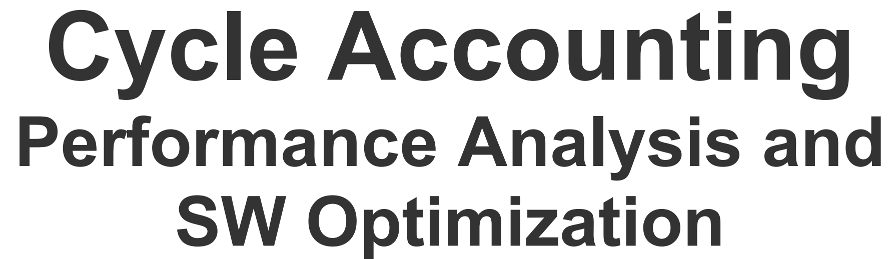


Cycle Accounting

Performance Analysis and SW Optimization

David Levinthal
April 6, 2012

Disclaimer: This talk will never be finished and in a continual state of being updated. It also serves as documentation for the Gooda analyzer and as such updates for new processors and new analysis will be added whenever needed.

Basic Performance Analysis

Four step process that iterates

1. Make sure the platform is correctly configured
   - Do not take this for granted
2. Use a really good compiler for the job
   - different compilers have different strengths
   - Use the right compiler options
3. Analyze the interaction of the SW with the HW and tune the code accordingly
   - Gooda...or if you must..any of a host of other tools
4. Parallelize the execution
   - Batch queue (you cannot beat trivial parallelism)
   - MPI
   - Threading

Iterate on steps 3 and 4

Platform checkout

Make sure the platform is right

1. enough memory
   a. check page fault rate with vmstat, perf
      i. rate > 100/sec should be investigated
2. Make sure the DIMMS are right
   a. if there is a 3 channel memory controller the number of dimms/socket must be a multiple of 3 and in the correct slots!
   b. Dimms should be matched!
3. Bios settings
   a. prefetchers on
   b. NUMA mode (not interleaved)
   c. Disks in SATA mode (not IDE)
   d. disable power management during measurements
      i. at least initially

Google™

Compiler Usage and Optimization

General rule: The less you ask the compiler to do,
the more likely you will find joy and fulfillment

Complex compilation (inlining + loop unrolling + load hoisting +.....)
may not be as effective as you think

Related axiom: Make the compiler's life easy

Write code that a chimpanzee with an abacus can code generate and schedule correctly

1. Only optimize the time consuming functions
   - speeds up build and avoid destabilizing the compiler

Example: The triad
Compilation can be the difference

Topologically equivalent to all programs
it has input, output and some calculations
for(i=0; i<len; i++) a[i] = b[i] + x*c[i];

BAD

```
28:  8b 45 fc           mov    -0x4(%rbp),%eax
2b:  48 98              cltq
2d:  48 c1 e0 03        shl    $0x3,%rax
31:  48 03 45 d8        add    -0x28(%rbp),%rax
35:  8b 55 fc           mov    -0x4(%rbp),%edx
38:  48 63 d2           movslq %edx,%rdx
3b:  48 c1 e2 03        shl    $0x3,%rdx
3f:  48 03 55 d0        add    -0x30(%rbp),%rdx
43:  f2 0f 10 0a        movsd  (%rdx),%xmm1
47:  8b 55 fc           mov    -0x4(%rbp),%edx
4a:  48 63 d2           movslq %edx,%rdx
4d:  48 c1 e2 03        shl    $0x3,%rdx
51:  48 03 55 c8        add    -0x38(%rbp),%rdx
55:  f2 0f 10 02        movsd  (%rdx),%xmm0
59:  f2 0f 59 45 e0     mulsd  -0x20(%rbp),%xmm0
5e:  f2 0f 58 c1        addsd  %xmm1,%xmm0
62:  f2 0f 11 00        movsd  %xmm0,(%rax)
66:  83 45 fc 01        addl   $0x1,-0x4(%rbp)
6a:  8b 45 fc           mov    -0x4(%rbp),%eax
6d:  3b 45 ec           cmp    -0x14(%rbp),%eax
70:  7c b6              jl     28 <triad+0x28>
```

BETTER

```
60:  f2 0f 10 14 02     movsd  (%rdx,%rax,1),%xmm2
65:  41 83 c0 01        add    $0x1,%r8d
69:  f2 0f 10 24 01     movsd  (%rcx,%rax,1),%xmm4
6e:  66 0f 16 54 02 08  movhpd 0x8(%rdx,%rax,1),%xmm2
74:  66 0f 16 64 01 08  movhpd 0x8(%rcx,%rax,1),%xmm4
7a:  66 0f 28 ca        movapd %xmm2,%xmm1
7e:  66 0f 28 d4        movapd %xmm4,%xmm2
82:  66 0f 59 d3        mulpd  %xmm3,%xmm2
86:  66 0f 58 ca        addpd  %xmm2,%xmm1
8a:  66 0f 29 0c 06     movapd %xmm1,(%rsi,%rax,1)
8f:  48 83 c0 10        add    $0x10,%rax
93:  45 39 c8           cmp    %r9d,%r8d
96:  72 c8              jb     60 <triad+0x60>
```

BEST

```
1d:  0f 28 14 c1        movaps (%rcx,%rax,8),%xmm2
21:  66 0f 59 d1        mulpd  %xmm1,%xmm2
25:  66 0f 58 14 c2     addpd  (%rdx,%rax,8),%xmm2
2a:  66 0f 2b 14 c6     movntpd %xmm2,(%rsi,%rax,8)
2f:  48 83 c0 02        add    $0x2,%rax
33:  48 3b c7           cmp    %rdi,%rax
36:  72 e5              jb     1d <triad+0x1d>
```

Google™

Compiler Usage and Optimization

General rule: The less you ask the compiler to do,
the more likely you will find joy and fulfillment

Complex compilation (inlining + loop unrolling + load hoisting +.....)
may not be as effective as you think

Related axiom: Make the compiler's life easy

Write code that a chimpanzee with an abacus can code generate and schedule correctly

1. Only optimize the time consuming functions
   - speeds up build and avoid destabilizing the compiler
2. Check that the compiler actually generated good code

General Optimization

- Do everything possible with data while you have it
- Optimize cache line layout to use every byte while you have it
- Optimization takes advantage of the specifics
  - Avoid overhead of generalization

All this leads to "Write large hand optimized functions"

Bandwidth or Latency limited?

Triad

```
for(i=0; i<len; i++) a[i] = b[i] + x*c[i];
```

Linked list walk

```
i=0;
while(i<len){
      p=*p; //  *p = &p + 64,  p[last] = &p[0]    circular buffer
      i++;
      }
```

Gather

```
for(i=0; i<len; i++) a[i] = b[address[i]];  //  address is "random"
```

Application classes

Server apps break down into 2 dominant groups:
- Enterprise and HPC
- Enterprise apps are characterized by branch dominated execution of small functions (OOP/C++)
- no dominant hotspots

HPC are dominated by loops
- 5-50 hot loops will account for 95% of cycles

Client apps break down to interactive and video game
interactive apps may not have performance as a "feature"
Video games have a lot in common with HPC
Except for smaller data sets

Google™

Optimizing Loops

What is the most critical thing to know about a loop you want to optimize?

- The trip count
- The Trip count
- The TRIP COUNT!
- THE TRIP COUNT!
- THE TRIP COUNT!
- Variations in the tripcount
- Some other things...

whose solutions all depend on the trip count

And getting the tripcount of a loop is not easy

Code Optimization is Minimizing 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
- Cycle accounting computation is architecture specific
  - events do not map consistently between architectures
    - even instructions_retired
- Cycles can be grouped into architecture independent groups
  - some groups will be meaningless on some architectures
  - unlikely all groups are discussed in this talk
  - forms a hierarchical tree

Google™

Hardware Event Collection

Two modes: Counting and Interrupt

- Counting mode: Workload Characterization
  - program counter to count desired event
  - initialize to zero
  - read value of counter after a fixed time
  - Good for giving feedback to processor architects
  - Most events are targeted for this
    - cache hit rates/MESI state, Intel "matrix event"
  - Can assist in optimizing machine configuration
    - Page sizes
- A spreadsheet analysis demo will be done later to illustrate usage

Google™

Hardware Event Collection

Two modes: Counting and Interrupt

- Interrupt mode: profile where events occur vs asm/source
  - enables methodical code optimization
    - program counter to count desired event
    - initialize to overflow - sampling period
    - capture IP, PID, TID, CPU and other data on interupt
  - Post Processing tool needed for generated data file
  - Main focus here

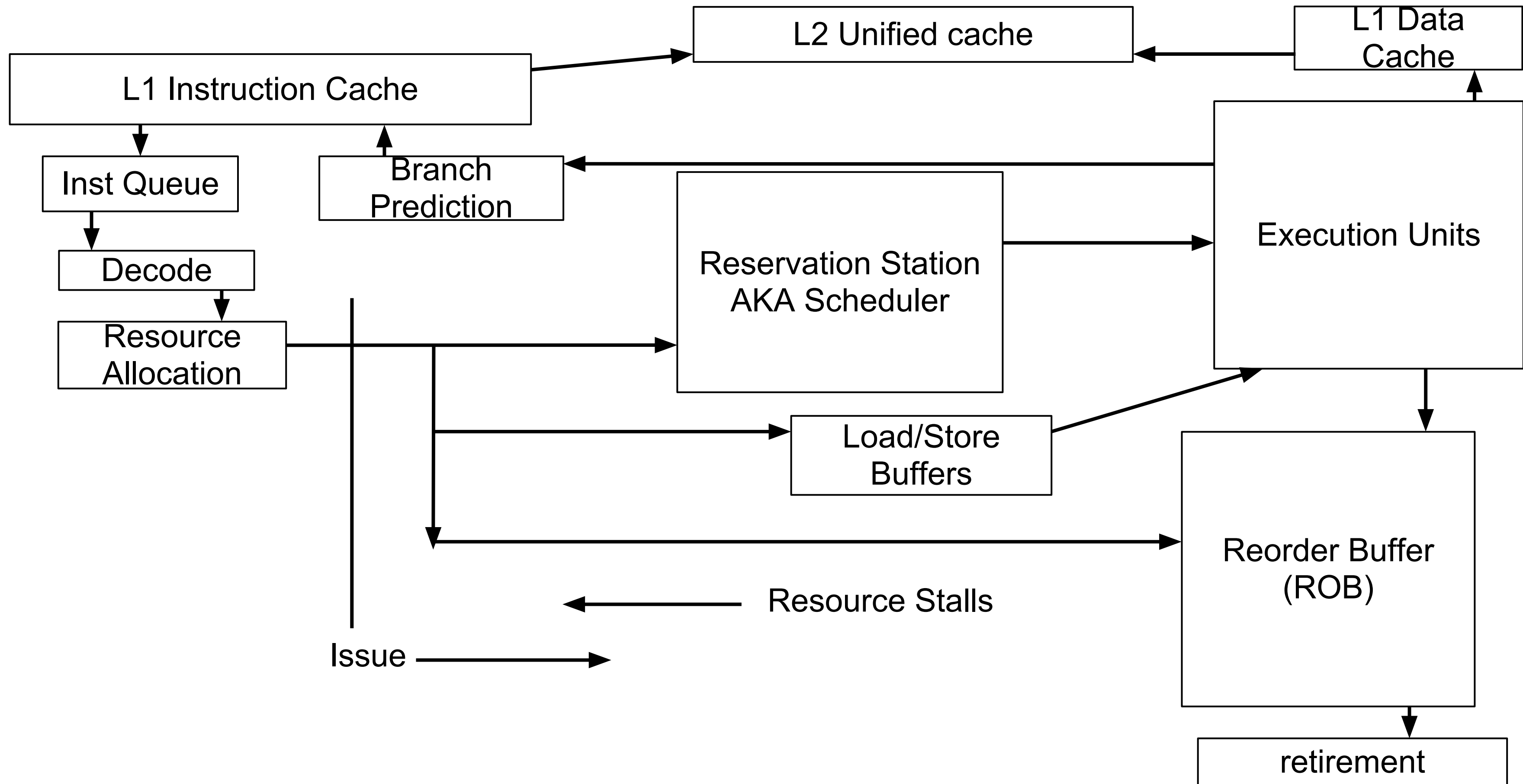
<u>Cycle accounting methodology works for both modes</u>
<u>But not always at finest granularities (source/asm)</u>
sampling IPs likely do not reproduce true IP distribution
causing differences and ratios evaluated for single instructions to be nonsense

Cycle Groups Form a Hierarchical Tree

- Cycles divide into halted and unhalted
- Unhalted cycles can be divided into "stalled" and "unstalled"
  - exact definition can vary
  - decomposition will always lead to same components on a given architecture
- Definition of cycles must be considered
  - reference cycles != core pipeline cycles
- Most activity is defined in core pipeline cycles
  - instruction latencies, on core cache latencies are in pipeline cycles
  - usually best to stick with that
- Halted cycles are better measured in reference cycles
  - Most processors drop the core frequency when in the halted state

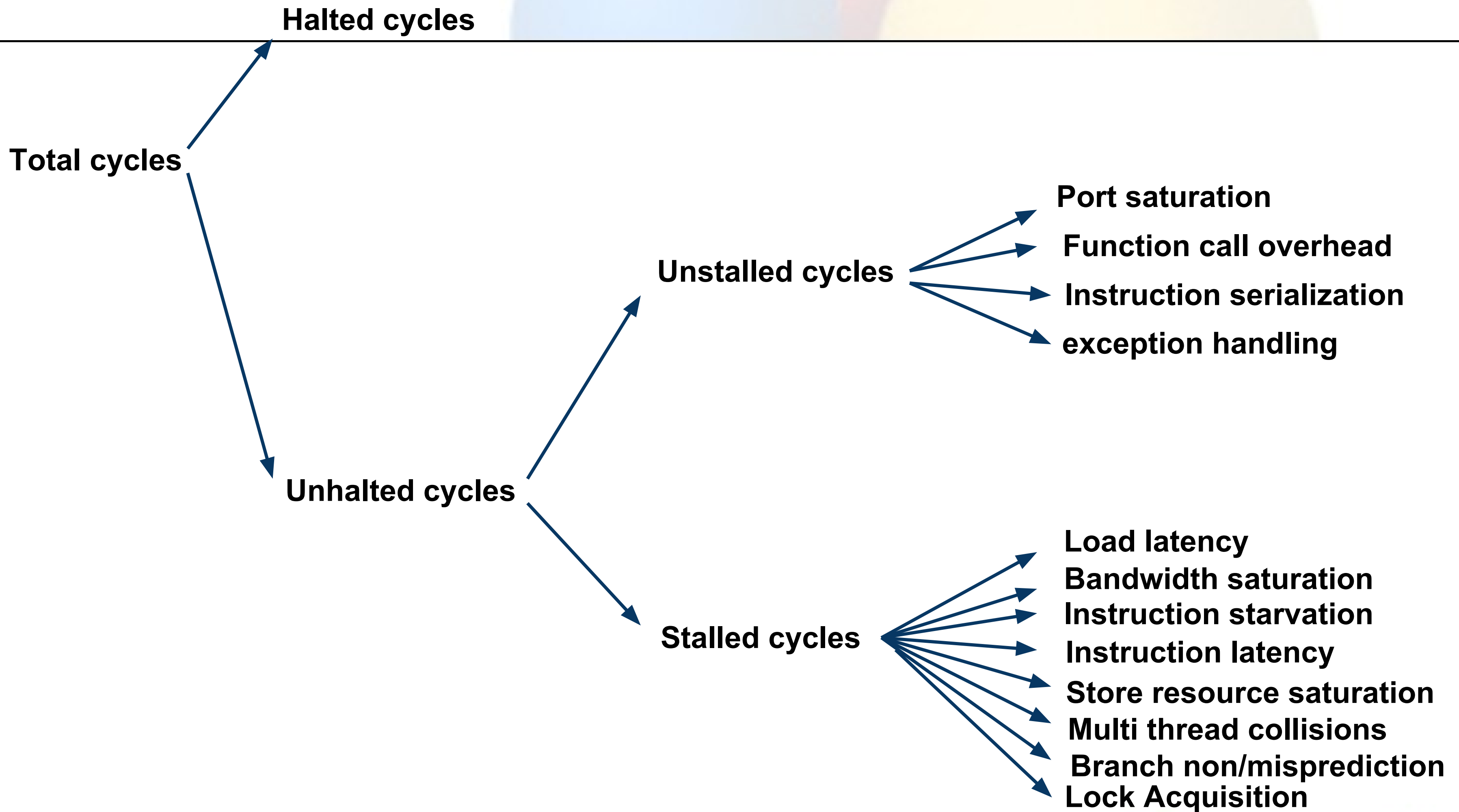
Google™

Artistic Out of Order Pipeline



Google

Cycle Accounting



Cycle Accounting on Westmere

Halted cycles*

Total cycles*

Unhalted cycles*

Unstalled cycles*

- Port saturation*
- Function call overhead**
- Instruction serialization***
- exception handling*

Stalled cycles*

- Load latency*
- Bandwidth saturation*
- Instruction starvation****
- Instruction latency**
- Store resource saturation*
- Multi thread collisions**
- Branch non/misprediction*
- Lock Acquisition**

* well covered on WSM

**** accurate count, inaccurate profile

** partially covered on WSM

*** not covered on WSM

Cycle Decomposition

- Stalled/unstalled cycles are decomposed as a sum of count(event)*cost(event)
    - this is effectively serializing execution
        - does not handle temporally overlapping stall conditions
    - requires that events have a well defined cost
- In some cases "covering" events can be used to estimate the upper limit of the total cost
    - Correcting for overlapping penalties
    - offcore_requests_outstanding:demand_reads:cmask=1
        - cycles with at least 1 offcore demand read (load) in flight
        - would count total cycles attributable to offcore load latency
        - Problem: "Demand" includes L1d HW prefetch
            - offcore_requests.demand_read/SUM(mem_load_retired:"offcore") > 1
            - you can't always count what you want
    - (Uops_issued:stall_cycles - resource_stalls:any) covers instruction starvation
      in the FE uop delivery, but differences of event counts make lousy profiles

Google™

Stalls

- Define stalls as "retirement stalls" = cycles with no retirement
- Stalls can be ~decomposed using other events into
  - Load latency
  - Memory bandwidth saturation
  - Instruction starvation
  - Instruction latency
  - Store resource saturation
  - Branch non/misprediction
  - Multi thread collisions (only for cores with shared pipelines)
  - Lock acquisition
  - and probably some others
- Top 3 tend to dominate HPC and Enterprise applications

Google™

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
      - data_cache_misses:l2_cache_miss
- "Generic" events count all sorts of things
  - generic l1_miss counts l2_hw_prefetch that hit L2 on WSM
- Decomposition is easiest with exclusive "hit" events
  - load_hit_here
    - accurate penalty/event can be determined
    - a difference must be used with "miss" events to define a penalty
      - <u>making profiling extremely inaccurate</u>
  - SUM(Count(event)*Penalty(event)) = load_latency
- Events must be "precise" to identify asm line
  - Skid can be into another function!
  - PEBS on Intel, IBS on AMD are examples

Google™

EX: Load Latency on Westmere

- Includes load accesses to caches and memory, load DTLB costs and blocked store forwarding (A lot of events!)
  - For example:
    - 6*mem_load_retired:l2_hit
    - 52*mem_load_retired:l3_unshared_hit (should be called l3_hit_no_snoop)
    - 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
    - 450*mem_uncore_retired:remote_hitm
    - 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
  - Latency can depend on specific configuration
    - Need to measure and verify with micro benchmarks
    - small penalties (< 10 cycles) have large fractional errors
- Tool makers need to know methodology so users don't
  - The analysis methodology should be in the data presentation
  - Predefine collection scripts
  - Data viewer/analysis should absorb the calculations

Instruction Starvation

- Uops_issued:stalls - resource_stalls:any on Intel
    - Uops_issued:stalls = Uops_issued:any:c=1:i=1
    - covering event
    - This difference cannot be used when profiling
- Decomposed on WSM as
    - 55*l2_rqsts:ifetch_miss
    - 8*l2_rqsts:ifetch_hit
    - 7*(itlb_misses:0x10 + itlb_misses:walk_completed) + itlb_misses:walk_cycles
    - 6*ild_stall:lcp
- Very large skid! (100 instructions)
- On SNB rs_events:empty_cycles counts cycles scheduler is completely empty
    - Dominated by ifetches from L3 and br mispredictions

Google™

Measuring Bandwidth Saturation

- Bandwidth saturation results in cacheline requests backing up in the queuing hardware.
- Measuring the cycles a code is BW limited involves measuring the cycles the queues have a "lot" of entries
- Measuring the BW itself is pretty much useless
  - the BW limits will depend on the number of threads using BW, where they are and where the lines are coming from
  - a multi dimensional surface
  - BW limitations occur when the thread is "close" to the appropriate spot on this surface
  - measuring queue occupancy is just easier
- Total queue occupancies can be measured on some processors
  - Many processors do not support this at this time

Google™

Understanding BW Measurement Data

- Counting BW limited cycles on WSM can be done with offcore_requests_outstanding:any:c=6
  - any request in the queue, cycles with at least 6 entries
  - empirically determined to work well
  - need to count everything, loads, stores and prefetches
- Use offcore_response:data_in:local_dram & offcore_response:data_in: remote_dram for NUMA
- Drawbacks: Counts transfers from L3
- Complications
  - BW limited execution can fire PEBS load events
    - for(i=0;i<len;i++)a[i] = b[addr[i]];   //gather operation
      - will fire PEBS load events
      - OOO exec allows many loads to be in flight simultaneously for such a loop
  - Ignore load latency cycles when execution is BW limited
    - this might get into the tool in the future
    - for now understand the issue / do the correction

Branch Misprediction is Complicated

- Many execution phases in a branch misprediction
  1. cycles pursuing wrong path
  2. cycles flushing pipeline (not in NHM/WSM/SNB)
  3. cycles fetching correct instructions (instruction starvation)
  4. cycles pushing correct instructions through FE to exec
- 1 and 3 have variable duration
  - example data dependent conditional branch where data must be fetched
- Wrong path ~ (uops_issued - uops_retired)/uop_issue_rate
- 4 has variable duration on SNB due to DSB (trace cache)

Google™

Unstalled Cycle Accounting

- Just because cycles are not stalled, does not mean they are effective
- Some issues that can result in poor performance with no stalls are
  - Port saturation: one port (loads) dispatches on >= 75% of cycles
  - Call overhead: several cycles worth of instructions to execute call and return. Inlining might be called for
  - Serial execution: limiting ILP to low values
  - Exception handling: like denormals
- This list is surely incomplete

Google™

Workload Characterization

Spreadsheet: data normalized to fixed time (75 sec)
gooda running on WSM

Google™

Rule of Threes for Enterprise Applications

An enterprise application's execution can be roughly described as:

- stalled, waiting for instructions 1/3 of the time (bit less)
- stalled waiting for data 1/3 of the time (bit more)
- executing/retiring 1/3 of the time @ 3 uops/cycle
  - IPC = 1
- Thus an in order core would increase execution time from 1 to 1 and 2/3 (~1.7)
  - wimpy cores are a bad idea

Google™

What is Gooda

- Open sourced PMU analysis tool
- Presents PMU event profiles of executing code
  - Most critical feature of any profiler is as a CPU sensitive asm editor
- Processes perf.data file created with "perf record"
- Intrinsically incorporates hierarchical generic cycle accounting tree methodology
  - Architecture specific events build the generic tree from the bottom up
- Automates the analysis and optimization methodology described in: https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/00_News/News_pages/2010/10-15_Meeting_David_Levinthal/10-15_Meeting_David_Levinthal.htm
  - There are several presentations, and lab exercises available through the link

Google™

Gooda

- Two component analyzer
  - Analyzer and Visualizer
- Gooda-analyzer reads perf.data files and creates .csv/JSON tables and dot/svg files for CFG
  - does cycle accounting and adds column control data to tables
  - tables stored in directory tree for portability
  - tables contain everything needed to support all display actions

Google™

Gooda Visualizer

- Gooda-visualizer
- Web based Gooda viewer reads .csv/JSON and generates tabular displays in html5 compatible browsers
- Measurements are converted to cycles by default (shown in green)
- Cycle tree is expanded through column expansions
- Displays include
  - events/process, expanding to show event/module/process
  - events/function, selecting a function opens source view tab
  - source view contains coupled displays of:
    - annotated asm, organized by basic blocks
    - annotated source
    - CFG
    - Function count based call graph
- All displays can be sorted by column header
- Top row shows total for the display

DEMO

Backup

Google™

Measuring Bandwidth Saturation

- Bandwidth saturation results in cacheline requests backing up in the queuing hardware.
- Measuring the cycles a code is BW limited involves measuring the cycles the queues have a "lot" of entries
- Measuring the BW itself is pretty much useless
  - the BW limits will depend on the number of threads using BW, where they are and where the lines are coming from
  - a multi dimensional surface
  - BW limitations occur when the thread is "close" to the appropriate spot on this surface
  - measuring queue occupancy is just easier
- Total queue occupancies can be measured on some processors
  - Many processors do not support this at this time

Google™

Understanding BW Measurement Data

- **Counting BW limited cycles on WSM can be done with offcore_requests_outstanding:any:c=6**
  - **any request in the queue, cycles with at least 6 entries**
  - **empirically determined to work well**
  - **need to count everything, loads, stores and prefetches**
- **Use offcore_response:data_in:local_dram & offcore_response: data_in:remote_dram for NUMA**
- **Drawbacks:** **Counts transfers from L3**
- **Complications**
  - **BW limited execution can fire PEBS load events**
    - `for(i=0;i<len;i++)a[i] = b[addr[i]];     //gather operation`
      - **will fire PEBS load events**
      - **OOO exec allows many loads to be in flight simultaneously for such a loop**
  - **Ignore load latency cycles when execution is BW limited**
    - **this might get into the tool in the future**
    - **for now understand the issue / do the correction**

Store Resource Saturation

- **Stores retire before the data is in a cacheline**
  - **store buffers hold the data until the line is in L1D**
- **Stores must commit the data to visibility by other threads in order**
  - **Writes to cache arrays MUST be in order**
  - **Can cause store buffers to all be in use**
  - **stalling the FE from issuing more uops**
- **resource_stalls:store counts this condition**
- **informational data can be found from RFO events**
  - **offcore_requests_outstanding:any_rfo:c=1**
  - **offcore_requests_outstanding:demand_rfo:c=1**
    - **includes l1d HW prefetches**
  - **offcore_response:demand_rfo:local_dram etc**
- DTLB_misses:walk_cycles-dtlb_load_misses:walk_cycles

Google™

Instruction Latency

- Chained instructions decrease ILP and can cause instruction latency to result is stalls
  - a = b + c + d + e + f + g + h + i;
    - 3 cycles of stalls/add for FP data if evaluated left to right
- This will need to be identified through asm analysis
  - LBR mini traces run through a pipeline simulator
    - assume all cache access are L1 hits
- Exception: Arith:cycles_div_busy counts cycles for non pipelined divide and sqrt

Google™

Branch Non/Misprediction

- Mis/non predict detection cost can be determined as
  (uops_issued:any - uops_retired:slots)/
  (uops_issued:any/uops_issued:any:c=1)
  - non retired uops/uop_issue rate
- L1I resident kernel measured non predicted branches and mispredicted branches total cost
  - baclears:clear (6 cycle minimum penalty)
  - br_misp_retired:all_branches (6 cycle minimum penalty)
  - This is certainly dominated by pipeline reload

Google™

Multithread Collisions on WSM

- Only an issue on machines with HT enabled
- Pipeline collisions can occur at FE, Exec and retirement
- Additional effects can occur in caches due to mutual evictions
  - not included here as I know of no way to measure this directly
    - requires difference of HT on - HT off
      - load latency
      - instruction starvation
      - BW saturation
      - store resource saturation

Google™

Multithread Collisions on WSM

- FE collisions
  - control of (in order) FE alternates between threads when both have instructions/uops
  - Dispatch collisions can occur when threads can dispatch
    - can be approximated as a 0.5 * product of probabilities
    - resource_stalls are likely highly correlated between threads
- Execution collisions
  - likely dominated by load port collisions
    - can be approximated as a 0.5 * product of probabilities
- Retirement collisions
  - control of (in order) retirment alternates between threads when both have instructions/uops
    - can be approximated as a 0.5 * product of probabilities

Google™

Multithread Collisions on WSM

- best execution: the same binary runs on both threads
- This makes it possible to evaluate both probabilities with one thread, as they are the same
- FE
  - FE Prob ~ uops_issued:any:c=1/(cpu_clk_unhalted - resource_stalls.any)
  - FE collision ~ (uops_issued:any:c=1/(cpu_clk_unhalted - resource_stalls.any))**2
- Exec collisions are usually largest for loads
  - collisions delay one load by 1 cycle, increasing the latency
  - difficult to estimate for ports 1,3,5 (ALU)
  - Exec load Prob ~ Uops_executed:port2_core/(2*cpu_clk_unhalted)
    - the factor of 2 is there because the event counts both threads
  - Exec collision ~ (Uops_executed:port2_core/(2*cpu_clk_unhalted))**2
- Retirement
  - collisions ~ (uops_retired:any:c=1/cpu_clk_unhalted)**2

Google™

Lock Acquisition

- this is not so great on WSM
- use the load latency event to identify very long latency loads
- these are usually highly contested locks
- this is can be checked against the disassembly

Google™

Port Saturation

- When one port is dispatching uops on almost every cycle, that will define a lower limit to the execution
- no optimization is possible unless the uops on the saturated port are reduced
- For example: A large number of distinct loops are created due explicit source (F90 style array notation) or by compiler loop distribution. This results in the same data being reloaded in every loop and port2 is saturated.
- Solution is to merge the loops and keep the data in a register

Google™

Port Saturation on WSM

- Memory ports count for both threads (ports 2,3,4)
- Saturation is determined by the maximum of:
  - Uops_executed:port0/cpu_clk_unhalted
  - Uops_executed:port1/cpu_clk_unhalted
  - Uops_executed:port2_core/cpu_clk_unhalted
  - Uops_executed:port3_core/cpu_clk_unhalted
  - Uops_executed:port4_core/cpu_clk_unhalted
  - Uops_executed:port5/cpu_clk_unhalted

  being above ~75%

  only way to get code to go faster is to reduce pressure on the port
- Port2 is usually the culprit
  - for an optimized dense matrix multiply ports 0 and 1 should be the constraint

Google™

Function Call Overhead

- Function calls can require several cycles of instructions just for set up (loading arguments to stacks or registers, call + trampolines) and tear down (restoring state, return)
  - assuming a 3 cycle penalty is probably reasonable
- There can be an additional penalty caused by missed compiler optimizations due to the compiler not knowing what the function is doing
  - loads cannot be hoisted above function calls for example
- cost ~ 3*br_inst_retired:near_call
    - WSM, suffers from shaddowing
- Even better to use LBR's filtered on return and sampled with br_inst_retired:near_return
  - get source and target
  - 16 measures/ sample

Google™

Exception Handling

- Exceptions handled by the microcode sequencer result in a large flow of uops through the pipeline.
- You are not stalled, but you are not making progress.
- Classic example might be handling denormals
- cost ~ uops_decoded:ms_cycles_active/cpu_clk_unhalted
  - this also results in an anomolous value for
    uops_retired:any/inst_retired:any
    - which may have less skid
  - Also counts repmov and sincos

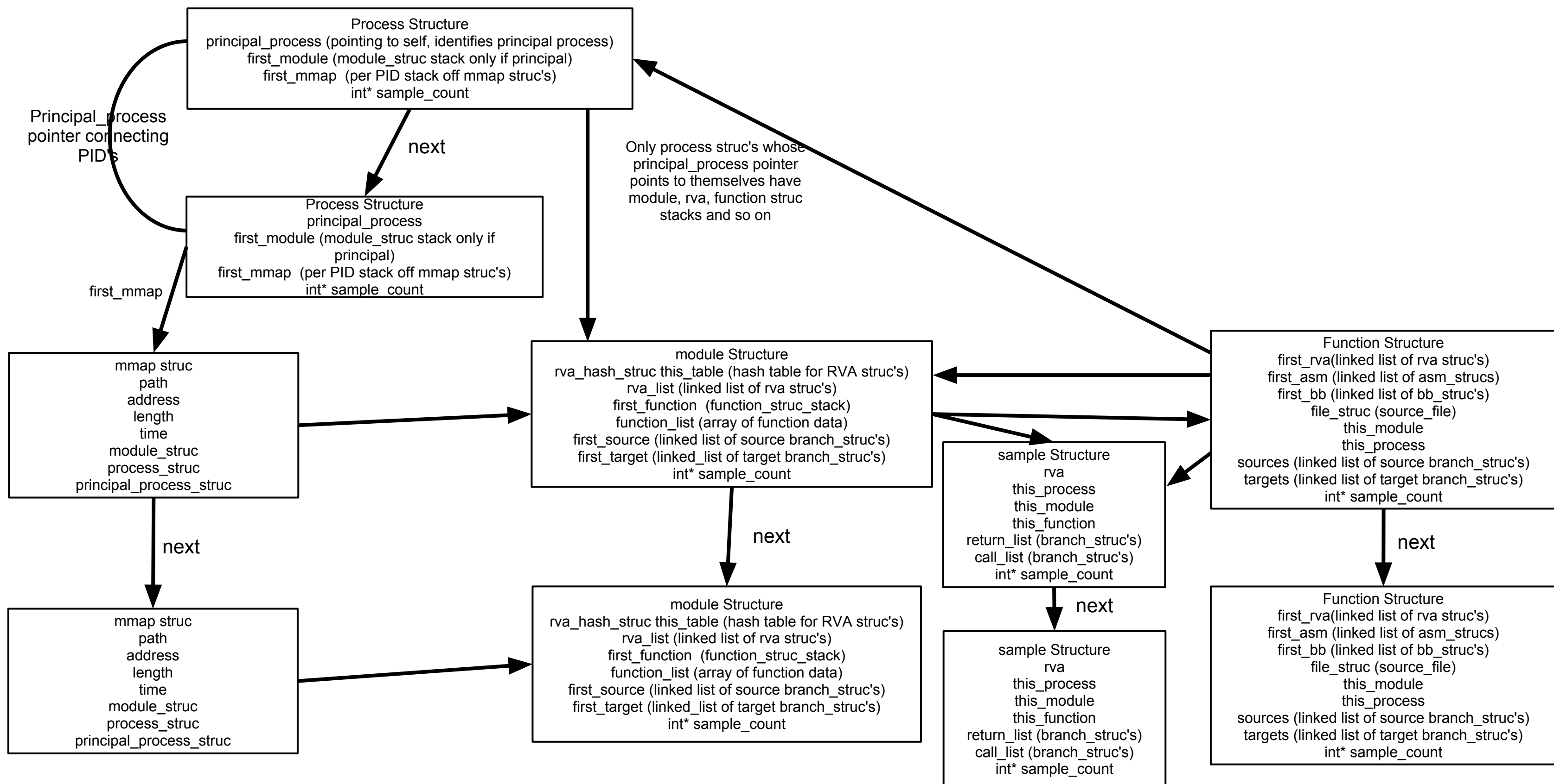
Google™

Instruction Serialization

- Dependencies between instructions can result in low instruction level parallelism (ILP)
- EX: a = b+c+d+e+f+g+h+i;
  - ANSI requires this be evaluated left to right creating dependencies
  - for FP operands this results in stalls and low ILP
  - recoding as   a = ((b+c)+(d+e))+((f+g)+(h+1));
    breaks the dependency
- At this time there are no HW events to identify this
- It will require static analysis or a pipeline simulator

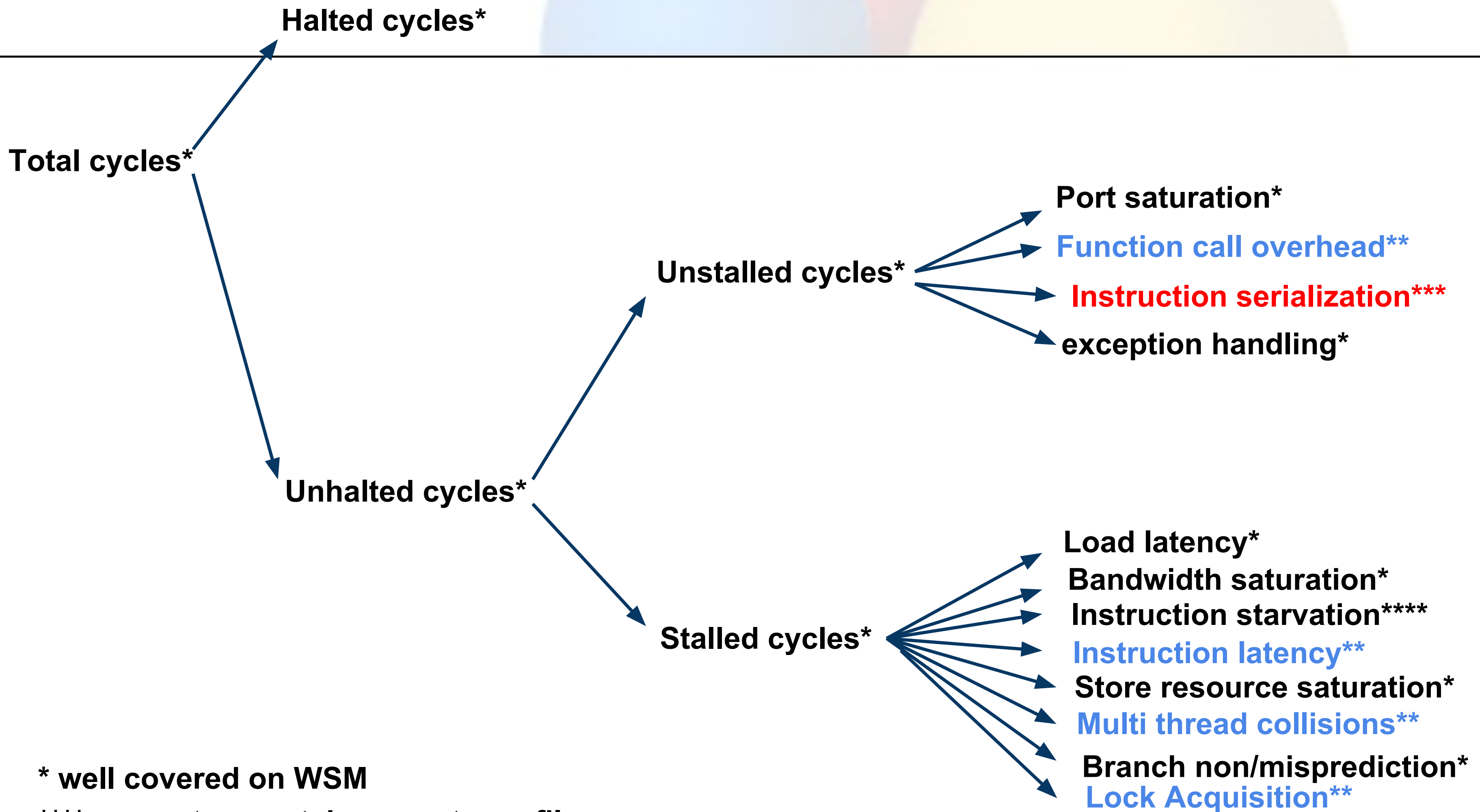
Google™

Greatly Simplified Gooda Analyzer Data Structures



Google™

Cycle Accounting on Westmere



Google