Efficient Memory Management

Giulio Eulisse - CERN

Original lectures by Lassi Tuura (FNAL, now Google)

About These Lectures

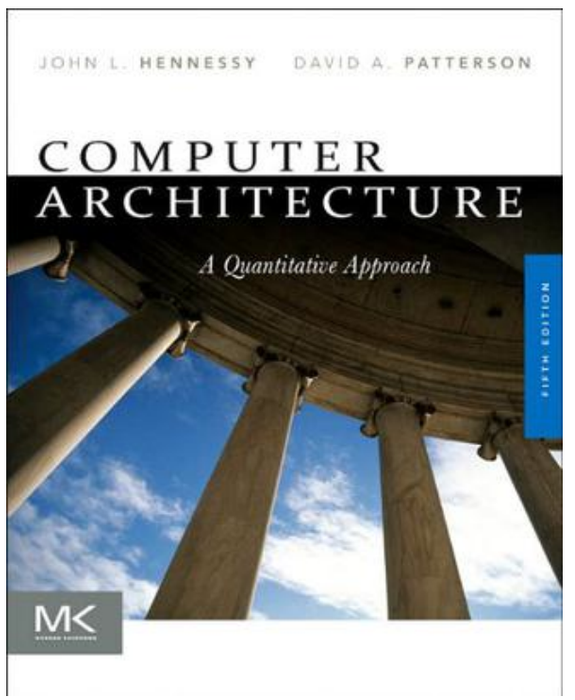
These lectures will address memory use and management in large scale scientific computing applications, with Linux/C++ focus.

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/esc15/memory

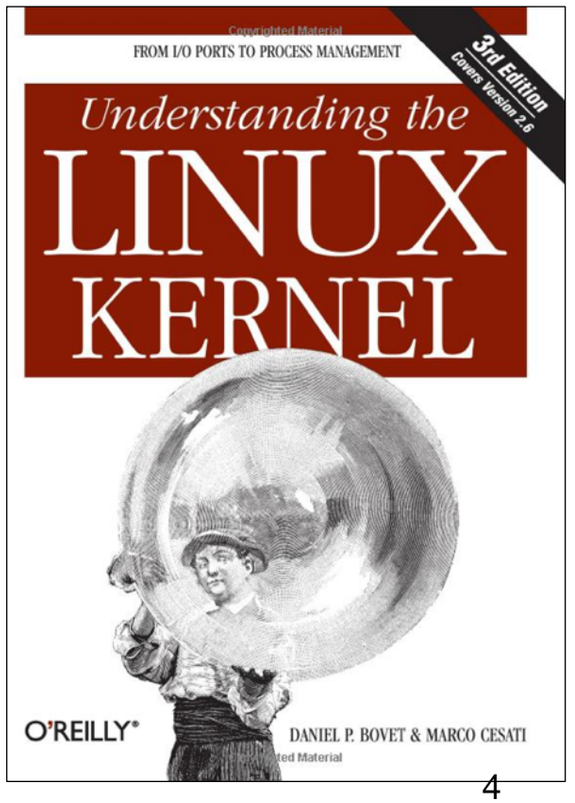
All the exercise material for these lectures

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

Why Memory Management Matters?

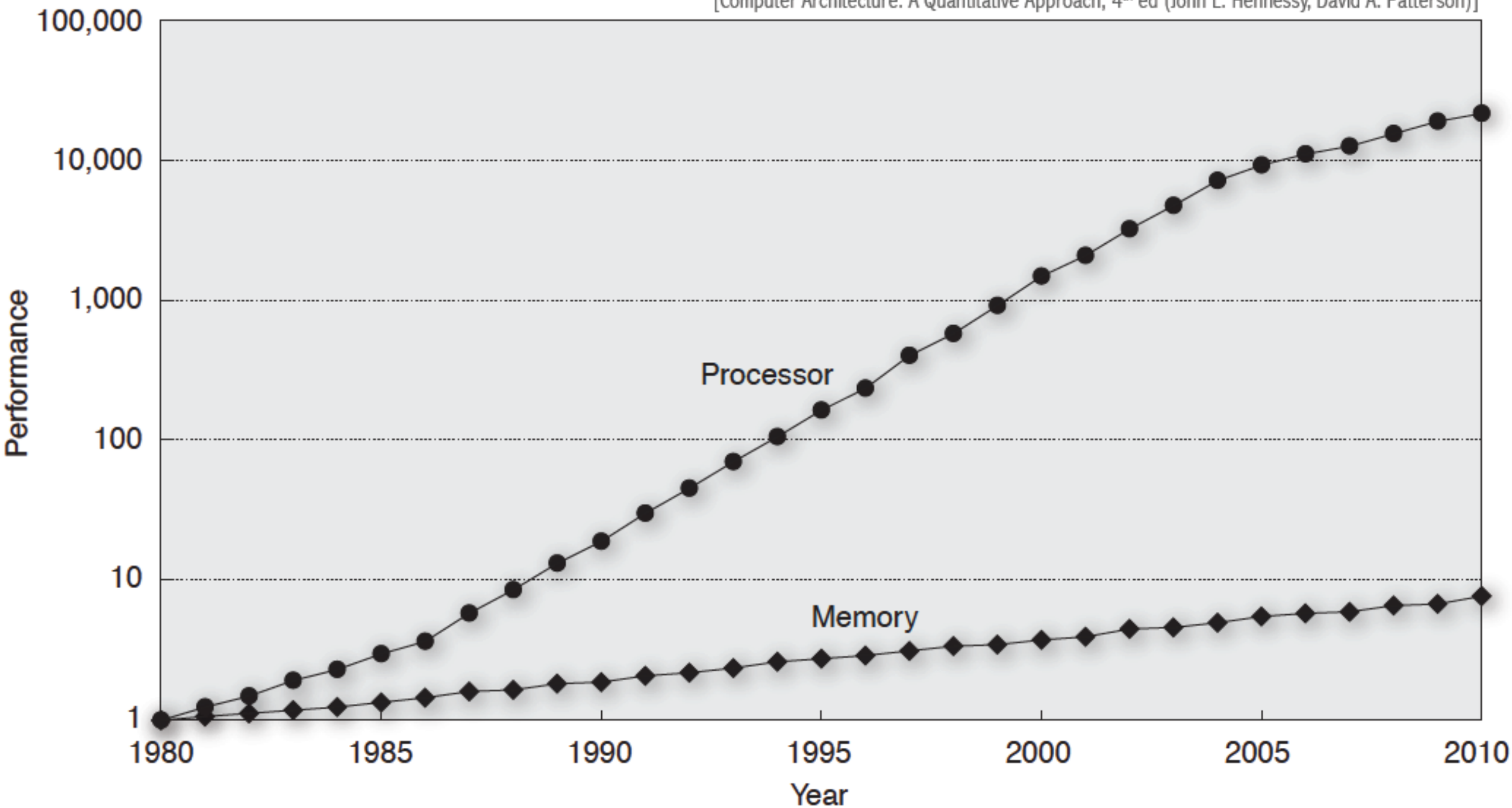
So, you've got a problem to solve. You've designed an algorithm to solve it. Now all you need is it code it up and you are done, right?

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.

The Performance Gap

[Computer Architecture: A Quantitative Approach, 4th ed (John L. Hennessy, David A. Patterson)]



Memory performance evolution compared with processor performance

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 practice! The small stuff still directly affects how much science you get out of your funding.*

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.

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 factor #1: correctness

Find the error:

```
#include <iostream>

int main(int argc, char **argv) {
  int *foo = new int[10];
  for (int i = 0; i <= 10; ++i)
    foo[i] = i;
}
```

Key factor #1: correctness

Find the error:

```
#include <iostream>

int main(int argc, char **argv) {
  int *foo = new int[10];
  for (int i = 0; i <= 10; ++i)
    foo[i] = i;
}
```

Key factor #1: correctness

- **Find the error:**

```
//_____________________________________________//
//                                             //
//    This macro reads ACORDE DDL Raw Data and  //
//    converts it into Digits                   //
//                                              //
//_____________________________________________ //


void ACORDERaw2Digits(Int_t nEvents = 1, char* fileName =
alien:///alice/data/2008/LHC08a_ACORDE/000016788/raw/080000167880l4.20.root")
{
  // Reads DDL data from fileName

  TStopwatch timer;
  timer.Start();

  TGrid::Connect("alien://");

  AliRawReader* rawReader = 0x0;
//  rawReader = new AliRawReaderFile(fileName); // DDL files
  rawReader = new AliRawReaderRoot(fileName); // DDL files

  AliACORDERawStream* rawStream  = new AliACORDERawStream(rawReader);
  for (Int_t i=0; i<nEvents; i++) {
    printf("========== EVENT  %d ==========\n",i);
    if (!rawReader->NextEvent())
      break;

    rawStream->Reset();
    if (!rawStream->Next())
      break;
    printf("Data size is %d\n",rawStream->DataSize());
    for (Int_t j=0; j<4; j++)
      printf(" %x",rawStream->GetWord(j));
    printf("\n");
  }

  delete rawReader;
  delete rawStream;

  timer.Stop();
  timer.Print();
}
```

```
//_____________________________________________//
//                                             //
//    This macro reads ACORDE DDL Raw Data and  //
//    converts it into Digits                   //
//                                              //
//_____________________________________________ //


void ACORDERaw2Digits(Int_t nEvents = 1, char* fileName = "rawdata.root")
{
  // Reads DDL data from fileName

  TStopwatch timer;
  timer.Start();

// Creates a TreeD to dump Digits

  AliRunLoader* rl = AliRunLoader::Open("galice.root");

  AliACORDELoader* loader = (AliACORDELoader*) rl->GetLoader("ACORDELoader");

  if(!loader) {
    AliError("no ACORDE loader found");
    return kFALSE; }

  TTree* treeD  = loader->TreeD();
  if(!treeD) {
      loader->MakeTree("D");
      treeD = loader->TreeD(); }

  AliACORDEdigit  digit;
  AliACORDEdigit* pdigit = &digit;
  const Int_t kBufferSize = 4000;

  treeD->Branch("ACORDE", "AliACORDEdigit",  &pdigit, kBufferSize);

  AliRawReader* rawReader = 0x0;
//  rawReader = new AliRawReaderFile(fileName); // DDL files
  rawReader = new AliRawReaderRoot(fileName); // DDL files

  AliACORDERawStream* rawStream  = new AliACORDERawStream(rawReader);
```

```
/**************************************************************************
 * Copyright(c) 1998-1999, ALICE Experiment at CERN, All rights reserved. *
 *                                                                        *
 * Author: The ALICE Off-line Project.                                    *
 * Contributors are mentioned in the code where appropriate.              *
 *                                                                        *
 * Permission to use, copy, modify and distribute this software and its   *
 * documentation strictly for non-commercial purposes is hereby granted   *
 * without fee, provided that the above copyright notice appears in all   *
 * copies and that both the copyright notice and this permission notice   *
 * appear in the supporting documentation. The authors make no claims     *
 * about the suitability of this software for any purpose. It is          *
 * provided "as is" without express or implied warranty.                  *
 **************************************************************************/



//---
//  Produces the data needed to calculate the quality assurance.
//  ACORDE QA for Hits, Digits, RAW and ESD's
//
//  Authors:
//  Luciano Diaz Gonzalez <luciano.diaz@nucleares.unam.mx> (ICN-UNAM)
//  Mario Rodriguez Cahuantzi <mrodrigu@mail.cern.ch> (FCFM-BUAP)
//  Arturo Fernandez Tellez <afernan@mail.cern.ch (FCFM-BUAP)
//
//  Created: June 13th 2008
//---

// --- ROOT system ---
#include <TClonesArray.h>
#include <TFile.h>
#include <TH1F.h>
#include <TH2F.h>
#include <TDirectory.h>
#include <TObject.h>
// --- Standard library ---

// --- AliRoot header files ---
#include "AliESDEvent.h"
#include "AliLog.h"
#include "AliACORDEdigit.h"
#include "AliACORDEhit.h"
#include "AliACORDERecPoint.h"
#include "AliACORDEQADataMaker.h"
#include "AliQAChecker.h"
#include "AliACORDERawReader.h"
#include "AliACORDERawStream.h"

ClassImp(AliACORDEQADataMaker)

//____________________________________________________________________
AliACORDEQADataMaker::AliACORDEQADataMaker():AliQADataMaker(AliQAv1::GetDetName(AliQAv1
:kACORDE), "ACORDE Quality Assurance Data Maker")
{
                    // Acorde QA Data Maker
}

//____________________________________________________________________
AliACORDEQADataMaker::AliACORDEQADataMaker(const AliACORDEQADataMaker& qadm):AliQADataMaker()
{
  SetName((const char*)qadm.GetName()) ;
  SetTitle((const char*)qadm.GetTitle());
}
//____________________________________________________________________
AliACORDEQADataMaker& AliACORDEQADataMaker::operator = (const AliACORDEQADataMaker& qadm )
{
  // Equal operator.
  this->~AliACORDEQADataMaker();
  new(this) AliACORDEQADataMaker(qadm);
  return *this;
}
//____________________________________________________________________
void AliACORDEQADataMaker::StartOfDetectorCycle()
{
  //Detector specific actions at start of cycle

}

//____________________________________________________________________
void AliACORDEQADataMaker::InitHits()
{

// create Hits histograms in Hits subdir
       TH1F *fAHitsACORDE[8];
```

```
void MakeACORDEFullMisAlignment(){
 // Create TClonesArray of full misalignment objects for ACORDE
 //
 const char* macroname = "MakeACORDEFullMisAlignment.C";
 // Activate CDB storage and load geometry from CDB
 AliCDBManager* cdb = AliCDBManager::Instance();
 if(!cdb->IsDefaultStorageSet()) cdb->SetDefaultStorage("local://$ALICE_ROOT/OCDB");
 cdb->SetRun(0);

 AliCDBStorage* storage;

 //load geom from local file till ACORDE is not switched on by default in standard config-files
 if( TString(gSystem->Getenv("TOCDB")) == TString("kTRUE") ){
  TString Storage = gSystem->Getenv("STORAGE");
  if(!Storage.BeginsWith("local://") && !Storage.BeginsWith("alien://")) {
   Error(macroname,"STORAGE variable set to %s is not valid. Exiting\n",Storage.Data());
   return;
  }
  storage = cdb->GetStorage(Storage.Data());
  if(!storage){
   Error(macroname,"Unable to open storage %s\n",Storage.Data());
   return;
  }
  AliCDBPath path("GRP","Geometry","Data");
  AliCDBEntry *entry = storage->Get(path.GetPath(),cdb->GetRun());
  if(!entry) Fatal(macroname,"Could not get the specified CDB entry!");
  entry->SetOwner(0);
  TGeoManager* geom = (TGeoManager*) entry->GetObject();
  AliGeomManager::SetGeometry(geom);
 }else{
  AliGeomManager::LoadGeometry(); //load geom from default CDB storage

 }
 //  AliGeomManager::LoadGeometry("geometry.root");

 TClonesArray *array = new TClonesArray("AliAlignObjParams",60);
 TClonesArray &alobj = *array;

 TRandom *rnd = new TRandom(4321);
 Int_t j = 0;
 Double_t dx, dy, dz, dpsi, dtheta, dphi;

 // RS = local
 // sigma translation = 1 mm
 // sigma rotation = 0.5 degrees
 Double_t sigmatr = 2;
 Double_t sigmarot = 1;

 TString symname;
 TString basename = "ACORDE/Array";
 Int_t iIndex=0;
 AliGeomManager::ELayerID iLayer = AliGeomManager::kInvalidLayer;
 UShort_t volid = AliGeomManager::LayerToVolUID(iLayer,iIndex);

 for (Int_t imod=0; imod<60; imod++){
  dx = rnd->Gaus(0.,sigmatr);
  dy = rnd->Gaus(0.,
  dz = rnd->Gaus(0.,
  dpsi = rnd->Gaus(
  dtheta = rnd->Gau
  dphi = rnd->Gaus(
  symname = basena
  symname += imod
  new(alobj[j++]) Al
 }
 if( TString(gSystem
  // save on file
  const char* filena
  TFile f(filename,"R
  if(!f){
   Error(macroname
   return;
  }
  Info(macroname,"
  f.cd();
  f.WriteObject(arra
  f.Close();
 }else{
  // save in CDB stora
  AliCDBMetaData*
  md->SetResponsib
  md->SetComment(
  md->SetAliRootVe
  AliCDBId id("ACOR
  storage->Put(arra
```

```
#ifndef ALIACORDEQADATAMAKER_H
#define ALIACORDEQADATAMAKER_H
/* Copyright(c) 1998-1999, ALICE Experiment at CERN, All rights reserved. *
 * See cxx source for full Copyright notice                               */

/* $Id: AliACORDEQADataMaker.h 25659 2008-05-030 15:13:46Z ldg $ */

//  Produces the data needed to calculate the quality assurance.
//  ACORDE QA for Hits, Digits, RAW and ESD's

//  Authors:
//
//  Luciano Diaz Gonzalez <luciano.diaz@nucleares.unam.mx> (ICN-UNAM)
//  Mario Rodriguez Cahuantzi <mrodrigu@mail.cern.ch> (FCFM-BUAP)
//  Arturo Fernandez Tellez <afernan@mail.cern.ch (FCFM-BUAP)
//
//  Created: June 13th 2008
//---

#include "AliQADataMaker.h"

class AliACORDEQADataMaker: public AliQADataMaker {

public:
  AliACORDEQADataMaker() ;          // constructor
  AliACORDEQADataMaker(const AliACORDEQADataMaker& qadm) ;
  AliACORDEQADataMaker& operator = (const AliACORDEQADataMaker& qadm) ;
  virtual ~AliACORDEQADataMaker() {;} // destructor

private:
  virtual void   InitHits() ;      //book hit QA histo
  virtual void   InitDigits() ;    //book Digit QA histo
  virtual void   InitRaws() ;    //book Digit QA histo
  virtual void   InitRecPoints();  //book cluster QA histo
  virtual void   InitESDs() ;      //book ESD QA histo
  virtual void   MakeHits(TTree * hits) ;       //Fill hit QA histo
  virtual void   MakeRaws(AliRawReader* rawReader) ;
  virtual void   MakeDigits(TTree* digitsTree) ;   //Fill Digit QA histo
  virtual void   MakeRecPoints(TTree * clusters)    ;  //Fill cluster QA histo
  virtual void   MakeESDs(AliESDEvent * esd) ;         //Fill hit QA histo

/*******/

  virtual Int_t  Add2DigitsList(TH1*, Int_t){return 0;};
  virtual Int_t  Add2HitsList(TH1*, Int_t){return 0;};
  virtual Int_t  Add2RecPointsList(TH1*, Int_t){return 0;};
  virtual Int_t  Add2RawsList(TH1*, Int_t){return 0;};
  virtual Int_t  Add2SDigitsList(TH1*, Int_t){return 0;};
  virtual void   Exec(AliQAv1::TASKINDEX_t, TObject*){};
  virtual void   EndOfCycle(AliQAv1::TASKINDEX_t){};
  virtual Int_t  Add2ESDsList(TH1*, Int_t){return 0;};
  virtual TH1*   GetDigitsData(Int_t){return 0;};
```

```
void MakeACORDEFullMisAlignment(){
  // Create TClonesArray of full misalignment objects for ACORDE
  //
  const char* macroname = "MakeACORDEFullMisAlignment.C";
  // Activate CDB storage and load geometry from CDB
  AliCDBManager* cdb = AliCDBManager::Instance();
  if(!cdb->IsDefaultStorageSet()) cdb->SetDefaultStorage("local://$ALICE_ROOT/OCDB");
  cdb->SetRun(0);

  AliCDBStorage* storage;

  //load geom from local file till ACORDE is not switched on by default in standard config-files
  if( TString(gSystem->Getenv("TOCDB")) == TString("kTRUE") ){
    TString Storage = gSystem->Getenv("STORAGE");
    if(!Storage.BeginsWith("local://") && !Storage.BeginsWith("alien://")) {
      Error(macroname,"STORAGE variable set to %s is not valid. Exiting\n",Storage.Data());
      return;
    }
    storage = cdb->GetStorage(Storage.Data());
    if(!storage){
      Error(macroname,"Unable to open storage %s\n",Storage.Data());
      return;
    }
    AliCDBPath path("GRP","Geometry","Data");
    AliCDBEntry *entry = storage->Get(path.GetPath(),cdb->GetRun());
    if(!entry) Fatal(macroname,"Could not get the specified CDB entry!");
    entry->SetOwner(0);
    TGeoManager* geom = (TGeoManager*) entry->GetObject();
    AliGeomManager::SetGeometry(geom);
  }else{
    AliGeomManager::LoadGeometry(); //load geom from default CDB storage

  }
  //  AliGeomManager::LoadGeometry("geometry.root");

  TClonesArray *array = new TClonesArray("AliAlignObjParams",60);
  TClonesArray &alobj = *array;

  TRandom *rnd = new TRandom(4321);
  Int_t j = 0;
  Double_t dx, dy, dz, dpsi, dtheta, dphi;

  // RS = local
  // sigma translation = 1 mm
  // sigma rotation = 0.5 degrees
  Double_t sigmatr = 2;
  Double_t sigmarot = 1;

  TString symname;
  TString basename = "ACORDE/Array";
  Int_t iIndex=0;
  AliGeomManager::ELayerID iLayer = AliGeomManager::kInvalidLayer;
  UShort_t volid = AliGeomManager::LayerToVolUID(iLayer,iIndex);

  for (Int_t imod=0; imod<60; imod++){
    dx = rnd->Gaus(0.,sigmatr);
```

ALICE Experiment Offline Software:
~5M lines of code
CMS Experiment Offline Software:
~8M lines of code

Key factor #1: correctness

Tools matter:

- **Valgrind**
- **IgProf**
- **Address Sanitizer**
- **Clang Static Analyzer**
- **GDB, strace, LD_DEBUG & /proc <pid>**

Real world scenarios is not something which can be captured on a slide, in a tutorial or in half an hour exercise. Mastering debug and profiling tools is as crucial as knowing your Algorithm Book, book to cover.

We need tools to help us dealing with large, mostly unfamiliar, codebases and to find our errors.

Key Factor #1: Correctness

VALGRIND…

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 memory leak data. It's very slow just for that however.*

… and friends

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: tools to enforce correctness

Address Sanitizer

When recompiling buggy code is an option, some compilers (like GCC and Clang) allow to add extra runtime instrumentation which will catch errors that can elude also Valgrind checks.

Clang Static Analizer

Due to C / C++ being very "liberal" languages, compilers do not flag bad practices even if they are usually not correct. At the cost of false positives, Static Analyzers (like clang-static-analyser) can detect possibly fatal bad behaviours.

GDB, strace, LD_DEBUG, /proc/<pid>

There is a number of system features / tools which can be extremely helpful when debugging memory problems. If none of the above ring a bell, make sure you google for them and become familiar.

Key Factor #1: Correctness

IgProf

The IgProf profiling suite is complimentary 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.

Instruments (Mac only), Google Perf Tools..

There is a number of tools out there which we will not cover, each with its strengths and deficiencies. While we will limit our exercises to those that I personally use, that does not mean there is no alternatives.

Key Factor #1: Memory Leaks

Unreachable but still allocated:

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 reachable garbage

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

Exercise

Catching Memory Errors

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.*

Know your enemies: simple types

The first thing to do when dealing with memory is to know the size of the objects you are dealing with.

Type	*Memory used (B)*
1 bit	
char	
short	
int	
long / int64_t	
int *	

Know your enemies: simple types

The first thing to do when dealing with memory is to know the size of the objects you are dealing with.

Type	*Memory used (B)*
1 bit	1
char	1
short	2
int	4
long / int64_t	8
int *	8 (64-bit)

Know your enemies: simple types

The first thing to do when dealing with memory is to know the size of the objects you are dealing with.

Type	*Memory used (B)*
1 bit	1
char	1
short	2
int	4
long / int64_t	8
int *	8 (64-bit)

Know your enemies: simple types

This is a first (simple) example on how "on paper" behaviour and hardware actually can be different.

Type	*Memory used (B)*
1 bit	1
8 bits	1
char	1
short	2
int	4
long / int64_t	8
int *	8 (64-bit)

Know your enemies: padding

How much memory is occupied by the following struct?

```
struct A {
  char a;
  int b;
  char c;
};
```

?

a. 6 bytes

b. 8 bytes

c. 12 bytes

d. 16 bytes

Know your enemies: padding

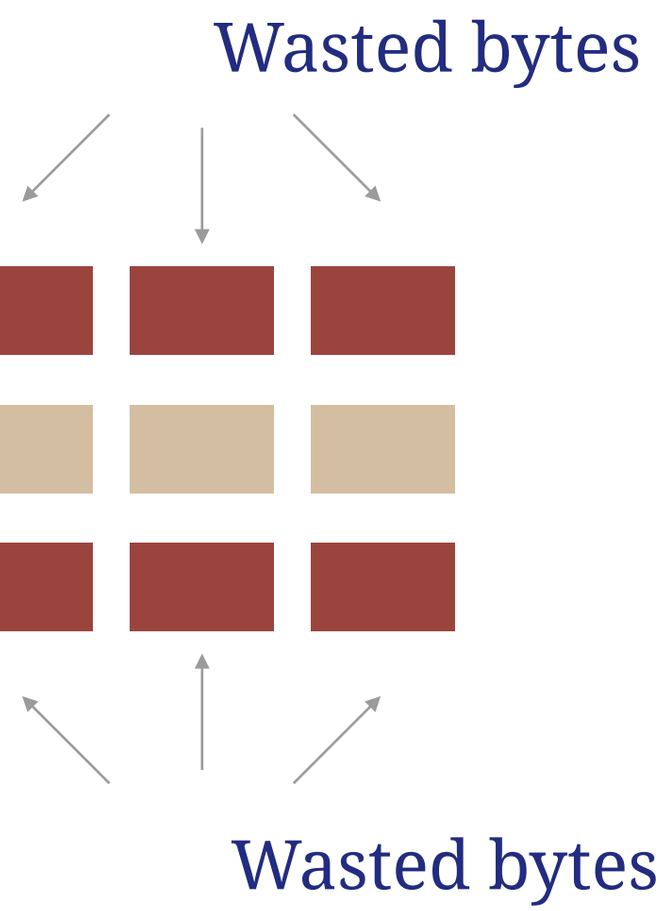
6 bytes come from the actual member sizes.

```
struct A {
  char a;
  int b;
  char c;
};
```

Know your enemies: padding

Another 6 bytes come from the required padding.

```
struct A {
  char a;
  int b;
  char c;
};
```

Know your enemies: padding

A much better layout…

```
struct A {
  int b;
  char a;
  char c;
};
```


Only two bytes wasted

*By default, C / C++ imposes **alignment** of data members within a struct / class to their size.*

Know your enemies: padding

End of the story?

```
struct A {
  int b;
  char a;
  char c;
};

A *a = new A;
```

How much memory is used by the above construct?

Know your enemies: padding

End of the story?

```
struct A {
  int b;
  char a;
  char c;
};

A *a = new A;
```

8 bytes for the data itself

8 bytes pointer to the object

~16 extra bytes for alignment of the structure on the heap and extra heap book-keeping.

Whenever dealing with small structures always think about the overhead introduced by the required book-keeping!

C, C++ Run Time Memory Management

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

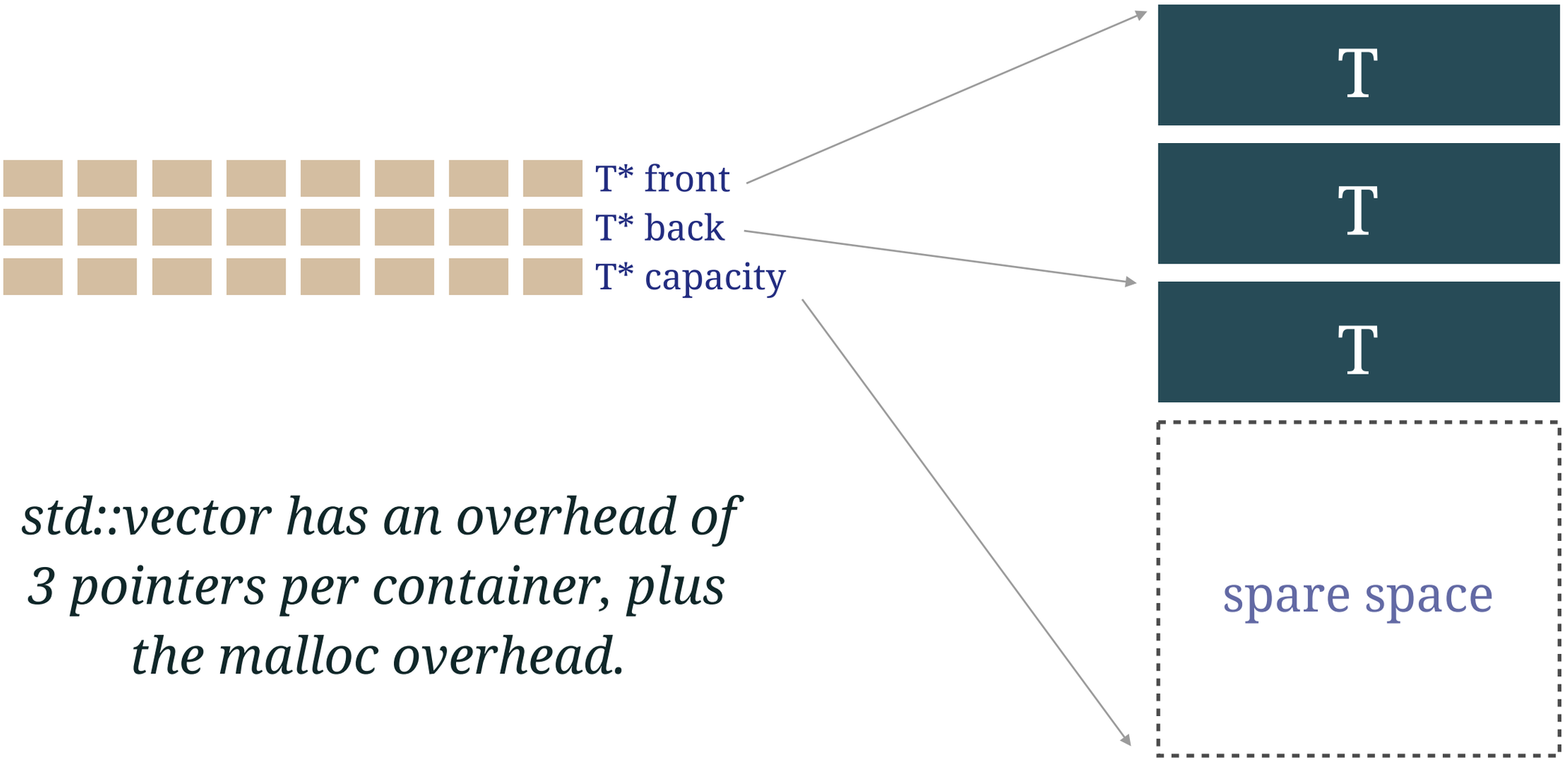
operator new/delete; object constructors, destructors and copy constructors; standard library containers and allocator objects; smart pointers, etc.; does map easily on top of malloc + free, somewhat painfully on anything else

"C++: the power and elegance of a hand grenade"

STL containers are great implementations of abstract concepts, but they come with a cost. Knowing that cost is key to be able to use them efficiently.

Know your enemies: std::vector

std::vector<T>: a compact, variable size, collection of objects of type T.



std::vector has an overhead of 3 pointers per container, plus the malloc overhead.

Know your enemies: std::vector

A good and efficient data structure in general.

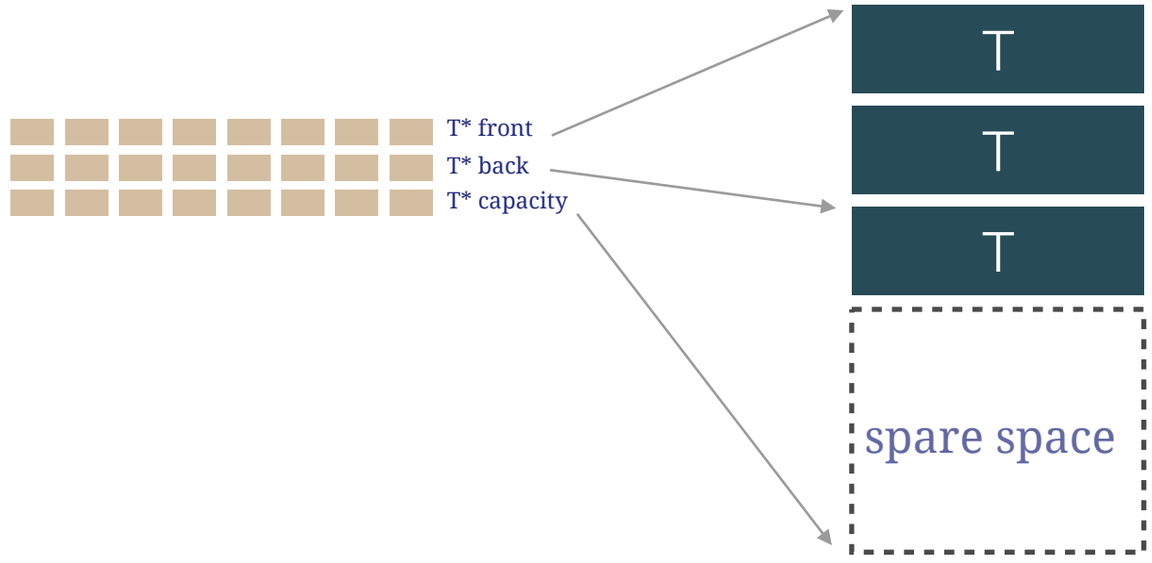
Good locality usually, guaranteed contiguous allocation.

Avoid small vectors *because of the overhead.*

Beware creating vectors incrementally *without reserve(). Grows exponentially and copies old contents on every growth step if there isn't enough space!*

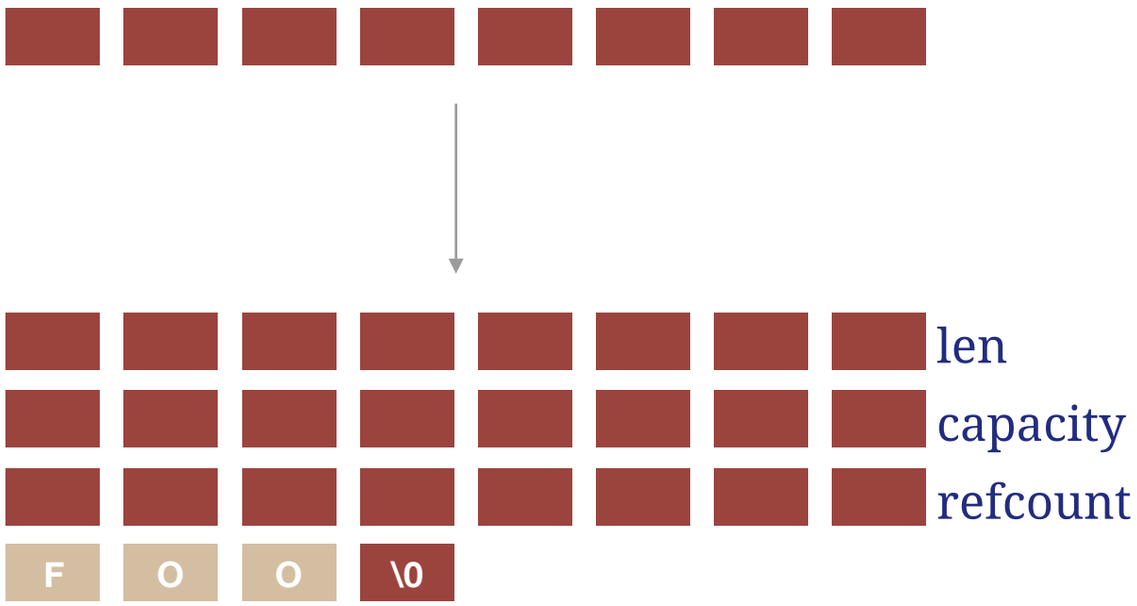
Beware making a copy*, if the dynamically allocated part is copied!*

Beware using erase(), it also causes incremental copying.



Know your enemies: std::string

std::string s("foo");



char *s[] = "foo";

*Strings are **highly overrated** and are most likely to point to bad design choices in your code. While they do have a use and can have smart(er) implementations, in general you should **think twice if you really need their dynamic behaviour** or if what you want is really a constant string literal.*

Know your enemies: std::vector<uint16_t>

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 ≤ 19 elements, it'd better to just use a std::array<uint16_t,N> x.*
- *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).*

Know your enemies: std::list

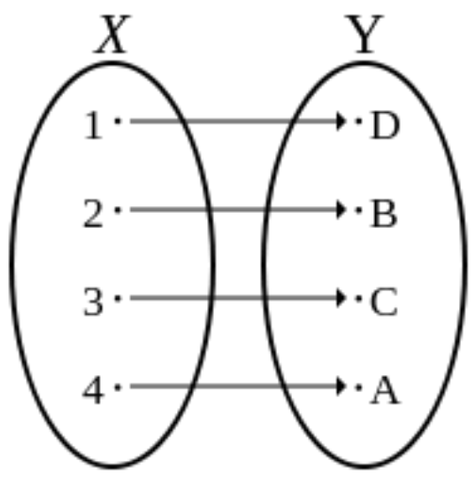
*Usually implemented as a **doubly linked list**. Usually, overhead of at least 2 pointers per item (implementation dependent).*

Here the cost is per item, not per container.

Moreover there is no guarantee that the objects will be close in memory potentially adding pressure to the memory subsystem.

Know your enemies: std::map<K,V>

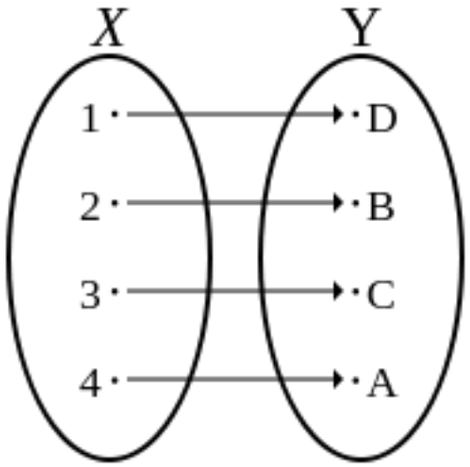
Maps are another extremely abused container, mostly because of their "intuitive" behaviour.



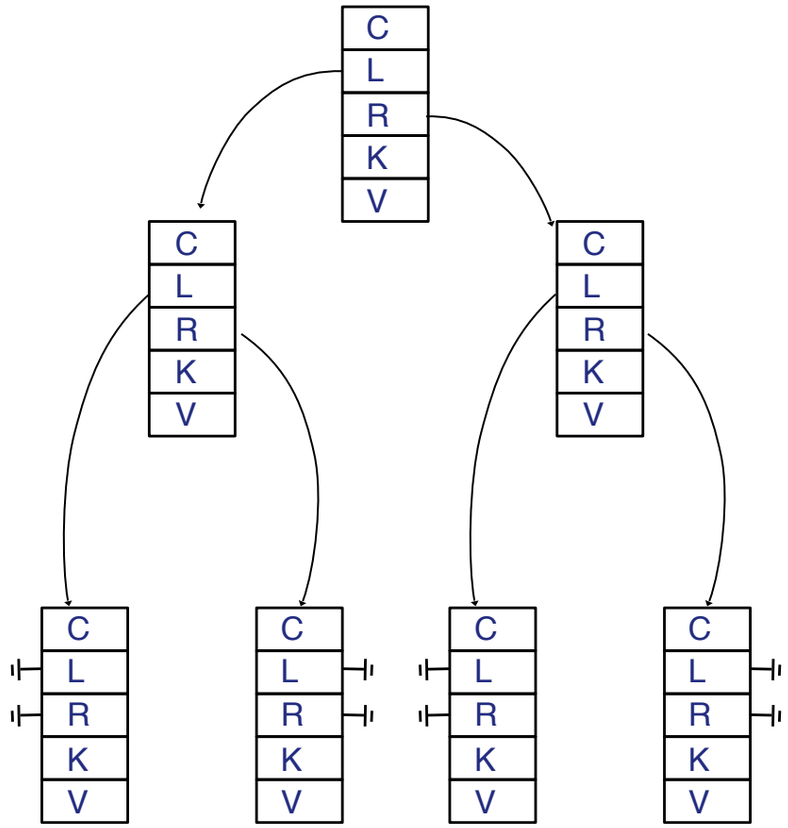
Platonic world

Know your enemies: std::map<K,V>

Maps are another extremely abused container, mostly because of their "intuitive" behaviour.



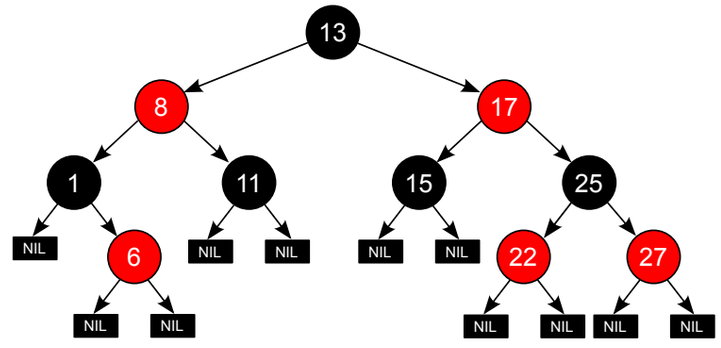
Platonic world



Reality: Balanced Binary Tree, e.g. Red-Black

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

std::map<K, V>



[Wikipedia / GFDL]

Maps are "pointer-fest"

*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.*

Avoid large maps and use inexpensive keys

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.

Memory Churn

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

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.*

If you do not use C++11, do yourself a favour and switch to it.

Object Life Cycle Management

Object life cycle management defines **who is responsible for allocation, creation and destruction of objects.**

The API specifies where objects are created, who owns and frees them, and when. It may also specify hooks for memory allocation allowing client to decide where memory gets allocated.

One policy is to take standard library like objects. It implies memory allocation is hard-wired to types, and copies happen frequently, and as such is a very significant design choice.

Opt-in approach to life cycle management doesn't work.

The APIs define the object management policy. You cannot avoid this by ignoring it; you'll just make your clients confused and guess (wrong).

Changing life cycle policy usually implies API + library rewrite.

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, 6'640 bytes (64-bit; proper use of reserve() gets this.)*
- *Reality (C++98): 870 allocations, total 36'184 bytes alloc'd, 7'168 at end, 12'096 peak.*
- *Reality (C++11): 555 allocations, total 20'584 bytes alloc'd, 7'168 at end, 11'040 peak.*
- *+400% # allocs, +210% bytes alloc'd, 66% working and 8% residual overhead!*
- *Versus 1 allocation, 4'880 bytes and some pointer setup had we used a real matrix (or 1 allocation and 4000 bytes, had we used a linear array).*

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, +6'640 bytes (= naïve / full reserve() allocation).***
- ***An allocation storm is inevitable if you copy nested containers by value.*** *Evil bonus:* ***fragmentation.*** *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.*

Allocator template argument

*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 arenas – 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.

Custom "plug-in" allocators

*Finally, there is a number of "plug-in" allocators, like **jemalloc** or **TCmalloc**, which might enormously improve the performance of your application under certain specific conditions, e.g. in a highly multithreaded environment. Noticeable gains can be obtained by using them, however make sure you **profile** their performance and that you keep looking at the improvements done in the standard allocator.*

Combating Memory Churn

Rewarding. Eliminating churn tends to yield big gains – x10 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.*

Hard. Solving memory churn varies from trivial to very hard.

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

Modern compilers *help a lot! With old compilers, passing / returning containers by value was also an issue, nowadays, when using* ***C++11 and C++14****, thanks to* ***move constructors****, one needs to try very hard to suffer from this.*

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.*

C++ Hazards Wrap-Up

C++ standard library is very easy to program.

With judicious use easy to write high performance programs. With poor judgement easy to make terrible mistakes which ruin the performance and require total rewrite to regain it.

C++ standard library types in interface = life cycle policy.

Passing C++ standard library like types in API interfaces effectively means segregating memory management to library. In practice library users will have little ability to manage library memory use.

Avoid containers nested by value and string abuse.

There are legitimate uses, but this is almost always a mistake.

Containers are dynamic entities.

Make sure you understand what their dynamic behaviour implies. Prefer vectors over maps and list. Prefer string constants over std::string where possible.

Exercise
std::vector memory use

Exercise
std::string memory use

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.*

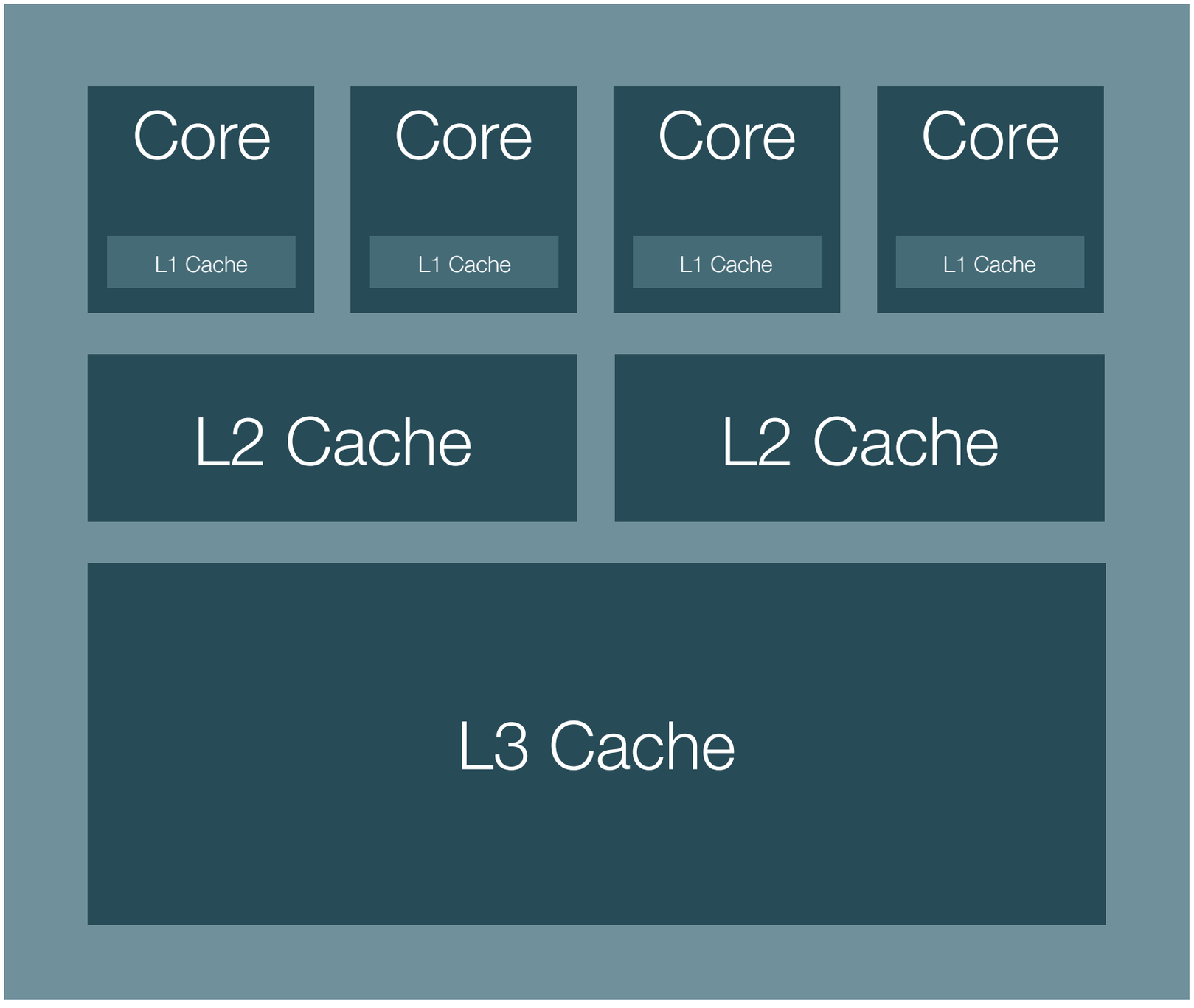
#2: Memory overhead, alignment & churn matter

- *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.*

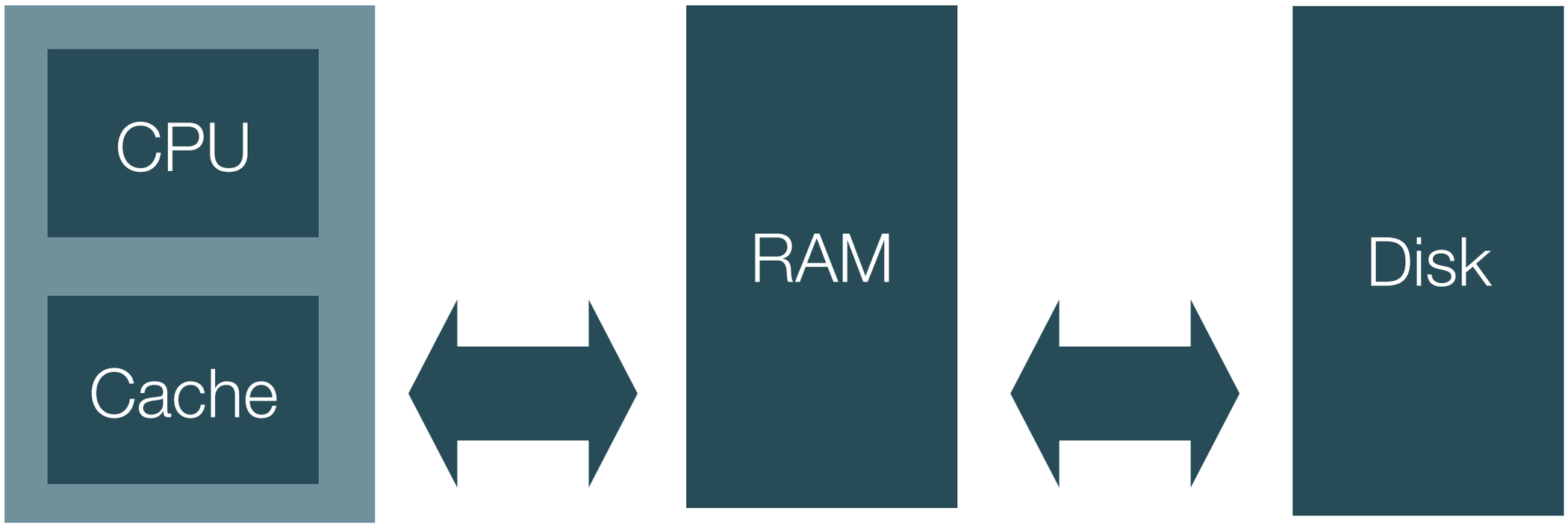
Memory hierarchy



2 - 8 **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

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.

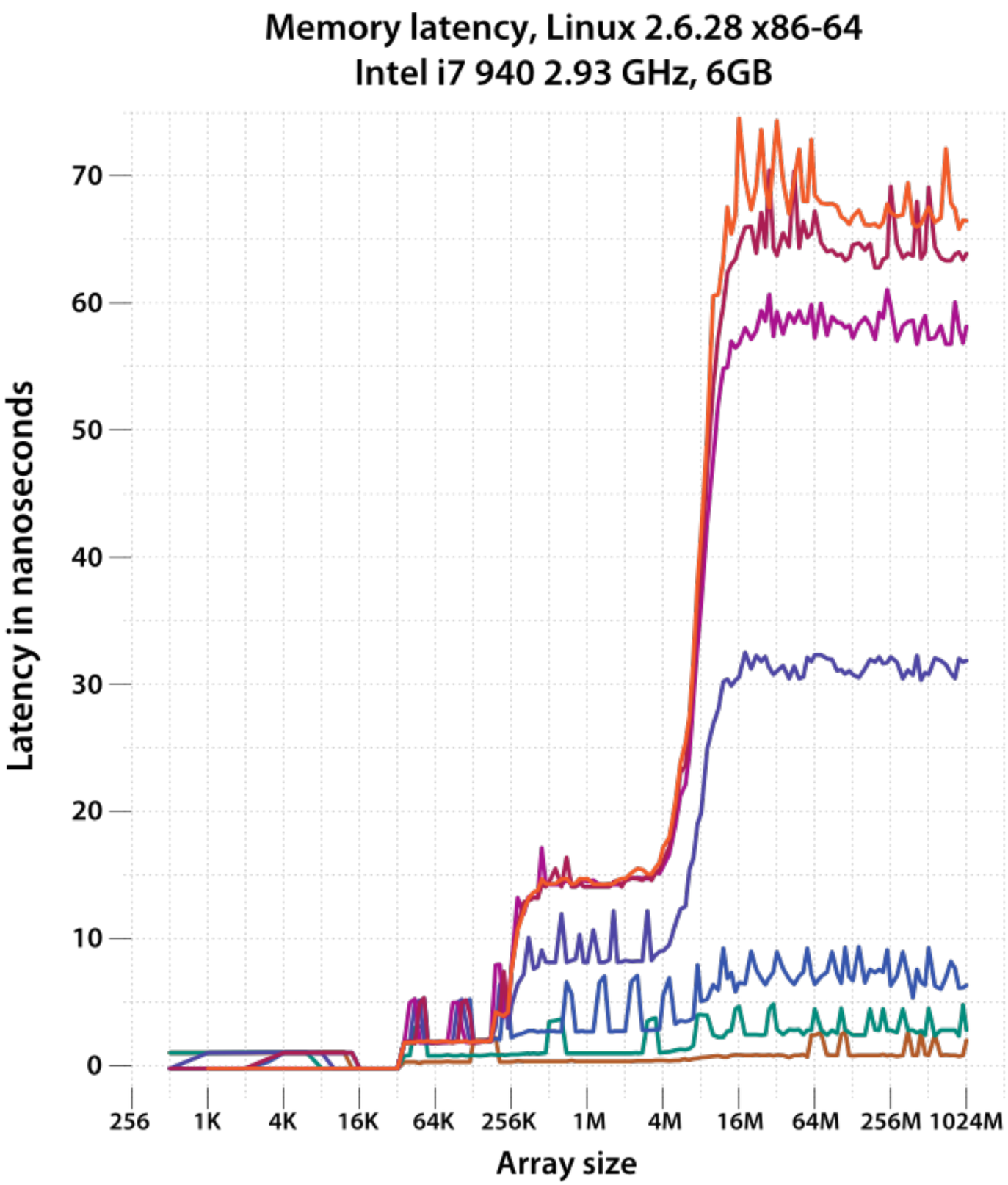
TLB: L1 miss, L2 miss =~ 30+ cycles.

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).

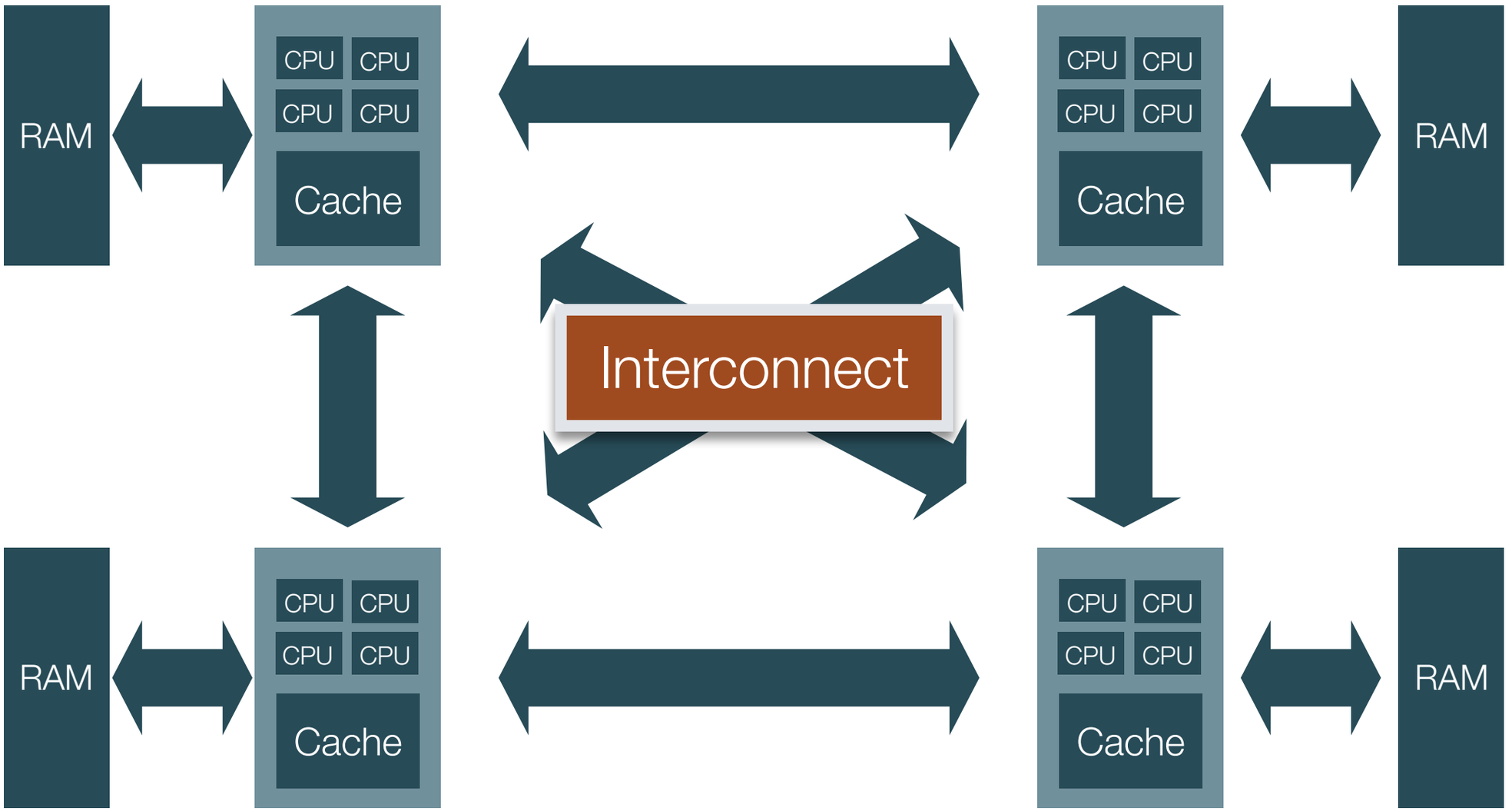
Good cache efficiency matters.



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

Non-Uniform memory access

RAM is not necessarily local anymore



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

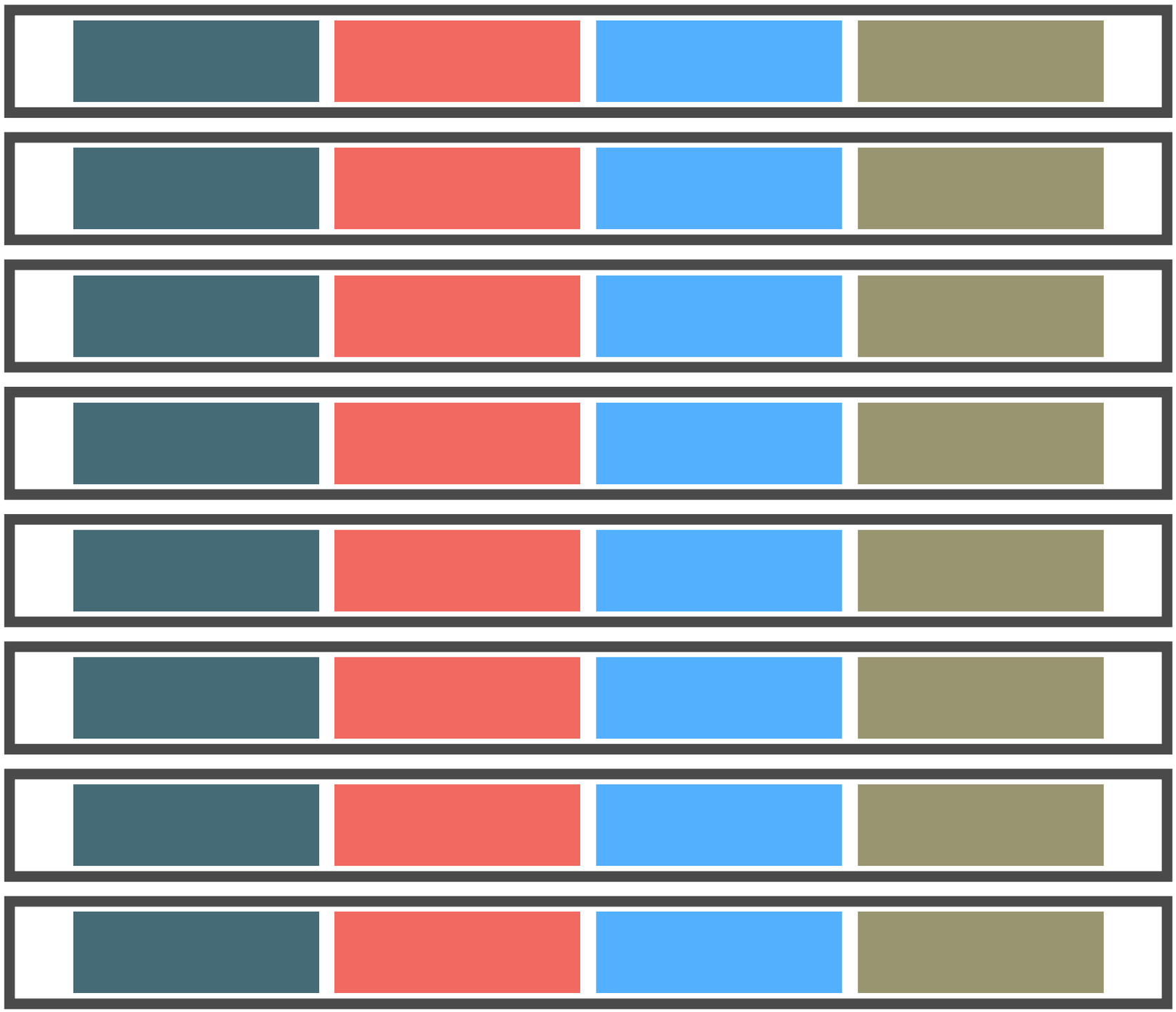
Peak performance requires effective cache use for low latency. **How that is achieved is less important.** Understanding the language mapping from high-level constructs to low-level behaviour helps.

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.*

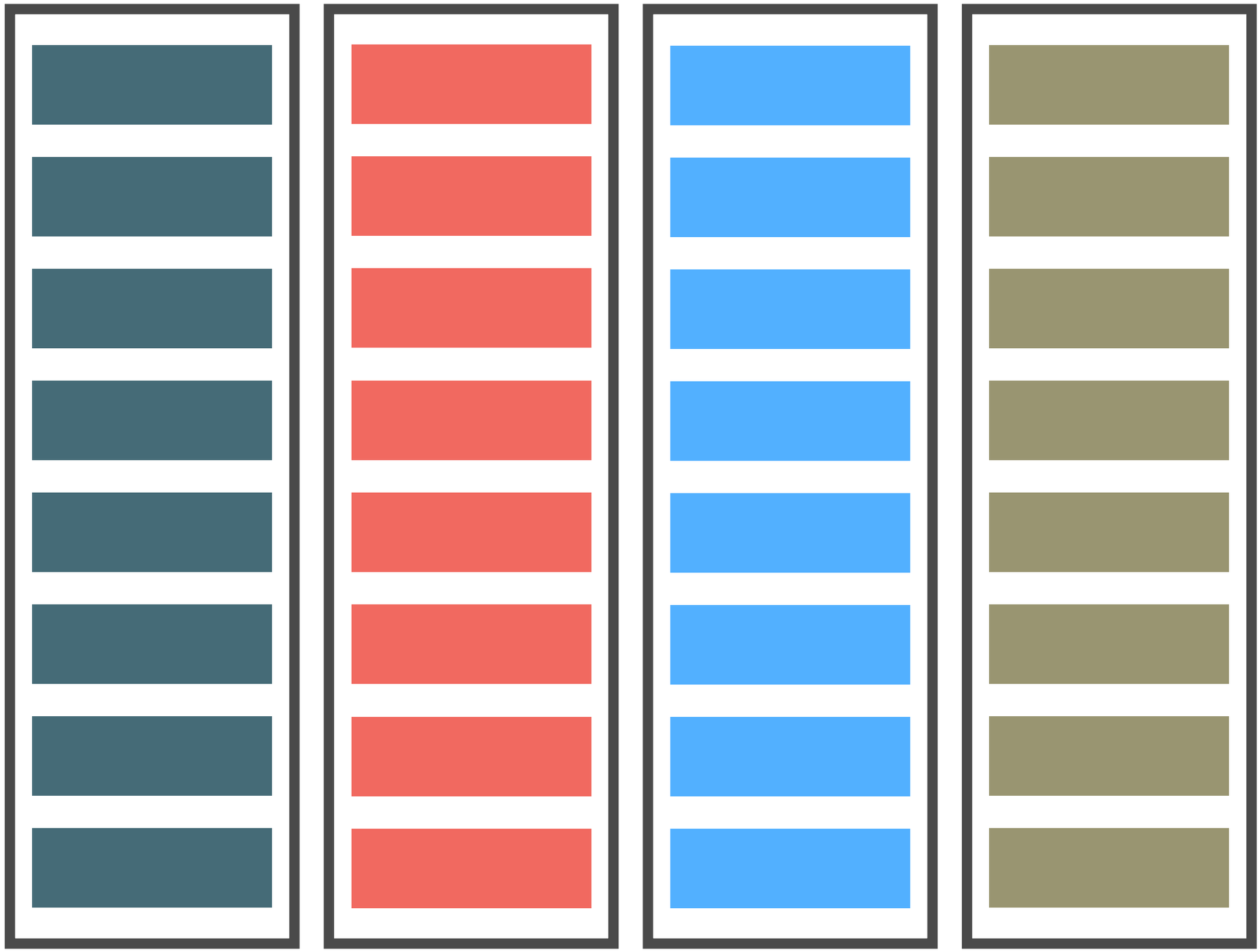
Pointer-rich "proper objects" do remain immensely useful – as long as caches are used very effectively, or performance simply doesn't matter, for example in GUIs, support data structures and rarely used infrastructure.

Array of Structures



std::vector<  >

Structure of Array



struct {std::vector<■>; std::vector<■>; std::vector<■>; std::vector<■>;}

Why Structures of Arrays?

```
class X {
  ...
  vec3 pos_;
  ...
  float boost_;
  ...
  float dir_;
  ...
  void update(vec3 target)
  { dir_ = dot3(pos_, target) * boost_; }
};
```

Why Structures of Arrays?

```
class X {
  ...
  vec3 pos_;
  ...
  float boost_;
  ...
  float dir_;
  ...
  void update(vec3 target)
  { dir_ = dot3(pos_, target) * boost_; }
};
```

I-Cache miss

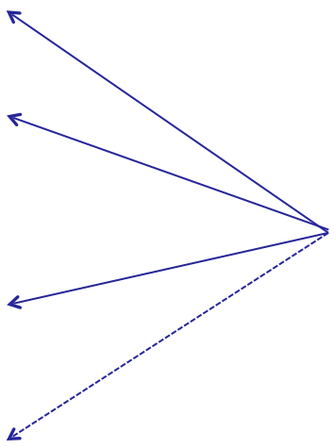
Why Structures of Arrays?

```
class X {
  ...
  vec3 pos_;
  ...
  float boost_;
  ...
  float dir_;
  ...
  void update(vec3 target)
  { dir_ = dot3(pos_, target) * boost_; }
};
```

D-Cache
miss

Why Structures of Arrays?

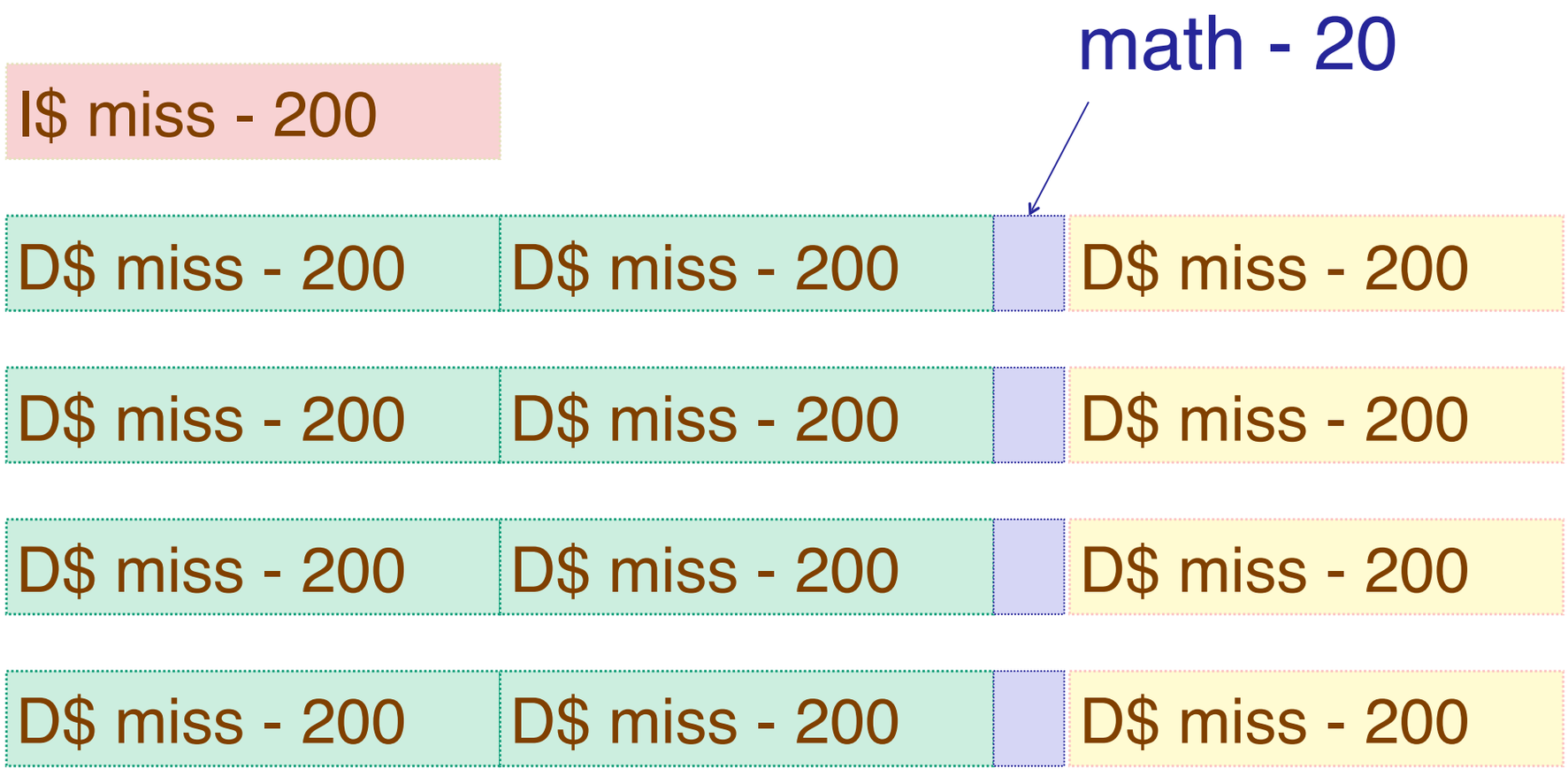
```
class X {
  ...
  vec3 pos_;
  ...
  float boost_;
  ...
  float dir_;
  ...
  void update(vec3 target)
    { dir_ = dot3(pos_, target) * boost_; }
};
```



Cache occupied but not used here

4 executions on 4 objects – how many cycles?

```
void update(vec3 target)
{ dir_ = dot3(pos_, target) * boost_; }
```



Change Abstraction to SoA

```
void update(float *dir, TargetData *data,
            vec3 target, size_t count)
{
  for (size_t i = 0; i < count; ++i)
    dir[i] = dot3(data->pos[i], target) * data->boost[i];
}
```

Same kind of data is grouped together,
maximizing data cache usage.

Change Abstraction to SoA

Update an array

Minimal input data

```
void update(float *dir, TargetData *data,
            vec3 target, size_t count)
{
   for (size_t i = 0; i < count; ++i)
      dir[i] = dot3(data->pos[i], target) * data->boost[i];
}
```

Loop over data

Calculation mostly unchanged

Code has been separated out

Timings difference

```
void update(float *dir, TargetData *data,
            vec3 target, size_t count)
{
  for (size_t i = 0; i < count; ++i)
    dir[i] = dot3(data->pos[i], target) * data->boost[i];
}
```

math - 20

I$ miss - 200

D$ miss - 200 | D$ miss - 200 | | D$ miss - 200 | | | |

Structures of Arrays and Design

*Try to view SoA vs. objects as a **change in abstraction**, not as a "Do I really have to break everything I was taught about encapsulation?"*

*When designing for SoA, you create **higher-level abstractions** with operators and **kernels which are applied to collections** of data.*

*You apply SoA design to the **largest masses of data** in the **most computation intensive** parts. There are still places for polymorphisms and more complex data structures, e.g. graphs, but they operate in different levels, or sections of code which are not performance sensitive.*

Wrapping Up

The CPU – memory performance difference has profound impact.

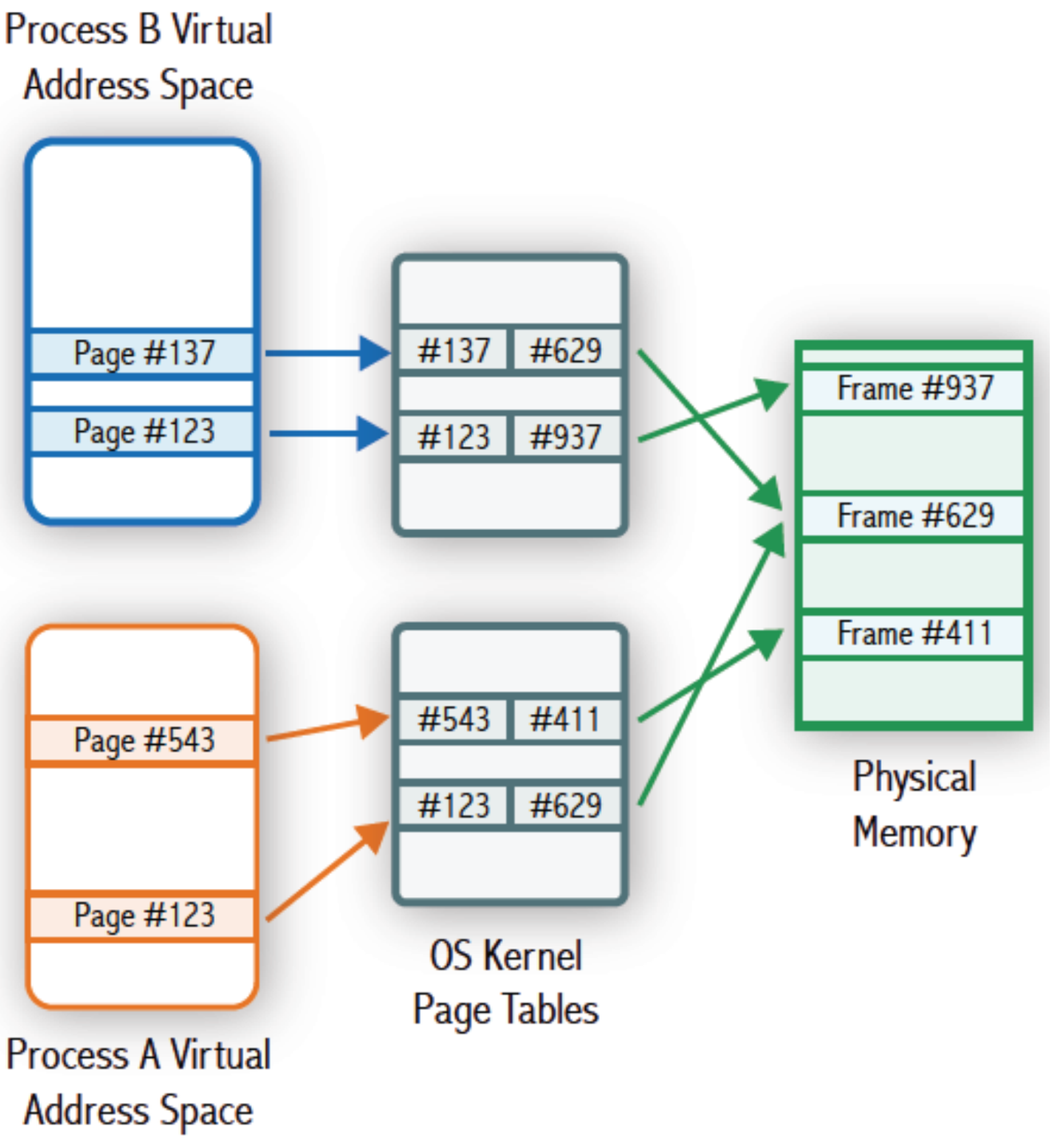
> *Memory management choices have orders of magnitude performance impact, and among the most important design criteria after selection of algorithms. A performance-oriented design must consider all the layers from application to libraries/language to operating system to processor and memory interconnect.*

Bandwidth is usually adequate for all but the most demanding applications. 1-3 layers of cache help hide a very significant memory access latency ***if and only if*** *you structure the application to have excellent locality for code, data and virtual memory pages and their tables. Best performance for large data volumes requires hardware-predictable access of structures-of-arrays / arrays-of-structures.*

Memory access latency is non-uniform and depends on physical design and interconnect distance to memory, thus operating system allocation strategy. Application may need to guide operating system on strategy choices.

Exercise
AoS vs SoA

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.

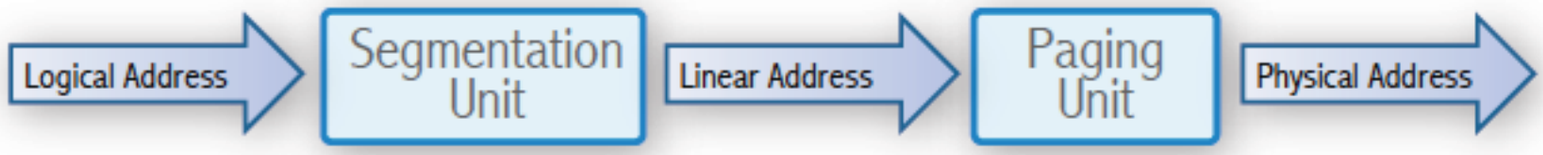
Page tables themselves **use memory, consume L2+ cache space,** and are never swapped out.

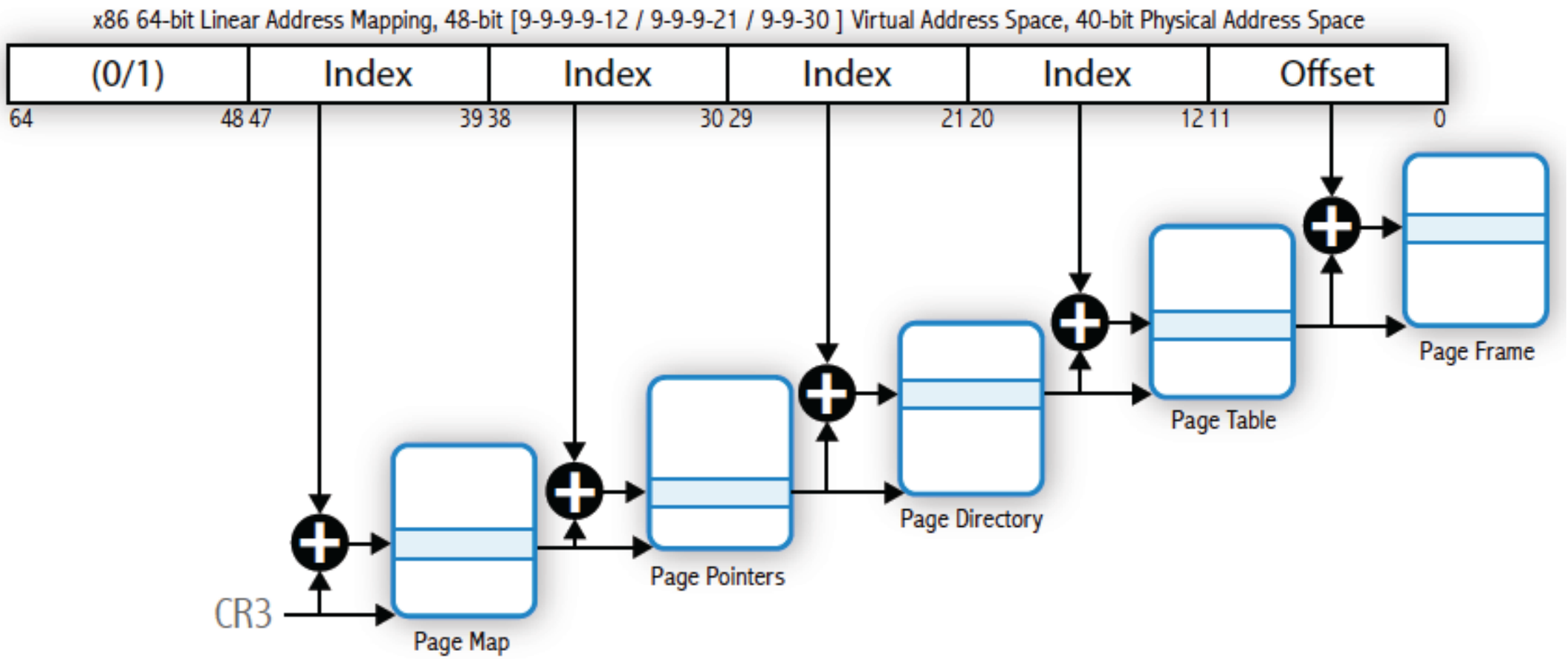
Even if processes share physical page frames, the **page tables are not shared.** With 4kB pages, large address spaces mean *big page tables*, even if the memory itself is shared: there's over 2MB of page tables for every 1GB of committed address space.+

** 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.*

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

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").

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:
```

```
$ 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]
```

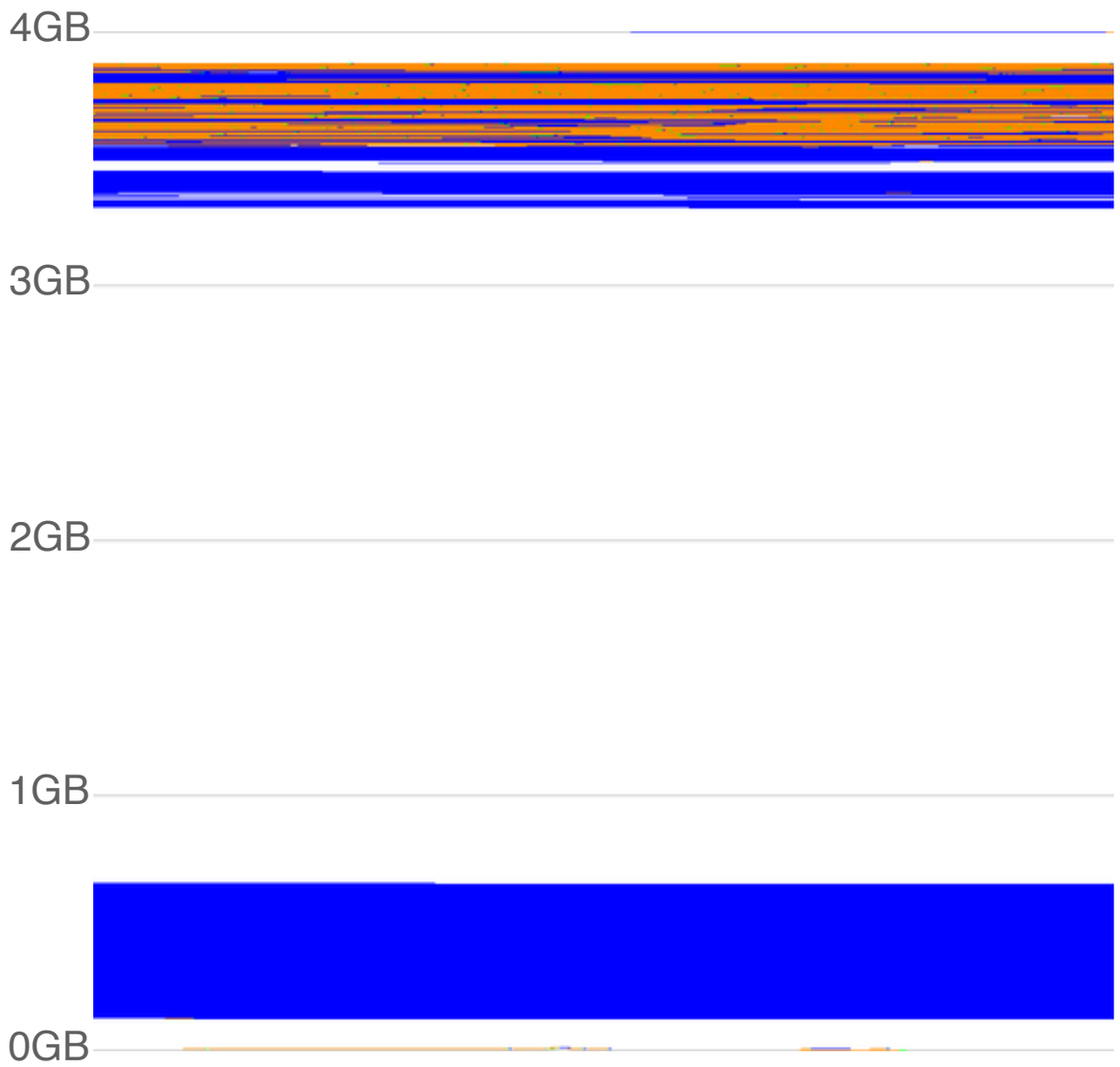
After a while…

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(!)

Exercise
Process Address Space

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.**

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:

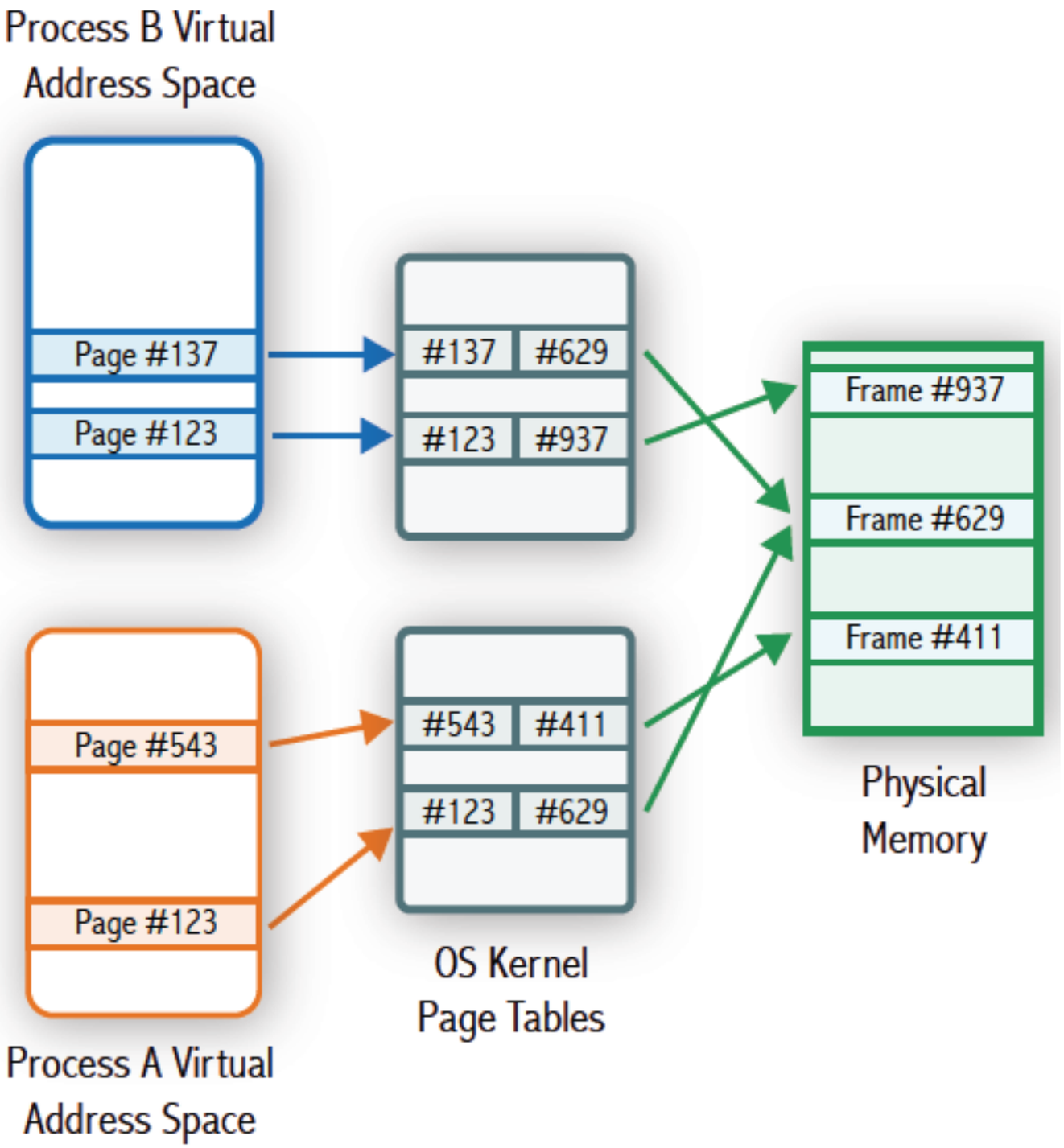
Calling **mmap() on a file in multiple processes** can be used to create shared read-only or read-write mappings, on any file region. Example: shared library **position independent code**. One way to share static read-only data is to wrap and load it as a shared library. Suitable use of mmap() + {f,m}advise()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

About Shared Memory



B's page #137 and A's page #123 are mapped to the same physical frame #629, creating shared memory.

#629 could be a read-only page of common library code, writeable memory created with mmap() + fork() or shmget().

Exercise
Zero Pages

Key Factor #3: Locality

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

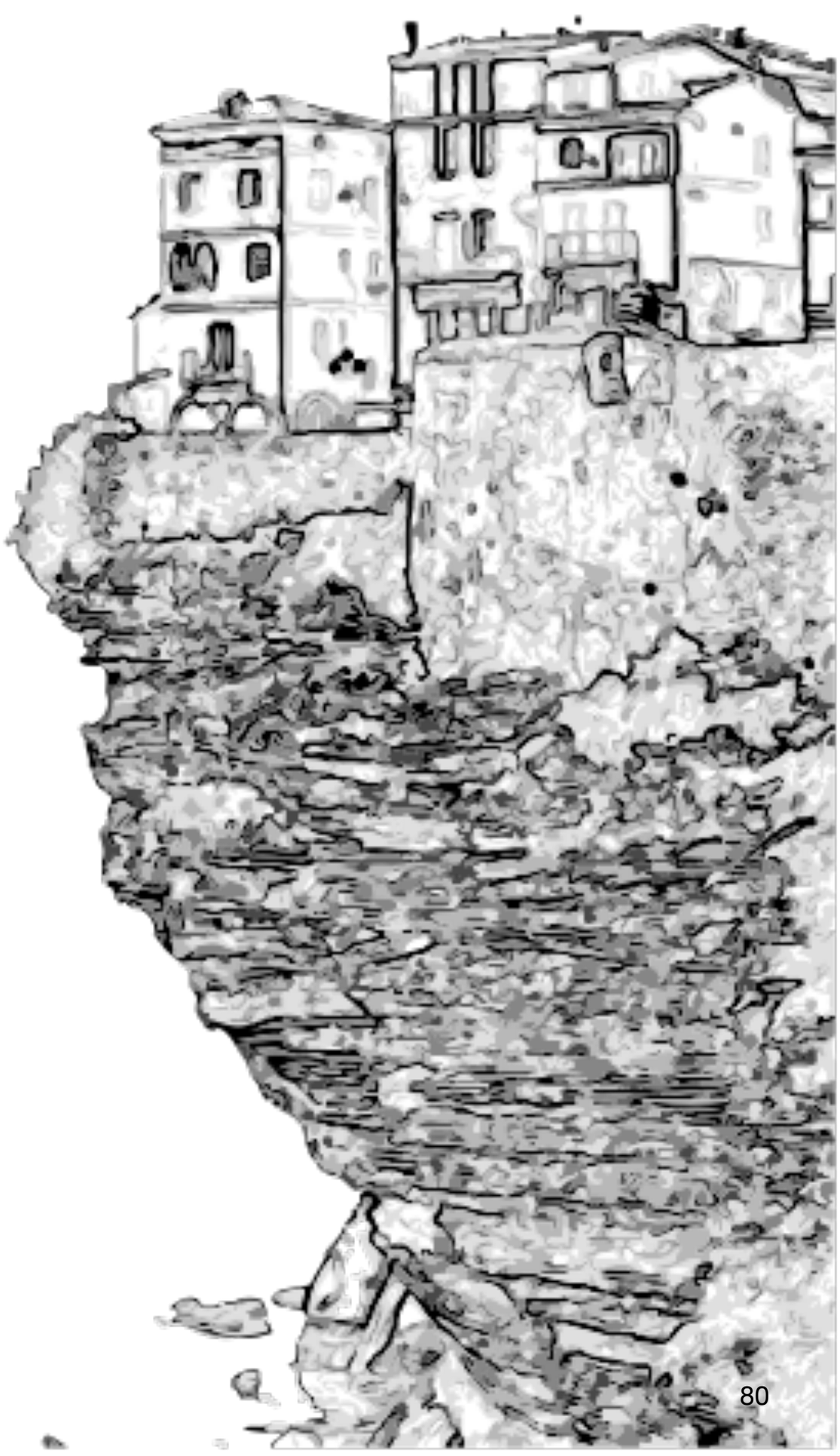
> *Using suitable pool allocators is known to help, but no easy-to-use analysis tools. You can try evaluate heap trashing and allocation size distribution to some extent with e.g. igprof heap snapshots, even GLIBC's memusage. In general the better your unit and regression test collection, the easier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

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

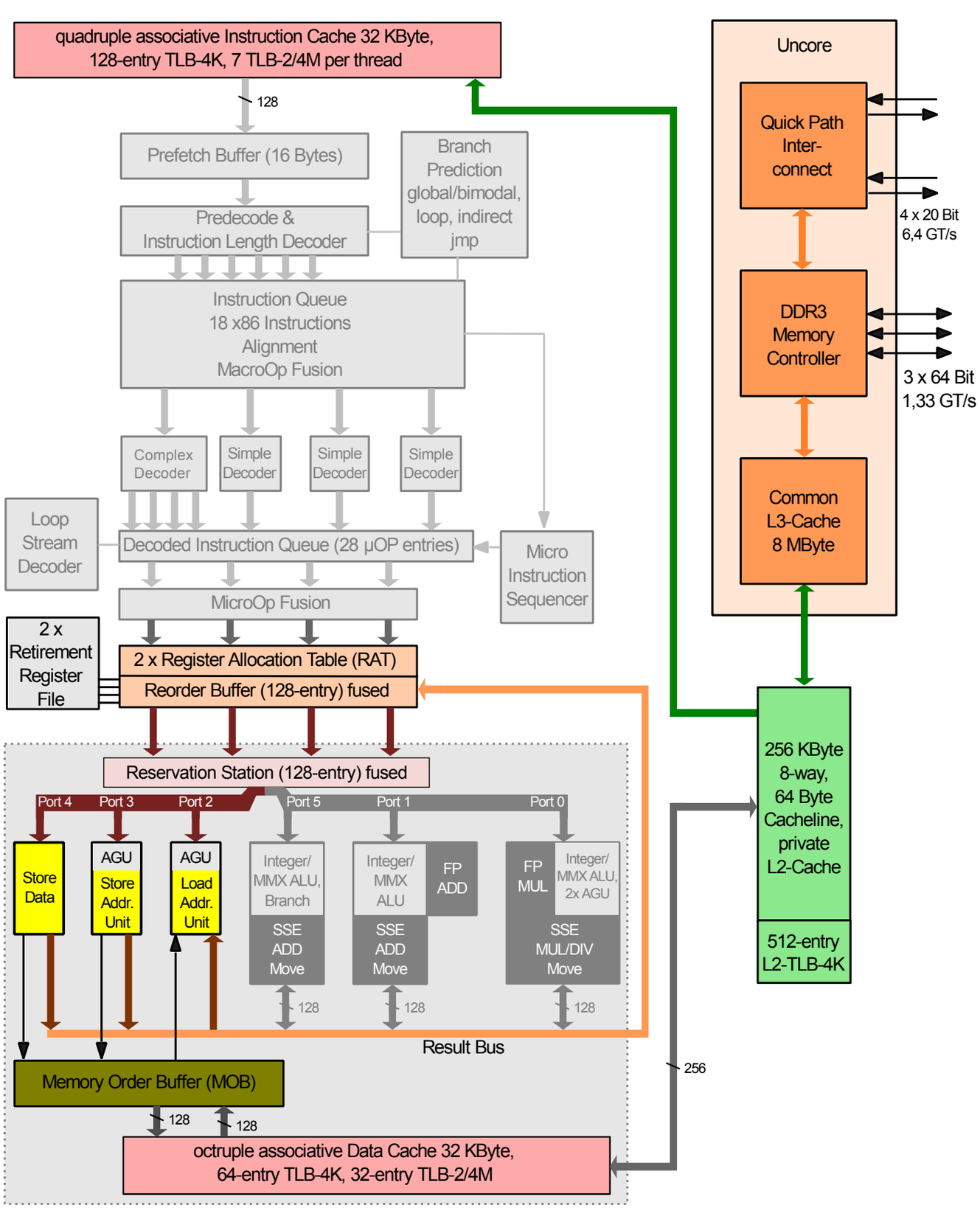
> *A good rule of thumb is the larger the process, the slower it gets, with a few well designed applications an exception to this. 200+ MB of machine code from 500+ shared libraries is usually just preposterously bad packaging and / or large-scale code bloat. Fix packaging, make big shared libraries 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.*

Memory Crisis

Closer look at locality



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



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.

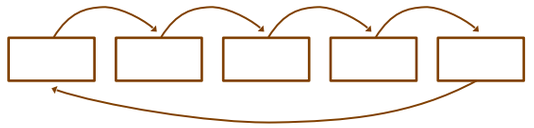
512-entry L2 TLB ≅ 2MB code + data addressing capacity – less than fits in L3 cache, but more than one core share of L3.

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

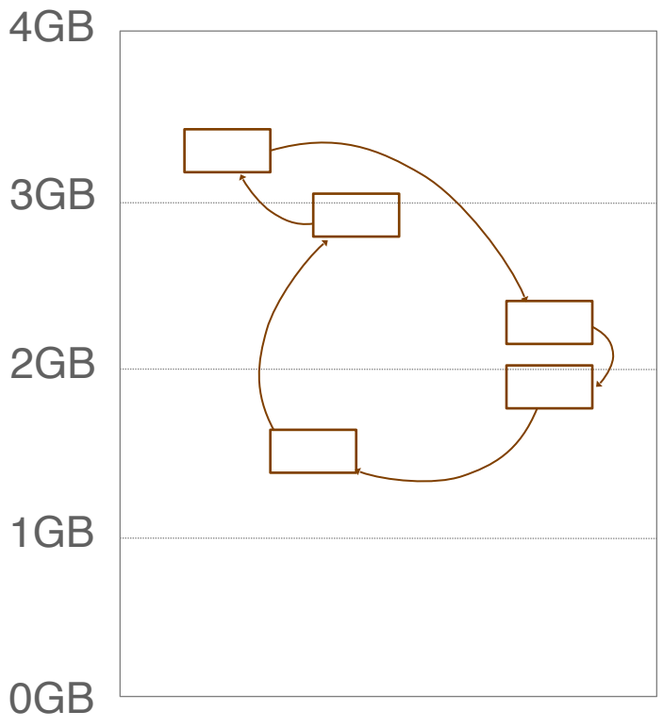
BUT for all practical purposes a modern CPU performs well on **large data volumes** only if organised as **arrays-of-structures** (AoS) or **structures-of-arrays** (SoA) – pointer-rich "objects" *will* perform poorly.

Logical vs. Real Data Structures

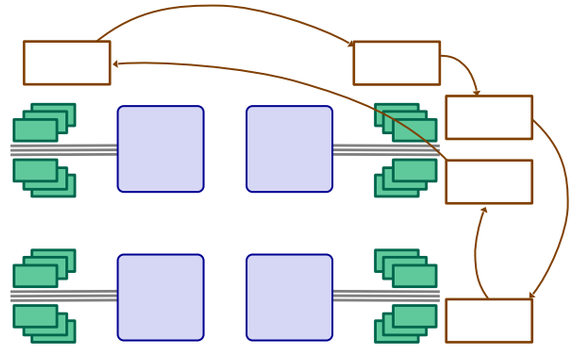
This logical linked list…



Could be scattered in virtual address space like this…



And in physical memory like this…

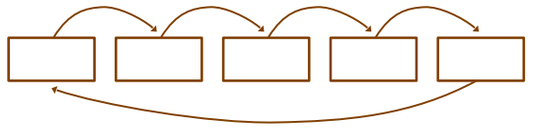


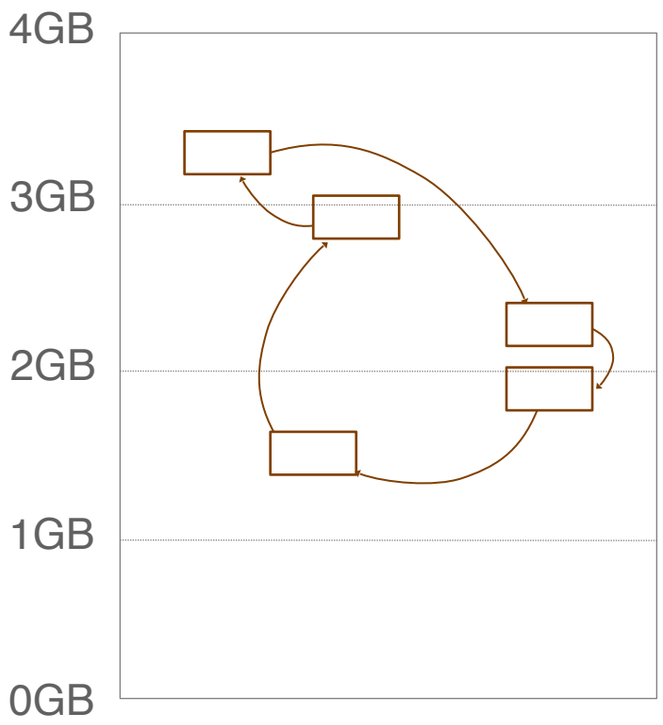
Logical vs. Real Data Structures

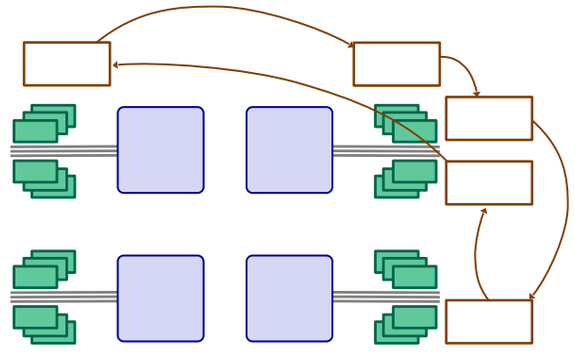
The scatter is unimportant as long as L*n* and TLB caches hide all latencies. Otherwise **you must explicitly arrange for** a better memory ordering.

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.







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!

Exercise

Shared libraries

Exercise

Word list filter

Exercise

Large Application

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