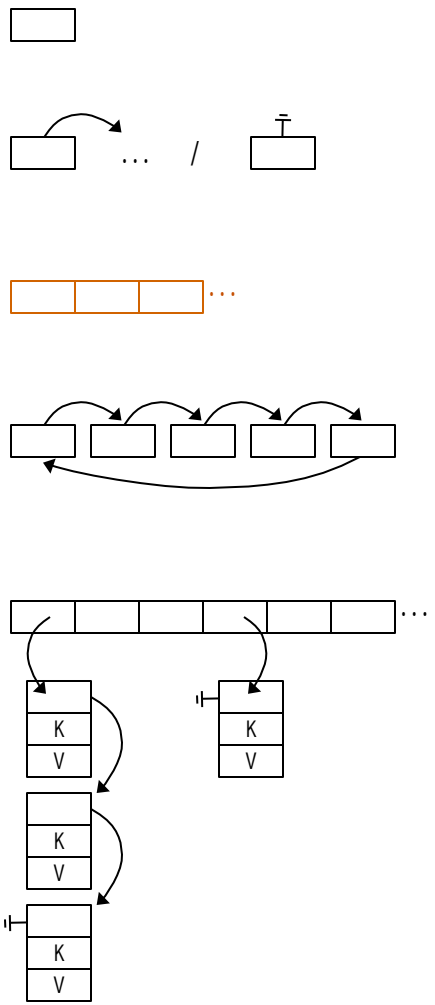
Scalar

Pointer

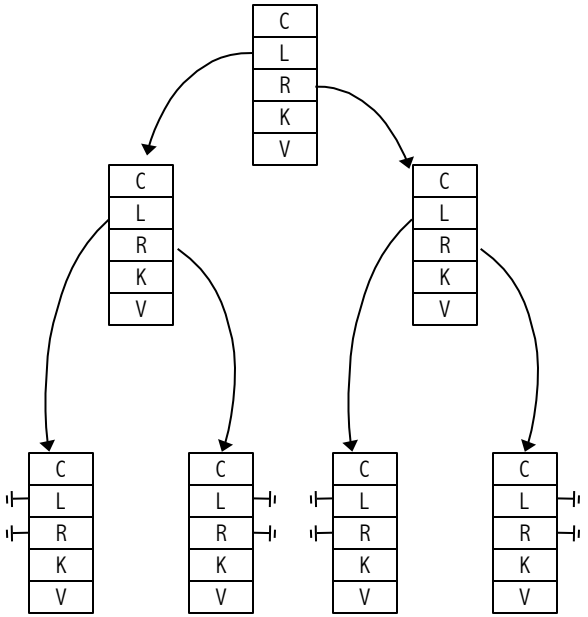
Structure / Array

Linked list

Hash



Balanced Binary Tree, e.g. Red-Black

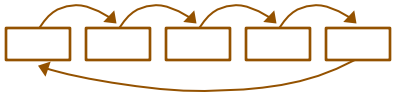


K = key, V = value, C = color, L = left, R = right

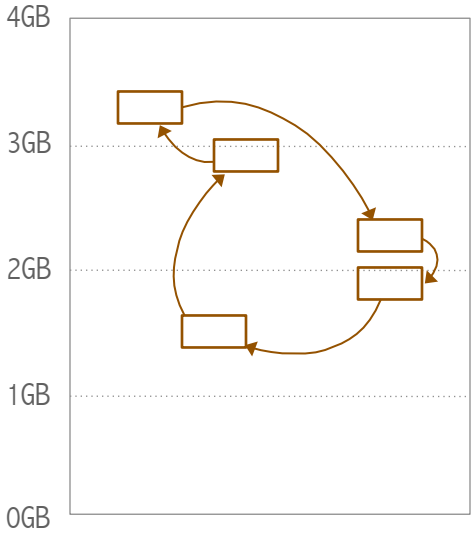
= by far the most efficient

Logical vs. Real Data Structures

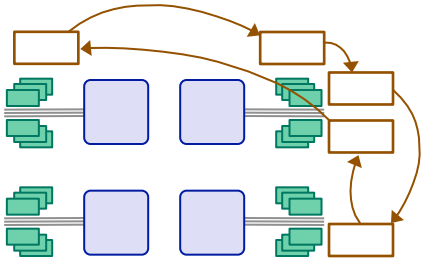
This logical linked list…



Could be scattered in virtual address space like this…



And in physical memory like this…

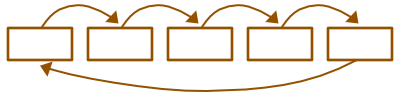


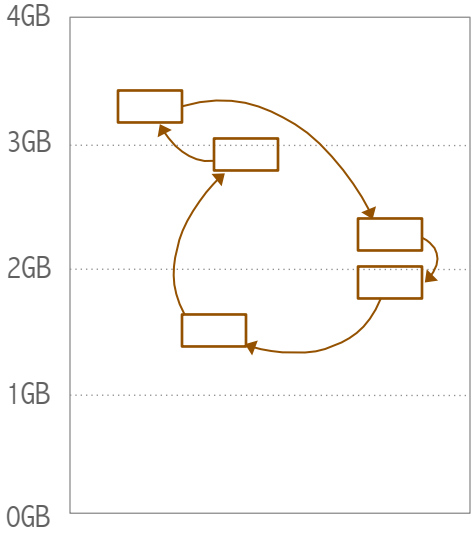
Logical vs. Real Data Structures

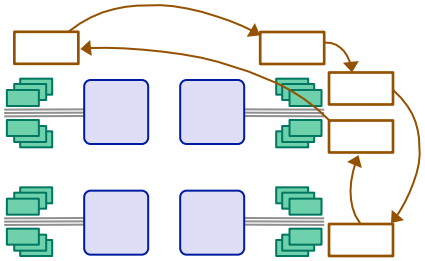
The scatter is unimportant as long as L*n* and TLB caches hide all latencies. Otherwise **you must explicitly arrange for** a better memory ordering.

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.

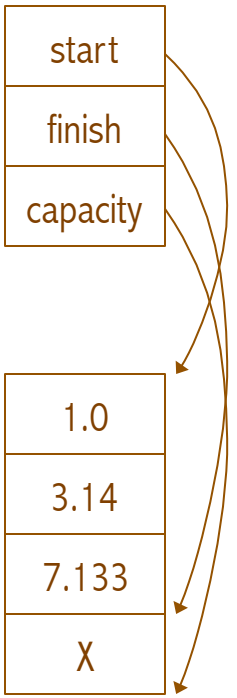






Getting Hands Dirty: C++ Types

std::vector<double>




```
std::vector<double> vec;
vec.reserve(4);
vec.push_back(1.0);
vec.push_back(3.14);
vec.push_back(7.133);
```

A good and efficient data structure in general.

- Good locality usually, guaranteed contiguous allocation.
- Avoid **small vectors** because of the **overhead**; more on this in a moment.
- **Beware** creating vectors incrementally without reserve(). Grows exponentially and copies old contents on every growth step if there isn't enough space!
- **Beware** making a copy, the dynamically allocated part is copied!
- **Beware** using erase(), it also causes incremental copying.

Getting Hands Dirty: C++ Types

std::vector<std::vector<std::vector<int>>>

```
typedef std::vector<int> VI;
typedef std::vector<VI> VVI;
std::vector<VVI> vvvi;
for (int i = 0, j, k; i < 10; ++i)
  for (vvvi.push_back(VVI()), j = 0; j < 10; ++j)
    for (vvvi.back().push_back(VI()), k = 0; k < 10; ++k)
      vvvi.back().back().push_back(k);
```

A very common mistake. C++ vectors of vectors are expensive, and **not** contiguous matrices. Let's measure just how lethal this nested containment by value combined with incremental growth is.

- Naively: 111 allocations, 5'320 bytes. (IA32; proper use of reserve() gets this.)
- Reality: 980 allocations, total 30'402 bytes allocated, 5'632 at end, 9'508 peak.
- +780% # allocs, +460% bytes alloc'd, 79% working and 6% residual overhead!
- *Versus 1 allocation, 4'440 bytes and some pointer setup had we used a real matrix.*

Getting Hands Dirty: C++ Types

std::vector<std::vector<std::vector<int>>>

```
std::vector<VVI> vvvi, vvvi2;
for (/* ... */) { /* ... */ }
vvvi2 = vvvi;
```

Why you should **avoid making container copies by value…**

- +111 allocations, +5'320 bytes (= naïve / full reserve() allocation).
- **An allocation storm is inevitable if you copy nested containers by value.** Evil bonus: **memory churn.** Because of the allocation/free pattern, by-value copies are an effective way to scatter the memory blocks all over the heap.
- "A nested container" does not have to be a standard library container. It can refer to any object type which makes an expensive deep copy – for instance almost any normal type with std::string, std::vector or std::map data members, or objects which "clone" pointed-to objects on copy.
- The simple "=" line may also generate *lots* of code.

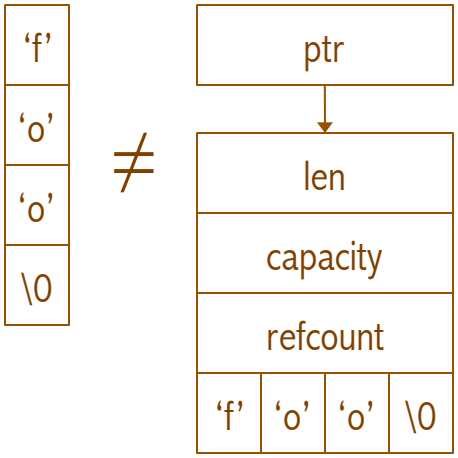
Getting Hands Dirty: C++ Types
std::vector<uint16_t> x

Typical std::vector<uint16_t> overhead is 40 bytes [64-bit system].

- 3 pointers × 8 bytes for vector itself, plus average 2 words × 8 bytes malloc() overhead for the dynamically allocated array data chunk.
- So, if x always has N ≤ 20 elements, it'd better to just use a uint16_t x[N].
- More generally, if 95+% of uses of x have only N elements for some small N, it may be better to have a uint16_t x[N] for the common case, and a separate dynamically allocated "overflow" buffer for the rare N = large case. Somewhat more complex code may be offset by reduction in overheads – measure to see!
- Even more generally, this applies to any small object allocated from heap. Examples abound in almost any large code base – at one point our software made many heap copies of 1-byte strings (yes, just the trailing null byte).

Getting Hands Dirty: C++ Types

const char * and std::string




```
const char array[] = "foo";
const char *ptr_to_literal = "foo";
std::string dynstr; dynstr.reserve(3);
dynstr += 'f'; dynstr += 'o'; dynstr += 'o';
```

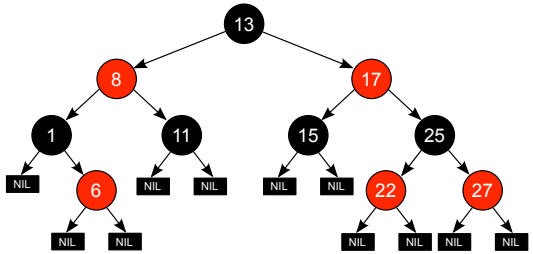
Character arrays are filled in by compiler – if local, at run-time. String literals are statically allocated by compiler and linker.

- It's **foolish** to copy string literals unmodified into std::strings – you store the same character data twice, once in .rodata, another time on heap. Avoid defining APIs taking a std::string if 99% of callers will use a string literal!

C++ std::string is a container much like std::vector<>.

- Same caveats apply. Even though strings *may* be reference-counted and copy-on-write, avoid relying on that extensively as consequences are usually awful.
- **Strings are highly overrated** and spread like rats through bad interfaces. Our apps have ~15% of dynamic std::string data, majority misguided use.

Getting Hands Dirty: C++ Types
std::map<std::string, X>



[Wikipedia / GFDL]

std::map<K,V> is a balanced binary tree, usually red-black tree

- Each tree node is a separately allocated [R/B, LeftPtr, RightPtr, Key, Value] tuple. Key comparison determines whether to follow left or right pointer. The **recursive pointer chasing is poison** to modern CPUs if data is not in cache.
- Since the map is a balanced binary tree, it has $\log_2(size)$ levels. If you have 1M entries in the map, it will take up to 20 key comparisons to find a match. If each key is a container such as std::string, every key comparison involves another pointer dereference, then key data match – for 1M entries, up to 40 pointer dereferences and up to 20 key comparisons before you get to data. If you fill the map slowly, the tree nodes and key and value data can be scattered all over virtual address space. **Avoid large maps and use inexpensive keys.**
- Beware temporaries in x["foo"] = abc(); x["foo"].call()! Beware code growth when using maps inside loops: for (…) { std::map<K,V> mymap; … }

Getting Hands Dirty: C++ Types

All C++ standard containers take an **allocator** template argument.

- Usually by default the containers just grab memory with operator new when they need something. This can lead to highly inefficient memory layouts.
- We are meant to use the template argument and constructor parameter to specify an alternate allocator, such as a pool allocator to improve locality. Pointer-rich containers (maps, lists) do need pool allocators for performance.
- Do be advised this is even more **invasive decision** than starting to use slabs, obstacks, talloc, or purpose-built areanas – it affects the *type*. In general the decision needs to be made early on, retrofitting custom allocators into a large code base is a significant effort.

I personally have rarely customised C++ allocators, mainly because it affects API types. I have used custom (= handwritten, non-std-like) allocators and containers extensively, with great benefit.

Getting Hands Dirty: C++ Types

Hey wait, aren't you going to talk about objects!?

Peak performance requires effective cache use for low latency. **How that is achieved is less important.** Understanding the language mapping from high-level constructs to low-level behaviour helps.

With big data the answer tends to translate to hardware-aware and -friendly **Arrays of Structures** (AoS) and **Structures of Arrays** (SoA) organisation, e.g. partitioning problem so it fits in L1 cache, strides hardware can prefetch or is vectorisation-friendly. Cache-defeating pointer chasing will simply not work.

Based on what we know of future processor roadmaps, **the performance gap** between AoS/SoA and pointer chasing data structures **will only stay or grow bigger.** If streaming units get prominent, code locality will also matter more.

Pointer-rich "proper objects" do remain immensely useful – as long as caches are used very effectively, or performance simply doesn't matter, for example in GUIs, support data structures and rarely used infrastructure.

Key Factor #1: Correctness

VALGRIND is one of the most valuable tools to verify correctness of any memory related operations. It **will save you hours of work.**

It's not a toy – it's one of the most useful software developer tools I have ever used. Always verify your **regression test suite** under valgrind; if nothing is flagged there's reasonable chance there are no silent memory access problems.

Any time you run into a problem, and certainly if you have a memory fault such as a segmentation violation, run the program under valgrind.

It will also provide useful leak data. It's very slow just for that however.

The same suite has other tremendously useful associated tools.

HELGRIND for finding multi-threaded data races, **MASSIF** for generating run time heap snapshot profiles and **CACHEGRIND** for CPU simulation.

Key Factor #1: Correctness

IgProf profiling suite is complementary to the Valgrind family.

IgProf can profile memory allocations, and can report the full stack trace for every allocated memory block. It's particularly useful for detecting **leaks,** generating run-time **heap snapshots**, and generally **tracking memory use.**

Recommended use: check correctness with Valgrind, then use IgProf to create heap profiles, in particular to identify leaks. IgProf has much less overhead than Valgrind (50-100% vs 1000%), but assumes correctness.

Memory leaks come in broadly two flavours: **unreachable** but still allocated, and **accumulated reachable garbage.**

Unreachable memory is created by *forgetting to free data past last reference.* In C++ it is usually a sign of fairly poor object ownership design – see talloc for ideas. Accumulated garbage happens when *object lifetime extends long beyond the time the object is needed.* Fattens virtual memory use and slows apps down.

Combating Memory Leaks

#1: Design clear object ownership – it won't just happen!

The most common reason for leaks is developers don't know who owns the object or how long it will be live. Most likely to happen at API boundaries. Design clear ownership rules; see for example talloc library. [Causes knock-on issues: developers copy objects when they don't know who owns them.]

#2: Use RAII idiom where possible *(Resource Acquisition Is Initialisation)*

The owner object will release resources when destructed. Numerous idioms. **A)** Prefer **memory pools** when you can define en-masse clear ownership; **B)** Use **by-value containers** – std::string, std::vector; **C)** Use **reference counting** smart pointers – std::auto_ptr, boost::intrusive_ptr, boost::shared_ptr; good for internal use, be cautious of using them in APIs: prefer #1 over #2.

#3: Proactively verify correctness using leak detection tools

Key Factor #2: Memory Churn

Memory churn is excessive reliance on dynamic heap allocation, usually in the form of numerous very short-lived allocations.

Every HEP C++ application I have looked at has suffered from **extreme memory churn.** Our software performs 1M memory allocations per second *on average, over hours of running*. That's a malloc() + free() every ~2500 cycles!

Memory churn has several highly undesirable side effects.

Time is spent in memory management, not in your algorithms. We've seen up to 40% in malloc()+free(); 10%+ is a strong sign of bad problems.

Tends to cause **poor heap locality** and to **increase heap fragmentation.** Churn on large allocations can cause **frequent, costly page table updates.**

Contaminates I, D and TLB caches with memory management code and data structures. CPU performance counter profiling less useful because the caches will seem to perform *extremely well* – they just contain the wrong data.

Combating Memory Churn

Eliminating churn tends to **yield big gains** – *x*10 is not unusual.

Unless the code suffers from even greater algorithmic flaws, **memory churn tends to mask any other properties,** rendering other profiling ineffective.

Detecting memory churn is relatively easy: **memory use profiling,** such as IgProf MEM_TOTAL stats, tends to flag the problems almost trivially.

Solving memory churn varies from trivial to very hard.

Easy to fix **mistakes** like passing / returning containers by value, std::vector push_back()/erase(), containers defined inside of loops rather than outside.

Maybe **caching,** a std::vector ("poor man's arena"), replacing local variable with a data member, or **a proper pool allocator** will provide sufficient relief.

Next hardest are changes to **specific common types,** e.g. replacing small heap-allocated matrix objects with compile-time sized array matrices.

By far the hardest is to address **systematic poor design** – code "thinks" too locally and you have to touch tens to hundreds of thousands of lines of code to cut string use or introduce new object ownership or pool / slab allocators.

Key Factor #3: Locality

Detecting, measuring and fixing poor locality: discussed extensively in other sessions this week and somewhat already in this one.

Using suitable pool allocators is known to help, but no easy-to-use analysis tools. You can try evaluate heap trashing and allocation size distribution to some extent with e.g. igprof heap snapshots, even GLIBC's memusage. In general the better your unit and regression test collection, the easier the job.

Do pay attention to excessive virtual memory use – code and data.

A good rule of thumb is the larger the process, the slower it gets, with a few well designed applications an exception to this. 200+ MB of machine code from 500+ shared libraries is usually just preposterously bad packaging and/or large-scale code bloat. Fix packaging, make big shared libraries only, use coverage testing to figure out what you really need, fix coding problems, if nothing else works, reorder binaries to separate "hot" and "cold" segments.

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).

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

Summary

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.

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.

There are tools out there which will reduce the mysteries a lot.

And we will try them out in the exercises part!