Modular Software Performance Monitoring

Daniele Francesco Kruse – CERN – PH / SFT

Karol Kruzelecki – CERN – PH / LBC

Summary

1. Motivation and Goal

2. Introduction
   - Performance Monitoring
   - Performance Counters

3. Challenge
   - *Monolithic* vs. *Modular* Monitoring
   - Modularity in *CMSSW*, *Gaudi* and *Geant4*

4. Solution
   - Analysis Overview
   - What do we monitor?

5. Simple example of successful usage

6. Conclusions

Motivation and Goal

- **HEP** software is huge and complex and is developed by a multitude of programmers often unaware of performance issues
- The software produced is suboptimal in terms of efficiency and speed
- Unfortunately CPU speed is not likely to be increased in the near future as we were used to in the past
- The ***goal*** is then to find an effective method to improve SW through **monitoring and optimization**
- Better *performance* (more *throughput*) means savings both in hardware and power needed

Performance Monitoring

DEF : The action of collecting information related to how an application or system performs

HOW : Obtaining micro-architectural level information from hardware performance counters

WHY : To identify bottlenecks, and possibly remove them in order to improve application performance

Performance Counters

- All recent processor architectures include a processor–specific **PMU**

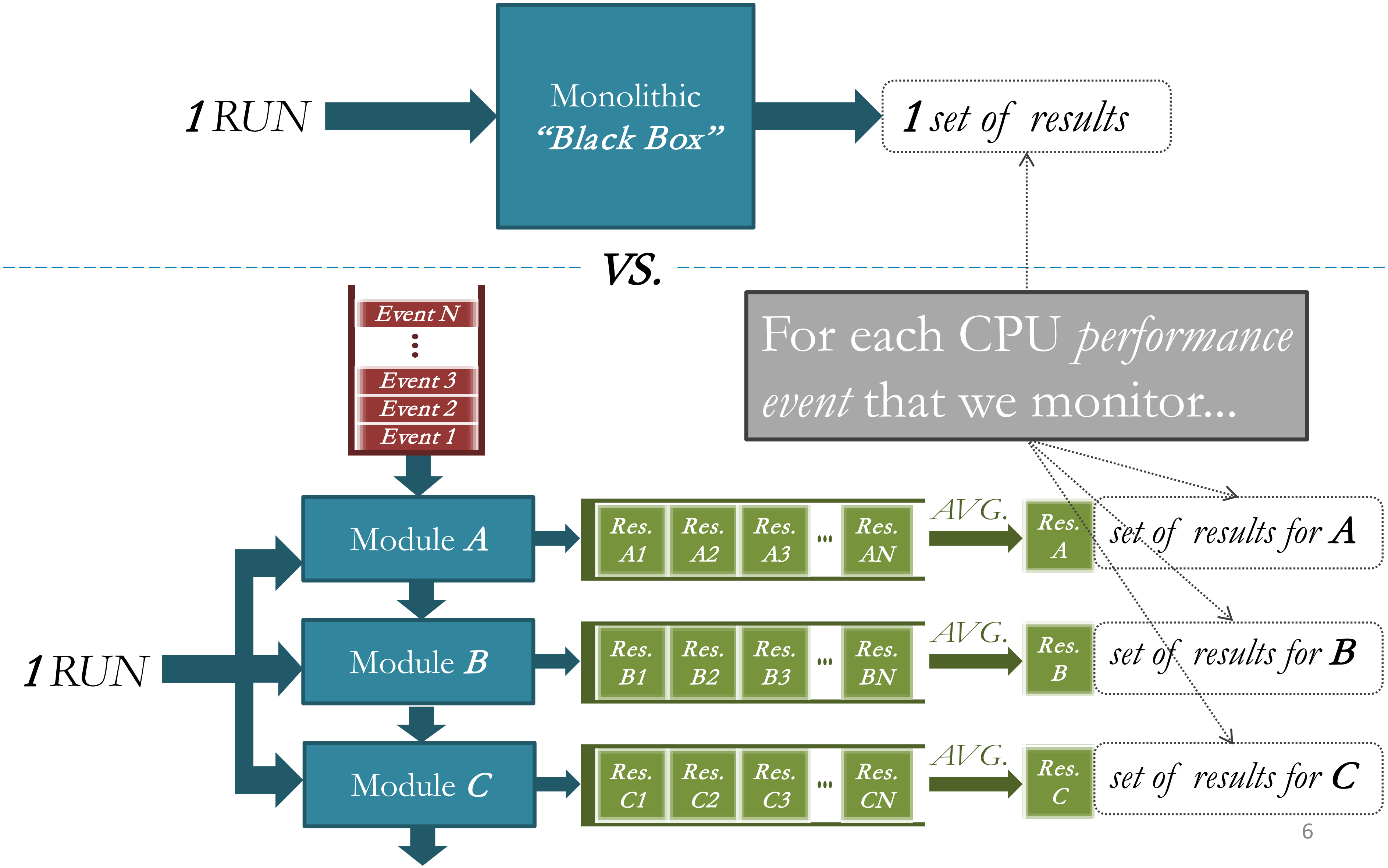
- The **P**erformance **M**onitoring **U**nit contains several <u>performance counters</u>

- Performance counters are able to count micro-architectural events from many hardware sources (cpu pipeline, caches, bus, etc…)

- We focus on the two main **Intel®** cpu families currently on the market: ***Core*** and ***Nehalem***

- *Nehalem* processors feature 4 programmable counters while *Core* processors have 2 programmable counters

Monolithic vs. *Modular* Monitoring



Monolithic vs. *Modular* Monitoring

- When we face large and complex software *monolithic* analysis becomes less useful
- "Traditional" monitoring tools (using performance counters) are monolithic. Examples: **PTU** and **pfmon**
- Even *sampling* over symbols (functions) is not enough for code division. Solution: ***modular monitoring!***
- Code instrumentation (minimal in HEP software) and ad-hoc interface to the monitoring tool needed
- *Advantage*: narrowing down the possible location of performance problems leads to ***easier optimization***

Modularity in HEP SW

- ***<u>CMSSW</u>*** code is organized into ***modules*** that are sequencially executed during each event processed, and it provides <u>hooks</u> to execute user defined actions at the beginning and at the end of modules

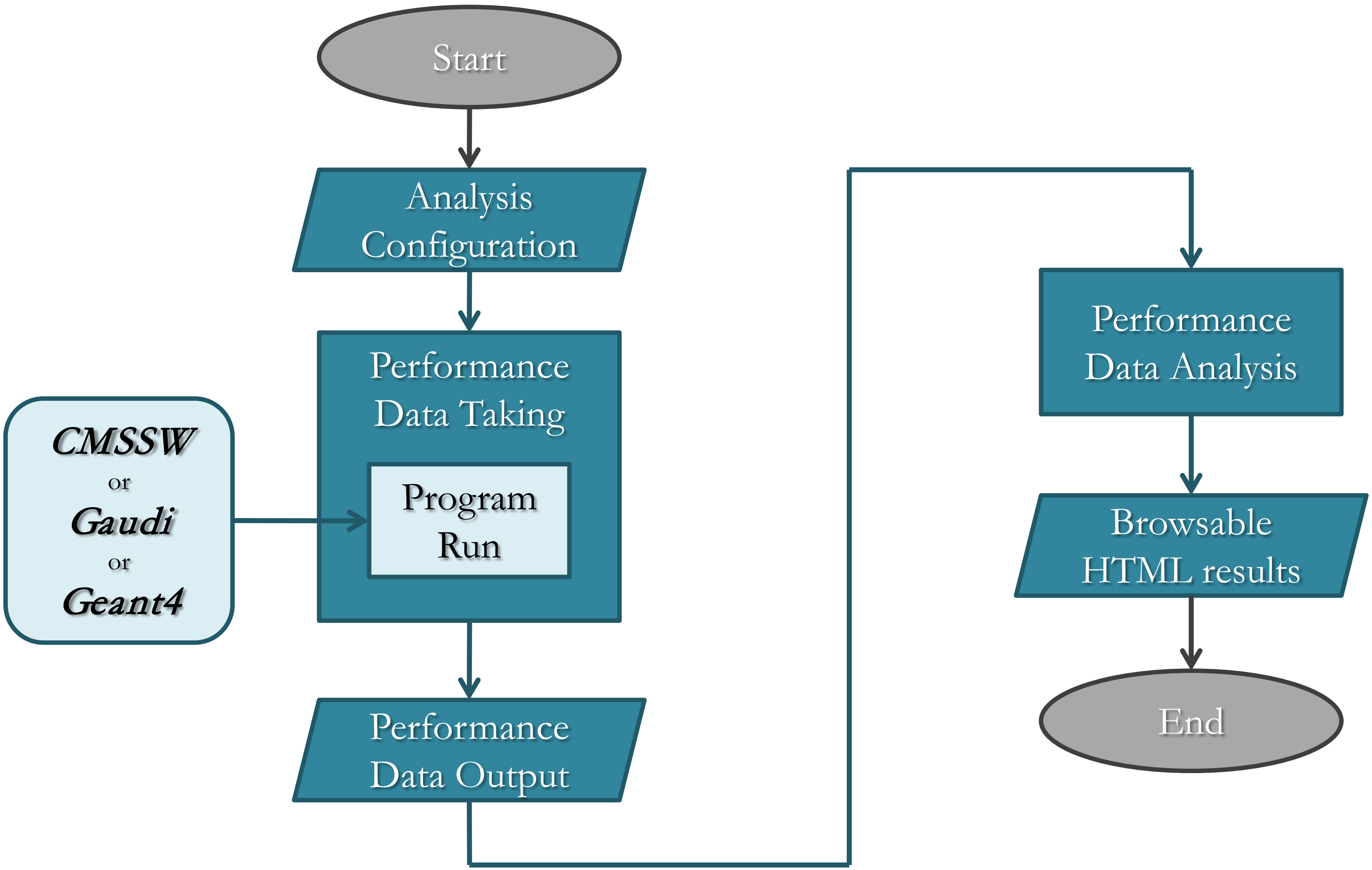
- Hooks is what we use to *start* and *stop* the **monitoring process** and to collect results for each module

- More on *CMSSW* performance in *Matti Kortelainen*'s talk

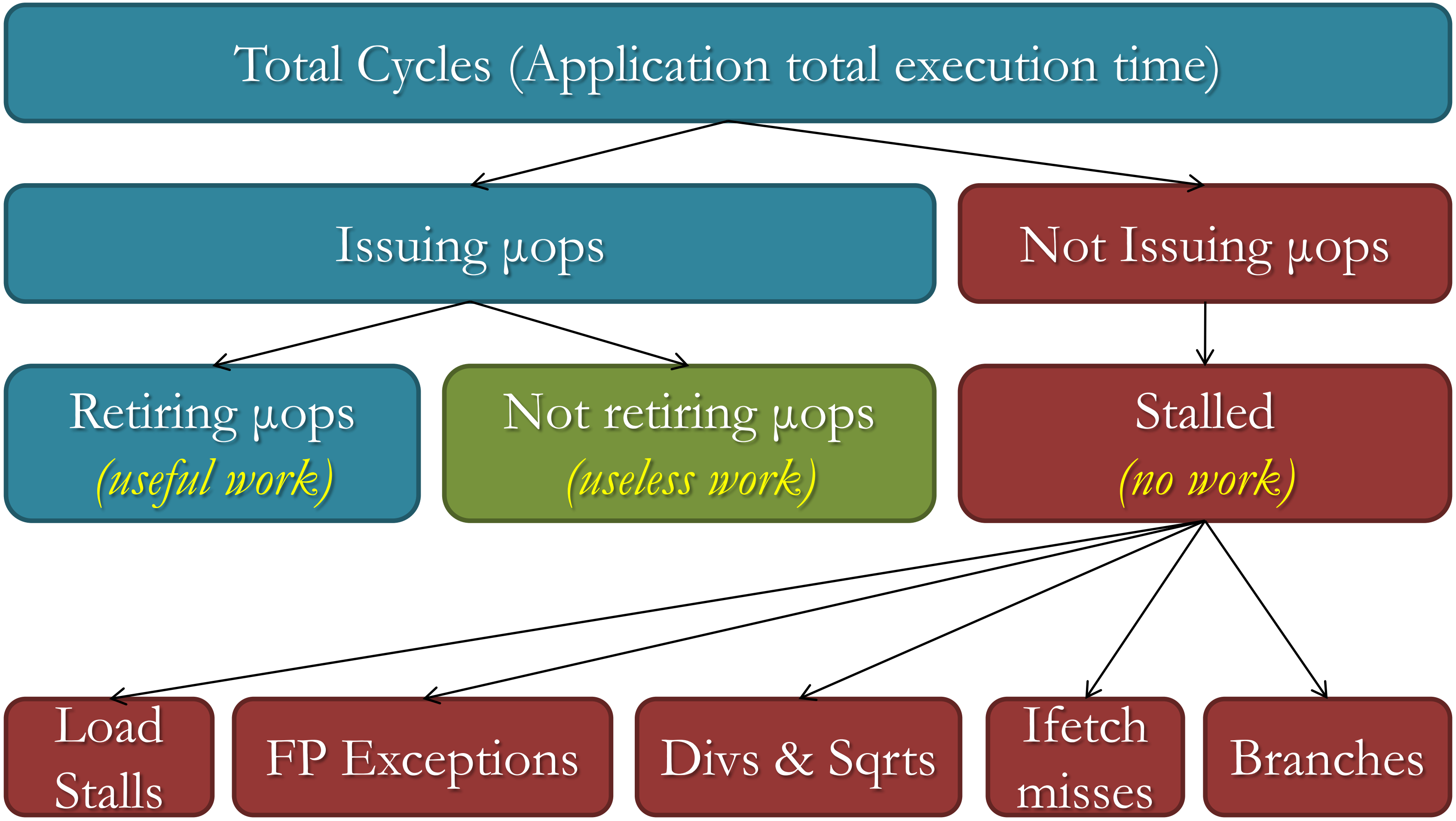
- ***<u>Gaudi</u>*** provides a similar mechanism to instrument its code (modules are called ***algorithms***)

- ***<u>Geant4</u>*** is handled differently: binning into triples ***<particle type, energy range, physical volume>***

Analysis flow-graph



What and why do we monitor?



Example of usage – LHCb (Gaudi)

- Tested on *Brunel v37r7*

  `GaudiRun <options>`

  `GaudiProfiler <options>`

No code instrumentation needed

- **GaudiProfiler** – python script handling sequential run of application for all the necessary counters and postprocessing

MODULE NAME	Total Cycles	Instructions Retired ▲	CPI	iMargin	iFactor
FitBest	88172186.85	113337618.59	0.78	11.94	2.28
CreateOfflinePhotons	62577023.13	53160705.52	1.18	9.84	1.83
FitSeedForMatch	38233332.96	52203155.51	0.73	5.03	0.74
MuonIDAlg	36209823.66	36911424.03	0.98	5.39	0.63
RichOfflineGPIDLLIt0	29723192.21	32871302.13	0.90	4.29	0.98
CreateOfflineTracks	19249568.96	20883977.44	0.92	2.80	0.35
FitVelo	17809787.32	19615917.09	0.91	2.58	0.31
RichOfflineGPIDLLIt1	17759552.39	18178152.35	0.98	2.64	0.88
TsaSeed	15595472.48	16022706.13	0.97	2.31	0.33
PatVeloTT	11575267.16	13407311.72	0.86	1.64	0.22

Choice of an *alghorithm*

- **<u>Objective:</u>** to reduce the number of *Total cycles* (execution time) of one *algorithm*
- As a simple example we choose to focus on reducing the cycles in which instructions were retired
- **<u>How:</u>** This is done by reducing the number of *Instructions Retired* – a very stable and reliable counter

MODULE NAME	Total Cycles ▲	Instructions Retired	CPI	iMargin	iFactor
FitBest	88172186.85	1133[illegible]618.59	0.78	11.94	2.28
CreateOfflinePhotons	62577023.13	53160705.52	1.18	9.84	1.83
FitSeedForMatch	38233332.96	52203155.51	0.73	5.03	0.74
MuonIDAlg	36209823.66	36911424.03	0.98	5.39	0.63

Symbols – Looking deeper in

- After choosing *Algorithm*, in the *detailed symbol view* → .cpp file and function
- *Inlined* functions are not shown, they are counted in the "parent" functions

INST_RETIRED:ANY_P -- Total Samples: 51437 -- Sampling Period: 100000			
Samples	**Percentage**	**Symbol Name**	**Library Name**
6911	13.435854%	solveQuarticEq	libRichRecPhotonTools.so
2042	3.969905%	reconstructPhoton	libRichRecBase.so
1977	3.843537%	photonPossible	libRichRecPhotonTools.so
1867	3.629683%	reconstructPhoton	libRichRecPhotonTools.so

Detailed profiling of one function

```
#include "GaudiProfiling/PfmCodeAnalyser.h"
```

```
PfmCodeAnalyser::Instance("INSTRUCTIONS_RETIRED").start();
```

- We modify the body of function adding *start()* and *stop()* commands for profiler
- Results is shown after the run of application is over

```
Event: INSTRUCTIONS_RETIRED
Total count:1697360246
Number of counts:1548592
Average count:1096.066779
Overhead removed:42
```

```
PfmCodeAnalyser::Instance().stop();
```

```
//============================================================================
// Setup and solve quartic equation in the form
// x^4 + a x^3 + b x^2 + c x + d = 0
//============================================================================
bool PhotonRecoUsingQuarticSoln::
solveQuarticEq ( const Gaudi::XYZPoint& emissionPoint,
                 const Gaudi::XYZPoint& CoC,
                 const Gaudi::XYZPoint& virtDetPoint,
                 const double radius,
                 Gaudi::XYZPoint& sphReflPoint ) const
{
  // vector from mirror centre of curvature to assumed emission point
  Gaudi::XYZVector evec = emissionPoint - CoC;
  const double e2 = evec.Mag2();
  if ( e2 < 1e-99 ) { return false; }

  // vector from mirror centre of curvature to virtual detection point
  const Gaudi::XYZVector dvec = virtDetPoint - CoC;
  const double d2 = dvec.Mag2();
  if ( d2 < 1e-99 ) { return false; }

  // various quantities needed to create quartic equation
  // see LHCB/98-040 section 3, equation 3
  const double e        = std::sqrt(e2);
  const double d        = std::sqrt(d2);
  const double cosgamma = evec.Dot(dvec) / (e*d);
  const double singamma = std::sqrt( 1.0 - cosgamma*cosgamma );
  const double dx       = d * cosgamma;
  const double dy       = d * singamma;
  const double r2       = radius * radius;

  // Fill array for quartic equation
  const double a0 =   4 * e2 * d2;
  const double a1 = - ( 4 * e2 * dy * radius ) / a0;
  const double a2 =  ( (dy * dy * r2) + ((e+dx) * (e+dx) * r2) - a0 ) / a0;
  const double a3 =  ( 2 * e * dy * (e-dx) * radius ) / a0;
  const double a4 =  ( ( e2 - r2 ) * dy * dy ) / a0;

  // use simplified RICH version of quartic solver
  const double sinbeta = solve_quartic_RICH( a1, // a
                                             a2, // b
                                             a3, // c
                                             a4  // d
                                             );
  // normal vector to reflection plane
  const Gaudi::XYZVector nvec2 = evec.Cross(dvec);

  // Set vector mag to radius
  evec *= radius/e;

  // create rotation
  const Gaudi::Rotation3D rotn( Gaudi::AxisAngle(nvec2,asin(sinbeta)) );

  // rotate vector and update reflection point
  sphReflPoint = CoC + rotn*evec;

  return true;
}
```

Improving small parts of code

```
//=============================================================================
// Setup and solve quartic equation in the form
// x^4 + a x^3 + b x^2 + c x + d = 0
//=============================================================================
bool PhotonRecoUsingQuarticSoln::
solveQuarticEq ( const Gaudi::XYZPoint& emissionPoint,
                 const Gaudi::XYZPoint& CoC,
                 const Gaudi::XYZPoint& virtDetPoint,
                 const double radius,
                 Gaudi::XYZPoint& sphReflPoint ) const
{
  // vector from mirror centre of curvature to assumed emission point
  Gaudi::XYZVector evec = emissionPoint - CoC;
  const double e2 = evec.Mag2();
  if ( e2 < 1e-99 ) { return false; }

  // vector from mirror centre of curvature to virtual detection point
  const Gaudi::XYZVector dvec = virtDetPoint - CoC;
  const double d2 = dvec.Mag2();
  if ( d2 < 1e-99 ) { return false; }

  // various quantities needed to create quartic equation
  // see LHCB/98-040 section 3, equation 3
  const double e        = std::sqrt(e2);
  const double d        = std::sqrt(d2);
  const double cosgamma = evec.Dot(dvec) / (e*d);
  const double singamma = std::sqrt( 1.0 - cosgamma*cosgamma );
  const double dx       = d * cosgamma;
  const double dy       = d * singamma;
  const double r2       = radius * radius;

  // Fill array for quartic equation
  const double a0 =    4 * e2 * d2;
  const double a1 = - ( 4 * e2 * dy * radius ) / a0;
  const double a2 =  ( (dy * dy * r2) + ((e+dx) * (e+dx) * r2) - a0 ) / a0;
  const double a3 =  ( 2 * e * dy * (e-dx) * radius ) / a0;
  const double a4 =  ( ( e2 - r2 ) * dy * dy ) / a0;

  // use simplified RICH version of quartic solver
  const double sinbeta = solve_quartic_RICH( a1, // a
                                             a2, // b
                                             a3, // c
                                             a4  // d
                                             );

  // normal vector to reflection plane
  const Gaudi::XYZVector nvec2 = evec.Cross(dvec);

  // Set vector mag to radius
  evec *= radius/e;

  // create rotation
  const Gaudi::Rotation3D rotn( Gaudi::AxisAngle(nvec2,asin(sinbeta)) );

  // rotate vector and update reflection point
  sphReflPoint = CoC + rotn*evec;

  return true;
}
```

- Optimization procedure loop:
  1. modify code
  2. compile
  3. profile
- Compare average count of *Instructions Retired*

```
//=============================================================================
// Setup and solve quartic equation in the form
// x^4 + a x^3 + b x^2 + c x + d = 0
//=============================================================================
bool
PhotonRecoUsingQuarticSoln::
solveQuarticEq ( const Gaudi::XYZPoint& emissionPoint,
                 const Gaudi::XYZPoint& CoC,
                 const Gaudi::XYZPoint& virtDetPoint,
                 const double radius,
                 Gaudi::XYZPoint& sphReflPoint ) const
{
  // vector from mirror centre of curvature to assumed emission point
  Gaudi::XYZVector evec = emissionPoint - CoC;
  const double e2 = evec.Mag2();
  if ( e2 < 1e-99 ) { return false; }

  // vector from mirror centre of curvature to virtual detection point
  Gaudi::XYZVector dvec = virtDetPoint - CoC;
  const double d2 = dvec.Mag2();
  if ( d2 < 1e-99 ) { return false; }

  // various quantities needed to create quartic equation
  // see LHCB/98-040 section 3, equation 3
  const double e        = std::sqrt(e2);
  const double d        = std::sqrt(d2);
  const double cosgamma = evec.Dot(dvec) / (e*d);
  const double dx       = d * cosgamma;
  // dy = d * singamma = d * sqrt( 1 -cosgamma^2 )
  const double dy       = d * std::sqrt( 1.0 - cosgamma*cosgamma);
  const double r2       = radius * radius;

  // Fill array for quartic equation
  const double a0 =   4 * e2 * d2;

  evec *= radius/e;

  sphReflPoint = CoC + ( Gaudi::AxisAngle( evec.Cross(dvec),
    asin(solve_quartic_RICH( - ( 4 * e2 * dy * radius ) / a0,
    ( (dy * dy * r2) + ((e+dx) * (e+dx) * r2) - a0 ) / a0,
    ( 2 * e * dy * (e-dx) * radius ) / a0,
    ( ( e2 - r2 ) * dy * dy ) / a0))))*evec;

  return true;
}
```

```
Event: INSTRUCTIONS_RETIRED
Total count:1697360246
Number of counts:1548592
Average count:1096.066779
Overhead removed:42
```

-6.5 %

```
Event: INSTRUCTIONS_RETIRED
Total count:1587412930
Number of counts:1548592
Average count:1025.068533
Overhead removed:42
```

Re-run after changes

MODULE NAME	Total Cycles ▲	Instructions Retired	CPI	iMargin	iFactor
FitBest	88172186.85	113337618.59	0.78	11.94	2.28
CreateOfflinePhotons	62577023.13	53160705.52	1.18	9.84	1.83
FitSeedForMatch	38233332.96	52203155.51	0.73	5.03	0.74
MuonIDAlg	36209823.66	36911424.03	0.98	5.39	0.63

- Even small changes are visible in *Instructions Retired*
- *Total Cycles* decreased – it is faster

MODULE NAME	Total Cycles ▲	Instructions Retired	CPI	iMargin	iFactor
FitBest	88148530.27	113326950.94	0.78	11.98	2.28
CreateOfflinePhotons	61094302.97	52116553.54	1.17	9.63	1.76
FitSeedForMatch	38168708.13	52208985.23	0.73	5.03	0.74
MuonIDAlg	36069940.44	36920672.53	0.98	5.38	0.62

- ~6.5% improvement in one function gave ~2% improvement in the *algorithm*

Conclusions

- We implemented a modular ad-hoc **performance counters-based monitoring tool** for three major HEP frameworks: *CMSSW*, *Gaudi* and *Geant4*
- This tool is supposed to help developers optimizing existing code to improve its performance without the need for code instrumentation
- The tool has been successfully used to optimize code in *Gaudi* and has shown the potential to be used for other applications as well
- *GaudiProfiling* package will be available in the next release of Gaudi

Thank you, Questions ?

backup slides

BACKUP: The 4-way Performance Monitoring

	Overall (pfmon)	*Modular*
Counting	1. Overall Analysis	3. Module Level Analysis
Sampling	2. Symbol Level Analysis	4. Modular Symbol Level Analysis

BACKUP: *Core* and *Nehalem* PMUs - Overview

Intel *Core* Microarchitecture PMU

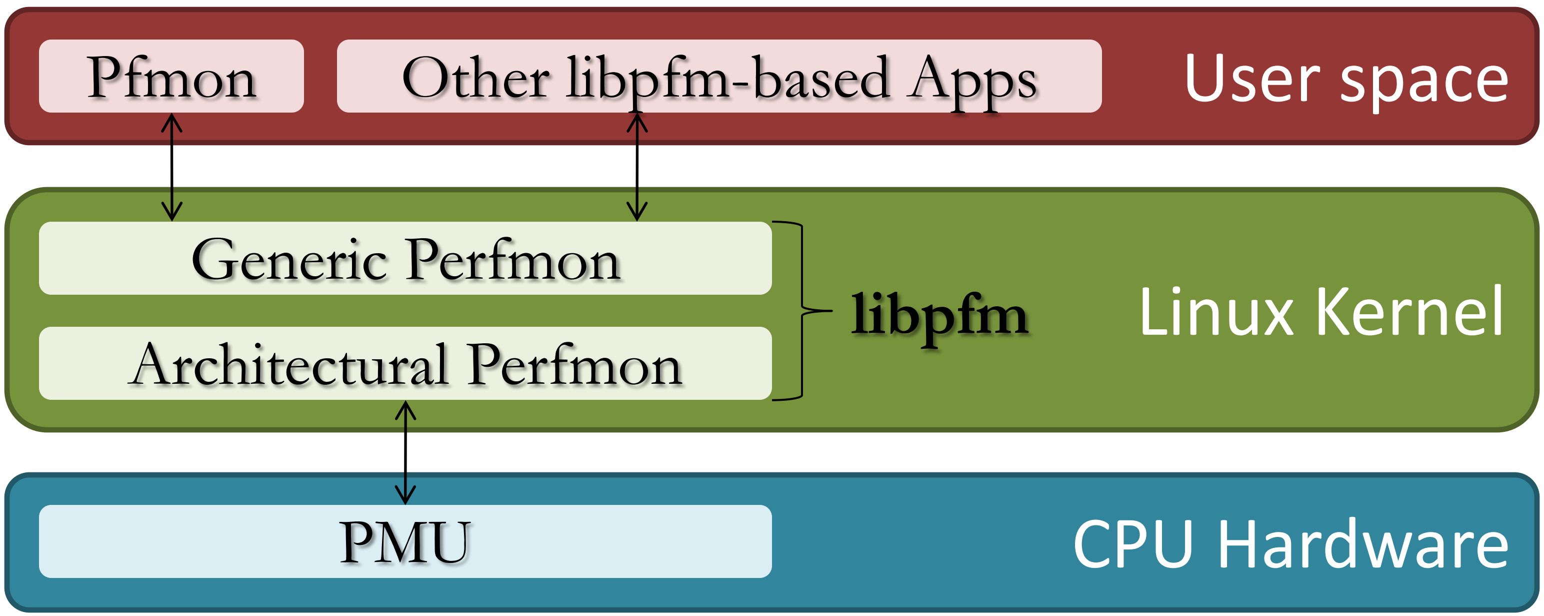
- 3 fixed counters
  (INSTRUCTIONS_RETIRED, UNHALTED_CORE_CYCLES, UNHALTED_REFERENCE_CYCLES)
- 2 programmable counters

Intel *Nehalem* Microarchitecture PMU

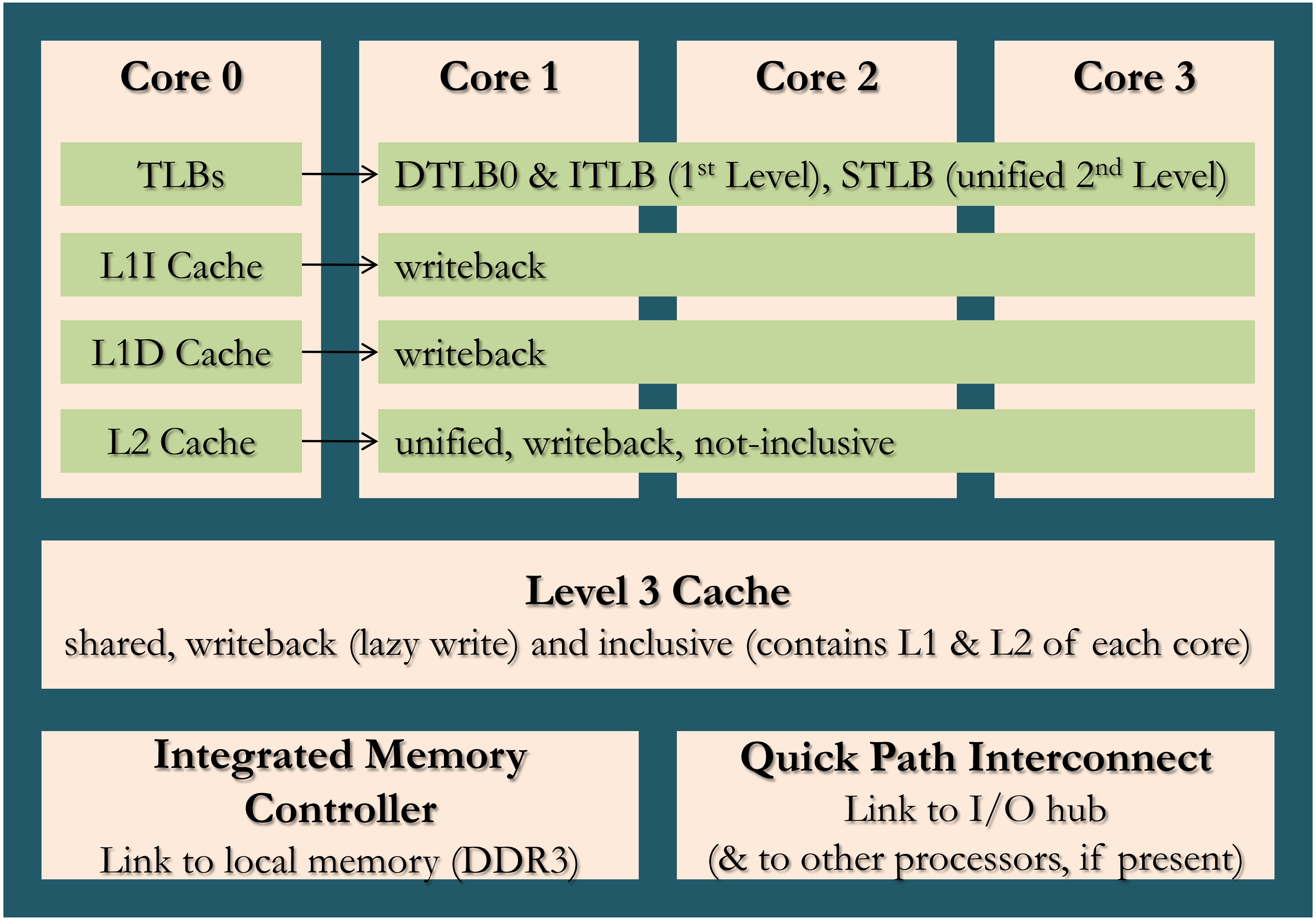
- 3 fixed *core*-counters
  (INSTRUCTIONS_RETIRED, UNHALTED_CORE_CYCLES, UNHALTED_REFERENCE_CYCLES)
- 4 programmable *core*-counters
- 1 fixed *uncore*-counter (UNCORE_CLOCK_CYCLES)
- 8 programmable *uncore*-counters

BACKUP: Perfmon2

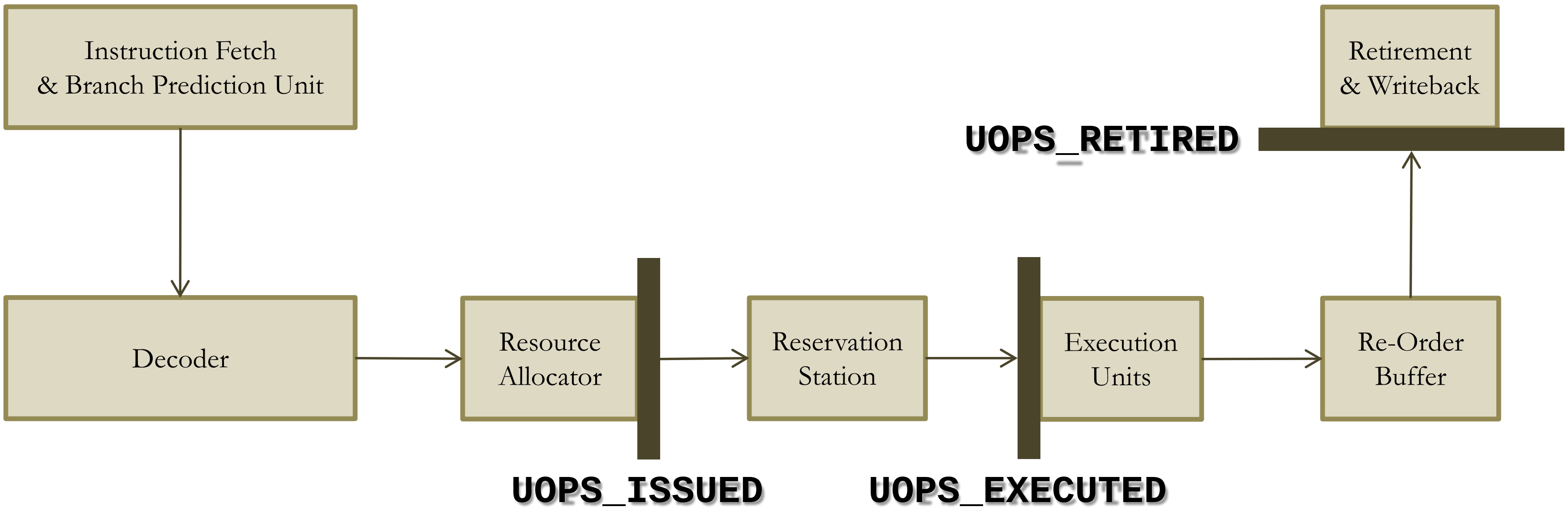
- A generic API to access the PMU (`libpfm`)
- Developed by Stéphane Eranian
- Portable across all new processor micro-architectures
- Supports system-wide and per-thread monitoring
- Supports counting and sampling



BACKUP: *Nehalem* : Overview of the architecture



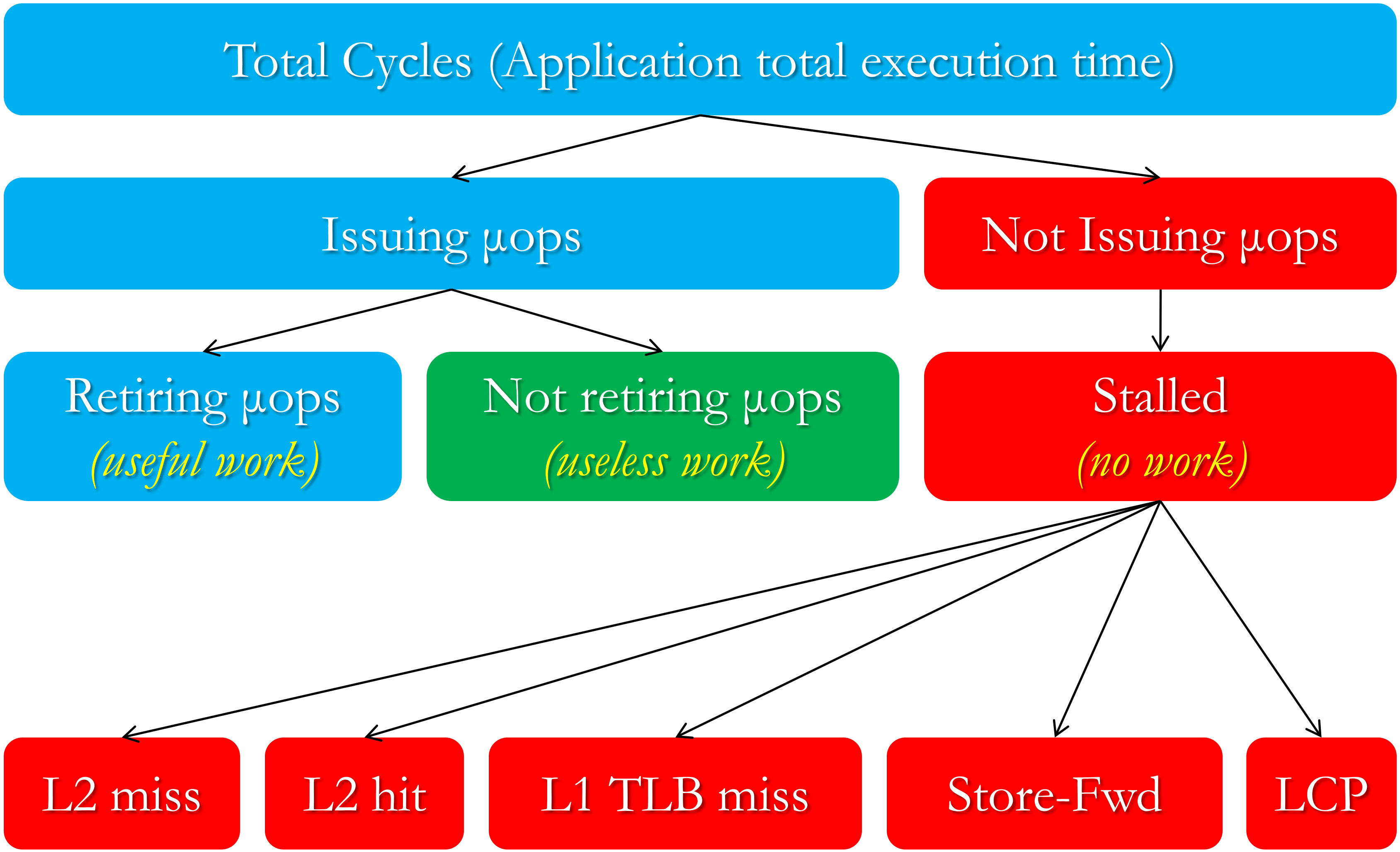
BACKUP: μops flow in *Nehalem* pipeline



- We are mainly interested in `UOPS_EXECUTED` (dispatched) and `UOPS_RETIRED` (the useful ones).

- Mispredicted `UOPS_ISSUED` may be eliminated before being executed.

BACKUP: Cycle Accounting Analysis



BACKUP: New analysis methodology for *Nehalem*

BASIC STATS: Total Cycles, Instructions Retired, CPI;

IMPROVEMENT OPPORTUNITY: iMargin, iFactor;

BASIC STALL STATS: Stalled Cycles, % of Total Cycles, Total Counted Stalled Cycles;

INSTRUCTION USEFUL INFO: Instruction Starvation, # of Instructions per Call;

FLOATING POINT EXCEPTIONS: % of Total Cycles spent handling FP exceptions;

LOAD OPS STALLS: L2 Hit, L3 Unshared Hit, L2 Other Core Hit, L2 Other Core Hit Modified, L3 Miss -> Local DRAM Hit, L3 Miss -> Remote DRAM Hit, L3 Miss -> Remote Cache Hit;

DTLB MISSES: L1 DTLB Miss Impact, L1 DTLB Miss % of Load Stalls;

DIVISION & SQUAREROOT STALLS: Cycles spent during DIV & SQRT Ops;

L2 IFETCH MISSES: Total L2 IFETCH misses, IFETCHes served by Local DRAM, IFETCHes served by L3 (Modified), IFETCHes served by L3 (Clean Snoop), IFETCHes served by Remote L2, IFETCHes served by Remote DRAM, IFETCHes served by L3 (No Snoop);

BRANCHES, CALLS & RETS: Total Branch Instructions Executed, % of Mispredicted Branches, Direct Near Calls, Indirect Near Calls, Indirect Near Non-Calls, All Near Calls, All Non Calls, All Returns, Conditionals;

ITLB MISSES: L1 ITLB Miss Impact, ITLB Miss Rate;

INSTRUCTION STATS: Branches, Loads, Stores, Other, Packed UOPS;

BACKUP: *PfmCodeAnalyser*, fast code monitoring

- Unreasonable (and useless) to run a complete analysis for every change in code
- Often interested in only small part of code and in one single event
- Solution: a fast, precise and light "singleton" class called *PfmCodeAnalyser*
- How to use it:

```
#include<PfmCodeAnalyser.h>

PfmCodeAnalyser::Instance("INSTRUCTIONS_RETIRED").start();

   //code to monitor

PfmCodeAnalyser::Instance().stop();
```

BACKUP: *PfmCodeAnalyser*, fast code monitoring

```
PfmCodeAnalyser::Instance("INSTRUCTIONS_RETIRED",  0, 0,
                          "UNHALTED_CORE_CYCLES",  0, 0,
                          "ARITH:CYCLES_DIV_BUSY", 0, 0,
                          "UOPS_RETIRED:ANY",      0, 0).start();
```

```
Event: INSTRUCTIONS_RETIRED
     Total count:105000018525
Number of counts:10
   Average count:10500001852.5

Event: UNHALTED_CORE_CYCLES
     Total count:56009070544
Number of counts:10
   Average count:5600907054.4

Event: ARITH:CYCLES_DIV_BUSY
     Total count:28000202972
Number of counts:10
   Average count:2800020297.2

Event: UOPS_RETIRED:ANY
     Total count:138003585913
Number of counts:10
   Average count:13800358591.3
```

BACKUP: *What can we do with counters?*

Question:

Is all this useful?

Answer:

We don't know, but we shall see

- Lack of papers and literature about the subject
- An empirical study is underway to find out:
  1. A relationship between counter results and coding practices
  2. A practical procedure to use counter results to optimize a program
- A procedure has already been developed and will be tested
- The trial study will be conducted on Gaudi together with *Karol Kruzelecki* (PH-LBC group)

BACKUP: The 3-step optimization procedure

- We start from counter results and choose one algorithm to work on using the *Improvement Margin* and the *iFactor*.

- We then apply the following procedure:

MyAlg : Total Cycles: 1000 , Total Instructions: 1000 → CPI: 1.00

1. Change to a more efficient algorithm and vectorize it

MyAlg : Total Cycles: 300 , Total Instructions: 250 → CPI: 1.20

2. Remove stall sources (L1 & L2 misses, store-fwd, etc..)

MyAlg : Total Cycles: 250 , Total Instructions: 250 → CPI: 1.00

3. Remove misprediction sources (branches, calls, etc..)

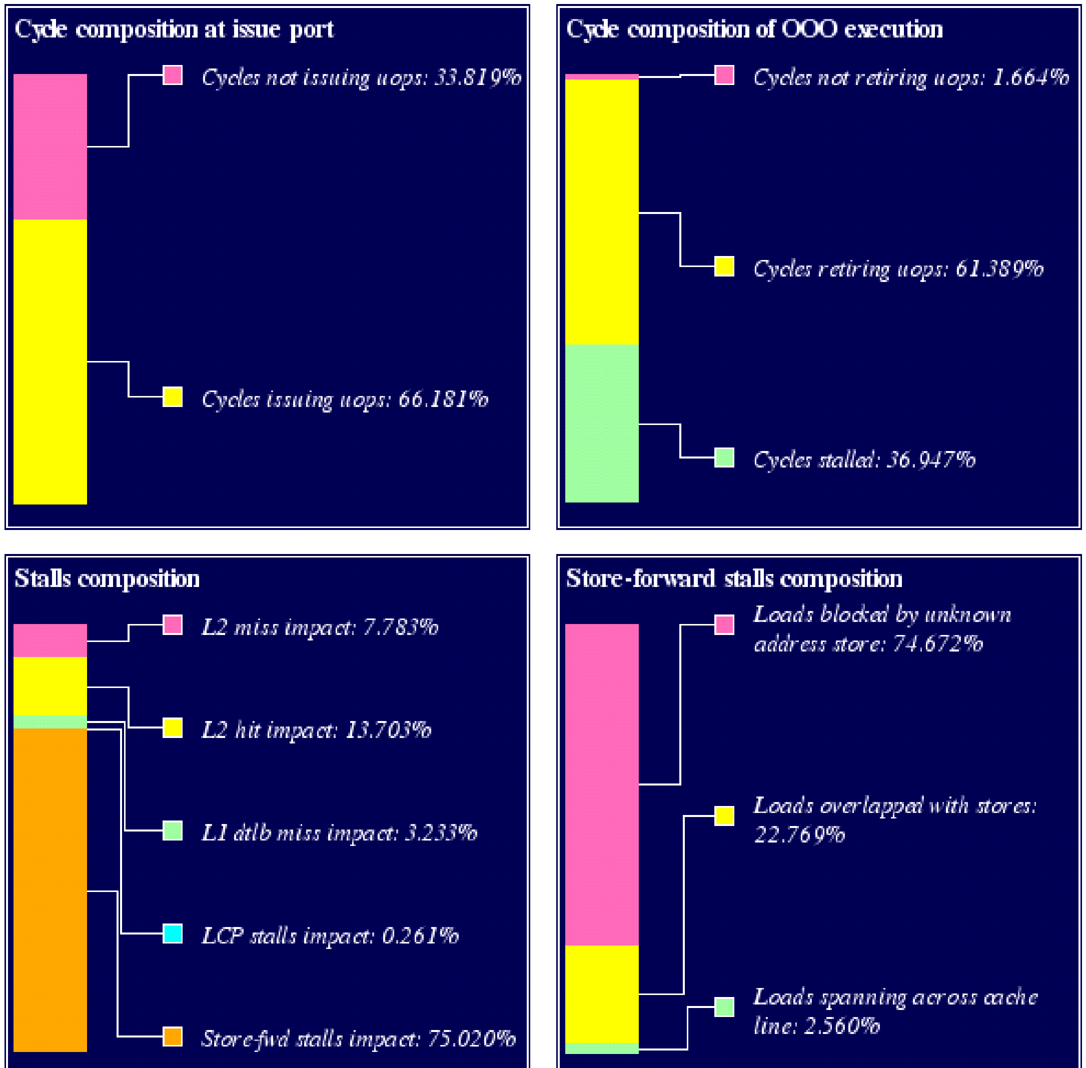
MyAlg : Total Cycles: 230 , Total Instructions: 250 → CPI: 0.92

<u>BACKUP:</u> Overall Analysis

- Uses Pfmon and it is based on the *Cycle Accounting Analysis*
- Good for showing **overall performance** and for checking improvements
- Good for identifying general software problems
- Good for comparing different versions of the code
- ***NOT*** enough for
  - finding inefficient parts of the software
  - finding bad programming practices

BACKUP: Overall Analysis

TOTAL CYCLES: 1408291621561



BACKUP: Symbol Level Analysis

- Uses *sampling* capabilities of pfmon
- Good for identifying general **bad programming practices**
- Can identify problems of functions which are frequently used
- Shows **functions** that use most of the execution cycles and functions that spend a lot of time doing nothing (stalling)
- ***NOT*** good for finding specific problems in the code

BACKUP: Symbol Level Analysis

Total Cycles

counts	%self	symbol
54894	3.79%	_int_malloc
50972	3.52%	__GI___libc_malloc
41321	2.85%	__cfree
36294	2.51%	ROOT::Math::SMatrix::operator=
31100	2.15%	__ieee754_exp
25636	1.77%	ROOT::Math::SMatrix::operator=
24833	1.72%	do_lookup_x
23206	1.60%	ROOT::Math::SMatrix::operator=
22970	1.59%	__ieee754_log
21741	1.50%	__atan2
20467	1.41%	ROOT::Math::SMatrix::operator=
19922	1.38%	_int_free
18354	1.27%	G__defined_typename
16026	1.11%	strcmp
15979	1.10%	TList::FindLink
14601	1.01%	G__defined_tagname

Stalled Cycles

counts	%self	symbol
24955	5.09%	_int_malloc
19797	4.04%	do_lookup_x
19084	3.89%	__GI___libc_malloc
14282	2.91%	__ieee754_exp
13564	2.77%	strcmp
13065	2.66%	__cfree
9927	2.02%	__atan2
8998	1.83%	__ieee754_log
7666	1.56%	TList::FindLink
7575	1.54%	_int_free
5392	1.10%	std::basic_string::find
4911	1.00%	computeFullJacobian
4410	0.90%	malloc_consolidate
4285	0.87%	operator new
4104	0.84%	ROOT::Math::SMatrix::operator=
3949	0.81%	33.84% makeAtomStep

DEMO

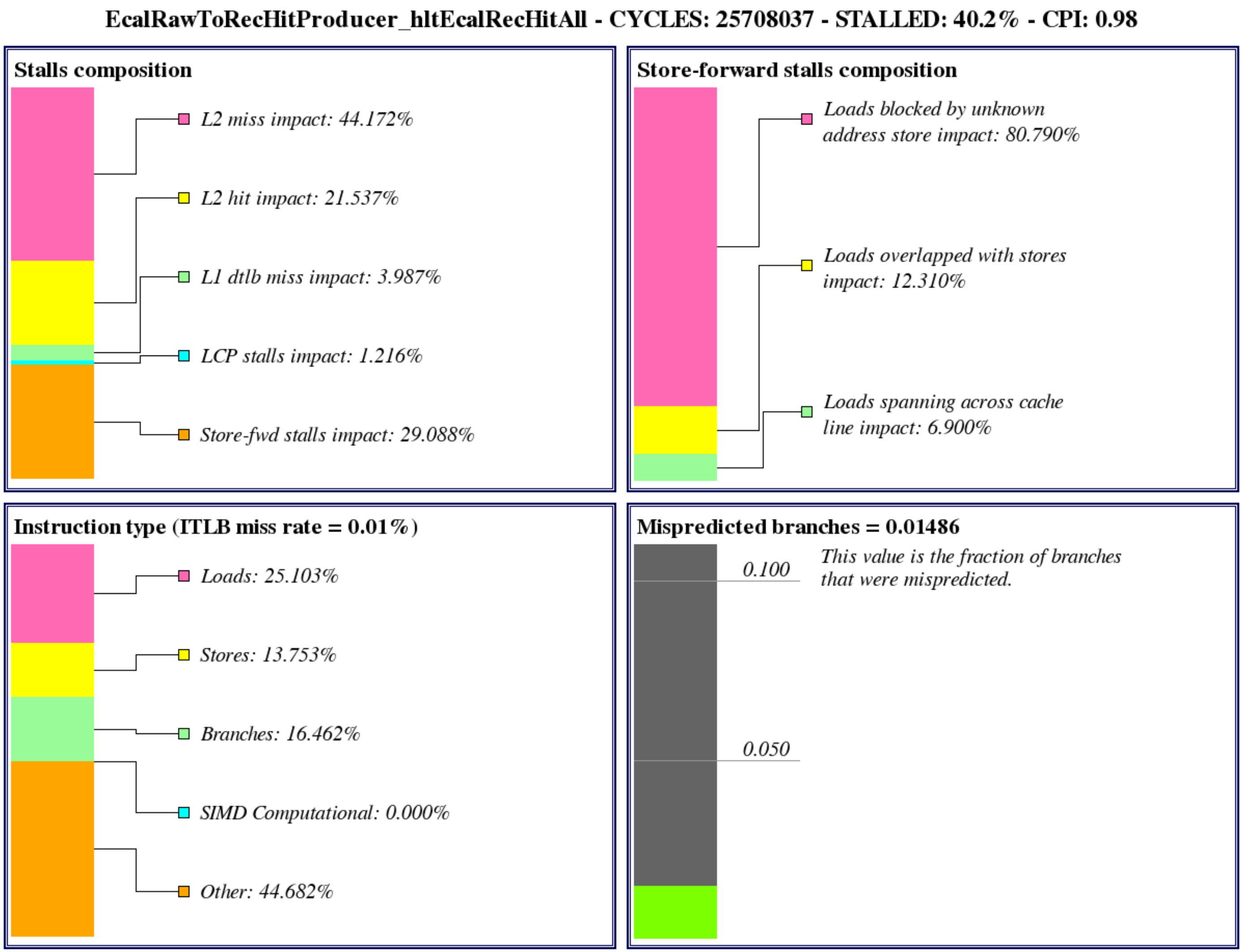
BACKUP: Module Level Analysis

- Uses the Perfmon2 interface (*libpfm*) directly
- Analyses each **CMSSW module** separately
- Allows the identification of *"troubled"* modules through a sortable HTML table
- Gives **instruction statistics** and produces **detailed graphs** to make analysis easier
- It requires 21 identical `cmsRun`'s (no multiple sets of events are used → more accurate results), but it can be parallelized so (*using 7 cores*): `time = ~3 runs`
- Code outside modules is not monitored (framework)

BACKUP: Module Level Analysis - Results Snapshot

Total Cycles	Cycles Stalled	% of Cycles Stalled	CPI Ratio	Improvement Margin	iFactor	L2 Miss Impact	% of Total Stalls	L2 Hit Impact	% of Total Stalls	L1 DTLB Miss Impact	% of Total Stalls
205757078	73002049	35.5%	1.13	12.3562%	2.2116	860596	2.5%	13252348	38.1%	1088361	3.1%
107164496	41917192	39.1%	1.22	6.5664%	1.0426	725514	3.7%	7881293	39.9%	667089	3.4%
87458073	36272929	41.5%	1.20	5.3403%	0.8352	740136	3.9%	9606730	50.0%	1044767	5.4%
87257150	36845111	42.2%	1.22	5.3479%	0.8539	1074574	5.6%	9509844	49.2%	1085790	5.6%
69623622	27098126	38.9%	1.18	4.2375%	0.6077	938659	7.5%	4538021	36.3%	438571	3.5%
56071416	20759387	37.0%	1.15	3.3863%	0.4269	523417	5.2%	3706486	36.9%	321848	3.2%
53481805	15844810	29.6%	0.88	2.9479%	0.2833	284279	2.6%	4416432	41.0%	580631	5.4%
40712678	17601276	43.2%	1.25	2.5106%	0.3449	947122	9.7%	4447021	45.8%	560107	5.8%
40508212	16633763	41.1%	1.20	2.4735%	0.3225	533202	5.6%	4506761	47.6%	535283	5.7%
27315830	11298221	41.4%	1.25	1.6858%	0.2158	459206	8.4%	2069060	37.7%	194246	3.5%
25820246	11264500	43.6%	1.22	1.5825%	0.2049	1531928	24.9%	1948879	31.7%	209941	3.4%
25708037	10323668	40.2%	0.98	1.4793%	0.1753	2703393	44.7%	1318111	21.8%	244025	4.0%
22901445	10618268	46.4%	1.29	1.4250%	0.1941	1720125	30.4%	1714358	30.3%	185592	3.3%

BACKUP: Single Module Graphs



BACKUP: Modular Symbol Level Analysis

- Uses the Perfmon2 interface (*libpfm*) directly and analyses each **CMSSW module** separately

- **Sampling periods** are specific to each event in order to have reasonable measurements

- The list of modules is a HTML table sortable by number of samples of *UNHALTED_CORE_CYCLES*

- For each module the complete set of usual events (*Cycle Accounting Analysis* & others) is sampled

- Results of each module are presented in separate HTML pages in tables sorted by decreasing sample count

BACKUP: The List of Modules

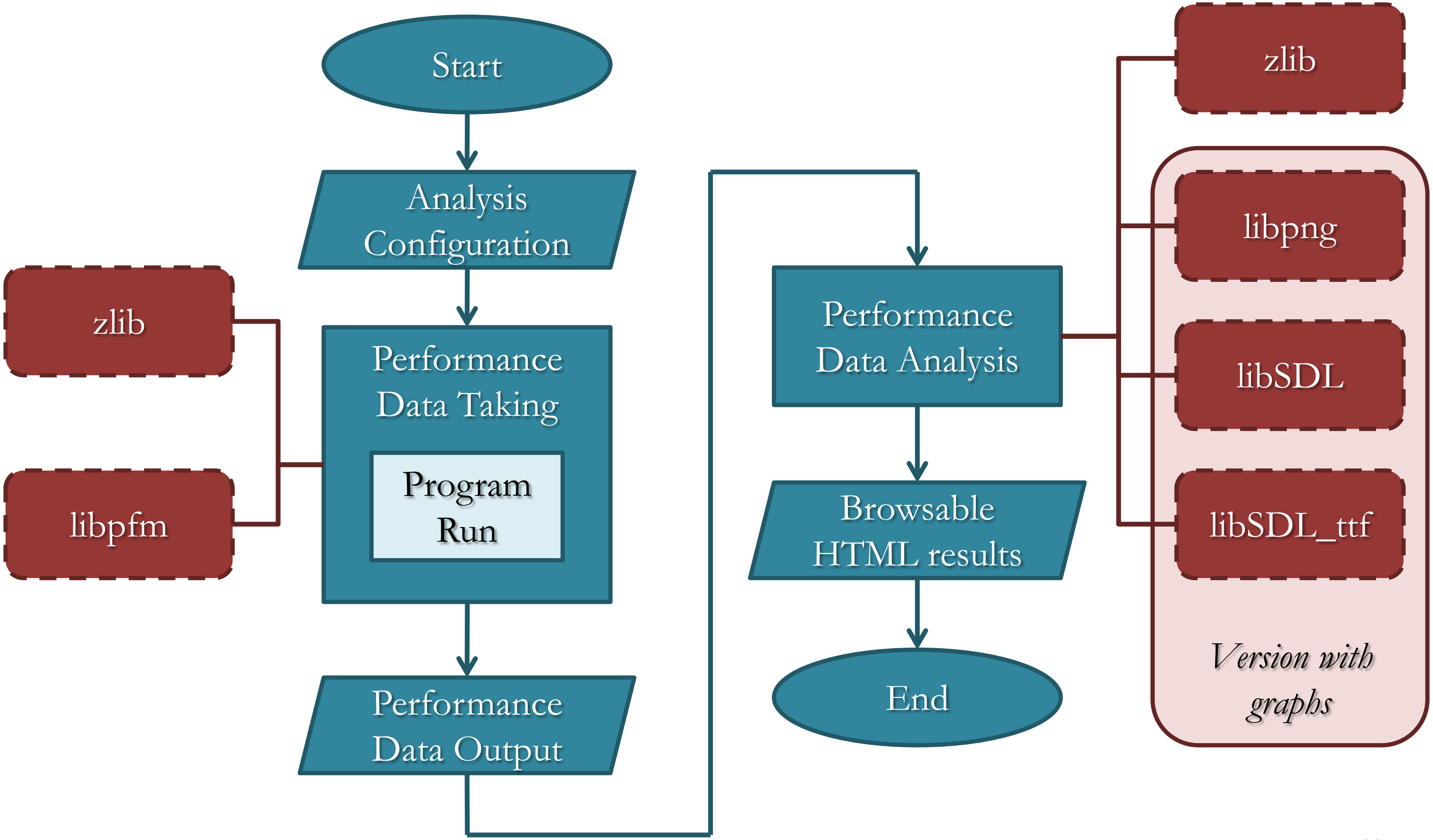
Sampling Analysis Results

#SAMPLES (UNHALTED_CORE_CYCLES) ▲	MODULES
118696	CkfTrackCandidateMaker_hltL1NonIsoEgammaRegionalCkfTrackCandidates
69756	ElectronSeedProducer_hltL1NonIsoLargeWindowElectronPixelSeeds
66556	ElectronSeedProducer_hltL1NonIsoStartUpElectronPixelSeeds
61888	CkfTrackCandidateMaker_hltL1IsoEgammaRegionalCkfTrackCandidates
53719	PixelTrackProducer_hltPixelTracksForMinBias
43667	CkfTrackCandidateMaker_hltBLifetimeRegionalCkfTrackCandidatesStartupU
33560	CkfTrackCandidateMaker_hltCkfL1NonIsoLargeWindowTrackCandidates
32355	ElectronSeedProducer_hltL1IsoLargeWindowElectronPixelSeeds
30706	ElectronSeedProducer_hltL1IsoStartUpElectronPixelSeeds
27055	EcalRawToRecHitProducer_hltEcalRecHitAll
20868	L2TauNarrowConeIsolationProducer_hltL2TauNarrowConeIsolationProducer

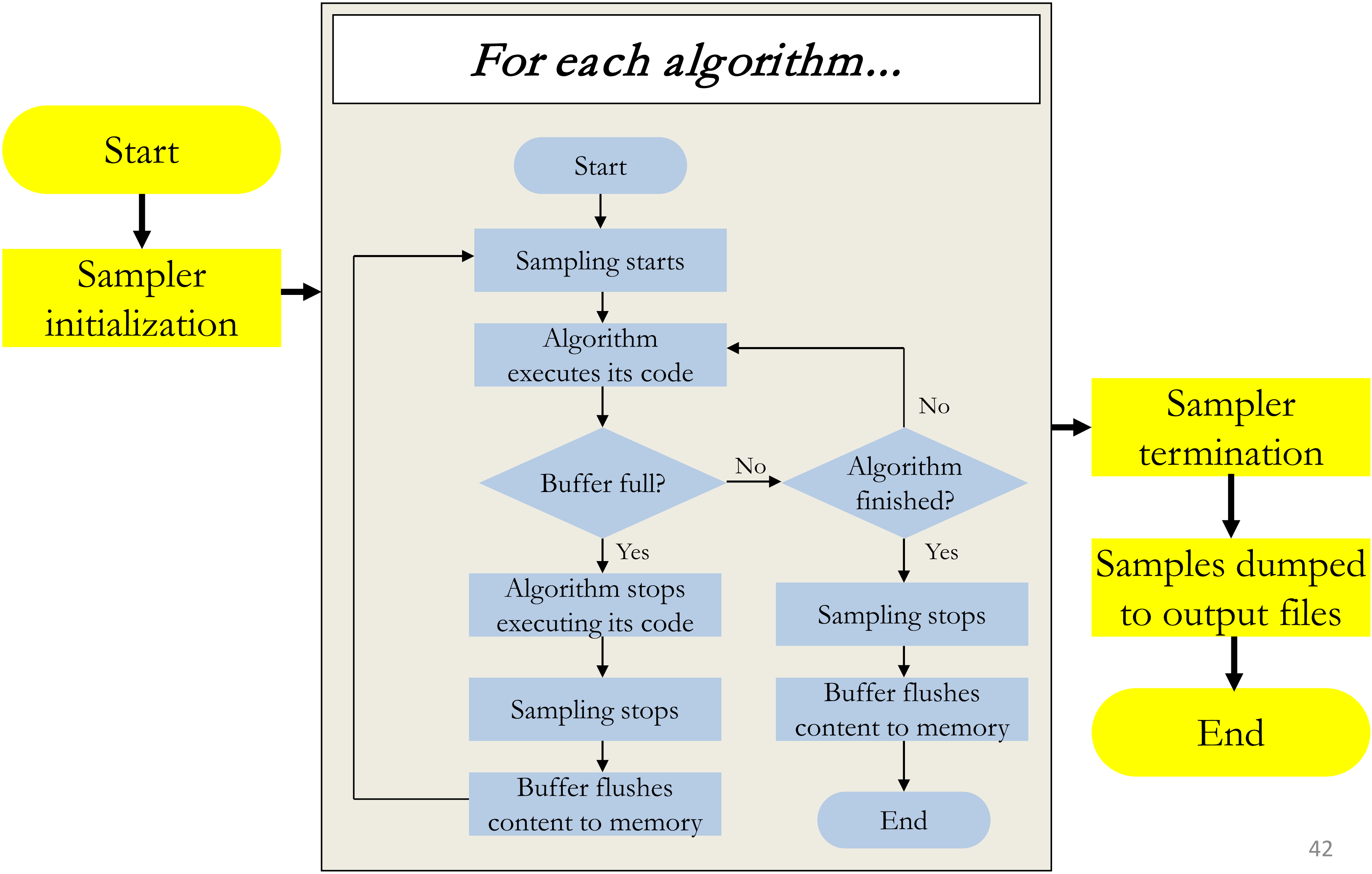
BACKUP: Table Example of a Module

RS_UOPS_DISPATCHED_INV_1_CMASK_1 -- Total Samples: 41333 -- Sampling Period: 100000				
Samples	**Percentage**	**Symbol Name**	**Library Name**	**Complete Signature**
3427	8.291196%	__ieee754_atan2	libm-2.5.so	__ieee754_atan2
1281	3.099219%	_int_malloc	libc-2.5.so	_int_malloc
1202	2.908088%	tan	libm-2.5.so	tan
908	2.196792%	match	libRecoLocalTrackerSiStripRecHitConverter.so	SiStripRecHitMatcher::match(SiStripRecHit2D const*, __gnu_cxx::__normal_iterator<SiStripRecHit2D const* const*, std::vector<SiStripRecHit2D const*, std::allocator<SiStripRecHit2D const*> > >, __gnu_cxx::__normal_iterator<SiStripRecHit2D const* const*, std::vector<SiStripRecHit2D const*, std::allocator<SiStripRecHit2D const*> > >, boost::function<void ()(SiStripMatchedRecHit2D const&)>&, GluedGeomDet const*, Vector3DBase<float, LocalTag>) const
899	2.175018%	fesetenv	libm-2.5.so	fesetenv
849	2.054049%	computeFullJacobian	libTrackingToolsAnalyticalJacobians.so	AnalyticalCurvilinearJacobian::computeFullJacobian (GlobalTrajectoryParameters const&, Point3DBase<float, GlobalTag> const&, Vector3DBase<float, GlobalTag> const&, Vector3DBase<float, GlobalTag> const&, double const&)
568	1.374205%	JacobianCurvilinearToLocal	libTrackingToolsAnalyticalJacobians.so	JacobianCurvilinearToLocal::JacobianCurvilinearToLocal (Surface const&, LocalTrajectoryParameters const&, MagneticField const&)
566	1.369366%	localMomentum	libTrackingToolsTrajectoryState.so	BasicSingleTrajectoryState::localMomentum() const
566	1.369366%	__ieee754_exp	libm-2.5.so	__ieee754_exp

BACKUP: Structure and libraries



BACKUP: The Sampling Process



BACKUP: Module Level Analysis Results Snapshot

RESULTS:

Click for graphs...

MODULE NAME	Total Cycles	Cycles Stalled	% of Cycles Stalled	CPI Ratio	Improvement Margin	iFactor	L2 Miss Impact	% of Total Stalls
CreateOfflinePhotons	66099336	25827555	39.1%	1.27	7.9836%	1.5116	2786991	14.8%
FitForward	52049150	15468001	29.7%	0.99	5.8517%	0.7391	5391694	28.5%
FitMatch	48407093	14049725	29.0%	0.97	5.3973%	0.6471	4874571	27.4%
MuonIDAlg	44410064	15731374	35.4%	1.17	5.2587%	0.9655	4277709	20.0%
FitDownstream	40519279	12222142	30.2%	0.97	4.5191%	0.5449	3686451	25.1%
RichOfflineGPIDLLIt0	37708627	12480747	33.1%	1.06	4.3366%	0.4460	3594169	28.7%
FitSeedForMatch	36451853	10623706	29.1%	0.94	4.0237%	0.4471	3079995	24.0%
RefitSeed	35875307	10299942	28.7%	0.93	3.9507%	0.4273	2813026	22.4%
RichOfflineGPIDLLIt1	25363184	7987786	31.5%	1.12	2.9631%	0.2486	2810843	24.0%
SpdMon	21383173	5174171	24.2%	0.93	2.3552%	0.2267	911354	8.9%
FitVelo	21243866	5834260	27.5%	0.97	2.3692%	0.2037	933870	12.9%

BACKUP: *Improvement Margin* and *iFactor*

iMargin *(CPI reduction effects)*

data: *cur_CPI, exp_CPI, local_cyc_bef, glob_cyc_before.*

- *loc_imp_ratio = cur_CPI/exp_CPI*
- *loc_cyc_after = local_cyc_bef/ loc_imp_ratio*
- *glob_cyc_after = glob_cyc_before – loc_cyc_before + loc_cyc_after*
- *improvement_margin = 100 – (glob_cyc_after/glob_cyc_before) * 100*

iFactor *(Improvability Factor)*

data: *simd_perc, missp_ratio, stalled_cycles.*

- *simd_factor = 1 – normalized(simd_perc)*
- *missp_factor = normalized(missp_ratio)*
- *stall_factor = normalized(stalled_cycles)*
- *iFactor = stall_factor * (simd_factor + missp_factor + stall_factor)*

<u>BACKUP:</u> " *Vertical* " vs. " *Horizontal* " cut

- For Gaudi and CMSSW we used a " *horizontal* " cut
- Geant4 doesn't provide *hooks* for any horizontal cut
- Modular analysis through *User Actions*
- " *Time division* " instead of " *Code division* "
- Useful or not? maybe... taking **particles**, **energies** and **volumes** into consideration
- Moreover **modular symbol analysis** still provides " *Code division* "

<u>BACKUP:</u> Choice of granularity

- Different levels of granularity were considered (run, event, track and step) as each offered *User Actions*
- **Step-level granularity** was the final winner
- At each step the **particle**, its **energy** (at the *beginning*) and the **physical volume** that it is running through are used
- *Interesting* volumes (at any level in the geometry tree) are given through an input file and used in the results view
- Other volumes are labeled as “OTHER”
- Results are browsable by any of the above variables

BACKUP: " *Total* " vs. " *Average* " count

- *CMSSW* and *Gaudi* used **average counts** of performance events
- All *modules* were "used" the same number of times during a single execution
- No longer true in Geant4 steps since "modules" here are a combination of physics variables
- Therefore we chose to display **total counts** of all performance counters
- *Exception*: for the number of **`UNHALTED_CORE_CYCLES`** we provide both average and total counts

BACKUP: (1/3) How to use it?

- Unpack the following archive in your application directory:
  http://dkruse.web.cern.ch/dkruse/G4_pfm.tar.gz

- Add the following lines to your GNUmakefile (to link *libpfm*):

```
CPPFLAGS += -I/usr/include/perfmon
EXTRALIBS += -lpfm -ldl -
    L/afs/cern.ch/sw/lcg/external/libunwind/0.99/x86_64-slc5-gcc43-
    opt/lib -lunwind
EXTRALIBSSOURCEDIRS +=
    /afs/cern.ch/sw/lcg/external/libunwind/0.99/x86_64-slc5-gcc43-opt
```

- Edit the **"RUN_CONFIG"** attribute in the **pfm_config_*arch*.xml** file inserting the normal run command. Example:

```
RUN_CONFIG="~/geant4/bin/Linux-g++/full_cms bench10.g4"
```

- Edit the **pvs.txt** file inserting the interesting physical volumes:

```
CALO     MUON
VCAL     BEAM
TRAK
```

BACKUP: (2/3) How to use it?

- Add the following lines to your **main()** before **runManager->Initialize()**:

```
#include "PfmSteppingAction.hh”
...
int base_arg_no = 2; //number of standard arguments including executable name
runManager->SetUserAction(new PfmSteppingAction(argv[base_arg_no],
   argv[base_arg_no+1],  atoi(argv[base_arg_no+2]),
(unsigned int)atoi(argv[base_arg_no+3]),  (unsigned int)atoi(argv[base_arg_no+4]),
   argv[base_arg_no+5],  atoi(argv[base_arg_no+6]),
(unsigned int)atoi(argv[base_arg_no+7]),  (unsigned int)atoi(argv[base_arg_no+8]),
   argv[base_arg_no+9],  atoi(argv[base_arg_no+10]),
(unsigned int)atoi(argv[base_arg_no+11]),(unsigned int)atoi(argv[base_arg_no+12]),
   argv[base_arg_no+13], atoi(argv[base_arg_no+14]),
(unsigned int)atoi(argv[base_arg_no+15]),(unsigned int)atoi(argv[base_arg_no+16]),
(unsigned int)atoi(argv[base_arg_no+17]), argv[base_arg_no+18],
(bool)atoi(argv[base_arg_no+19])) );
```

- Compile and link:

```
gmake
g++ -Wall -o create create_config_files_from_xml.cpp -lxerces-c
g++ -Wall -lz -o analyse pfm_gen_analysis.cpp
```

BACKUP: (3/3) How to use it?

- Create results directory:
  ```
  mkdir results
  ```
- Create python run script **`“G4perfmon_runs.py”:`**
  ```
  ./create pfm_config_nehalem.xml
  ```
- Run the application with the *perfmon* monitor:
  ```
  python G4perfmon_runs.py &
  ```
- Analyse the results (optionally generating **`csv`** file):
  ```
  ./analyse results/ --caa [--csv]
  ```
- Check your results using your favourite browser:
  ```
  firefox results/HTML/index.html
  ```

BACKUP: XML configuration file

```xml
<?xml version="1.0" ?>
<PFM_CONFIG>
 <PROPERTIES NAME="GeneralAnalysis" RUN_CONFIG="hlt_HLT.py" OUTPUT_DIR="results/" />
 <CONFIG START_AT_EVENT="4" PARALLEL="0" />
 <EVENTS>
  <EVENT_SET>
   <EVENT NAME="BR_INST_RETIRED:ALL_BRANCHES" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
   <EVENT NAME="ILD_STALL:ANY" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
   <EVENT NAME="MEM_INST_RETIRED:LOADS" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
   <EVENT NAME="MEM_INST_RETIRED:STORES" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
  </EVENT_SET>
  <EVENT_SET>
   <EVENT NAME="INST_RETIRED:ANY_P" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
   <EVENT NAME="ITLB_MISS_RETIRED" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
   <EVENT NAME="MEM_LOAD_RETIRED:DTLB_MISS" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
   <EVENT NAME="MEM_LOAD_RETIRED:L2_HIT" CMASK="0" INVMASK="0" SMPL_PERIOD="0" />
  </EVENT_SET>
  <EVENT_SET>
   <EVENT NAME="ARITH:CYCLES_DIV_BUSY" CMASK="0" INVMASK="0" SMPL_PERIOD="1000" />
  </EVENT_SET>
 </EVENTS>
</PFM_CONFIG>
```