

An approach to Performance and Bottleneck Analysis

**Sverre Jarp
CERN openlab**



**Gelato Spring Meeting, San José
26 April 2006**

AGENDA

- Introduction
- Path to Optimization
- Basics
- Hardware Review
- PMU
- More on SWP
- More on compilers
- Conclusion

INTRODUCTION

Initial Question

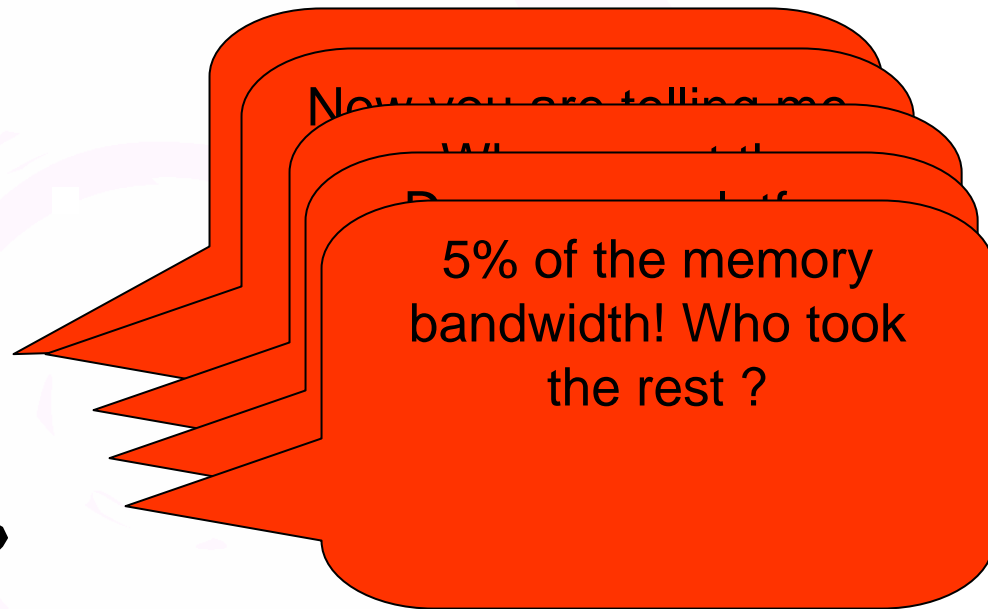
- How come we (too) often end up in the following situation?



Why the heck
doesn't it
perform as it
should!!!

Initial Answer

- In my opinion, there are many (complicated) details to worry about!

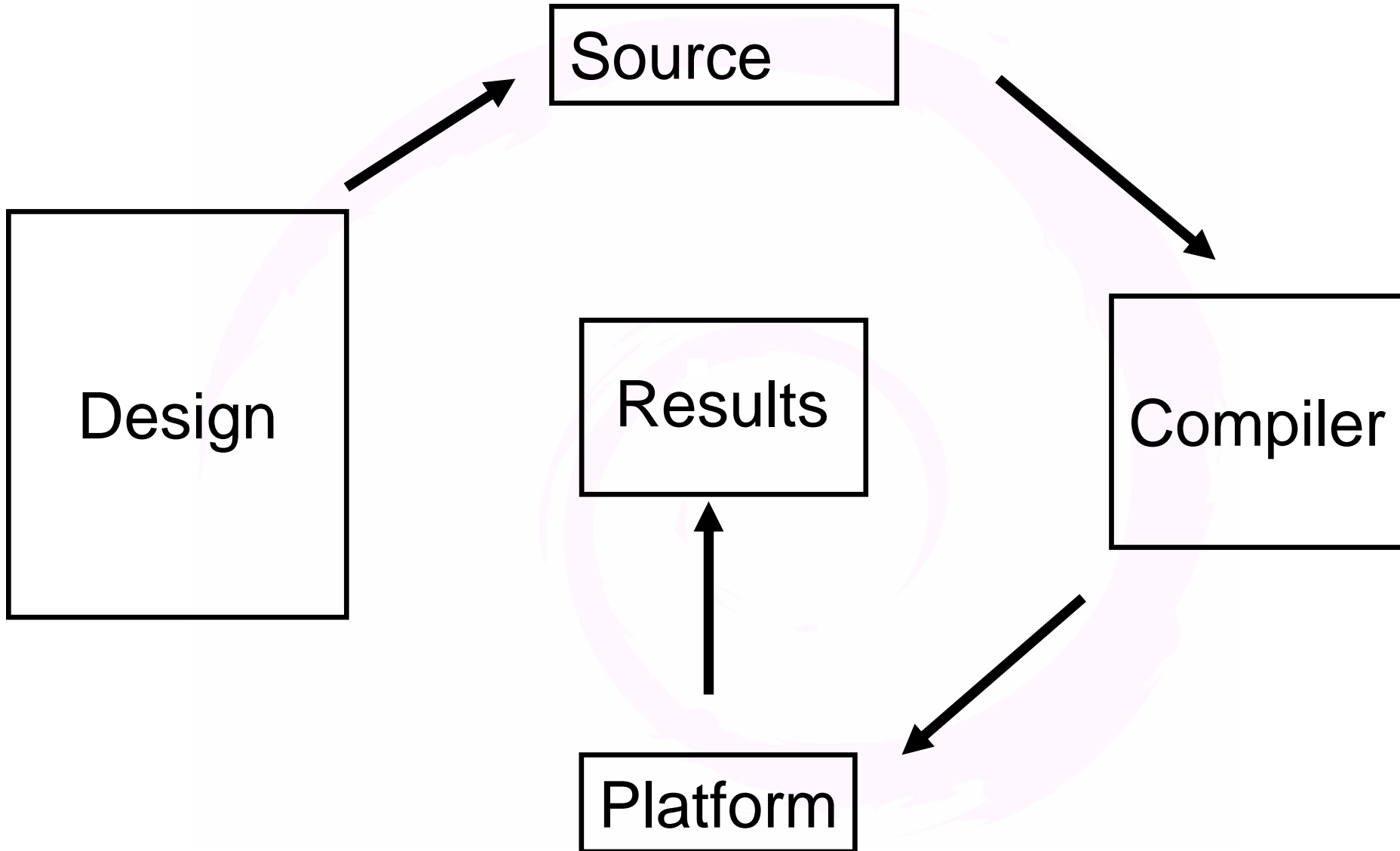


Before we start

- **This is an effort to pass in review a somewhat “systematic approach” to tuning and bottleneck analysis**
 - Main focus is on understanding the “spiral to success”
- **The introduction of the elements is done “top-down”**
- **But, it is important to understand that in real-life, this is rarely the case**

PATH TO OPTIMIZATION

Path to optimization



Step 0: Correctness

A MUST

- **Before undertaking any tuning effort**
 - Excellent regression/correctness tests
 - For all critical algorithms, all important use cases
 - Otherwise,
 - Too many tuning efforts get left by the wayside
 - Which options will work ?
 - “IPF_fp_relax”
 - “ansi_alias”
 - “ffast-math”

In an case, needed for the basic development/maintenance effort

No point in speeding up an incorrect program!

Step 1: Application design

DESIRABLE

- **Regular reviews of the design (globally or partially)**
 - Data structures
 - Arrays; structs; data members
 - Choice of algorithms
 - Accuracy, robustness, rapidity
 - Design of classes
 - Domain decomposition
 - Hierarchy
 - Interrelationship
- **Is there a time gap?**
 - Design \leftrightarrow Today's microprocessor (tomorrow's ?)
 - Did we design for low ILP, small caches, single core,..... ?

The arrival of multi-/many-core may force partial redesign of many applications

- **Review all aspects of implementation**
 - Choice of language (Fortran, C, C++, Java, ...)
 - Use of language features
 - Templates (STL with map, lists, etc.)
 - Precision of data (FLP)
 - Single, double, double extended
 - Intermediate calculations
 - Stored results
 - Code split between .cpp and .h files
 - Aggregation or decomposition ?
 - Reliance on preprocessor
 - Platform dependencies
 - Such as endianness
 - Reliance on external libraries
 - Smartheap, Math kernel/vector libraries, etc.

DESIRABLE

Correct organization of source can greatly impact the application's efficiency

Step 3: Compiler/compilation

A MUST

- **Access to the best compiler**
 - On Linux/IPF we have a limited choice
 - Intel or GNU (other coming?)
 - But, it is worth trying both
 - Mix and match (thanks to common ABI) ?
 - Inform the other camp when they are behind
 - Upgrade to latest versions regularly
 - Choose from hundreds of flags
- **Build procedure**
 - One class at a time ?
 - Archive/shared libraries ?
 - Monolithic executable or dynamic loading ?
- **And (to a large extent)**
 - Machine code is chosen for you

IPF is still not bogged down in too many microarchitectural options (à la x86)

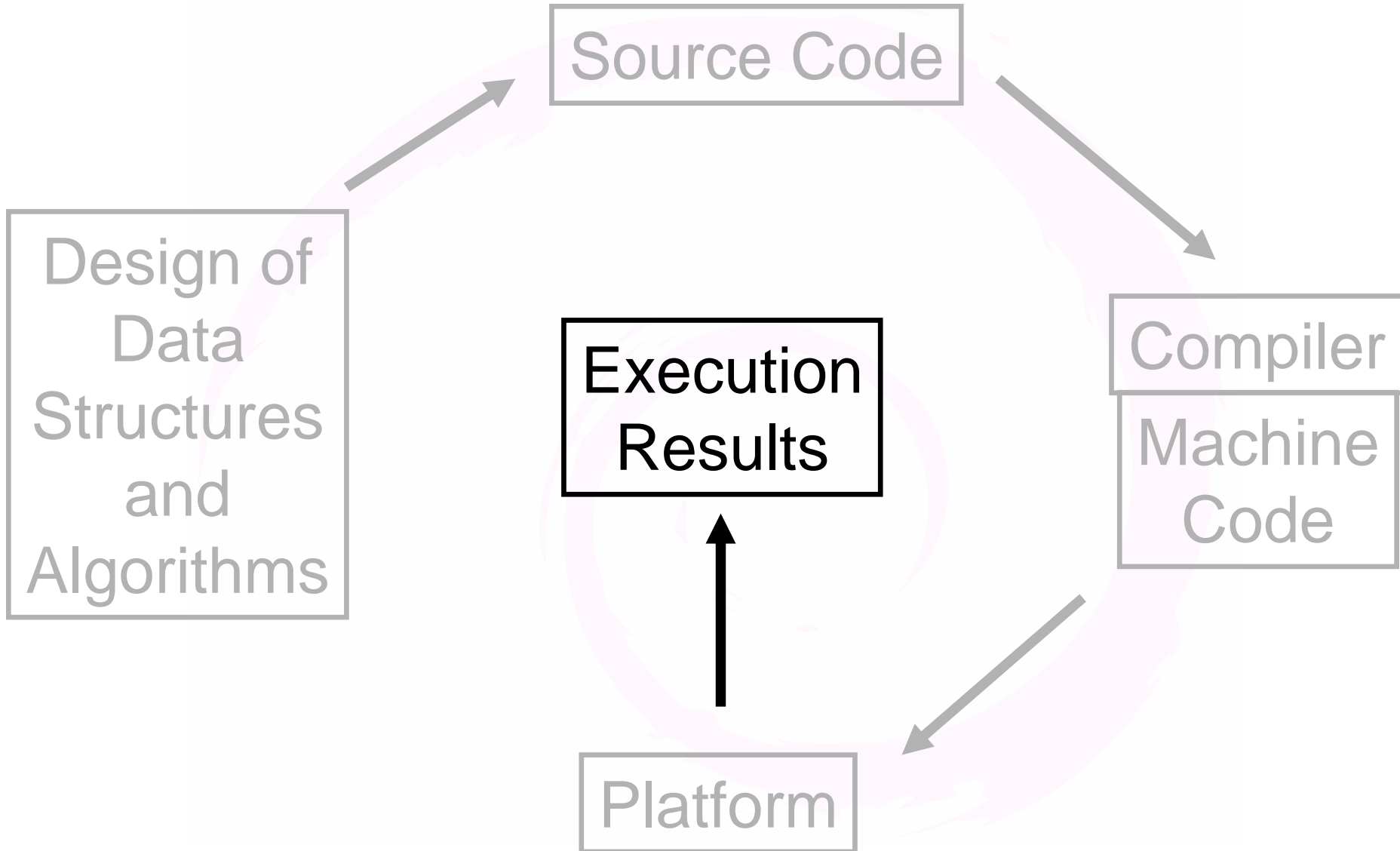
Step 4: Platform

- **The best hardware for the job**
 - Manufacturer
 - Server type
 - Entry, mid-range, large SMP, NUMA, etc.
 - Processor characteristics
 - Madison (single core), Montecito (dual core)
 - Frequency, cache size
 - Further (important) factors
 - Bus speed
 - Memory speed
 - Price/performance ratio

A MUST

However, the IPF platform ecosystem is still somewhat limited

In the end: Execution Results

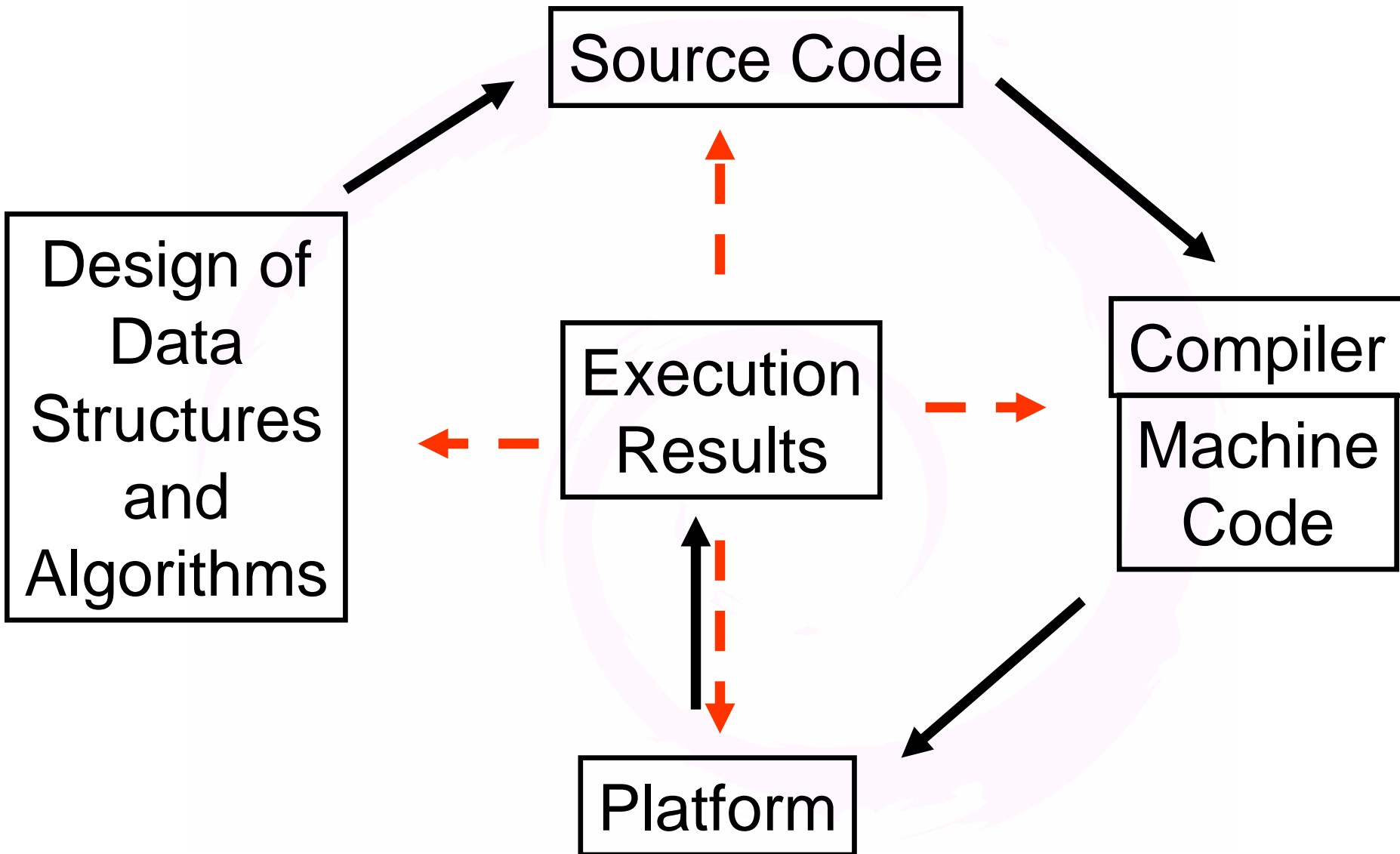


Back to our cartoon



- **As already said, first of all, we must guarantee correctness**
- **If we are unhappy with the performance**
 - ... and by the way, how do we know when to be happy?
- **We need to look around**
 - Since the culprit can be anywhere

Where to look ?



THE BASICS

Price_out_impl (mcf)

- VTUNE screenshot:

Address	Line	Clockticks	Source
01:03CA	246	1426	while(arcin)
	247		{
01:03F0	248	2329	tail = arcin->tail;
01:03F2	250	323207	if(tail->time + arcin->org_cost > latest)
	251		{
01:03FC	252	1485	arcin = (arc_t *)tail->mark;
	253		continue;
	254		}
01:0410	256	657	red_cost = compute_red_cost(arc_cost, tail, head_potential);
01:0424	258	2233	if(red_cost < 0)
	259		{
01:04F6	260	296	if(new_arcs < MAX_NEW_ARCS)
	261		{
	262		insert_new_arc(arcnew, new_arcs, tail, head,
01:0530	263	39	arc_cost, red_cost);"
01:0558	264	16	new_arcs++;
	265		}
01:04FE	266	301	else if((cost_t)arcnew[0].flow > red_cost)
	267		replace_weaker_arc(arcnew, tail, head,
01:050B	268	65	arc_cost, red_cost);
	269		}
01:042C	271	1664	arcin = (arc_t *)tail->mark;
	272		}

- **The language spoken by the processor is**
 - MACHINE CODE !!
- **To understand it, we need what I call “Assembler awareness”:**
 - Looking into compiler-generated code, there may be a need to:
 - Modify (repeatedly) the HLL code (or compiler options) and inspect the result
 - When available, add inline assembly or intrinsics for localized impact
 - Today, we are not dealing with the case of writing Assembly code
 - But the issues are the same

Machine code

- It may be necessary to read assembly code directly

```

Bool_t TGeoCone::Contains(Double_t *point) const
{
// test if point is inside this cone
if (TMath::Abs(point[2]) > fDz) return kFALSE;

Double_t r2 = point[0]*point[0] + point[1]*point[1];
Double_t rl = 0.5*(fRmin2*(point[2] + fDz) + fRmin1);
Double_t rh = 0.5*(fRmax2*(point[2] + fDz) + fRmax1);
if ((r2<rl*rl) || (r2>rh*rh)) return kFALSE;
return kTRUE;
}

```

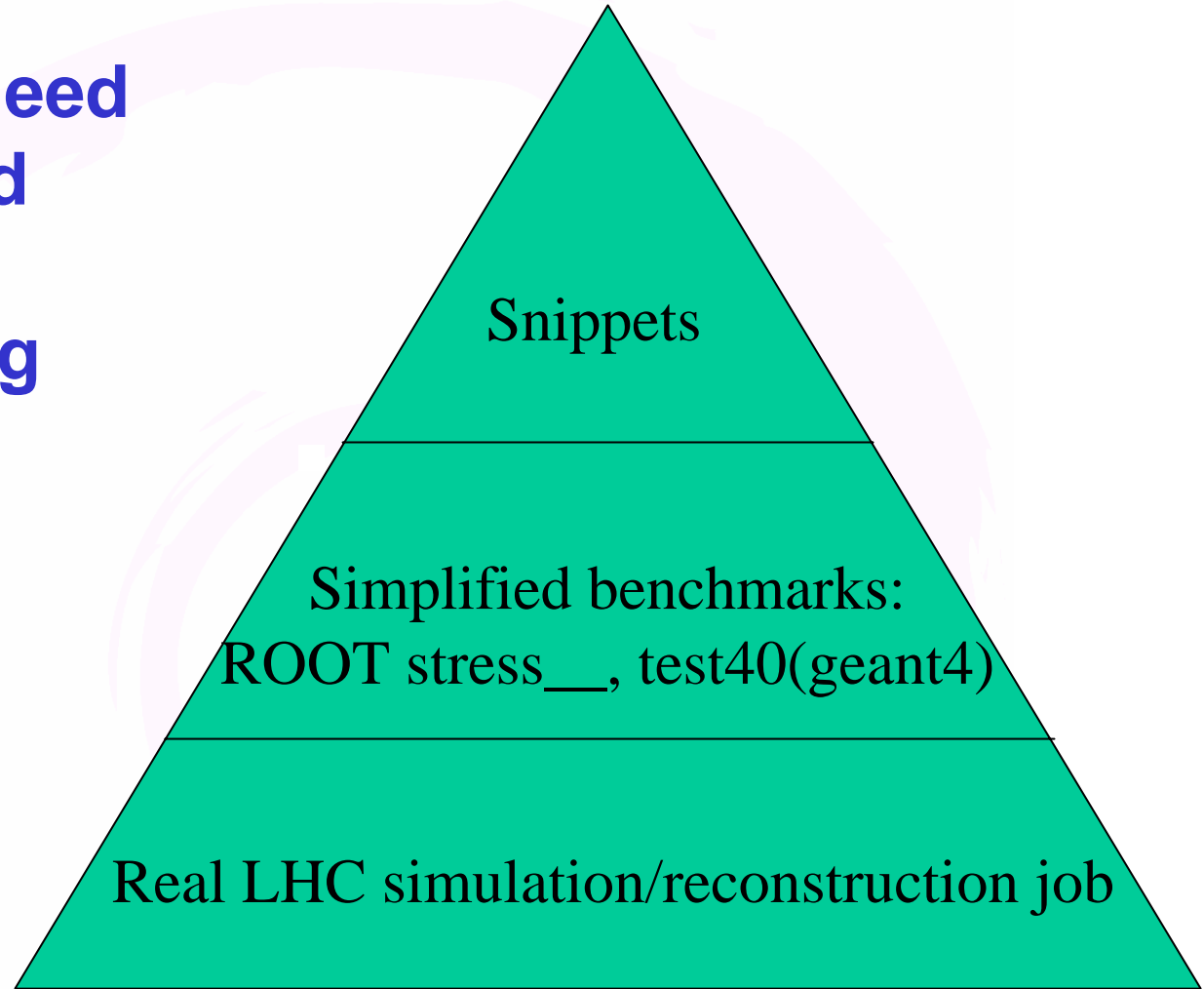
```

_ZNK8TGeoCone8ContainsePd:
[.LFB1785:]
    .prologue
    .body
    .mmi
    adds r14 = 16, r33
    adds r15 = 16, r32
    adds r16 = 32, r32
    .mmi
    adds r17 = 24, r32
    adds r18 = 40, r32
    adds r32 = 8, r32 ;;
    .mmi
    ldfd f11 = [r14]
    ldfd f15 = [r32]
    mov r8 = r0 ;;
    .mfb
    fcmp.ge p6, p7 = f11, f0
    .mfi
    mov f6 = f11 ;;
    .mmf
    (p7) fneg f6 = f11 ;;
    .mmf
    fcmp.gt p6, p7 = f6, f15;;
    .bbb
    (p6) br.ret.dptk.many rp
    (snip)

```

The testing pyramid

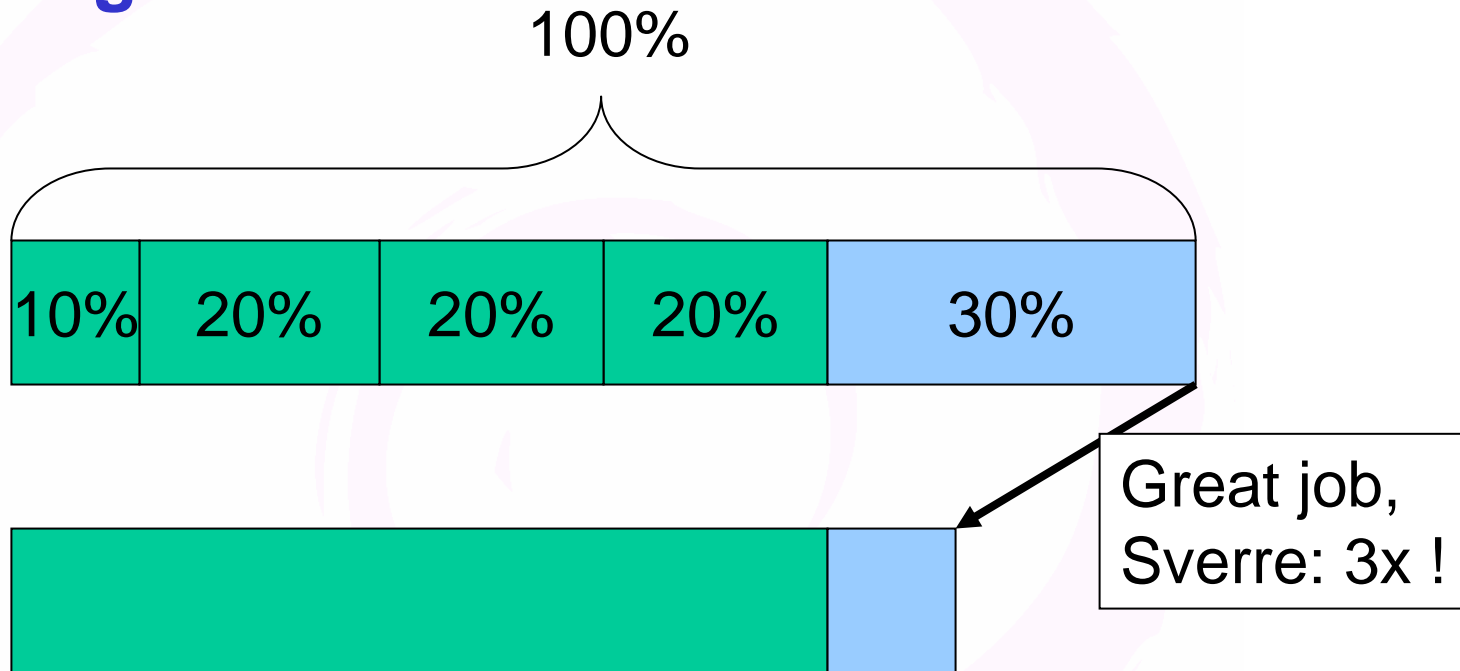
- You always need to understand what you are benchmarking



CERN's pyramid

Amdahl's Law

- The incompressible part ends up dominating:



Total speedup is "only": $(100/80): 1.25$

Profiling with q-tools

- Test40: Physics simulation job

Command: ./test40icc8002fz

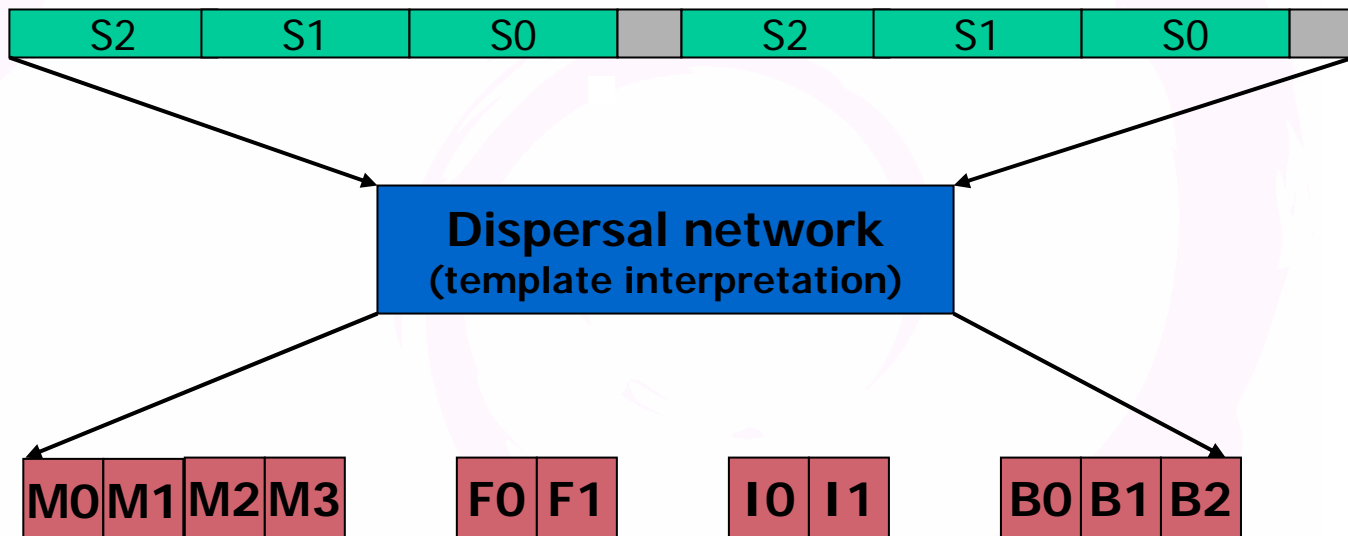
Flat profile of CPU_CYCLES in test40icc8002fz-pid24686-cpu0.hist#0: Each histogram sample counts as 1.00034m seconds

% time	self	cumul	calls	self/call	tot/call	name
14.28	7.23	7.23	1.72M	4.21u	4.90u	G4VoxelNavigation::LevelLocate(G4NavigationHistory&, G4VPhysicalVolume const*, int, Hep3Vector const&, Hep3Vector const*, bool, Hep3Vector&)
5.63	2.85	10.08	35.4M	80.5n	80.5n	RanecuEngine::flat()
3.57	1.81	11.89	3.80M	476n	2.94u	G4Navigator::LocateGlobalPointAndSetup(Hep3Vector const&, Hep3Vector const*, bool,...)
3.46	1.75	13.64	5.12M	343n	2.54u	G4SteppingManager::DefinePhysicalStepLength()
2.30	1.16	14.80	898k	1.30u	2.84u	G4VEnergyLoss::GetLossWithFluct(G4DynamicParticle const*, G4Material*, double,)
2.29	1.16	15.96	28.1M	41.2n	41.2n	G4Tubs::Inside(Hep3Vector const&) const
2.27	1.15	17.11	7.50M	153n	2.27u	G4SteppingManager::InvokePSDIP(unsigned long)
2.23	1.13	18.24	5.17M	219n	7.53u	G4SteppingManager::Stepping()
2.17	1.10	19.34	4.92M	223n	2.03u	G4Transportation::PostStepDoIt(G4Track const&, G4Step const&)
2.12	1.08	20.41	15.8M	67.9n	67.9n	G4PhysicsLogVector::FindBinLocation(double) const
1.93	0.98	21.39	5.23M	186n	901n	G4Transportation::AlongStepGetPhysicalInteractionLength(G4Track const&, double, double, double&, G4GPILSelection*)
1.90	0.96	22.35	1.11M	864n	864n	G4MuPairProduction::ComputeDDMicroscopicCrossSection(G4ParticleDefinition const*, ...)
1.80	0.91	23.26	1.01M	897n	1.50u	G4MultipleScattering::PostStepDoIt(G4Track const&, G4Step const&)
1.78	0.90	24.16	19.1M	47.3n	47.3n	G4Track::GetVelocity() const
1.62	0.82	24.98	4.48M	182n	720n	G4Navigator::ComputeStep(Hep3Vector const&, Hep3Vector const&, double, double&)
1.51	0.77	25.75	1.89M	405n	863n	G4MultipleScattering::GetContinuousStepLimit(G4Track const&, double, double, double&)
1.41	0.71	26.46	4.62M	155n	503n	G4ReplicaNavigation::ComputeStep(Hep3Vector const&, Hep3Vector const&, Hep3Vector const&, Hep3Vector const&, double, double, double&, G4NavigationHistory&, bool&, Hep3Vector&, bool&, bool&, G4VPhysicalVolume**, int&)
1.27	0.64	27.10	3.27M	196n	196n	G4Tubs::DistanceToOut(Hep3Vector const&, Hep3Vector const&, bool, bool*, Hep3Vector*) const
1.08	0.55	27.65	550k	999n	2.28u	G4eBremsstrahlung::PostStepDoIt(G4Track const&, G4Step const&)
1.07	0.54	28.20	5.01M	109n	170n	G4Transportation::AlongStepDoIt(G4Track const&, G4Step const&)
0.98	0.50	28.69	7.00M	70.7n	70.7n	_int_malloc
0.96	0.49	29.18	4.80M	102n	3.65u	G4SteppingManager::InvokePostStepDoItProcs()
0.93	0.47	29.65	25.8k	18.2u	18.6u	G4MultipleScattering::ComputeTransportCrossSection(G4ParticleDefinition const&,)
0.92	0.47	30.12	4.97M	93.8n	146n	Em2SteppingAction::UserSteppingAction(G4Step const*)
0.91	0.46	30.58	4.95M	93.4n	1.04u	G4SteppingManager::InvokeAlongStepDoItProcs()
0.91	0.46	31.04	9.64M	47.5n	230n	G4VDiscreteProcess::PostStepGetPhysicalInteractionLength(G4Track const&, double, double, double&, G4GPILSelection*)

HARDWARE REVIEW

Instruction delivery

- **Must match**
 - instructions to issue ports
 - w/corresponding execution units attached

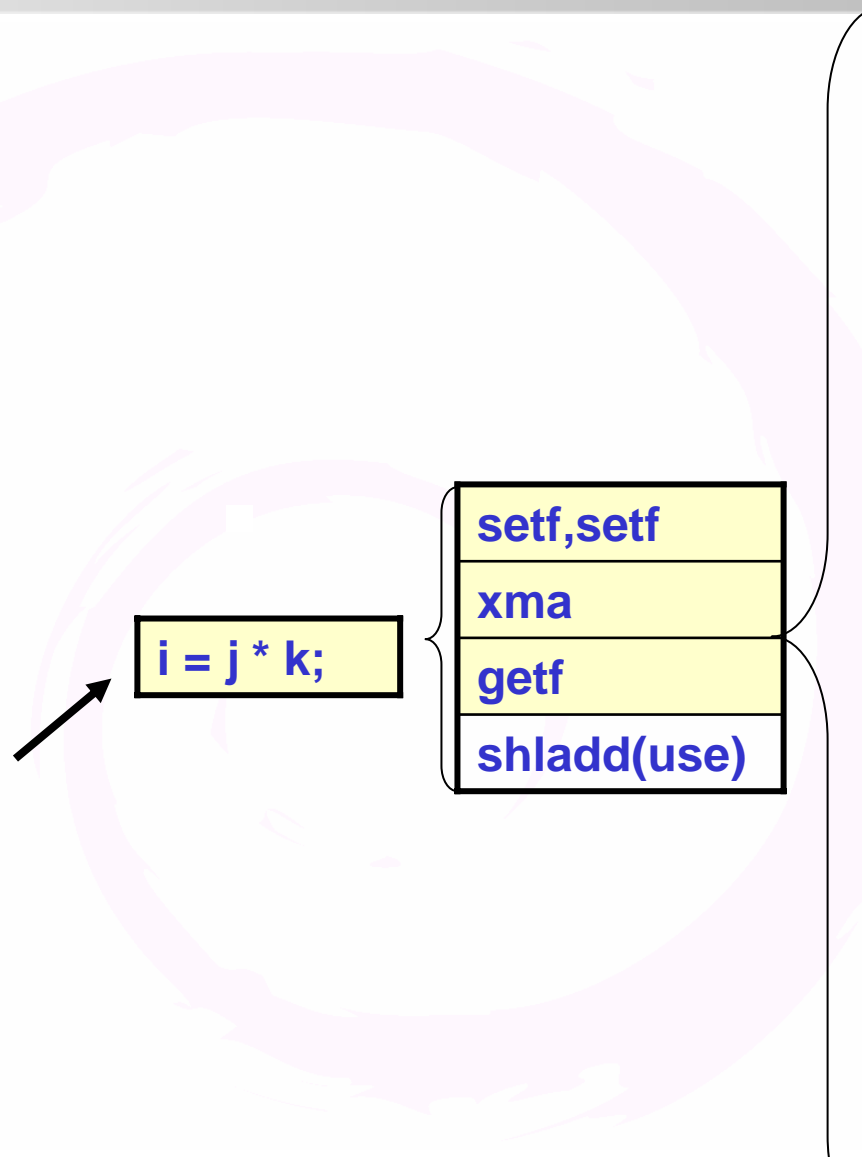


Max 6 instructions to 11 available ports in total

useful

Understanding latency

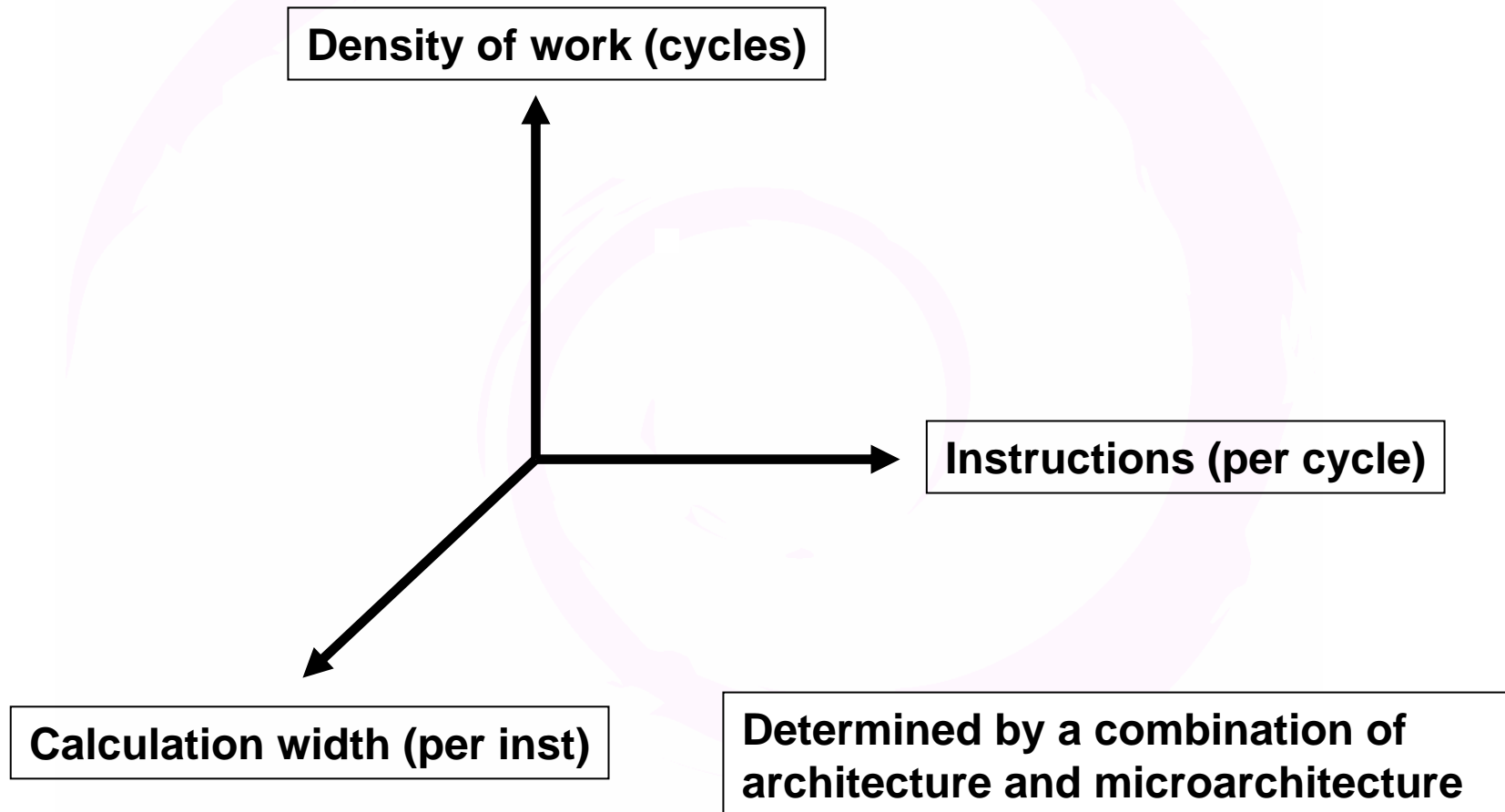
- All instructions have latency.
- “1” is normal
 - Ignore “0”
- What about “> 1” ?
- Affects (mainly) floating-point instructions
- But also such operations as integer multiply!



setf,setf
bubble
bubble
bubble
bubble
bubble
xma
bubble
bubble
bubble
bubble
getf
bubble
bubble
bubble
bubble
shladd(use)

CPU performance vector

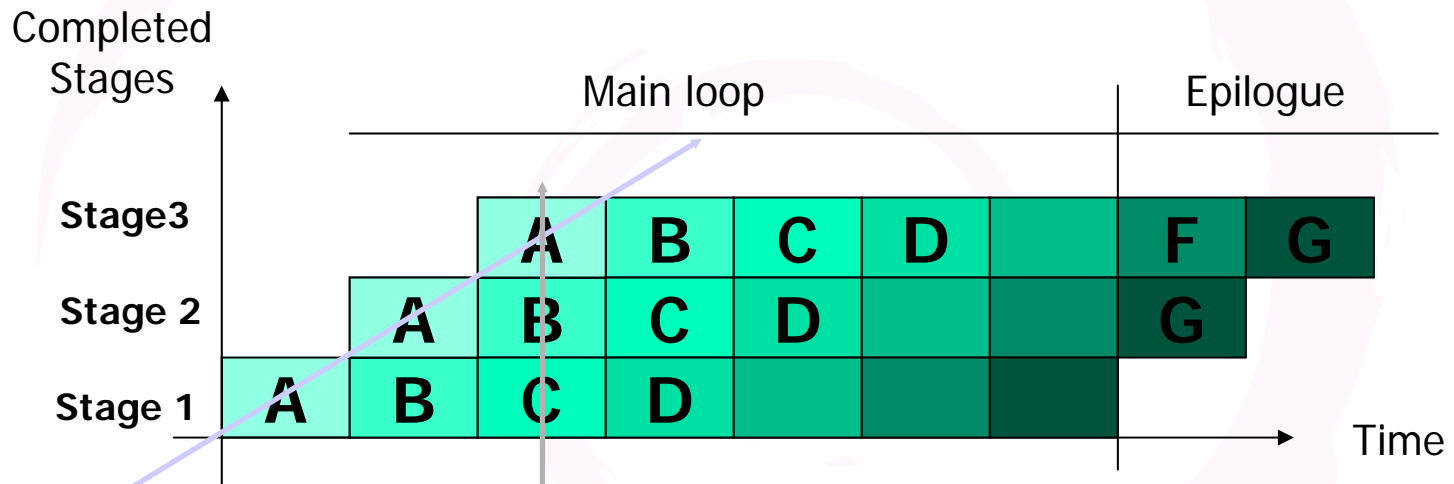
- **Defined in 3 dimensions**



Modulo Loops

– Graphical representation

- N loop traversals desired, but with skewed execution:
 - Stage 2 is offset relative to Stage 1
 - Stage 3 is offset relative to Stage 2



Analogy: Think of a bar where each customer (Red arrow) wants to:
1) order a drink, 2) drink it, 3) pay the bill.

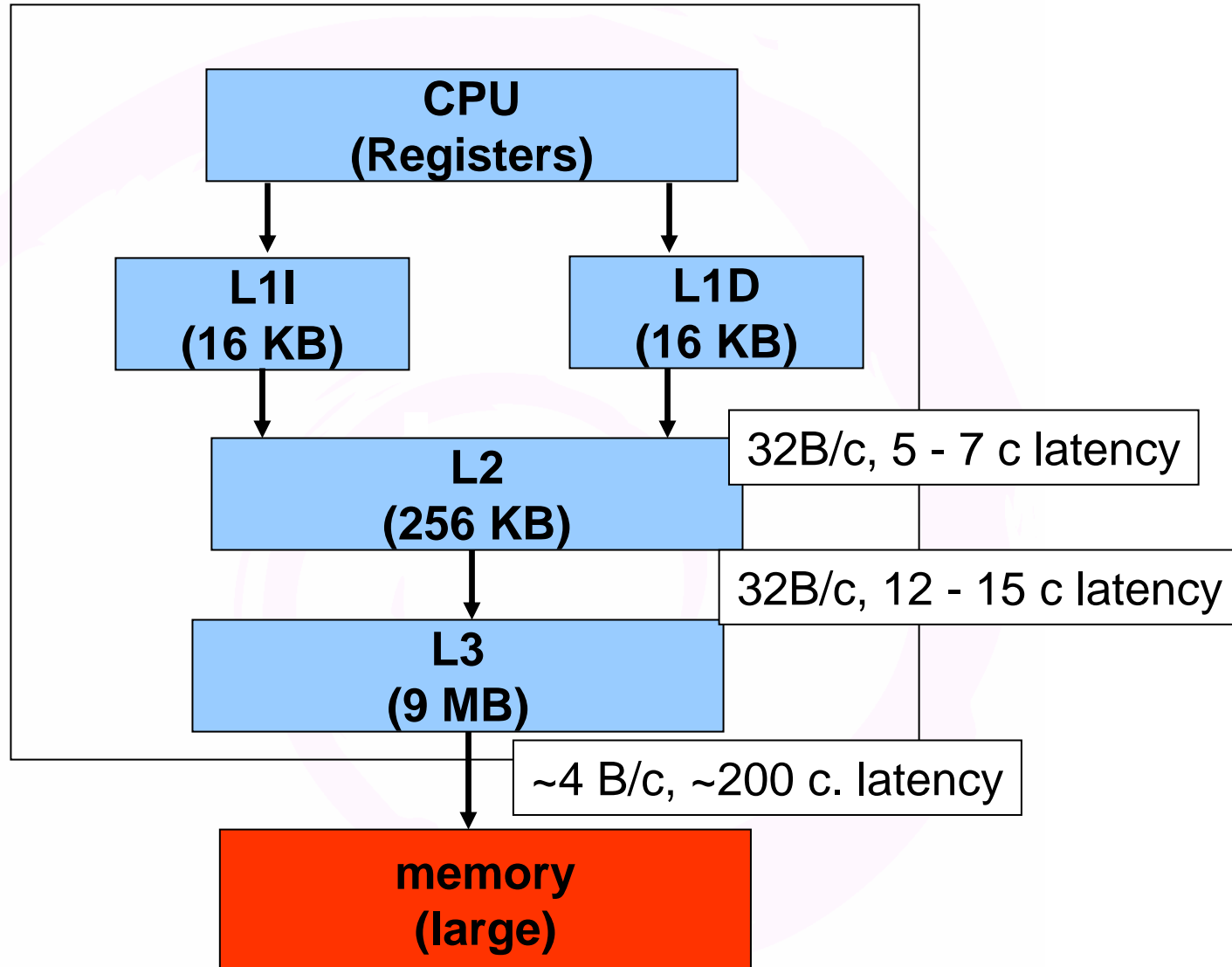
The waiter (Blue arrow) is working "flat out" by

1) taking an order from C, 2) serving a drink to B, 3) getting paid by A.

Memory Hierarchy

- **From CPU to main memory on Madison**

- With multicore, memory bandwidth is shared between cores on the same bus



Cache lines

- **L3 cache lines are 128B (16 * double)**
 - Minimum amount of data transferred between cache and memory.
 - Imagine what happens if your stride is 16 (or more)!



use

Programming the memory hierarchy is an art in itself.

In comes the PMU (Performance Monitoring Unit)

Quickly summarized:
4 counters (12 on Montecito)
~200 monitored events
Some very advanced features!

Geant 4 – Test40

- Overall counters in 10^9

Counter	Counts
IA64_INST_RETIRED	108.43
NOPS_RETIRED	39.73
CPU_CYCLES (CC)	74.98
BACK_END_BUBBLE_ALL	43.10

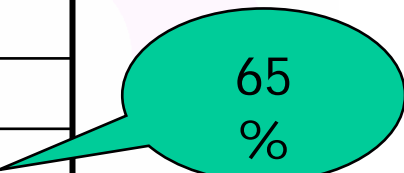
Useful instructions (UI)	68.70
Non-stalled cycles (NSC)	31.88

UI/CC	~ 1
UI/NSC	~ 2

Geant 4 – Test40

- Stall counters

Counter	Counts (10 ⁹)
BACK_END_BUBBLE_ALL	43.10
BE_EXE_BUBBLE_ALL	27.96
BACK_END_BUBBLE_FE	7.38
BE_L1D_FPU_BUBBLE_ALL	3.86
BE_FLUSH_BUBBLE_ALL	2.72
BE_RSE_BUBBLE_ALL	1.38



65
%

Why so many EXE bubbles?

Keep drilling down!

Geant 4 – Test40

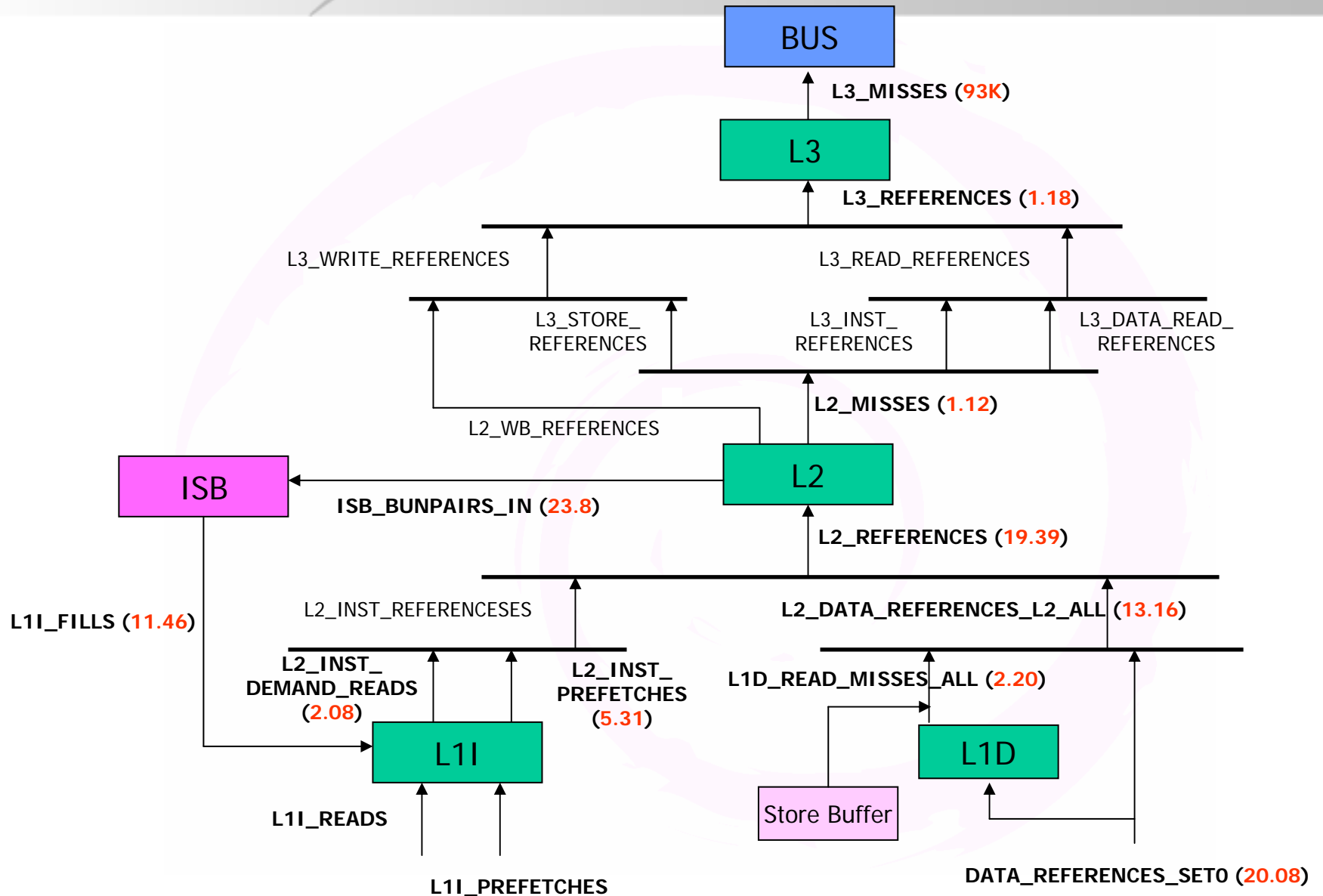
- EXE stall counters

Counter	Counts
BE_EXE_BUBBLE_ALL	27.96
BE_BUBBLE_GRALL	10.29
BE_BUBBLE_GRGR	~zero
BE_EXE_BUBBLE_FRALL	17.21

Counter	Counts
L2_REFERENCES	19.39
L2_DATA_REFERENCES_L2_ALL	13.16

ldf,ldf
bubble
bubble
bubble
bubble
bubble
fma
bubble
bubble
bubble
stf

Test40 - Cache counters



Software Pipelining

Mersenne Twister

```

Double_t TRandom3::Rndm(Int_t){
  UInt_t y;
  const Int_t kM = 397; const Int_t kN = 624; const UInt_t kTemperingMaskB = 0x9
  const UInt_t kTemperingMaskC = 0xefc60000; const UInt_t kUpperMask = 0x800
  const UInt_t kLowerMask = 0x7fffffff; const UInt_t kMatrixA = 0x990

  if (fCount624 >= kN) {
    register Int_t i;
    for (i=0; i < kN-kM; i++) { /* THE LOOPS */
      y = (fMt[i] & kUpperMask) | (fMt[i+1] & kLowerMask);
      fMt[i] = fMt[i+kM] ^ (y >> 1) ^ ((y & 0x1) ? kMatrixA : 0x0);
    }
    for ( ; i < kN-1 ; i++) {
      y = (fMt[i] & kUpperMask) | (fMt[i+1] & kLowerMask);
      fMt[i] = fMt[i+kM-kN] ^ (y >> 1) ^ ((y & 0x1) ? kMatrixA : 0x0);
    }
    y = (fMt[kN-1] & kUpperMask) | (fMt[0] & kLowerMask);
    fMt[kN-1] = fMt[kM-1] ^ (y >> 1) ^ ((y & 0x1) ? kMatrixA : 0x0);
    fCount624 = 0;
  }
  y = fMt[fCount624++]; /*THE STRAIGHT-LINE PART*/
  y ^= (y >> 11); y ^= ((y << 7) & kTemperingMaskB);
  y ^= ((y << 15) & kTemperingMaskC); y ^= (y >> 18);
  if (y) return ((Double_t) y * 2.3283064365386963e-10); // * Power(2,-32)
  return Rndm();
}

```

The “MT” loop is full

- **Highly optimized**

- Here depicted in 3 Itanium cycles

- But similarly dense on other platforms

0	Load	Test Bit	XOR	Load	Add	No-op
1	AND	AND	Shift	Add	Load	Move
2	Store	OR	XOR	Add	Add	Branch

The sequential part is not!

0	Add	Mov long	No-op	No-op	No-op	No-op
1	Load	Mov long	Mov long	No-op	No-op	No-op
2	Shift,11	Set float	No-op	No-op	No-op	No-op
3	XOR	Move	No-op	No-op	No-op	No-op
4	Shift,7	No-op	No-op	No-op	No-op	No-op
5	AND	No-op	No-op	No-op	No-op	No-op
6	XOR	No-op	No-op	No-op	No-op	No-op
7	SHL,15	No-op	No-op	No-op	No-op	No-op
8	AND	No-op	No-op	No-op	No-op	No-op
9	XOR	No-op	No-op	No-op	No-op	No-op
10	SHL,18	No-op	No-op	No-op	No-op	No-op
11	XOR	No-op	No-op	No-op	No-op	No-op
12	Set float	Compare	Branch	No-op	No-op	No-op
13	Bubble (no work dispatched)	<pre> y = fMt[fCount624++]; /*THE STRAIGHT-LINE PART*/ y ^= (y >> 11); y ^= ((y << 7) & kTemperingMaskB); y ^= ((y << 15) & kTemperingMaskC); y ^= (y >> 18); if (y) return ((Double_t) y * 2.3283064365386963e-10); </pre>				
14	Bubble (no work dispatched)					
15	Bubble (no work dispatched)					
16	Bubble (no work dispatched)					
17	Bubble (no work dispatched, because of FP latency)					
18	Mult FP	No-op	No-op	No-op	No-op	No-op
19	Bubble (no work dispatched, because of FP latency)					
20	Bubble (no work dispatched, because of FP latency)					
21	Bubble (no work dispatched, because of FP latency)					
22	Mult FP	Branch	No-op	No-op	No-op	No-op

Back to Compilers

“All created equal ?”

TestKalman [nx,ny] : kalman_win7.1

	2	3	4	5	6	7	8	9	10
2	0.41 0.40 1.01	0.55 0.56 1.29	0.80 0.79 1.65	1.26 1.31 2.18	2.16 2.36 3.16	3.91 3.84 7.13	5.70 5.14 8.85	7.43 6.97 11.24	9.66 8.90 13.66
3	0.52 0.51 1.24	0.70 0.70 1.50	0.98 0.98 1.97	1.49 1.52 2.61	2.43 2.62 3.68	4.35 4.15 7.82	6.07 5.46 9.53	8.23 7.55 12.26	10.09 9.17 14.95
4	0.63 0.62 1.50	0.85 0.85 1.88	1.24 1.17 2.19	1.73 1.77 3.09	2.79 2.86 4.31	4.77 4.46 8.53	6.65 5.93 10.56	8.64 7.93 13.77	10.86 10.01 16.58
5	0.78 0.83 1.81	1.04 1.09 2.24	1.41 1.45 2.91	2.11 2.10 3.49	3.12 3.22 5.02	5.12 4.90 9.40	7.17 6.53 11.56	9.64 8.70 14.88	11.45 10.56 17.61
6	0.85 0.98 2.13	1.16 1.29 2.65	1.68 1.72 3.40	2.28 2.49 4.37	3.50 3.72 5.49	5.57 5.44 10.36	8.12 7.07 12.53	9.94 9.22 16.09	12.50 11.42 19.24
7	1.04 1.10 2.44	1.50 1.48 3.09	2.01 1.99 3.95	2.79 2.80 4.95	4.03 4.15 6.47	6.24 5.89 10.88	8.48 7.64 13.44	10.76 9.80 17.59	13.30 11.96 20.76
8	1.22 1.26 2.81	1.69 1.71 3.57	2.30 2.30 4.48	3.18 3.16 5.57	4.59 4.57 7.28	6.89 6.47 12.02	9.24 8.69 14.23	11.67 10.78 18.69	14.35 13.03 22.77

N1,N2 <= 6 36.51
37.96
66.81
 N1,N2 > 6 261.15
242.13
421.54
 All N1,N2 297.67
280.08
488.35

SMatrix_Sym SMatrix TMatrix SMatrix_Sym better than TMatrix

TestKalman [nx,ny] : kalman_solaris.5.9

	2	3	4	5	6	7	8	9	10
2	2.29 1.39 2.49	4.53 2.41 2.95	7.49 3.52 3.84	13.36 5.57 5.15	27.92 9.88 8.18	30.68 20.13 34.74	42.12 29.90 42.52	56.27 41.89 51.08	74.79 56.08 61.38
3	3.36 2.05 2.83	6.28 3.43 3.49	9.88 5.09 4.74	17.60 7.79 6.23	33.32 12.81 9.61	37.58 24.26 36.25	51.27 35.10 44.61	69.29 50.11 53.69	88.88 66.09 63.78
4	4.70 2.92 3.50	8.39 4.82 4.41	13.02 7.16 5.68	21.30 10.45 7.46	38.09 16.27 11.42	44.86 28.23 38.35	62.55 43.15 47.23	83.96 60.94 56.35	108.25 79.72 67.32
5	6.45 3.84 3.87	11.09 6.42 5.10	16.75 9.42 6.78	26.01 13.43 8.88	45.35 20.22 12.96	52.92 32.57 40.86	73.96 50.84 49.51	100.57 72.06 59.60	127.69 94.24 70.80
6	8.77 5.36 4.58	14.55 8.67 6.12	21.27 12.45 8.33	32.27 17.49 10.55	51.35 25.37 14.90	63.23 39.55 43.39	87.57 60.54 52.76	118.44 84.29 63.18	152.52 112.31 75.01
7	12.58 6.85 5.27	20.21 10.88 7.13	29.16 15.45 9.41	42.12 21.34 12.36	64.82 29.96 17.47	78.81 44.96 45.91	107.49 68.27 55.53	142.03 96.24 66.99	183.05 128.52 79.38
8	17.68 10.79 6.08	28.33 17.19 8.30	40.40 24.55 10.98	57.23 33.55 14.26	84.57 46.08 19.58	103.45 64.54 48.60	139.67 95.46 58.98	184.39 132.12 70.57	232.32 170.91 83.94

N1,N2 <= 6 445.38
218.23
164.08
 N1,N2 > 6 3095.72
2099.65
1673.16
 All N1,N2 3541.10
2317.89
1837.24

SMatrix_Sym SMatrix TMatrix SMatrix_Sym better than TMatrix

TestKalman [nx,ny] : kalman_slc3_gcc323

	2	3	4	5	6	7	8	9	10
2	0.30 0.33 0.86	0.38 0.44 1.00	0.56 0.64 1.39	0.88 0.98 1.69	1.54 1.64 2.96	5.77 5.84 5.22	8.00 7.81 6.06	10.41 10.23 7.41	14.36 14.58 9.03
3	0.37 0.46 1.01	0.56 0.60 1.16	0.72 0.99 1.59	1.10 1.29 1.96	1.84 2.03 3.40	6.24 6.33 5.48	8.19 8.17 6.64	10.97 10.76 7.61	14.37 14.02 9.86
4	0.47 0.61 1.16	0.63 0.76 1.42	0.89 1.03 1.72	1.39 1.48 2.48	2.16 2.27 3.67	6.71 6.43 6.14	9.04 8.92 6.95	11.71 11.37 8.85	15.07 14.69 10.33
5	0.60 0.78 1.28	0.85 1.03 1.55	1.19 1.28 2.35	1.71 1.80 2.69	2.58 2.70 4.44	7.03 6.79 6.60	9.52 9.18 8.34	12.41 12.03 9.44	15.74 16.00 12.16
6	0.77 0.96 1.59	1.26 1.22 2.09	1.49 1.60 2.42	2.13 2.17 3.58	3.06 3.12 4.61	7.81 7.39 7.55	10.19 9.90 8.09	12.98 12.53 10.36	17.56 16.92 11.58
7	0.96 1.25 1.75	1.33 1.49 2.17	1.77 1.99 3.03	2.46 2.57 3.53	3.47 3.62 5.48	8.24 8.08 7.90	10.56 10.13 9.50	13.08 12.72 10.62	18.03 16.96 14.14
8	1.14 1.48 2.05	1.68 1.79 2.81	2.15 2.33 3.02	2.95 3.14 4.67	4.07 4.27 5.59	8.99 8.79 8.69	11.47 11.37 9.34	14.48 14.36 12.29	19.15 18.07 13.71

N1,N2 <= 6 29.45
32.23
54.07
 N1,N2 > 6 340.08
334.29
283.99
 All N1,N2 369.53
366.52
338.05

SMatrix_Sym SMatrix TMatrix SMatrix_Sym better than TMatrix

“Low- hanging fruit”

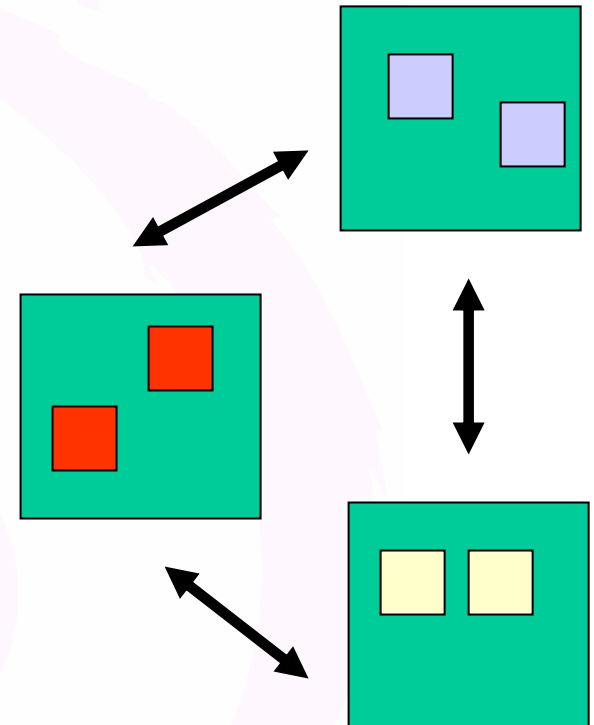
- Typically one starts with a given compiler, and moves to:
- **More aggressive compiler options**
 - For instance:
 - -O2 → -O3, -funroll-loops, -ffast-math (g++)
 - -O2 → -O3, -ipo (icc)
- **More recent compiler versions**
 - g++ version 3 → g++ version 4
 - icc version 8 → icc version 9
- **Different compilers**
 - GNU → Intel (or reverse?)

Some options can compromise accuracy or correctness

May be a burden because of potential source code issues

Interprocedural optimization

- **Let the compiler worry about interprocedural relationship**
 - “icc -ipo”
- **Valid also when building libraries**
 - Archive
 - Shared
- **Cons:**
 - Can lead to code bloat
 - Longer compile times



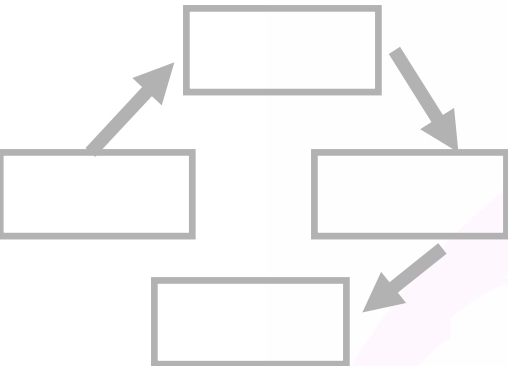
Probably most useful when combined with heavy optimization for “production” binaries or libraries!

- **Many compilers allow further optimization through training runs**
 - Compile once (to instrument binary)
 - g++ -fprofile-generate
 - icc -prof_gen
 - Run one (or several test cases)
 - ./test40 < test40.in (will run slowly)
 - Recompile w/feedback
 - g++ -fprofile-use
 - icc -prof_use (best results when combined with -O3,-ipo)

With icc 9.0 we get ~20% on root stress tests on Itanium, but only ~5% on x86-64

CONCLUSION

Conclusions



Itanium is definitely a great platform for bottleneck analysis and performance tuning!

- Understand which parts of the “spiral” you control
- Understand the Itanium hardware
- Equip yourself with good tools
 - Get access to hw performance counters
 - Exploit the power of Itanium-based performance tools
- Check how key algorithms map on to your hardware platform
 - Are you at 5% or 95% efficiency?
 - Where do you want to be?
- Cycle around the spiral frequently
 - It is hard to get to “peak” performance (and stay there!)

QUESTIONS?

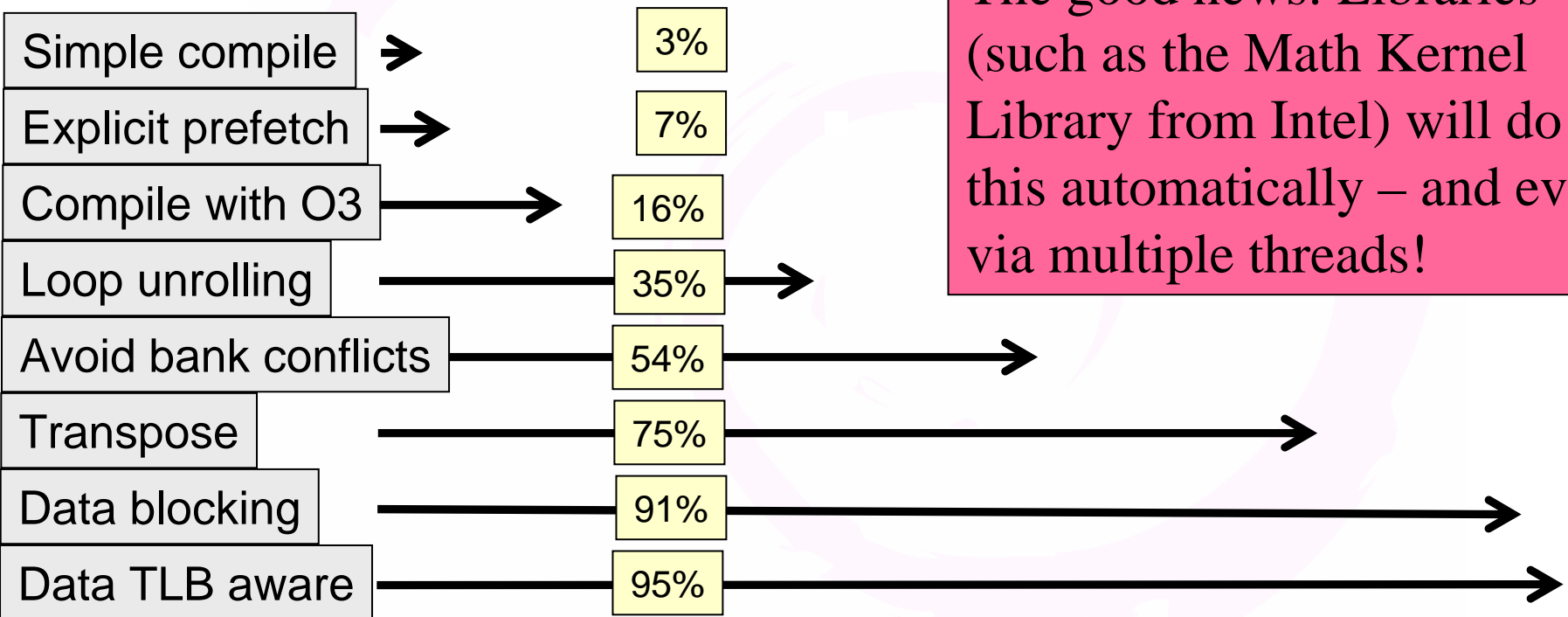
Backup

Useful to know the hardware ?

- **Matrix multiply example (IPF)**

- From D.Levinthal/Intel (Optimization talk at IDF, spring 2003)

- Basic algorithm: $C_{ik} = \text{SUM}(A_{ij} * B_{jk})$



The good news: Libraries (such as the Math Kernel Library from Intel) will do this automatically – and even via multiple threads!