GOODA TUTORIAL

Roberto A. Vitillo

Software & Computing Week, ATLAS Experiment, CERN, 11-15 June 2012

WHAT IS GOODA?

- **Low overhead** open source Performance Monitoring Unit (PMU) event data analysis package
  - A CPU profiler
- Developed in collaboration between Google and LBNL
- Logically composed of four main components:
  - A kernel subsystem that provides an interface to the PMU (perf-events)
  - An event data collection tool (perf)
  - An analyzer creates call graphs, control flow graphs and spreadsheets for a variety of granularities (process, module, function, source etc.)
  - A web based GUI displays the data

MOTIVATION

- What we were looking for:
  - Low overhead profiling
  - Call counts statistics
  - Micro-architectural insights
  - Lightweight & user friendly GUI
  - Open Source
  - Usable for Enterprise & HPC

APPLICATION CLASSES

- **Enterprise applications**
  - characterized by branch dominated execution of small functions (OOP)
  - no dominant hotspots
- **HPC applications**
  - dominated by loops
  - few loops will account for 95% of cycles
- Client applications
  - interactive, may not have performance as a feature
  - video games, lots in common with HPC except for smaller data sets

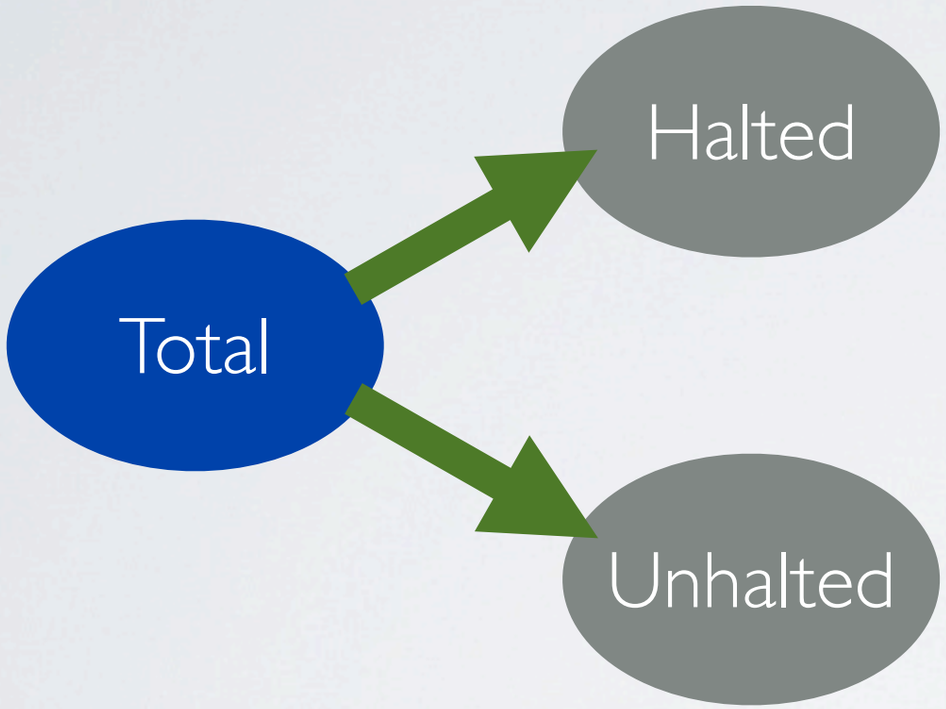
CODE OPTIMIZATION

- Code optimization is minimizing CPU cycles
  - nothing else matters
- Decisions of what code to work on must be based on reasonably accurate estimates of what can be gained... **in cycles!**
- Cycles can be grouped into architecture independent groups
  - forms an hierarchical tree

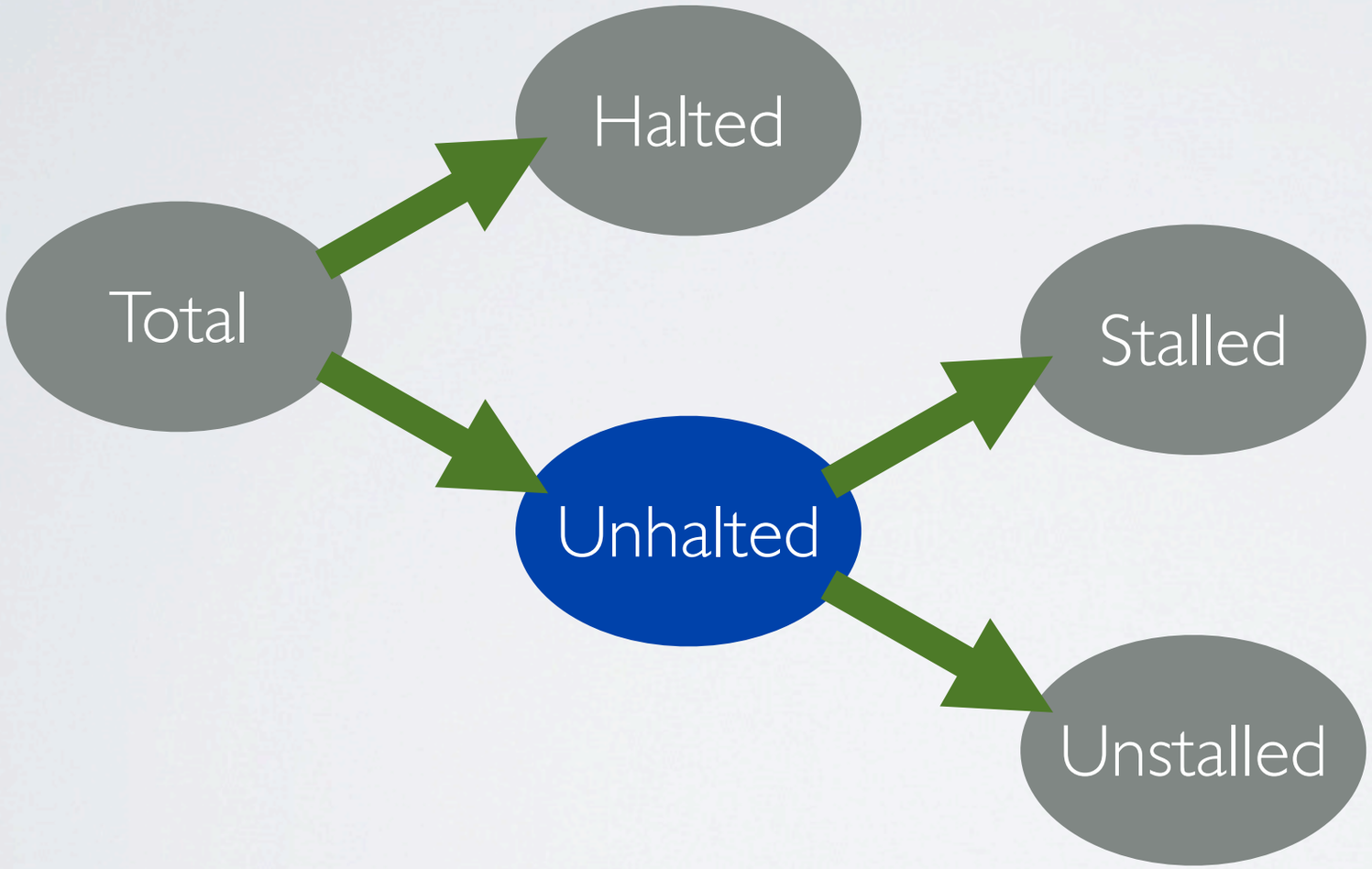
CYCLE ACCOUNTING

Total

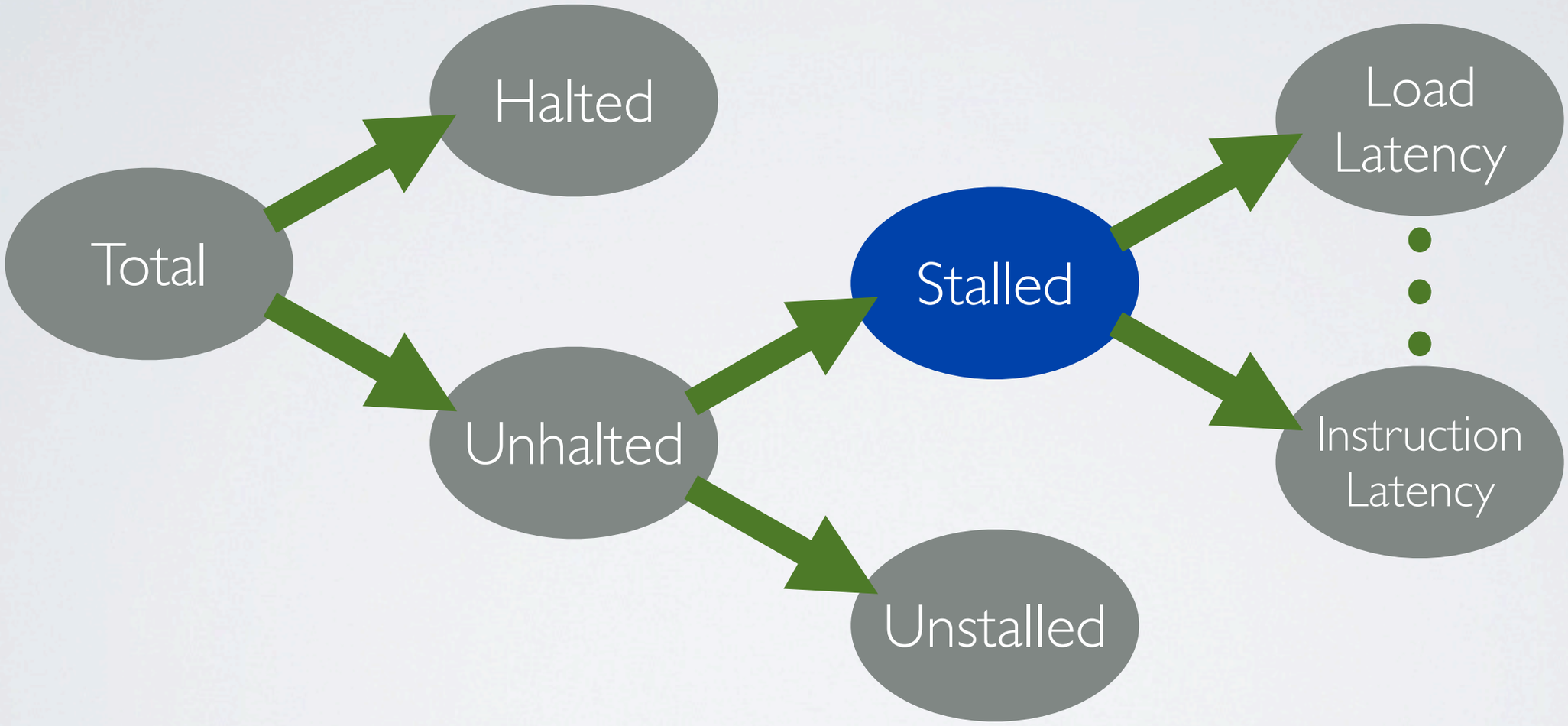
CYCLE ACCOUNTING



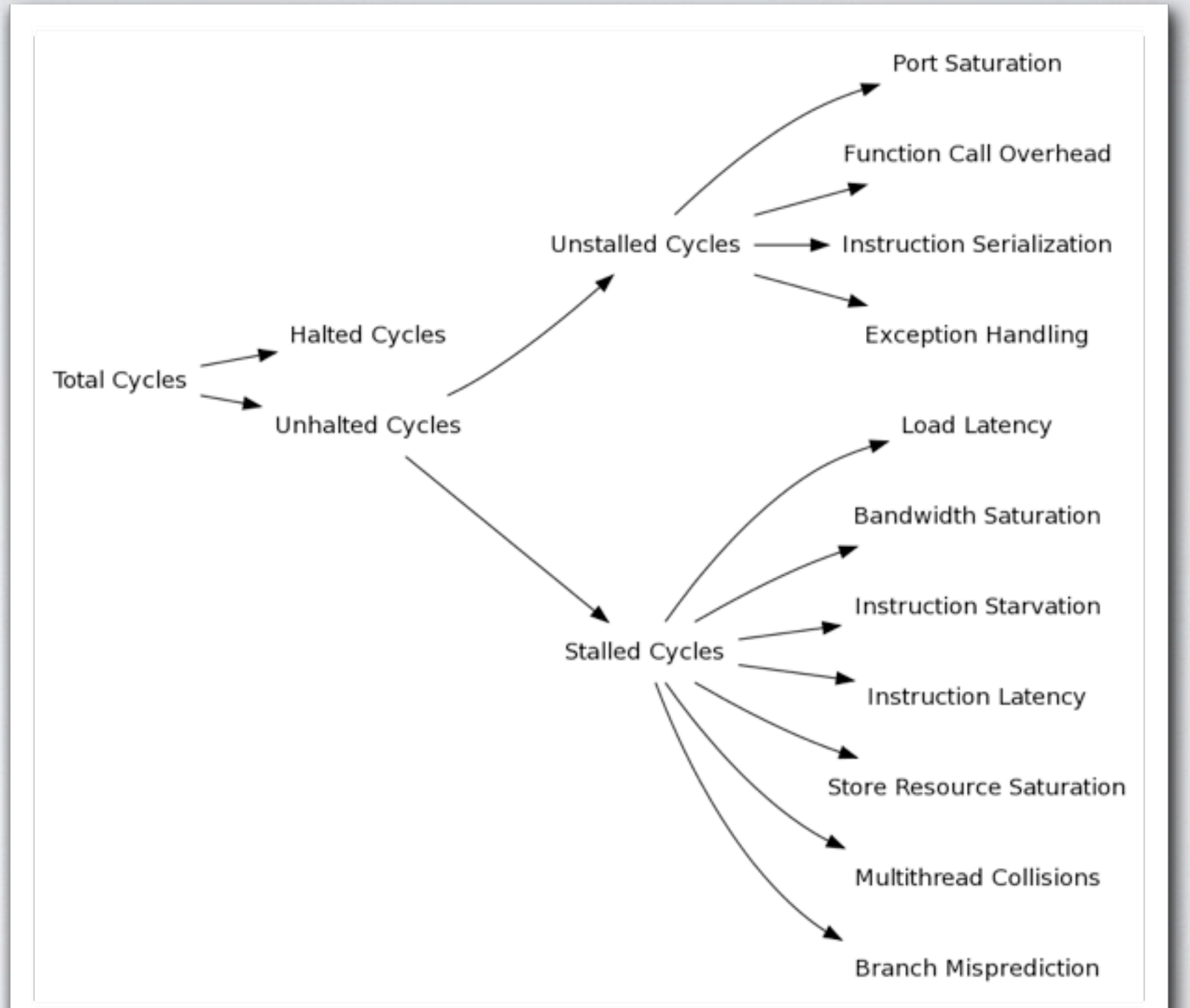
CYCLE ACCOUNTING



CYCLE ACCOUNTING



CYCLE ACCOUNTING



ENTERPRISE APPS

- Main branches of interest for ATLAS code:
  - **Load Latency**: idle cycles spent in waiting for data
  - **Instruction Starvation**: idle cycles spent in waiting for instructions
  - **Branch Misprediction**: idle cycles spent to pursue and flush the wrong path and to load the correct path
  - **Function Call Overhead**: cycles spent in building and tearing down stack frames

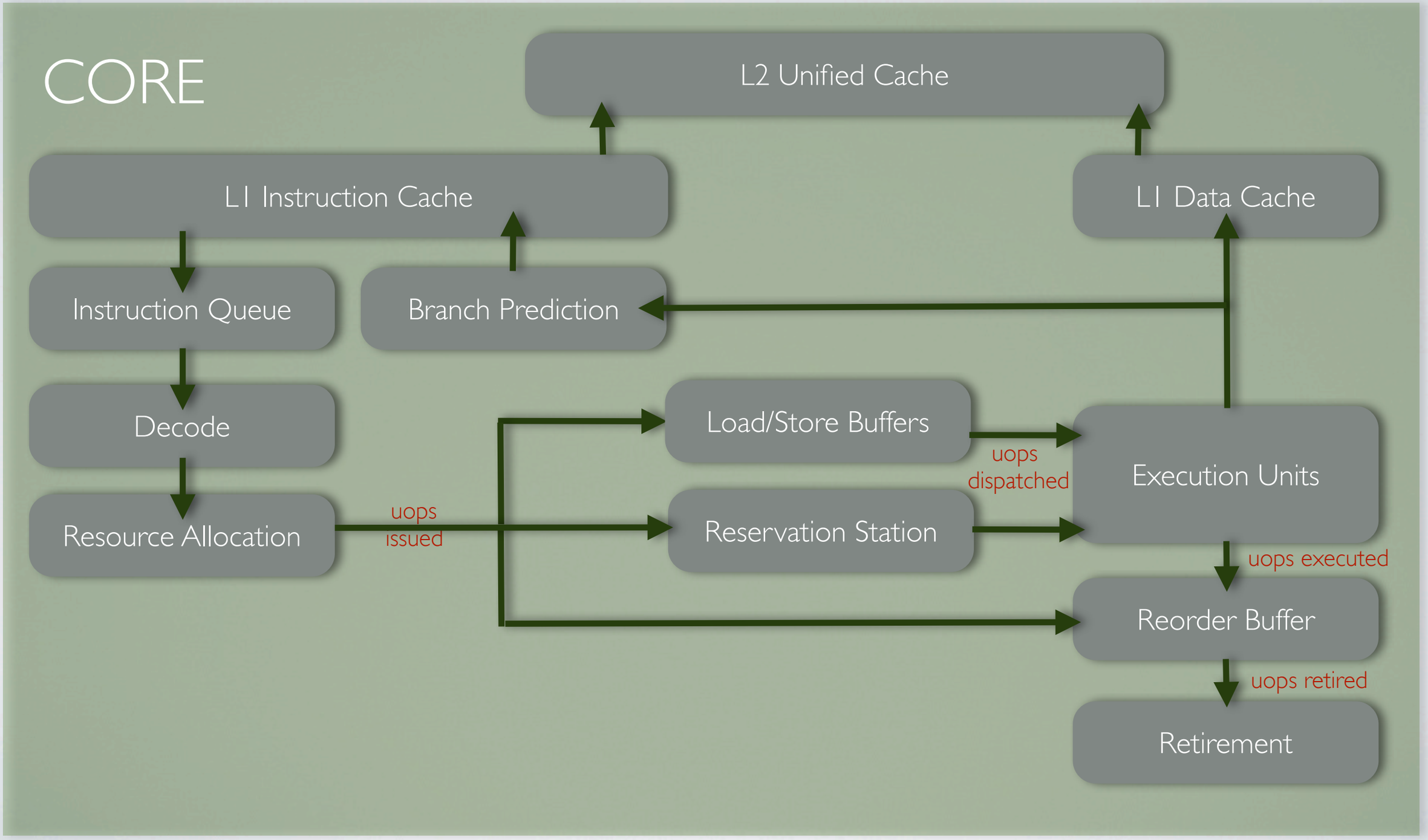
CYCLE ACCOUNTING

HOW DID WE REALIZE IT?

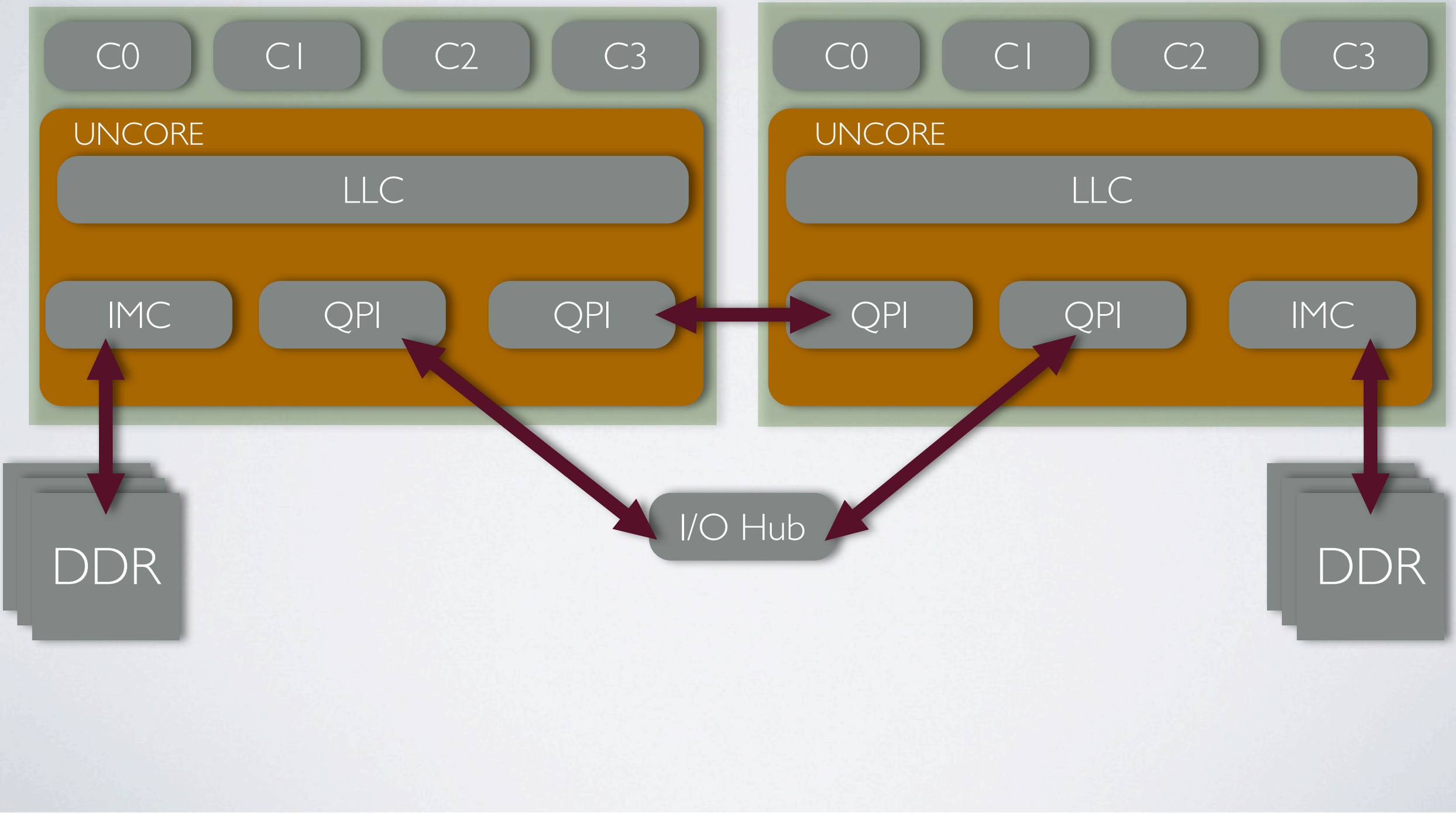
HARDWARE EVENT COLLECTION

- Modern CPU's include a Performance Monitoring Unit (PMU)
- Provides the ability to count the occurrence of micro-architectural events through a small set of counters, e.g.:
  - "executed" instructions
  - cache misses
- Events expose inner workings of the processor as it executes code
  - hundreds of events per architecture
  - **caveat**: events do not map consistently between different architectures

ARTISTIC OUT OF ORDER PIPELINE



ARTISTIC OUT OF ORDER PIPELINE



HARDWARE EVENT COLLECTION

- UOPS_ISSUED
- UOPS_EXECUTED
- UOPS_RETIRED
- Event Options:
  - c (event count threshold per cycle)
  - i (comparison logic, 0 is >= threshold)
  - E.g.: UOPS_EXECUTED:C=1:I=1

HARDWARE EVENT COLLECTION

- PMU counting mode: workload characterization
  - program counter to count desired event and initialize it to zero
  - read value of counter after fixed time
  - good for giving feedback to processor architects
  - most events are targeted for this use case

HARDWARE EVENT COLLECTION

- PMU interrupt mode: profile where events occur vs assembly and source
  - initialize counters to the sampling period, e.g.:
    UOPS_EXECUTED:C=1:I=1:period=2000000
  - an interrupt is triggered when counter is zero
  - capture IP, PID, TID, LBR, CPU and other data on interrupt
- How do we convert event samples to cycles?

CYCLE DECOMPOSITION

- Stalled/unstalled cycles are decomposed as a sum of count(event) * cost(event)
  - the cost is the penalty paid in cycles for a specific event
  - needs to be determined with micro benchmarks (*gooda/gooda-analyzer/kernels*)
- Main branches of interest for Enterprise & HPC apps:
  - Load Latency
  - Instruction Starvation
  - Branch Misprediction
  - Function Call Overhead

CYCLE DECOMPOSITION

LOAD LATENCY

- Load latency will stall the pipeline
  - store latency rarely will
  - events must **only** count loads
  - most cache miss events count loads and stores, e.g.: **data_cache_misses:l2_cache_miss**
  - events must be precise to identify assembly line

CYCLE DECOMPOSITION

LOAD LATENCY

- Count hits instead of misses
  - Use exclusive hit events
- Includes load accesses to caches and memory, load DTLB costs and blocked store forwarding... **lots of events!**
- Latency depends on specific configuration that needs to be determined with micro benchmarks

CYCLE DECOMPOSITION
LOAD LATENCY

- Load Latency on Westmere
  - 6 * mem_load_retired:l2_hit +
  - 52 * mem_load_retired:l3_unshared_hit +
  - 85 * (mem_load_retired:other_core_l2_hit_hitm - mem_uncore_retired:local_hitm) +
  - 95 * mem_uncore_retired:local_hitm +
  - 250 * mem_uncore_retired:local_dram_and_remote_cache_hit +
  - 450 * mem_uncore_retired:remote_dram +
  - 250 * mem_uncore_retired:other_llc_miss +
  - 7 * (dtlb_load_misses:stlb_hit + dtlb_load_misses:walk_completed) + dtlb_load_misses:walk_cycles +
  - 8 * load_block_overlap_store
- Tools needs to know methodology so users don't!

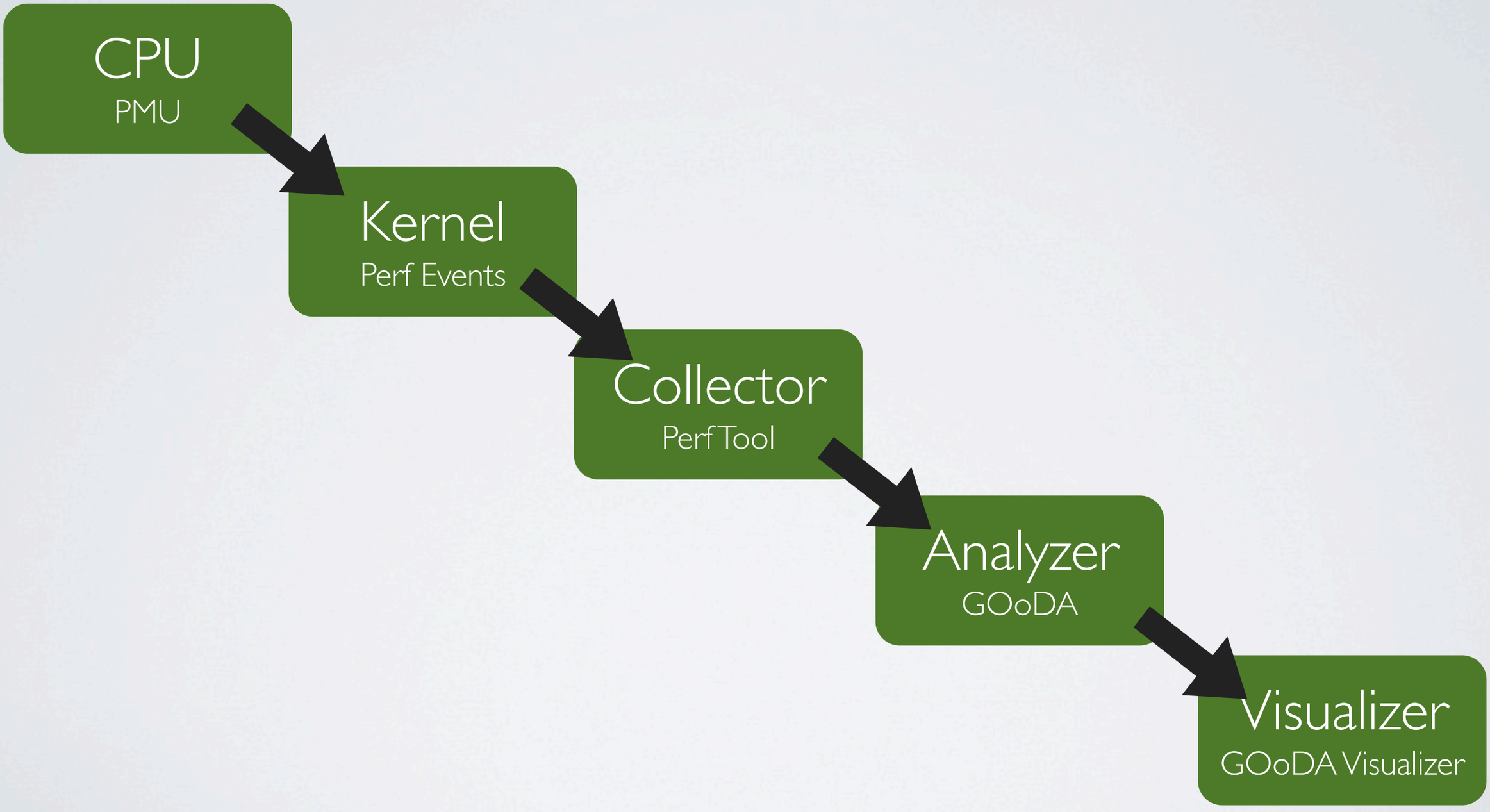
CYCLE DECOMPOSITION

- Same methodology applied to the other main branches
- Cycle decomposition expressions described in *gooda/gooda-analyzer/docs/CycleAccountingandPerformanceAnalysis.pdf*

HARDWARE EVENT COLLECTION

HOW DID WE IMPLEMENT IT?

HOW GOODA WORKS



PERF EVENTS

- Performance monitoring interface introduced in the kernel in 2009
- Unified interface to access hardware performance counters, kernel software counters and tracepoints
- System call interface that exposes an high level abstraction known as event
- Events are manipulated via file descriptor obtained through the perf_event_open system call
- Samples are saved into a kernel buffer which is made visible to tools via the mmap system call

PERF TOOL

- User space tool which allows counting and sampling of events
- abstracts CPU hardware differences in Linux performance measurements and presents a simple command-line interface
- Used by the GOoDA collection scripts to collect samples into a data file

PERF TOOL

PERF LIST

- List the supported events
  - raw events in the form of rNNN where NNN is a hexadecimal event descriptor [1]
  - libpfm patch for symbolic event names available in *gooda/gooda-analyzer/perf-patches*
  - An event can have sub-events (or unit masks) and modifiers which alter when and how an event is counted

[1] see Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide

PERF TOOL

PERF STAT

```
perf stat -B dd if=/dev/zero of=/dev/null count=1000000

1000000+0 records in
1000000+0 records out
512000000 bytes (512 MB) copied, 0.956217 s, 535 MB/s

 Performance counter stats for 'dd if=/dev/zero of=/dev/null count=1000000':

          5,099 cache-misses              #      0.005 M/sec (scaled from 66.58%)
        235,384 cache-references          #      0.246 M/sec (scaled from 66.56%)
      9,281,660 branch-misses             #      3.858 %     (scaled from 33.50%)
    240,609,766 branches                  #    251.559 M/sec (scaled from 33.66%)
  1,403,561,257 instructions              #      0.679 IPC   (scaled from 50.23%)
  2,066,201,729 cycles                    #   2160.227 M/sec (scaled from 66.67%)
            217 page-faults               #      0.000 M/sec
              3 CPU-migrations            #      0.000 M/sec
             83 context-switches          #      0.000 M/sec
    956.474238 task-clock-msecs           #      0.999 CPUs

    0.957617512  seconds time elapsed
```

- Use PMU in counting mode for workload characterization

PERF TOOL

PERF RECORD

- Collects event samples
  - *perf record -e cycles dd if=/dev/zero of=/dev/null count=100000*
  - system-wide collection on all CPUs with -a
  - set period with -c N
  - libpfm events can be specified with --pfm-events
  - GOoDA collection scripts are located in *gooda/gooda-analyzer/scripts/*

PERF TOOL

MULTIPLEXING

- Limited counters available
  - e.g. Nehalem has 4 generic and 3 fixed
  - with multiplexing an event is not measured all the time
  - events managed in round robin fashion
  - GOoDA scales event samples back by considering the multiplexing factor
    - can introduce errors

PERF TOOL

PERF REPORT

```
perf report

# Events: 1K cycles
#
# Overhead          Command                     Shared Object
                    Symbol
# ........  ..............  ..............................
.....................................
#
   28.15%      firefox-bin  libxul.so                          [.] 0xd10b45
    4.45%           swapper  [kernel.kallsyms]                 [k]
mwait_idle_with_hints
    4.26%           swapper  [kernel.kallsyms]                 [k] read_hpet
    2.13%      firefox-bin  firefox-bin                       [.] 0x1e3d
    1.40%  unity-panel-ser  libglib-2.0.so.0.2800.6          [.] 0x886f1
    [...]
```

- Reads a perf.data file and generates an execution profile
- Allows to annotate source and disassembly but only one event at the time

EXAMPLE: POINTER CHASING

```
for(int i = 0; i < len; i++)
{
    p = *p; // *p = &p + 64, p[last] = &p[0]
    ...     // non memory ops
}
```

EXAMPLE: POINTER CHASING

COUNTING LLC MISSES

```
for(int i = 0; i < len; i++)
{
    p = *p; // *p = &p + 64, p[last] = &p[0]
    ...     // non memory ops
}
```

What's wrong here?

```
 0.00 :        400b28:       inc    %ebx
 0.00 :        400b2a:       mov    (%rax),%rax
 0.00 :        400b2d:       xor    %r14,%r15
 0.00 :        400b30:       xor    %r14,%r15
 0.02 :        400b33:       xor    %r14,%r15
 0.00 :        400b36:       xor    %r14,%r15
 ....
 0.00 :        400c17:       xor    %r14,%r15
99.54 :        400c1d:       xor    %r14,%r15
 0.00 :        400c20:       xor    %r14,%r15
 ....
 0.00 :        400d0d:       cmp    %ecx,%ebx
 0.00 :        400d0f:       jl     400b28
```

EXAMPLE: POINTER CHASING

PEBS

LLC misses w PEBS [1]

```
  0.00 :          400b28:       inc    %ebx
100.00 :          400b2a:       mov    (%rax),%rax
  0.00 :          400b2d:       xor    %r14,%r15
  0.00 :          400b30:       xor    %r14,%r15
  0.00 :          400b33:       xor    %r14,%r15
  0.00 :          400b36:       xor    %r14,%r15
  ....
  0.00 :          400c17:       xor    %r14,%r15
  0.00 :          400c1d:       xor    %r14,%r15
  0.00 :          400c20:       xor    %r14,%r15
  ....
  0.00 :          400d0d:       cmp    %ecx,%ebx
  0.00 :          400d0f:       jl     400b28
```

LLC misses w/o PEBS [2]

```
 0.00 :          400b28:       inc    %ebx
 0.00 :          400b2a:       mov    (%rax),%rax
 0.00 :          400b2d:       xor    %r14,%r15
 0.00 :          400b30:       xor    %r14,%r15
 0.02 :          400b33:       xor    %r14,%r15
 0.00 :          400b36:       xor    %r14,%r15
 ....
 0.00 :          400c17:       xor    %r14,%r15
99.54 :          400c1d:       xor    %r14,%r15
 0.00 :          400c20:       xor    %r14,%r15
 ....
 0.00 :          400d0d:       cmp    %ecx,%ebx
 0.00 :          400d0f:       jl     400b28
```

note: walk_pebs available under gooda/gooda-analyzer/kernels/mem_latency

[1] perf record --pfm-events mem_load_retired:llc_miss:period=10000:precise=2 ./walk_pebs -i2 -r2 -l3200000 -s0 S64 -m1

[2] perf record --pfm-events mem_load_retired:llc_miss:period=10000 ./walk_pebs -i2 -r2 -l3200000 -s0 S64 -m1

EXAMPLE: POINTER CHASING

PEBS

- PEBS: Precise Event Based Sampling
- Enables the PMU to capture the architectural state at the completion of the instruction that caused the event
- Captured data often referred to as "IP + 1"
- Categorized in Memory Events and Execution Events

EXAMPLE: POINTER CHASING

PEBS: MEMORY EVENTS

- Moving the sample up by one instruction is enough to identify the offending memory instruction
- PEBS buffer contains the values of all 16 general registers
  - when coupled with the disassembly, the address can be reconstructed for data access profiling

LESSONS LEARNED

- You might **not** count what you think!
- Using hardware events correctly is **hard**!
- It gets worse: generic events count all sorts of things
  - generic **l1d_miss** counts **l2_hw_prefetch** that hit L2 on WSM
  - **l1d_miss** is actually **l2_rqst:umask=FF**

GOODA ANALYZER

- Reads and parses a perf.data file
- Implements the cycle accounting methodology
  - depends on the underlying architecture!
- Generates spreadsheets for:
  - hot processes and functions
  - source and assembly for the N hottest functions
- Generates SVG's of the Call Graph and the Control Flow Graph

GOODA ANALYZER

- *gooda* generates a spreadsheets directory for a report:
  - ***platform_properties.txt***
  - ***function_hotspots.csv***
  - ***process.csv***
  - ***asm/***
  - ***src/***
  - ***cfg/***
  - ***cg/***
- By default gooda will
  - try to open perf.data, use -i to use another filename
  - create asm, src and cfg for the hottest 20 functions, limit can be changed with the -n option

GOODA VISUALIZER

- HTML5, CSS3 & Javascript based GUI
- Reads, parses and displays the spreadsheets generated by the Analyzer
- Can be deployed on a webserver or on a client machine
- A modern browser is the only dependency

VISUALIZER

- *visualizer/reports* subdirectory contains the desired reports
- *visualizer/reports/list* contains a list of available reports
- Behavior will change in the future...

IN ACTION: HOT PROCESSES

process path	module path	unhalted_core_cycles		uop
		473185	(100%)	266508
⊞ athena.py		463031	(100%)	246143
⊞ vmlinux		9006	(100%)	19529
⊞ gnome-settings-		328	(100%)	156
⊞ irqbalance		253	(100%)	142
⊞ khugepaged		164	(100%)	142
⊞ perf		134	(100%)	85
⊞ flush-253:0				14
⊞ ksoftirqd/3		45	(100%)	

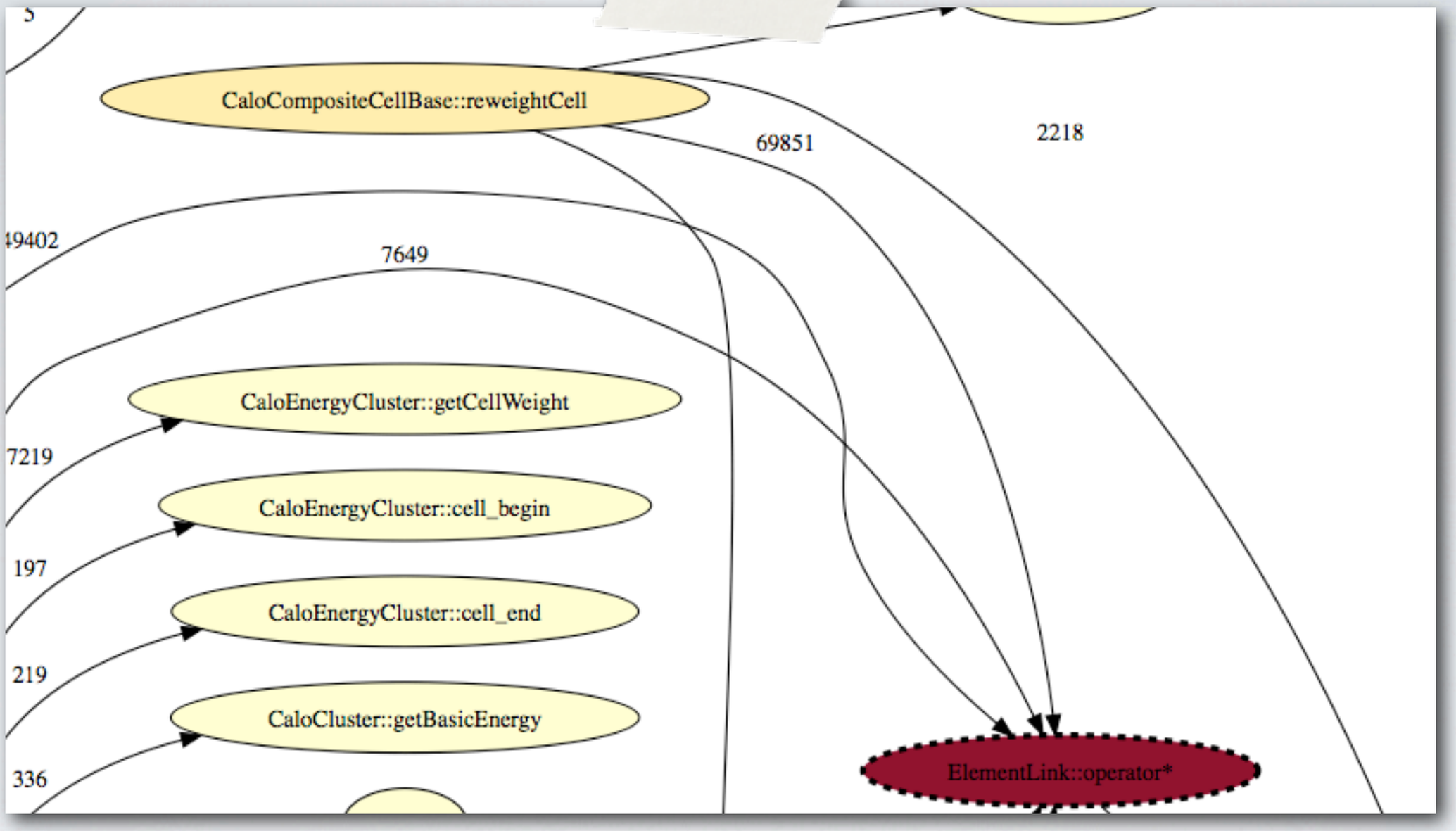
Processes ordered by hotness

IN ACTION: HOT MODULES

process path	module path	unhalted_core_cycles		uop
		473185	(100%)	266508
⊟ athena.py		463031	(100%)	246143
	libCaloEvent.so	28434	(100%)	15320
	libtcmalloc_minimal.so	28897	(100%)	14966
	libm-2.12.so	41526	(100%)	22066
	libBFieldStand.so	28792	(100%)	15122
	libstdc++.so.6.0.10	22440	(100%)	14030
	libCLHEP-Matrix-1.9.4.7.so	12644	(100%)	5017
	ld-2.12.so	11451	(100%)	4478
	libTrkAlgebraUtils.so	13464	(100%)	5456

Modules ordered by hotness

IN ACTION: CALLGRAPH



No instrumentation required: the callgraph is generated from LBRs

IN ACTION: HOT FUNCTIONS

function name	offset	length	module	process	unhalted_core_cycles		uop
					473185	(100%)	266508
operator new(unsigned lon…	0x134b0	0x3da	libtcmalloc_minimal.so	athena.py	12927	(100%)	5442
master.0.gbmagz_	0xfb80	0x4a0b	libBFieldStand.so	athena.py	13882	(100%)	5995
operator delete(void*)	0x12c10	0x2da	libtcmalloc_minimal.so	athena.py	7619	(100%)	3741
std::_Rb_tree_increment(s…	0x69c00	0x5a	libstdc++.so.6.0.10	athena.py	8633	(100%)	5697
get_bsfield_	0xed60	0xe16	libBFieldStand.so	athena.py	11407	(100%)	7809
Trk::STEP_Propagator::pro…	0x2b230	0x18e2	libTrkExSTEP_Propagator.so	athena.py	6337	(100%)	2792
Trk::RungeKuttaPropagator…	0x250e0	0x1051	libTrkExRungeKuttaPropagato…	athena.py	7589	(100%)	4478
ma27od_	0x22000	0x26ee	libTrkAlgebraUtils.so	athena.py	6397	(100%)	2083
Trk::FitMatrices::solveEq…	0x108a0	0x49a	libTrkiPatFitterUtils.so	athena.py	4935	(100%)	1701
deflate_slow	0x6850	0x976	libz.so.1.2.3	athena.py	5189	(100%)	2395

Dive into assembly and source code...

IN ACTION: SOURCE

line number	source	unhalted_core_cycles		uops_	
		6337	(100%)	2792	(44
1050	numSf++;	45	(100%)	14	(31
1051	} else {				
1052	// save the nearest distance to surface				
1053	m_currentDist.push_back(std::pair<int,std::pair<double,double> >(-1…	641	(100%)	184	(28
1054	}				
1055	}				
1056					
1057	if (distanceToTarget == maxPath || numSf == 0) {				
1058	//std::cout << "propagateWithJacobian: initial distance estimate faile…				
1059	if(m_currentDist.capacity() > m_maxCurrentDist) m_currentDist.reserv…				

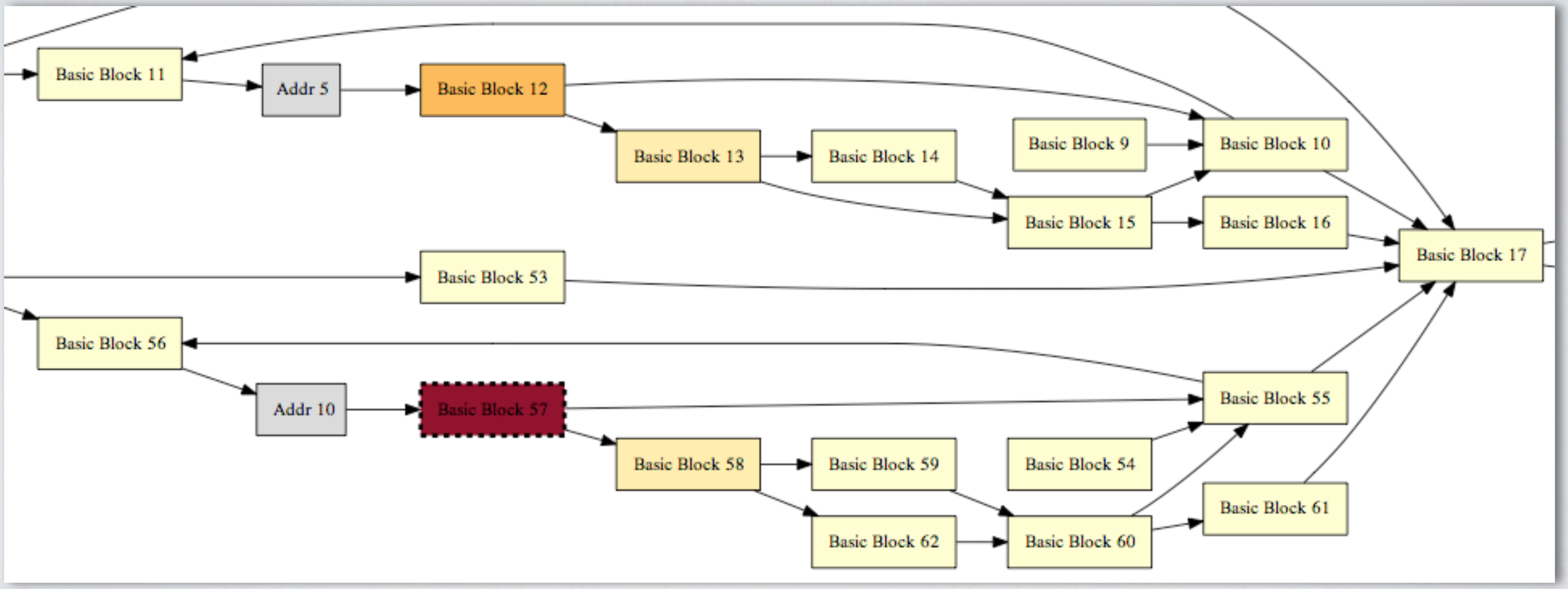
Pinpoint hot source lines

IN ACTION: ASSEMBLY

address	princ_l#	disassembly	unhalted_core_cycles		uops_r
			6397 (100%)		2083 (32%
0x23db4	1643	mov %esi,0xe67e(%rip)	15	(100%)	
0x23dba	1645	jl 23d48			
0x23dbc	1645	⊟ Basic Block 262 <0x23dc0>			
0x23dbc	1645	nopl 0x0(%rax)			
0x23dc0	1645	⊟ Basic Block 263 <0x23dc0><0x23e04>	5099	(100%)	1616 (31%
0x23dc0	1646	mov 0xe64e(%rip),%ecx	45	(100%)	
0x23dc6	1646	mov %ecx,%eax	119	(100%)	57 (47%
0x23dc8	1647	movslq %ecx,%rdx	567	(100%)	113 (19%
0x23dcb	1646	sub %edi,%eax			
0x23dcd	1647	sub $0x1,%rdx	30	(100%)	14 (46%
0x23dd1	1645	cmp %ecx,%esi	15	(100%)	14 (93%

Pinpoint hot basic blocks

IN ACTION: CFG



CYCLE ACCOUNTING TREE

:any	inst_ret...ear_return	load_latency	instruction_starvation	bandwidth_saturated	branch_misprediction	store_resources_saturated	instruction_latency	exceptio
258131 (54%)	65263 (13%)	6963 (1%)	13628 (2%)	56481 (11%)	29016 (6%)	7232 (1%)		
8782 (67%)	1342 (10%)	30 (0%)	328 (2%)	1998 (15%)	45 (0%)			
4488 (32%)	507 (3%)	15 (0%)	104 (0%)		3772 (27%)	268 (1%)		
4399 (57%)	731 (9%)		75 (0%)	746 (9%)		30 (0%)		
7798 (90%)	1088 (12%)	179 (2%)	820 (9%)		30 (0%)			
164 (1%)	15 (0%)				6695 (58%)			
1238 (19%)	910 (14%)		358 (5%)	2043 (32%)	537 (8%)	15 (0%)		

Branches can be expanded and explored

AN USE CASE

ATLAS RECONSTRUCTION

http://annwm.lbl.gov/~vitillo/visualizer/

RESOURCES

GOoDA

http://code.google.com/p/gooda/

GOoDA Visualizer

http://code.google.com/p/gooda-visualizer/

Build Script (kindly provided by Peter Waller)

https://github.com/pwaller/gooda/compare/dev