Generic Cycle Accounting GOODA

Generic Optimization Data Analyzer

What is Gooda

- **Open sourced PMU analysis tool**
- **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

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**

Hardware Event Collection
Two modes: Counting and Interupt

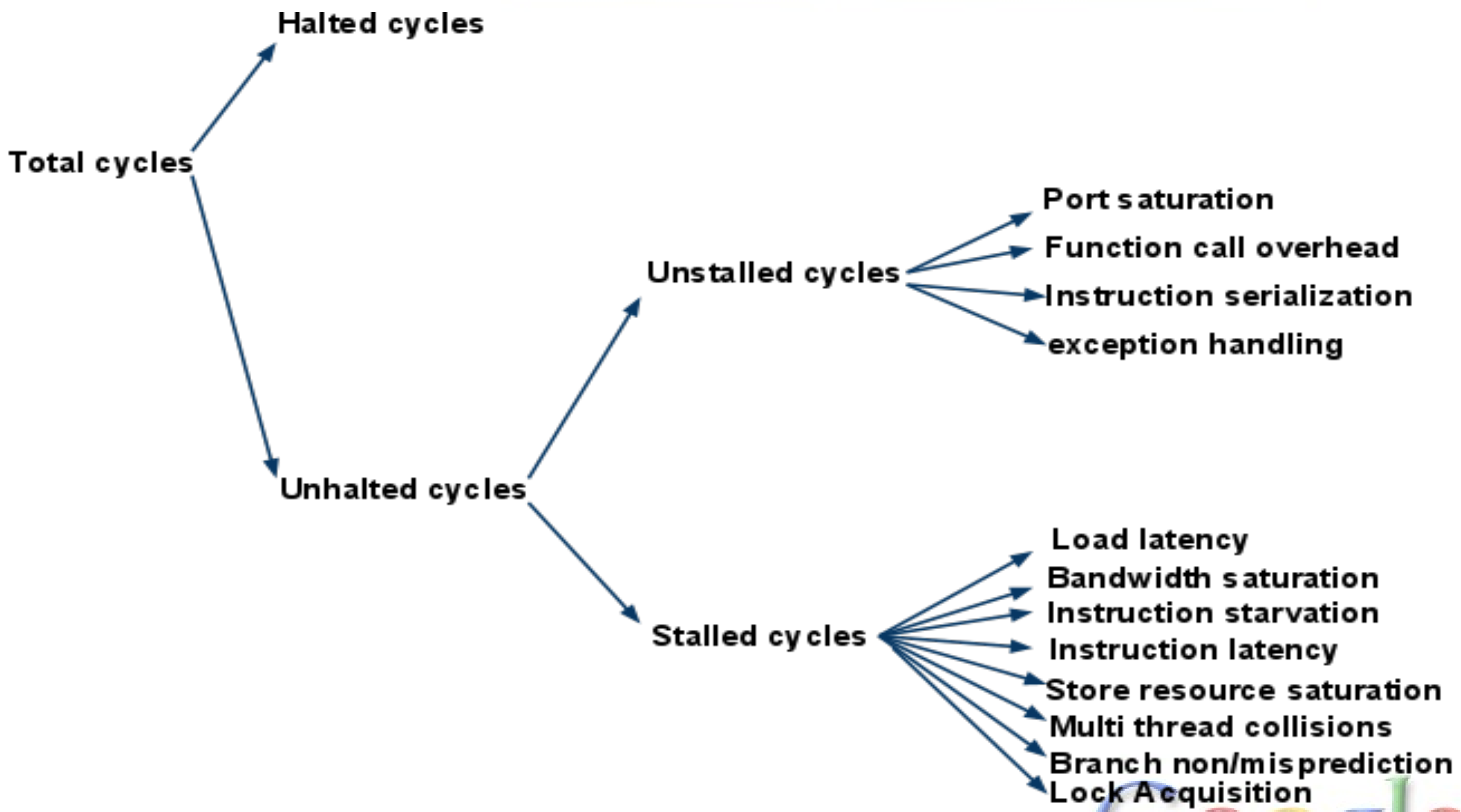
- **Counting mode: Workload Characterization**
  - program counter to count desired event
  - initialize to zero
  - read value of counter after a fixed time
  - Good for giving feedback to processor architects
  - Most events are targetted for this
    - cache hit rates/MESI state, Intel "matrix event"
- **Interupt 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

Cycle accounting methodology works for both

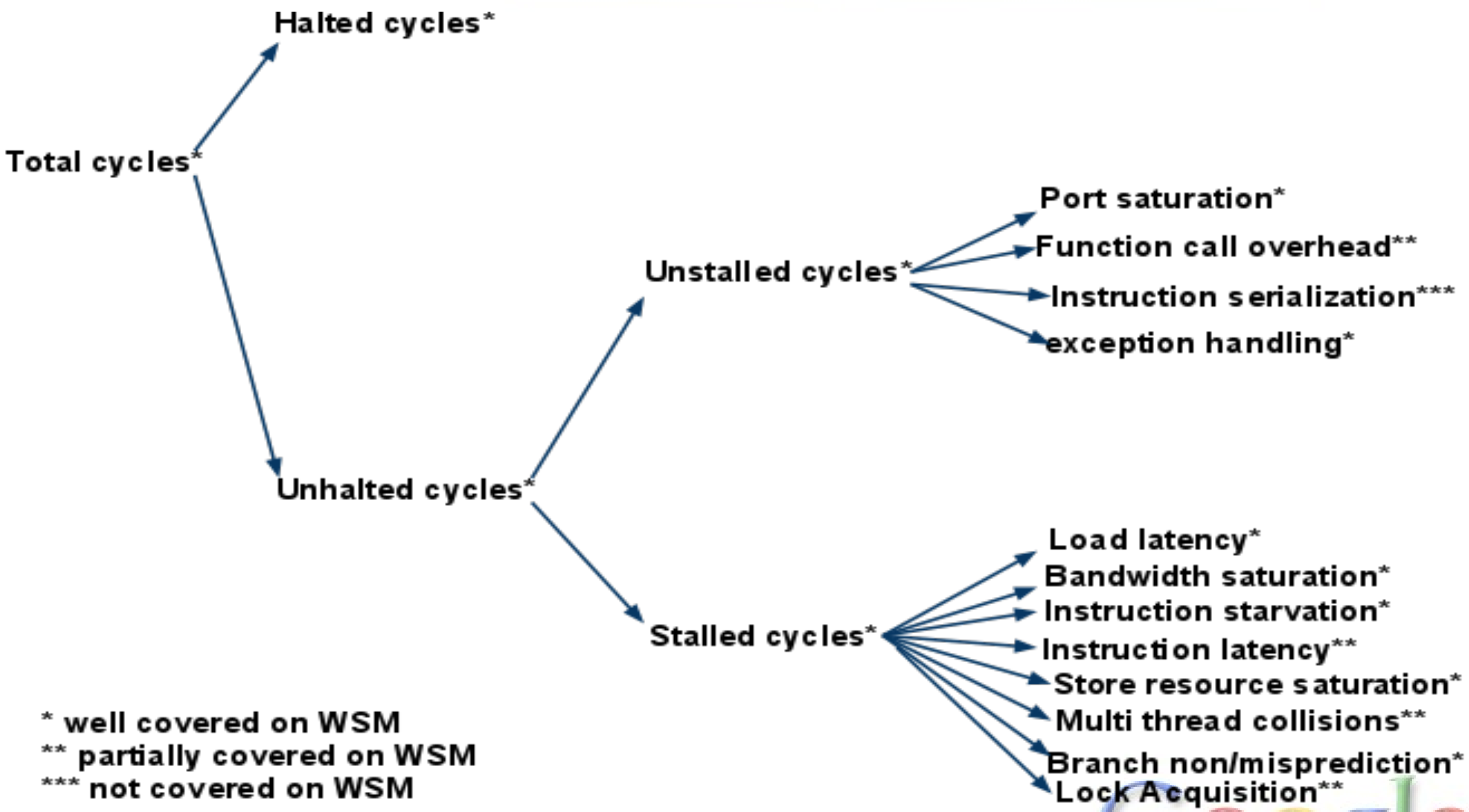
Cycle groups form a hierarchical tree

- 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 in the halted state

Cycle Accounting



Cycle Accounting on Westmere



* well covered on WSM
** 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
  - (uops_issued - uops_retired)/(uop_issue_rate) covers costs of all branch mispredict/non predicts
    - and this appears to work quite well

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

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**
            - **making profiling extremely inaccurate**
    - **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**

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
    - 45*mem_load_retired:l3_unshared_hit (should be called l3_hit_no_snoop)
    - 75*(mem_load_retired:other_core_L2_hit_hitm - mem_uncore_retired:local_hitm)
    - 85*mem_uncore_retired:local_hitm
    - 225*mem_uncore_retired:local_dram_and_remote_cache_hit
    - 400*mem_uncore_retired:remote_dram
    - 400*mem_uncore_retired:remote_hitm
    - 225*mem_uncore_retired:other_llc_miss
    - 7*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
- Tool makers need to know methodology so users don't
  - Collectors should enable all features the HW supports
    - The analysis methodology should be in the data presentation
  - Predefine collection scripts
  - Data viewer should absorb the calculations

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**

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**

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**

Gooda

- Two component analyzer
- Gooda reads perf.data files and creates .csv/JSON files
- Web based Gooda viewer reads .csv/JSON and applies cycle accounting analysis to generate tabular displays
- Measurements are converted to cycles by default
- Tree is expanded through column expansions
- Columns are ordered by importance in cycles

DEMO

backup

Instruction Starvation

- **Uops_issued:stalls - resource_stalls:any on Intel**
    - **Uops_issued:stalls = Uops_issued:any:c=1:i=1**
    - **covering event**
- **Decomposed on WSM as**
    - **48*l2_rqsts:ifetch_miss**
    - **7*l2_rqsts:ifetch_hit**
    - **7*itlb_misses:walk_completed + itlb_misses:walk_cycles**
    - **6*lld_stall:lcp**

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

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**

Branch non/misprediction

- **total 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**
- **decomposed into non predicted branches and mispredicted branches as**
  - **baclears:clear (6 cycle minimum penalty)**
  - **br_misp_retired:all_branches (6 cycle minimum penalty)**

Multi thread 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**

Multi thread 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**

Multi thread 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**

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

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**

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**
- **Port2 is usually the culprit**
  - **for an optimized dense matrix multiply ports 0 and 1 should be the constraint**

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

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**

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**