Optimizing CMS Software to the CPU

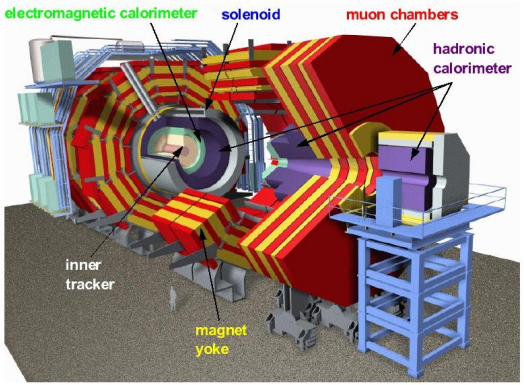
Peter Elmer – Princeton University
(for the CMS collaboration)
ACAT 2010, Jaipur, India
26 Feb, 2010

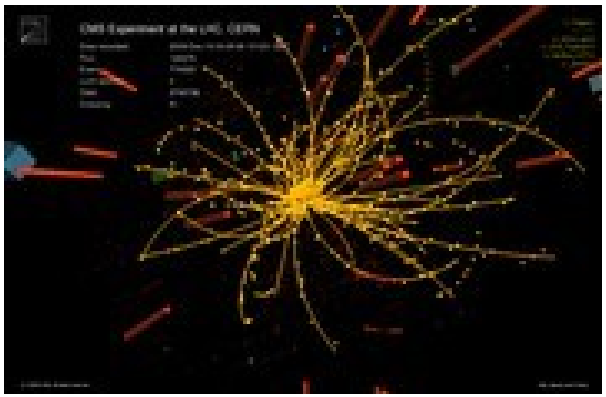
CMS at the LHC

CMS is a general purpose colliding beam experiment at the LHC at CERN

39 countries, 182 institutions, 3800+ scientists and engineers

Its total life cycle will undoubtedly run to 3-4 decades (from conception to end-of-life)





CMS Software Performance

- CMS has a very inclusive software development model. Many people are involved, with varying technical skill levels.
- CMS has had an ongoing effort since several years to improve the performance of the software

  From the top down: improvements in algorithms (reco, ...)

  From the bottom up: technical improvements

- The technical improvements have often focused on memory management and use, basic C++ performance issues, use of STL. This has included also larger migrations such as from CLHEP matrices to ROOT SMatrix as well as the basic compiler and 64bit migrations.

CMS Software Performance

- We feel we are in “reasonable shape” today (FLW)

  (Much) egregiously wasteful stuff has been removed

  Data taking and (really) hitting resource limits will test this in the next year or so

- Current performance work is thus more forward looking, with two regimes:

  Capitalization on new compiler features, 64bit, etc.

  Multicore – near term and longer term

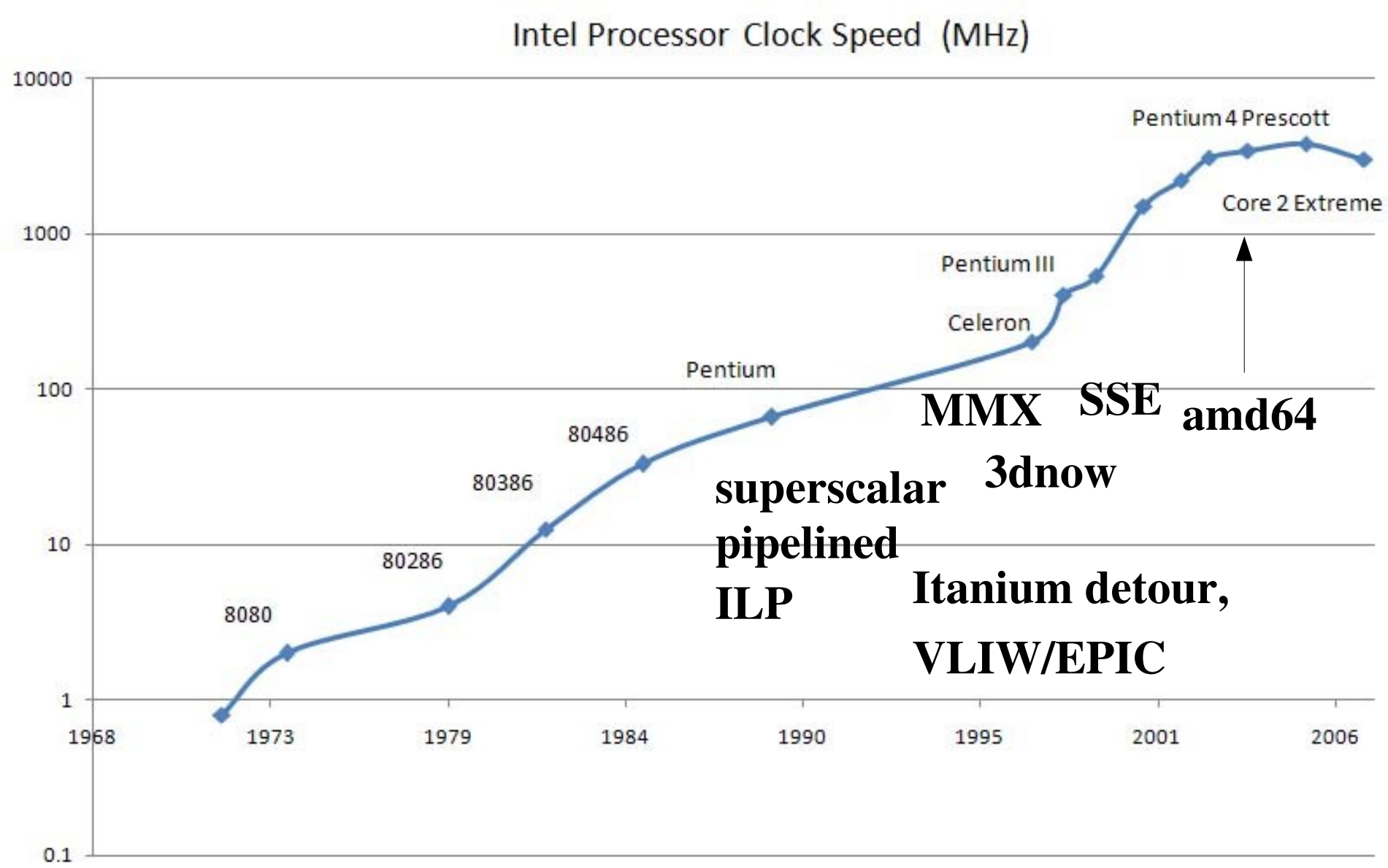
- Not just CMSSW, at times this includes work with the important software externals (Geant4, ROOT and others)

25 years of commodity CPU's

- i386 – 1985
- Pentium – 1993
- AMD 64 - 2003

The little CPU we had at home grew up into the CPU we use in our computer centers at work....

CPU Clockspeeds



Back to the Future

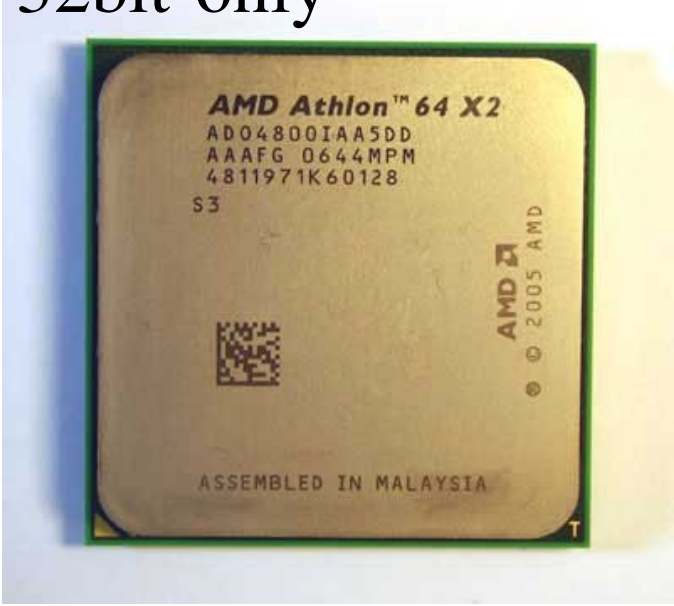
Even though it is now 2010, and HEP largely moved to commodity hardware in the late 1990's, we are in some sense still stuck in 1985.

- General tendency to build for the i386 and take whatever the CPU gives
- Clock frequencies increased in any case plus other improvements out-of-the-box
- In general little exploitation of newer CPU features or detailed performance study

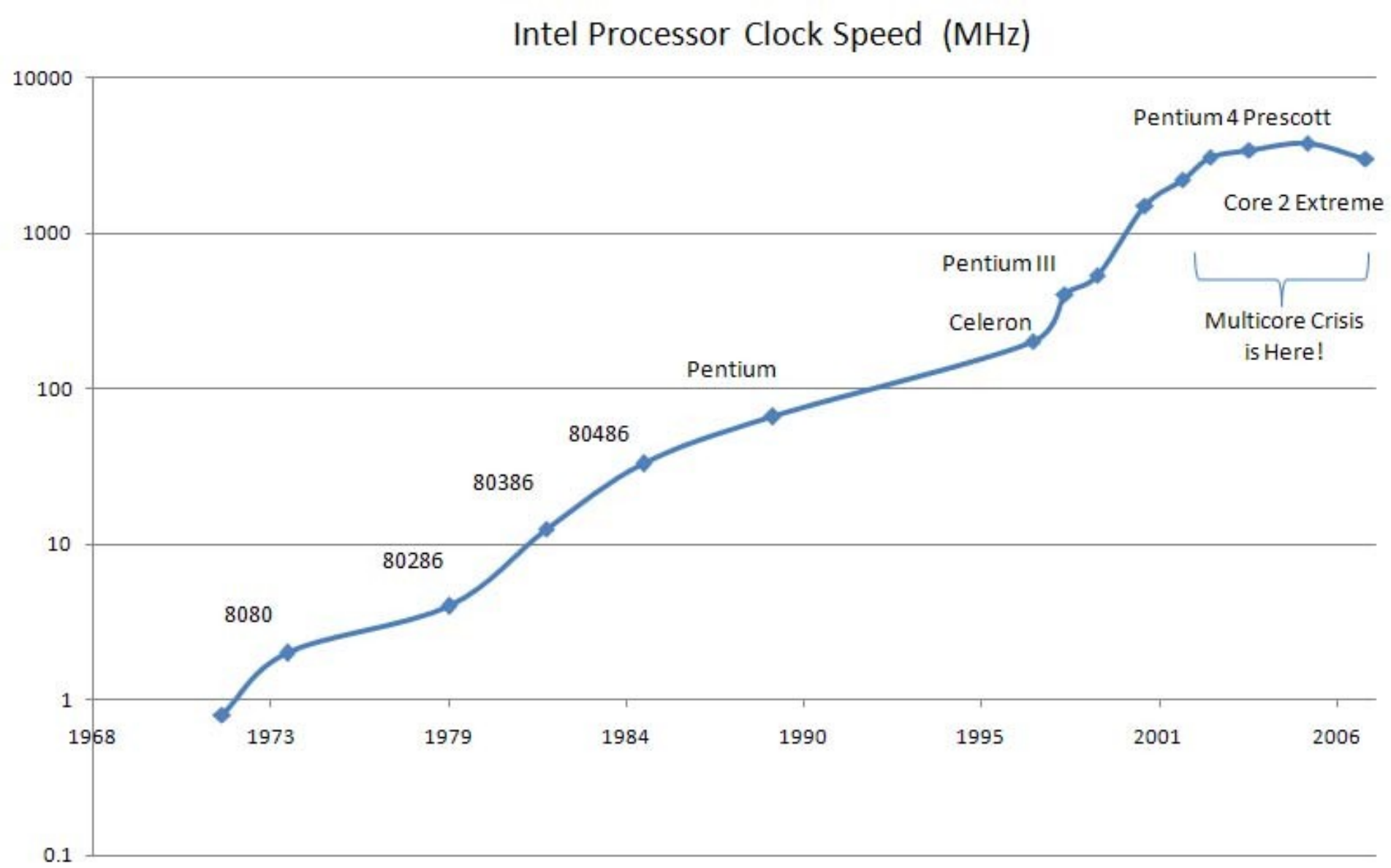
- mmx/3dnow/sse confusion, egcs/gcc confusion, no gain
- heterogeneous computing: want “build once, run anywhere”
- little experience with exploitation of new features anyway
- Community learning C++ (and OO): other problems.

Several leaps forward

- The biggest leap forward is x86_64, a “new architecture” implies a “renormalization” point for CPU features (e.g. SSE2), in addition to being a leap forward in and of itself
- Ongoing compiler improvements (gcc, in particular) + better tools!! (Still lacking experience at some level, though.)
- And in 2010 we have more or less flushed 32bit-only CPU's out of the system



CPU Clockspeeds

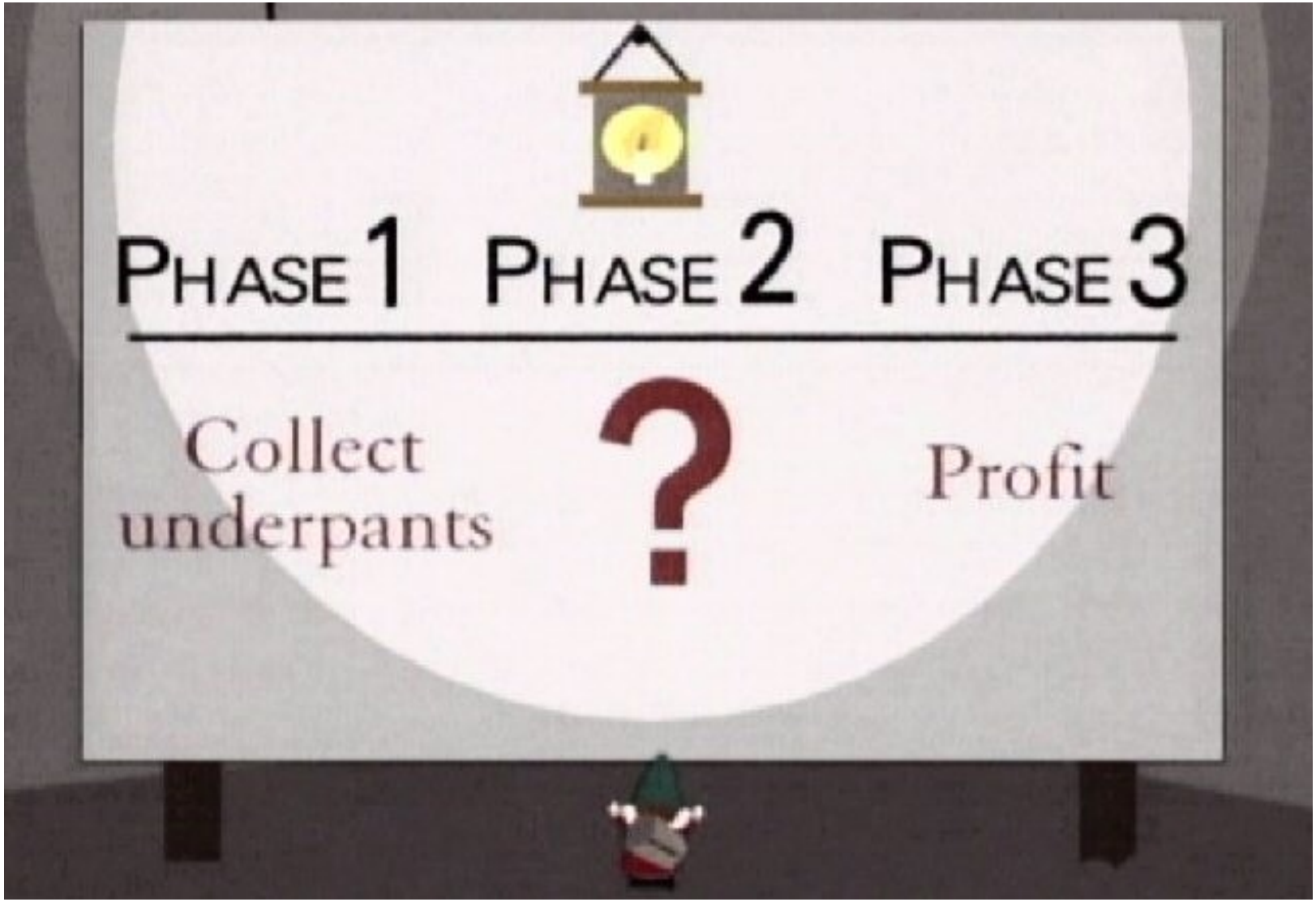


And add Multicore

- The frequency scaling era has ended, and additional transistors are now used to provide multiple cores → multicore
- So it isn't just that we now have the option of capitalizing on CPU and compiler improvements (perhaps with code changes), other changes are upon us which may eventually remove some the “guaranteed” gains with each generation of CPU
- In short, it appears that in the future gains with newer CPU's may not come for free. At some level we need to revisit our code and optimize it to the CPU.

In short, now what?

Like the famous 3-phase business plan of the “underpants gnomes”:



Performance Survey

- Systematic "top N" survey of performance of a number of CMS benchmarks, using multiple tools and types of profiles

  Look at both CPU and various aspects of memory use.

  Get more experience with some of the tools

- Discuss (in CMS HyperNews) and *document* what we see.

  Avoid physicist bias that "real men only do things ad hoc"

- "Forward looking" implies focusing only on the native 64bit builds at this point

  Seems obvious, but this step is where many real projects are!

  First step is comparing in detail 32bit and 64bit, though.

Performance Survey - Goals

- Compare 32bit/64bit builds to understand gains/losses
- Using 64bit builds, identify opportunities for:

  general algorithmic code performance improvements

  improvements to data structures in memory

  further reduction of dynamic memory use

  additional ILP, data parallelism through vectorization

  fine-grained (thread level) parallelism
- A sort of “Great Trigonometric Survey” of our software in terms of CPU performance, memory use, etc.

Tools used

- Igprof – a sampling performance profiler as well as memory profiler (heap, total dynamic memory allocations, leaks)

  64bit support added

  Web browsable reports added

  "allocation locality" metrics added

- pfmon (+ libpfm for selective profiling, e.g. by "module")
- Intel Performance Tuning Utility (PTU)

  On openlab machines at CERN

- Gain additional experience with using the latter tools

Overall memory use numbers

high pT QCD - GEN-SIM-DIGI-L1-DIGIZRAW-HLT-RAW2DIGI-L1Reco

	VSIZE MB	RSS MB	Heap size bytes	Heap size # allocs	codesize
32bit	1207.22	970.45	715'224'260	7'338'346	244352 kB
64bit	1487.68	1215.13	901'359'522	7'330'717	261644 kB
diff	280.46 MB	244.68 MB	177.5 MB	(~55MB?)	16.9 MB

sum ~ 249 MB

high pT QCD - RAW2DIGI-RECO

	VSIZE MB	RSS MB	Heap size bytes	Heap size # allocs	codesize
32bit	915.25	803.00	536'132'974	5'464'112	195632 kB
64bit	1146.44	1001.74	694'426'284	5'468'690	211380 kB
diff	231.19 MB	198.74 MB	151.0 MB	(~42MB?)	15.4 MB

sum ~ 208 MB

64bit memory use (footprint)

- As discussed elsewhere, VSIZE is a poor measure of actual use
- Comparing the 32bit/64bit igprof heap profiles in detail, we can identify *specifically* increases from one or more of the following:

  overhead/alignment from malloc'd memory

  Alignment padding in data structures

  Types who size actually increases (e.g. pointer sizes, size_t, …) + vtables

- Focus at first on once/job data structures: a very typical problem in our code is unnecessary maps or map-variants, often these can be simplified. Others involved lots of individual new's, but can simply be container-ized or made more compact.

Heap profile survey

Annotated list of allocation points + link to igprof heap profile call graph:

51	11.28	78,365,197	1,033	std::vector<char, std::allocator<char> >::_M_fill_insert(__gnu_cxx::__normal_iterator<char*, std::vector<char, std::allocator<char> > >, u

Most of the bytes here are due to blobbed buffers being read from the conditions DB, e.g. dominated by the pixel gains/calibrations
(Should follow up the get the full list of payloads, though)

Rank	Total %	Self (bytes)	Calls (allocs)	Symbol name - Being Investigated
110	5.45	37,821,696	1,628	TCint::CallFunc_Factory() const

Axel notes that the large size here is due to one (usually unnecessarily) huge type G__param
Philippe sent one possible solution

118	4.93	34,265,520	856,638	std::_Rb_tree<DetId, DetId, std::_Identity<DetId>, std::less<DetId>, std::allocator<DetId> >::_M_insert(std::_Rb_tree_node_base const*, s

This comes from DetIdAssociator::buildMap()
Now fixed by Dima with tag TrackingTools/TrackAssociator V04-02-01, reducing the size in the heap by an order of magnitude and getting rid of nearly all of the allocatio

125	4.50	31,263,136	577,632	std::basic_string<char, std::char_traits<char>, std::allocator<char> >::_Rep::_S_create(unsigned long, unsigned long, std::allocator<char>

This one increased from 24,286,215 bytes (577,623 allocations) in the 32bit profile
Some fraction (but not all) of this comes from ROOT, see for example here and other places

121	4.49	31,181,040	44,814	GeometricDet::GeometricDet(DDFilteredView*, GeometricDet::GDEnumType)

This one increased from 27,413,776 bytes (44,814 allocations) in the 32bit profile
This was discussed a bit in this thread and Vincenzo committed some changes (still to benchmark)

192	2.78	19,313,568	134,122	TMVA::DecisionTree::CreateNode()

This one increased from 15,021,664 bytes (134,122 allocations) in the 32bit profile
There was some discussion about this, including:
Flattening the pointer-chasing navigation later into some flat/pooled/indexed memory structure
Fixing the padding/alignment in DecisionTreeNode (minor effect)

Look both at the number of bytes *and* the number of individual allocations
Examples from the top of the list, but a longer list was actually investigated

Heap profile survey

One (bad) example that slipped in last year: array of array of std::set's (of ints)

Counter: MEM_LIVE

Before

Rank	% total	Counts: to / from this	Counts: Total	Calls: to / from this	Calls: Total	Paths: Including child / parent	Paths: Total	Symbol name
	4.93	34,261,280	40,240,016	855,098	855,694	1	1	DetIdAssociatorESProducer::produce(DetIdAssociatorRecord const&)
[119]	**4.93**	**0**	**34,261,280**	**0**	**855,098**	**1**	**1**	**DetIdAssociator::buildMap()**
	4.93	34,203,520	34,203,520	855,088	855,088	1	22	std::_Rb_tree<DetId, DetId, std::_Identity<DetId>, std::less<DetId>,
	0.01	57,384	57,384	5	5	1	1	HcalDetIdAssociator::getASetOfValidDetIds() const
	0.00	200	338	3	5	1	3	edm::LogVerbatim::~LogVerbatim()
	0.00	96	96	1	1	1	1	MuonDetIdAssociator::getDetIdPoints(DetId const&) const
	0.00	80	6,320	1	70	1	73	_dl_runtime_resolve

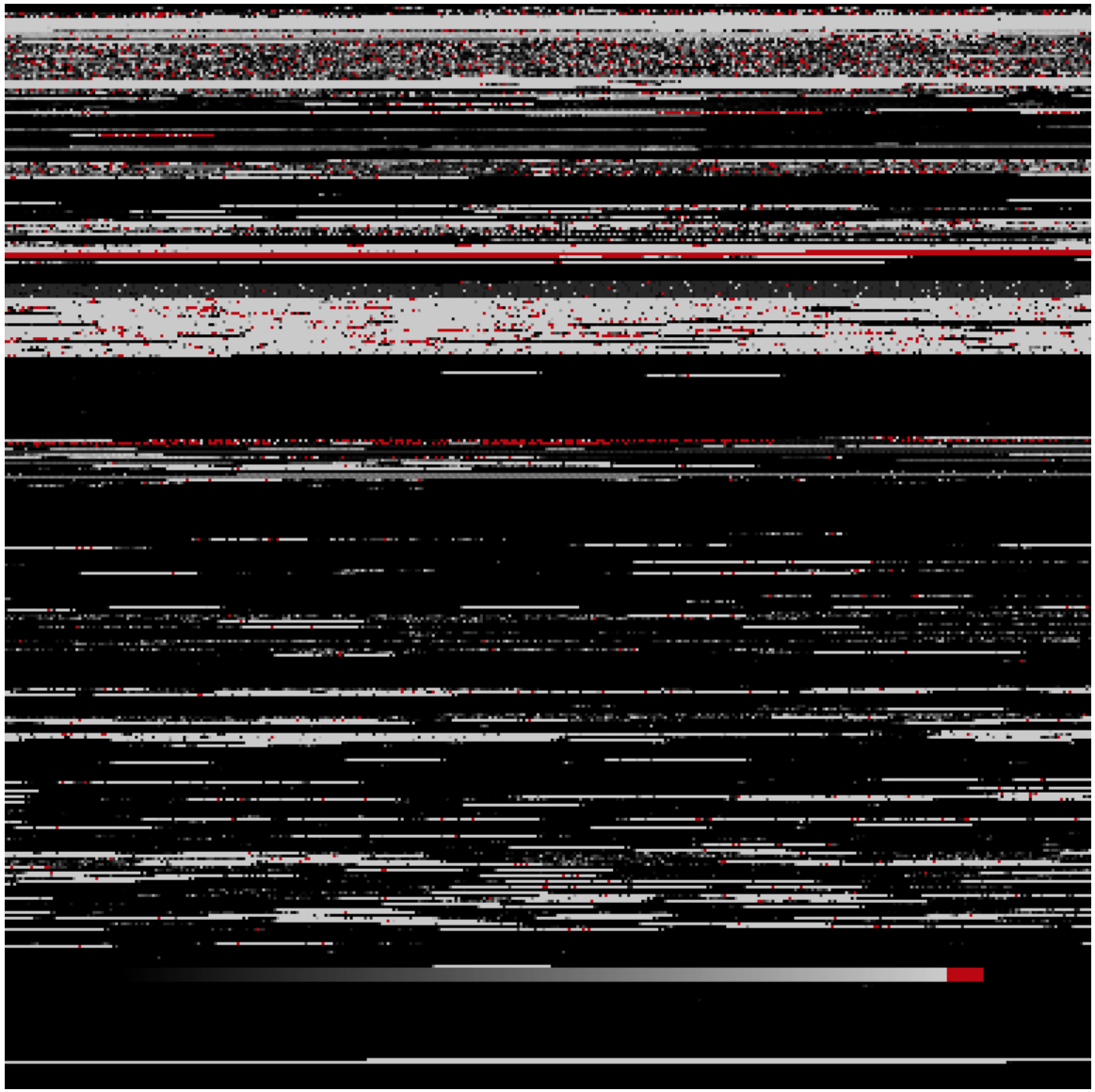
Back to summary

After

Rank	% total	Counts: to / from this	Counts: Total	Calls: to / from this	Calls: Total	Paths: Including child / parent	Paths: Total	Symbol name
	0.53	3,482,208	4,478,200	17	35	1	1	DetIdAssociatorESProducer::produce(DetIdAssociatorRecord const&)
[610]	**0.53**	**0**	**3,482,208**	**0**	**17**	**1**	**1**	**DetIdAssociator::buildMap()**
	0.52	3,420,352	3,474,624	6	1,702	1	2	std::vector<DetId, std::allocator<DetId> >::_M_fill_insert(__gnu_cxx::__norma
	0.01	57,384	57,384	5	5	1	1	CaloGeometry::getValidDetIds(DetId::Detector, int) const
	0.00	4,096	4,096	1	1	1	1	MuonDetIdAssociator::getValidDetIds(unsigned int) const
	0.00	200	338	3	5	1	3	edm::LogVerbatim::~LogVerbatim()
	0.00	96	96	1	1	1	1	MuonDetIdAssociator::getDetIdPoints(DetId const&) const
	0.00	80	7,560	1	83	1	89	_dl_runtime_resolve

Back to summary

Allocation Locality



We have also begun to look a bit at allocation locality, in addition to raw numbers of allocations in the heap

Virtual memory space plot
Each pixel is a page
White is full
Black is empty
Red indicates a page with allocations (strings) from a specific stacktrace

We also have quantitative reports, but plot illustrates the problem.

CPU performance (64bit)

32bit

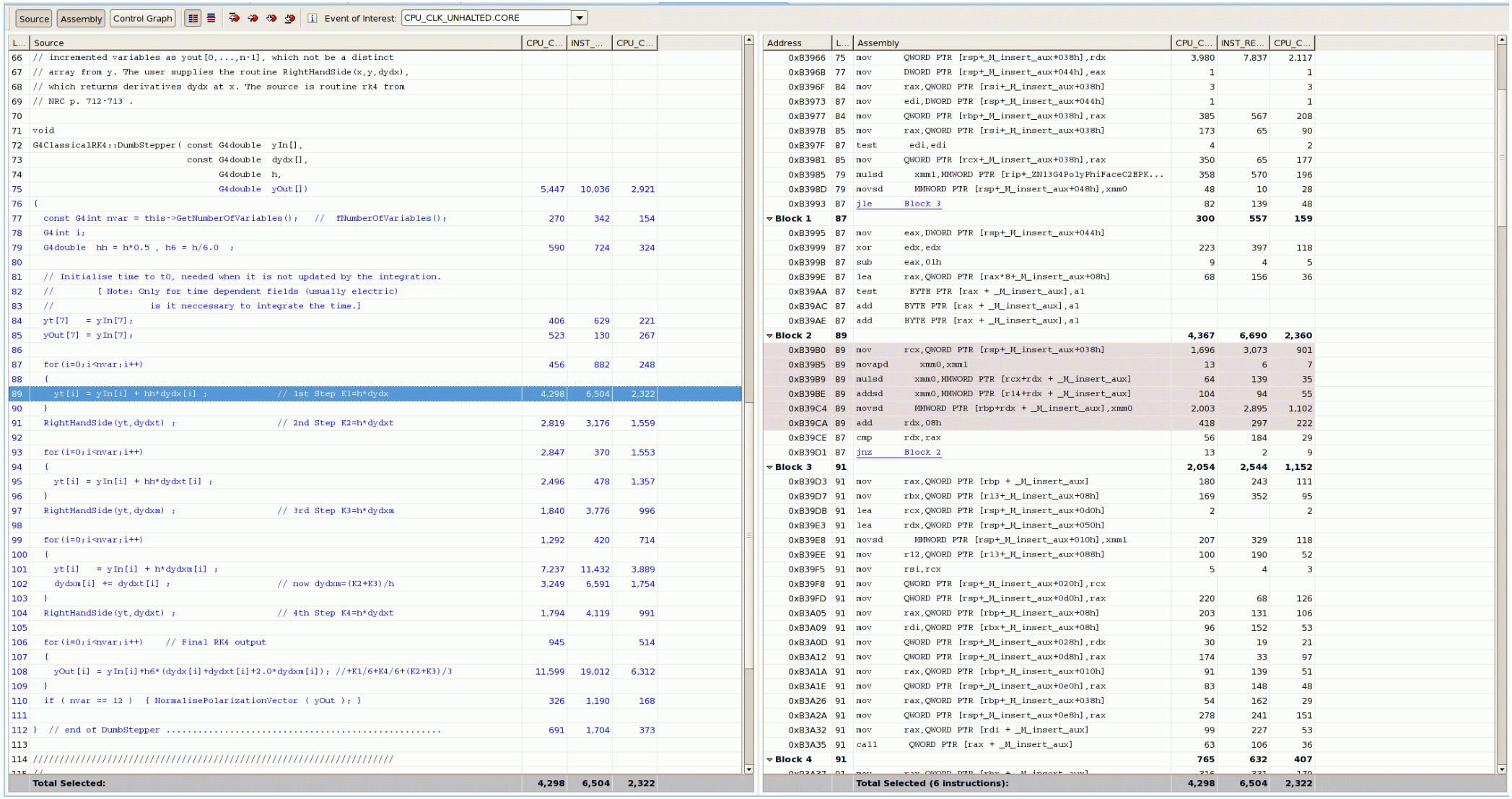
Rank	Total %	Self	Symbol name
51	2.93	1,097.55	__i686.get_pc_thunk.bx
60	2.25	841.34	G4Mag_UsualEqRhs::EvaluateRhsGivenB(double const*, double const*, double*) const
43	2.13	799.19	G4ClassicalRK4::DumbStepper(double const*, double const*, double, double*)
41	1.89	706.91	G4Navigator::LocateGlobalPointAndSetup(CLHEP::Hep3Vector const&, CLHEP::Hep3Vector const*, bool, bool)
70	1.72	645.61	atan2
72	1.67	625.23	CLHEP::HepJamesRandom::flat()
28	1.58	590.48	G4Navigator::ComputeStep(CLHEP::Hep3Vector const&, CLHEP::Hep3Vector const&, double, double&)
23	1.54	576.67	G4SteppingManager::DefinePhysicalStepLength()
77	1.53	573.74	_init
47	1.50	562.96	G4VEmProcess::PostStepGetPhysicalInteractionLength(G4Track const&, double, G4ForceCondition*)
93	1.34	501.27	__ieee754_exp

64bit

Rank	Total %	Self	Symbol name
47	3.56	1,061.33	__ieee754_log
48	3.51	1,044.86	__ieee754_atan2
33	2.80	833.81	G4Navigator::LocateGlobalPointAndSetup(CLHEP::Hep3Vector const&, CLHEP::Hep3Vector const*, bool, bool)
60	2.24	666.34	G4Mag_UsualEqRhs::EvaluateRhsGivenB(double const*, double const*, double*) const
22	2.24	666.07	G4SteppingManager::DefinePhysicalStepLength()
30	1.96	583.81	G4VoxelNavigation::ComputeStep(CLHEP::Hep3Vector const&, CLHEP::Hep3Vector const&, double, double&, G4Na
26	1.93	576.15	G4Navigator::ComputeStep(CLHEP::Hep3Vector const&, CLHEP::Hep3Vector const&, double, double&)
63	1.79	533.31	G4HadronCrossSections::CalcScatteringCrossSections(G4DynamicParticle const*, double, double)
42	1.74	517.57	G4VEmProcess::PostStepGetPhysicalInteractionLength(G4Track const&, double, G4ForceCondition*)
41	1.74	517.19	G4ClassicalRK4::DumbStepper(double const*, double const*, double, double*)
80	1.51	450.73	_int_malloc
81	1.50	447.35	G4VProcess::SubtractNumberOfInteractionLengthLeft(double)

Transcendental math slower from libm

Source code view with PTU



L...	Source	CPU_C...	INST_...	CPU_C...
66	// incremented variables as yout[0,...,n-1], which not be a distinct			
67	// array from y. The user supplies the routine RightHandSide(x,y,dydx),			
68	// which returns derivatives dydx at x. The source is routine rk4 from			
69	// NRC p. 712-713 .			
70				
71	void			
72	G4ClassicalRK4::DumbStepper(const G4double yIn[],			
73	const G4double dydx[],			
74	G4double h,			
75	G4double yOut[])	5,447	10,036	2,921
76	{			
77	const G4int nvar = this->GetNumberOfVariables(); // fNumberOfVariables();	270	342	154
78	G4int i;			
79	G4double hh = h*0.5 , h6 = h/6.0 ;	590	724	324
80				
81	// Initialise time to t0, needed when it is not updated by the integration.			
82	// [Note: Only for time dependent fields (usually electric)			
83	// is it neccessary to integrate the time.]			
84	yt[7] = yIn[7];	406	629	221
85	yOut[7] = yIn[7];	523	130	267
86				
87	for(i=0;i<nvar;i++)	456	882	248
88	{			
89	yt[i] = yIn[i] + hh*dydx[i] ; // 1st Step K1=h*dydx	4,298	6,504	2,322
90	}			
91	RightHandSide(yt,dydxt) ; // 2nd Step K2=h*dydxt	2,819	3,176	1,559
92				
93	for(i=0;i<nvar;i++)	2,847	370	1,553
94	{			
95	yt[i] = yIn[i] + hh*dydxt[i] ;	2,496	478	1,357
96	}			
97	RightHandSide(yt,dydxm) ; // 3rd Step K3=h*dydxm	1,840	3,776	996
98				
99	for(i=0;i<nvar;i++)	1,292	420	714
100	{			
101	yt[i] = yIn[i] + h*dydxm[i] ;	7,237	11,432	3,889
102	dydxm[i] += dydxt[i] ; // now dydxm=(K2+K3)/h	3,249	6,591	1,754
103	}			
104	RightHandSide(yt,dydxt) ; // 4th Step K4=h*dydxt	1,794	4,119	991
105				
106	for(i=0;i<nvar;i++) // Final RK4 output	945		514
107	{			
108	yOut[i] = yIn[i]+h6*(dydx[i]+dydxt[i]+2.0*dydxm[i]); //+K1/6+K4/6+(K2+K3)/3	11,599	19,012	6,312
109	}			
110	if (nvar == 12) { NormalisePolarizationVector (yOut); }	326	1,190	168
111				
112	} // end of DumbStepper	691	1,704	373
113				
114	//			
	Total Selected:	**4,298**	**6,504**	**2,322**

Address	L...	Assembly	CPU_C...	INST_RE...	CPU_C...
0xB3966	75	mov QWORD PTR [rsp+_M_insert_aux+038h],rdx	3,980	7,837	2,117
0xB396B	77	mov DWORD PTR [rsp+_M_insert_aux+044h],eax	1		1
0xB396F	84	mov rax,QWORD PTR [rsi+_M_insert_aux+038h]	3		3
0xB3973	87	mov edi,DWORD PTR [rsp+_M_insert_aux+044h]	1		1
0xB3977	84	mov QWORD PTR [rbp+_M_insert_aux+038h],rax	385	567	208
0xB397B	85	mov rax,QWORD PTR [rsi+_M_insert_aux+038h]	173	65	90
0xB397F	87	test edi,edi	4		2
0xB3981	85	mov QWORD PTR [rcx+_M_insert_aux+038h],rax	350	65	177
0xB3985	79	mulsd xmm1,MMWORD PTR [rip+_ZN13G4PolyPhiFaceC2EPK...	358	570	196
0xB398D	79	movsd MMWORD PTR [rsp+_M_insert_aux+048h],xmm0	48	10	28
0xB3993	87	jle Block 3	82	139	48
Block 1	**87**		**300**	**557**	**159**
0xB3995	87	mov eax,DWORD PTR [rsp+_M_insert_aux+044h]			
0xB3999	87	xor edx,edx	223	397	118
0xB399B	87	sub eax,01h	9	4	5
0xB399E	87	lea rax,QWORD PTR [rax*8+_M_insert_aux+08h]	68	156	36
0xB39AA	87	test BYTE PTR [rax + _M_insert_aux],al			
0xB39AC	87	add BYTE PTR [rax + _M_insert_aux],al			
0xB39AE	87	add BYTE PTR [rax + _M_insert_aux],al			
Block 2	**89**		**4,367**	**6,690**	**2,360**
0xB39B0	89	mov rcx,QWORD PTR [rsp+_M_insert_aux+038h]	1,696	3,073	901
0xB39B5	89	movapd xmm0,xmm1	13	6	7
0xB39B9	89	mulsd xmm0,MMWORD PTR [rcx+rdx + _M_insert_aux]	64	139	35
0xB39BE	89	addsd xmm0,MMWORD PTR [r14+rdx + _M_insert_aux]	104	94	55
0xB39C4	89	movsd MMWORD PTR [rbp+rdx + _M_insert_aux],xmm0	2,003	2,895	1,102
0xB39CA	89	add rdx,08h	418	297	222
0xB39CE	87	cmp rdx,rax	56	184	29
0xB39D1	87	jnz Block 2	13	2	9
Block 3	**91**		**2,054**	**2,544**	**1,152**
0xB39D3	91	mov rax,QWORD PTR [rbp + _M_insert_aux]	180	243	111
0xB39D7	91	mov rbx,QWORD PTR [r13+_M_insert_aux+08h]	169	352	95
0xB39DB	91	lea rcx,QWORD PTR [rsp+_M_insert_aux+0d0h]	2		2
0xB39E3	91	lea rdx,QWORD PTR [rsp+_M_insert_aux+050h]			
0xB39E8	91	movsd MMWORD PTR [rsp+_M_insert_aux+010h],xmm1	207	329	118
0xB39EE	91	mov r12,QWORD PTR [r13+_M_insert_aux+088h]	100	190	52
0xB39F5	91	mov rsi,rcx	5	4	3
0xB39F8	91	mov QWORD PTR [rsp+_M_insert_aux+020h],rcx			
0xB39FD	91	mov QWORD PTR [rsp+_M_insert_aux+0d0h],rax	220	68	126
0xB3A05	91	mov rax,QWORD PTR [rbp+_M_insert_aux+08h]	203	131	106
0xB3A09	91	mov rdi,QWORD PTR [rbx+_M_insert_aux+08h]	96	152	53
0xB3A0D	91	mov QWORD PTR [rsp+_M_insert_aux+028h],rdx	30	19	21
0xB3A12	91	mov QWORD PTR [rsp+_M_insert_aux+0d8h],rax	174	33	97
0xB3A1A	91	mov rax,QWORD PTR [rbp+_M_insert_aux+010h]	91	139	51
0xB3A1E	91	mov QWORD PTR [rsp+_M_insert_aux+0e0h],rax	83	148	48
0xB3A26	91	mov rax,QWORD PTR [rbp+_M_insert_aux+038h]	54	162	29
0xB3A2A	91	mov QWORD PTR [rsp+_M_insert_aux+0e8h],rax	278	241	151
0xB3A32	91	mov rax,QWORD PTR [rdi + _M_insert_aux]	99	227	53
0xB3A35	91	call QWORD PTR [rax + _M_insert_aux]	63	106	36
Block 4	**91**		**765**	**632**	**407**
		Total Selected (6 instructions):	**4,298**	**6,504**	**2,322**

Useful even just for sampling mode, in particular for “large functions” common in HEP code. People's intuition not always correct.

Source code view with PTU

Line	Source	CPU_CLK_UNHALTED.CORE	INST_RETIRED.ANY
1634	SteppingHelixPropagator::Result	0	0
1635	SteppingHelixPropagator::refToDest(SteppingHelixPropagator::DestType dest,	0	0
1636	const SteppingHelixPropagator::StateInfo& sv,	0	0
1637	const double pars[6],	0	0
1638	double& dist, double& tanDist,	0	0
1639	PropagationDirection& refDirection,	0	0
1640	double fastSkipDist) const{	3257	4415
1641	static const std::string metname = "SteppingHelixPropagator";	215	388
1642	Result result = SteppingHelixStateInfo::NOT_IMPLEMENTED;	0	0
1643	double curZ = sv.r3.z();	83	102
1644	double curR = sv.r3.perp();	3838	6134
1645		0	0
1646	switch (dest){	7871	12222
1647	case RADIUS_DT:	0	0
1648	{	0	0
1649	double cosDPhiPR = cos((sv.r3.deltaPhi(sv.p3)));	333	195
1650	dist = pars[RADIUS_P] - curR;	495	194
1651	refDirection = dist*cosDPhiPR > 0 ?	0	0
1652	alongMomentum : oppositeToMomentum;	265	326
1653	if (fabs(dist) > fastSkipDist){	400	264
1654	result = SteppingHelixStateInfo::INACC;	0	0
1655	break;	0	0
1656	}	0	0
1657	tanDist = dist/(sv.p3.perp()+1e-8)*sv.p3.mag();	1650	1787

CPU performance (examples)

- Investigations of the use of transcendental math, for example:

  atan2 calls from Hep3Vector::phi()

  found many of these to be repeated calculations of the same thing (both in our own code and in G4), also perp()/mag()

- Also encouraged some investigations into alternate math libraries
- Intel PTU source code view helps, especially for longer functions, easier to see inlines, etc.
- Identified places to lift calculations out of loops, remove divisions, etc.
- Shining a light on various technical aspects of our software also triggered useful discussions about appropriateness of algorithms (e.g. full stepping track propagator vs simple analytical extrapolation, etc.)

Module specific pfmon view

Click for details...

MODULE NAME	Total Cycles	Cycles Stalled	% of Cycles Stalled	CPI Ratio	L2 Miss Impact	% of Total Stalls
CkfTrackCandidateMaker_newTrackCandidateMaker	1405883341	684506320	48.7%	1.25	22756000	3.7%
ConversionTrackCandidateProducer_conversionTrackCandidates	1216890912	673825912	55.4%	1.11	6577600	0.8%
GsfTrackProducer_pixelMatchGsfFit	1148110826	260087145	22.7%	0.91	778800	0.2%
GsfTrackProducer_gsfPFtracks	618961542	127697046	20.6%	0.86	435200	0.2%
CaloMuonProducer_calomuons	552939456	189712951	34.3%	1.15	7865000	6.0%
CkfTrackCandidateMaker_thTrackCandidates	510372704	220202685	43.1%	1.39	10047600	5.8%
TrackProducer_preFilterFirstStepTracks	424368027	172101497	40.6%	1.28	6569200	6.0%
MuonIdProducer_muons	423798580	140229777	33.1%	1.22	5562800	5.6%
PFRecoTauProducer_pfRecoTauProducerHighEfficiency	417091298	300934845	72.2%	2.80	244313400	77.8%
SoftElectronProducer_btagSoftElectrons	373227607	122767413	32.9%	1.15	4807600	5.1%
PoolOutputModule_RECO	367355743	71736042	19.5%	0.79	8666200	8.4%

Tool developed to allow selective counting using pfmon, e.g. by framework module, here looking for stalls. To include in next passes of our Performance Survey

From Daniele Kruse

“Multicore Crisis”

- In practice the “multicore crisis” will manifest itself for CMS, if it does, as the softening or removal of some of the “Moore's Law” cost considerations in spreadsheets presented to the CERN Resources Review Board (RRB) a few years down the line.
- And of course in our inability to do some of the things we want to do with the resources we have in hand.
- It could affect not only CPU and memory need estimates, but also storage, due to I/O considerations.
- The software we have today has to evolve at some level.

CMS Multicore Strategy

- Phase 0 – independent jobs on each core (+ hyperthreading?)
- Phase 1 – development and deployment of CMS framework and WM system to fork sub-processes after loading bulk conditions

This year

Advantages: reduce memory needs scaling with #cores and limited changes to software

First steps with sites, grid providers, etc. to multicore

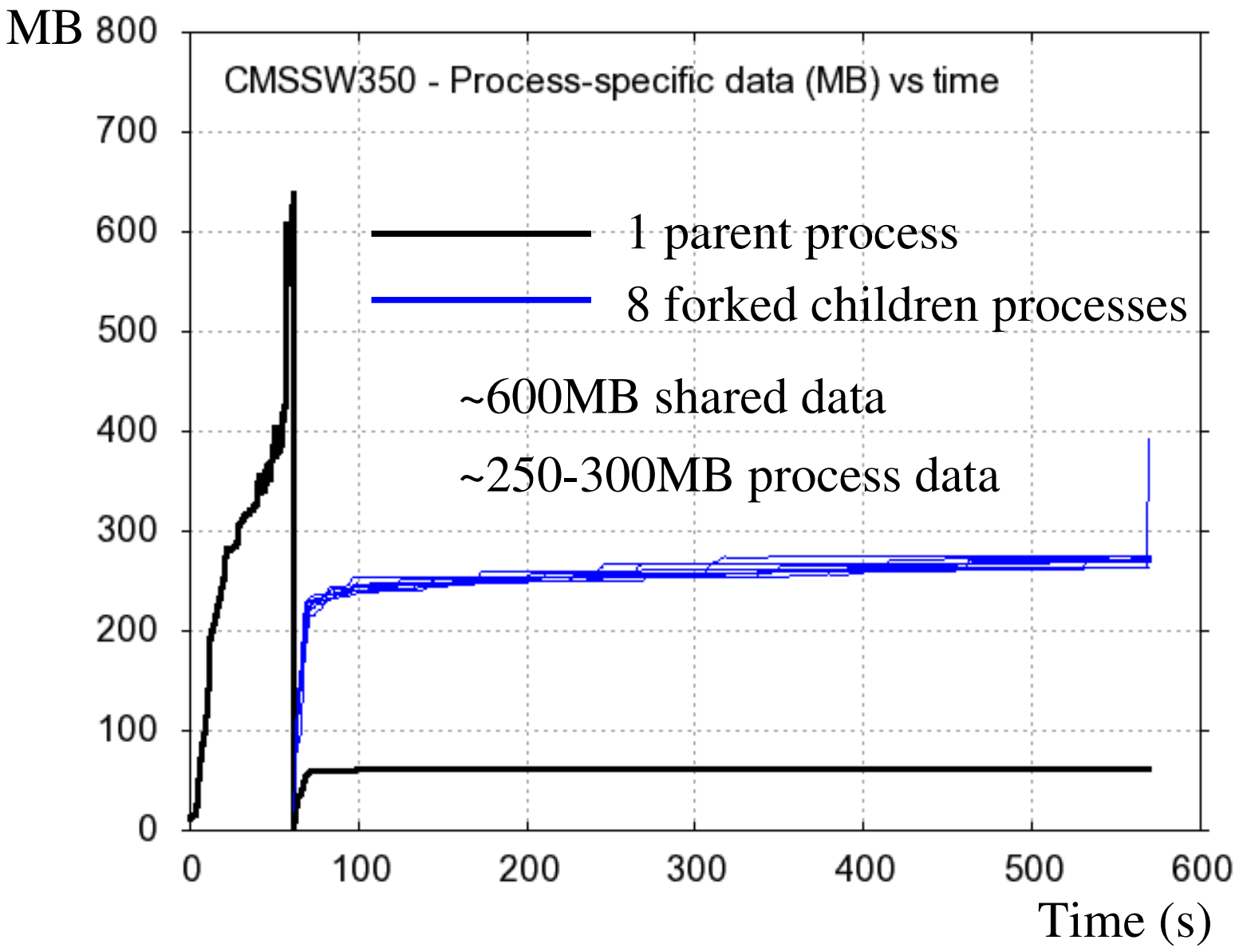
- Phase 2 – deployment of more fine grained parallelism

Next Year And beyond?

More difficult, requires greater changes to our software

Impacts software development model, may require more sophisticated software development in some cases

Process forking



CMS Framework features added to fork N child processes after prefetching the bulk of common conditions, etc.

(Catching up with others!)

Outputs simply fast merged on WN

Multicore deployment?

- How do we deploy applications which can use multiple cores?
- Two critical ingredients:

  Need to submit jobs to Grid sites, LSF at CERN, etc. that which can be assigned N cores

  Need discussion and agreement about proper accounting for CPU and memory from multicore jobs

- Sites then need to develop confidence in the accounting metrics to believe that we are indeed (for example) using less memory, using the N cores efficiently, etc.
- I hope we will achieve some of this during 2010

Summary

- CMS continues to work to maximize our software performance. Focusing primarily on capitalizing on top of 64bit builds at this point (including detailed comparison between 32bit and 64bit)
- We have recently begun a systematic "Performance Survey":

  Analyse in detail with multiple tools, discuss, document and fix. Look for opportunities for parallelism. gcc4.x and x86_64 are enabling!

  Look at CPU performance and all aspects of memory use.

  First pass (of course) already revealing many interesting things
- Multicore next – scalable solution much more difficult

  Basic framework features to fork/share-memory were added

  Near term enabling step is proper (grid) accounting/submission