HW based Performance Analysis Methodology

D. Levinthal

Overview

- Objectives
- Cycle Accounting
- Measuring performance limitations
  - Load latency
  - Memory bandwidth
  - Instruction starvation
  - Branch misprediction
  - Branch execution (calls and indirect branches)
  - Store resource limitations
  - Profiling and active binary size
- Microkernels and documentation
- Existing tools
- Actually running an application and collecting data
  - And common pitfalls

Performance analysis objectives

- Most performance analysis infrastructure developments driven by desire to improve code performance
  - Ability to identify largest easily fixed issue
  - Reduce execution cycle counts
- Performance event rates/instruction retired can guide processor design in simulators running traces
- Understand utilization at scale to identify deficiencies and guide requirements for next generation systems
  - CPU, mobo, addin cards

What matters is cycles

- Performance events count occurrences of signals in the processor
- These need to be turned into a single figure of merit for relative importance evaluation
- Cycles are the figure of merit that users care about
- Thus performance events that can be turned into cycle counts are the most valuable
  - Cache/dram hits can be multiplied by latencies
  - Cache misses cannot and ultimately tell you little
    - Profiling differences just does not work at all
- Dominant loss of performance on servers is due to stalled cycles
  - Cycles with zero instructions (issued, executed or retired)

IPC	0.7428
instructions/unstalled_cycle	2.776
uops(fused)/unstalled_cycle	3.000

What matters is cycles

- SUM of event_counts*penalties is essentially serializing “bottleneck model”
- OOO execution will result in overlapping issues
  - Penalties can be greater than total cycles
  - Need to see all issues to identify which to attack
  - Use multiple metrics for consistency checks when possible
- Forcing a sum to 1.0 is intrinsically misleading
  - And just a bad idea
- **Any hw event based micro architectural performance decomposition methodology should consider achieving a 10% accuracy in its components a triumph**
  - **Don’t expect better**

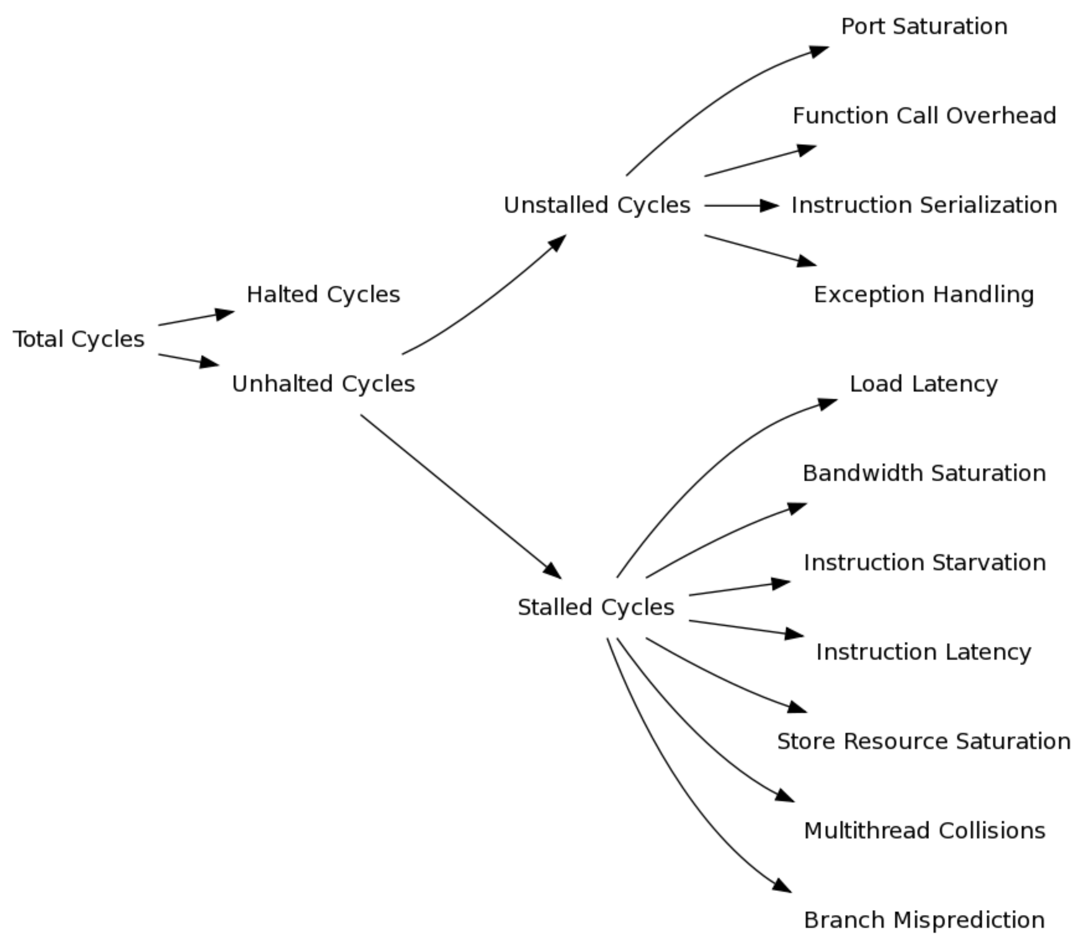
Identifying HW performance limitations in the Hardware

- Execution Characterization
  - Simple counting mode analysis based on well defined and validated performance events
  - Assign performance cost to architectural components
    - Latencies
    - resource saturation
    - Execution path discontinuities
- Can be done for a single application
- Or an entire data center
  - Statistical distributions

Identifying SW performance limitations in a single application

- Execution time is the figure of merit
- Improving that is made easier by precise knowledge of
  - Where in the code the time is spent
  - Why it is taking so long
- Standard approach is profiling code versus many performance events
- Event counters programmed to interrupt on overflow
  - ~ 1-4 khz interrupt rate/core results in a few % perf degradation
  - Interupt data collected on disk
  - post process data with binary, source and debug symbol file

Cycle accounting and the Gooda cycle tree



The leafs are constructed from bottom up sums, all using consistent units (cycles)
Consequently, drilling down to lower dominant components will always show consistent relative values

Generic Cycle accounting

pipeline status	issue	estimating event description
stall	load latency	number of cycles with pipeline stalls at execution stage while memory subsystem has an outstanding load
stall	bandwidth saturation	number of cycles where # of outstanding memory requests per core (not per thread) is near the minimum number measured at steady state with any number of triads running
stall	instruction starvation	number of cycles waiting at the execution stage for long latency code delivery (> 40 cycles). Also requiring the Scheduler/ROB being nearly empty
stall	instruction latency	Cycle count for divides, SQRT, transcendentals and other long latency execution streams not covered otherwise.
stall	store resource saturation	number of cycles all store buffers in use
stall	branch misprediction	total cost in cycles of branch mispredictions, including all time spent fetching/executing the wrong path, plus extra cycles needed to recover and resume progress on the correct path
stall	multithread collision	
unstalled	port saturation	cycles where single execution port getting uops nearly every cycle
unstalled	function call overhead	enable an estimated cost in cycles for calls including trampolines, stack manipulations, etc. This is done by using the metric instructions_retired/call_retired
unstalled	instruction serialization	cycles only executing chained dependent multi cycle latency uops
unstalled	exception handling	cycles spent issuing uops from a microcode scheduler for exception handling

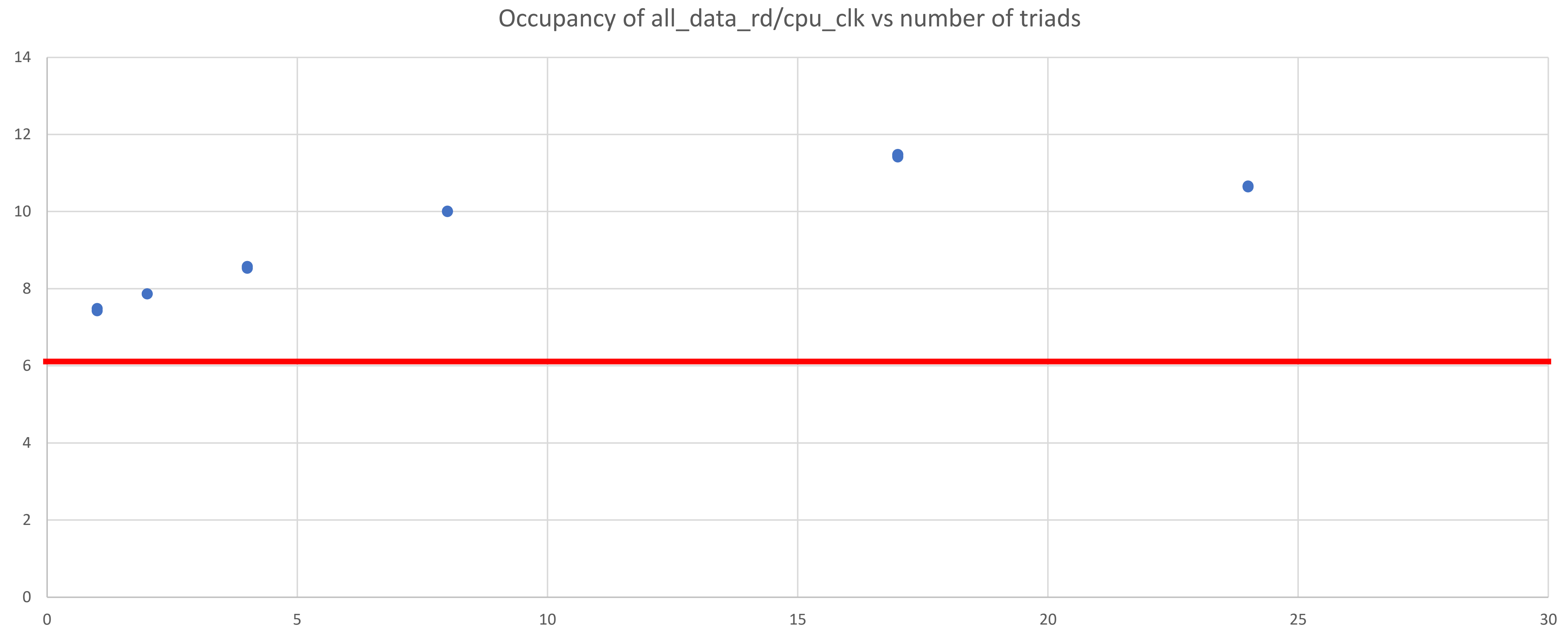
Load latency

```
<metric name="HIGH_LEVEL_CYCLE load latency penalty">
        <event alias="a">CPU_CLK_UNHALTED.REF_TSC</event>
        <event alias="b">MEM_LOAD_L3_MISS_RETIRED.LOCAL_DRAM</event>
        <event alias="c">MEM_LOAD_L3_MISS_RETIRED.REMOTE_DRAM</event>
        <event alias="d">MEM_LOAD_L3_MISS_RETIRED.REMOTE_HITM</event>
        <event alias="e">MEM_LOAD_L3_MISS_RETIRED.REMOTE_FWD</event>
        <event alias="f">MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM</event>
        <event alias="g">MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT</event>
        <event alias="l">MEM_LOAD_L3_HIT_RETIRED.XSNP_MISS</event>
        <event alias="h">MEM_LOAD_RETIRED.L3_HIT</event>
        <event alias="i">DTLB_LOAD_MISSES.WALK_COMPLETED</event>
        <event alias="j">DTLB_LOAD_MISSES.WALK_ACTIVE</event>
        <event alias="k">CPU_CLK_UNHALTED.THREAD</event>
        <formula>((167*b+368*c+335*(d+e))/a+(189*(f+g+l)+75*h+3.5*i+j)/k)</formula>
</metric>
```

Memory Bandwidth

- Memory bandwidth is a problem only when you get close to the limit
- Bandwidth is a time average
  - If the time scale of the saturation is too small you cannot detect the saturation by measuring BW
  - Ex: 10% of every 1 msec is running memcpy
  - Measuring BW would indicate you are using only 10% of available BW
  - No problem
- One needs to measure on each cycle if the BW is saturated
- This is done by monitoring cacheline request queue occupancies
- Punchline: >= 6 offcore_requests in the offcore queue is a saturated cycle

Memory BW saturation on CLX using parallel triads BW Total BW varies from 19GB/sec to 95



This approach can identify short durations BW saturations
But also even if only a single core is BW limited

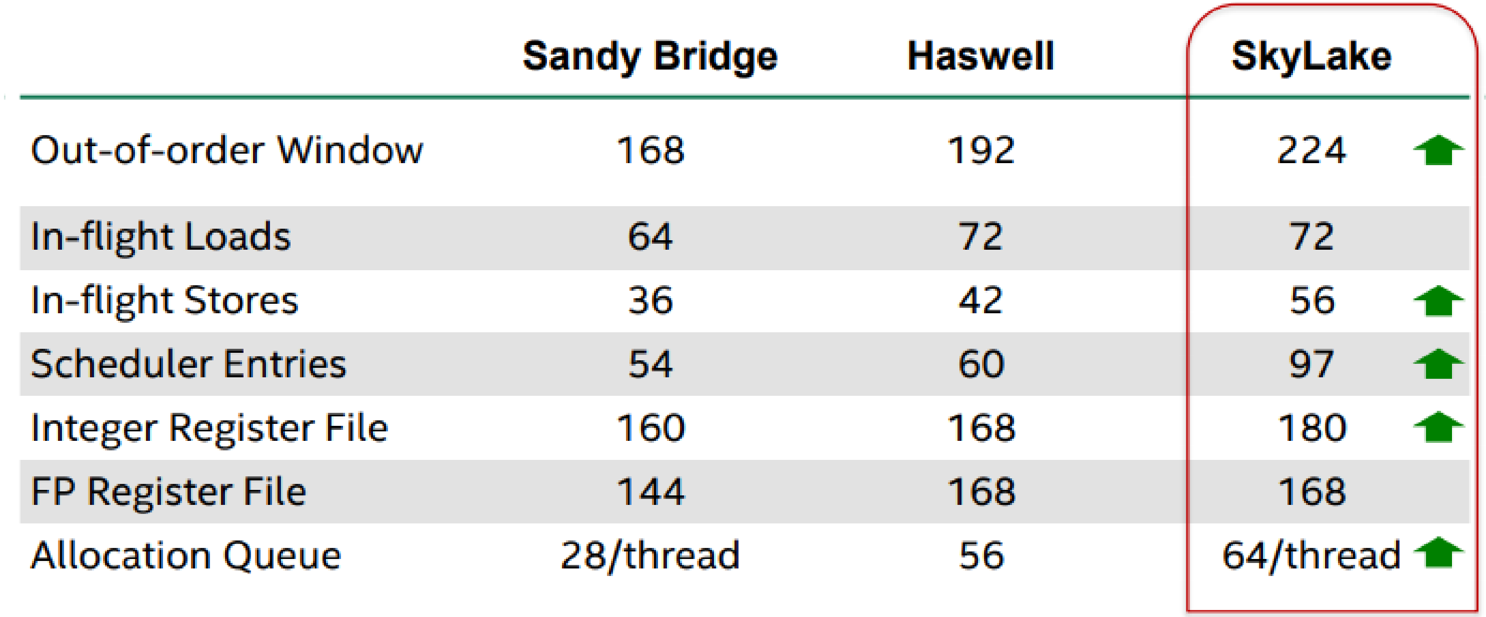
Branch misprediction

```
<metric name="HIGH_LEVEL_CYCLE branch misprediction penalty">
    <event alias="a">CPU_CLK_UNHALTED.THREAD</event>
    <event alias="b">BACLEARS.ANY</event>
    <event alias="c">BR_MISP_RETIRED.ALL_BRANCHES</event>
    <event alias="d">INT_MISC.RECOVERY_CYCLES</event>
    <event alias="e">UOPS_ISSUED.ANY</event>
    <event alias="f">UOPS_RETIRED.RETIRE_SLOTS</event>
    <event alias="g">UOPS_ISSUED.ANY:c1</event>
    <formula>((6*(b+c)+d+((e-f)*g/e))/a)</formula>
</metric>
```

Term (e-f)*g/e is cycles spent issuing uops for mispredicted path

Instruction starvation/Frontend stalls

- OOO structures are huge on modern processors
  - FE not delivering uops in a few slots or even cycles just doesn’t matter on servers
- If the scheduler has uops there is no problem
- So, only Ifetch from L3 is a real performance impact



	Sandy Bridge	Haswell	SkyLake
Out-of-order Window	168	192	224 ⬆
In-flight Loads	64	72	72
In-flight Stores	36	42	56 ⬆
Scheduler Entries	54	60	97 ⬆
Integer Register File	160	168	180 ⬆
FP Register File	144	168	168
Allocation Queue	28/thread	56	64/thread ⬆

Instruction Starvation/FE stalls

```
<metric name="HIGH_LEVEL_CYCLE instruction starvation cycle fraction">
    <event alias="b">CPU_CLK_UNHALTED.THREAD</event>
    <event alias="d">RS_EVENTS.EMPTY_CYCLES</event>
    <formula>(d/b)</formula>
</metric>

<metric name="metric_cycle L3_ifetch cycle fraction">
    <event alias="b">CPU_CLK_UNHALTED.THREAD</event>
    <event alias="d">FRONTEND_RETIRED.L2_MISS</event>
    <formula>65*(d/b)</formula>
</metric>
```

Store resource limitations

- There are a lot of store buffers (35-60) but servers still run out due to long latency of RFO from dram
  - Stores MUST finish in order
- FE blocks on first store after running out of store buffers

```
<metric name="HIGH_LEVEL_CYCLE no store buffer cycle fraction">
            <event alias="b">CPU_CLK_UNHALTED.THREAD</event>
            <event alias="d">RESOURCE_STALLS.SB</event>
            <formula>(d/b)</formula>
      </metric>
```

Branch misprediction

- **total cost in cycles of branch mispredictions, including all time spent fetching/executing the wrong path, plus extra cycles needed to recover and resume progress on the correct path**

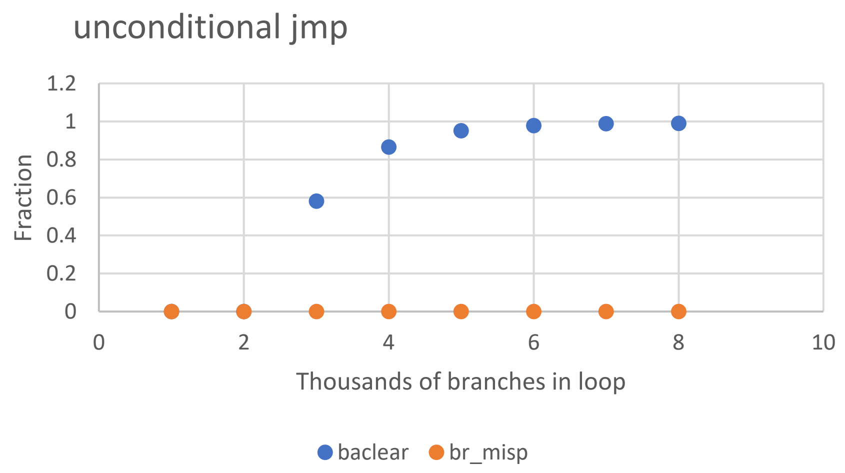
```
<metric name="HIGH_LEVEL_CYCLE branch misprediction penalty">
        <event alias="a">CPU_CLK_UNHALTED.THREAD</event>
        <event alias="b">BACLEARS.ANY</event>
        <event alias="c">BR_MISP_RETIRED.ALL_BRANCHES</event>
        <event alias="d">INT_MISC.RECOVERY_CYCLES</event>
        <event alias="e">UOPS_ISSUED.ANY</event>
        <event alias="f">UOPS_RETIRED.RETIRE_SLOTS</event>
        <event alias="g">UOPS_ISSUED.ANY:c1</event>
        <formula>((6*(b+c)+d+((e-f)*g/e))/a)</formula>
</metric>
```

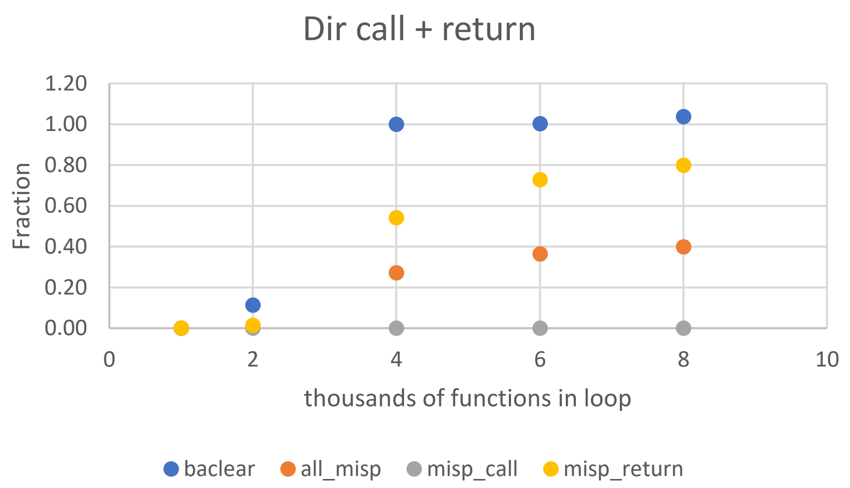
UOPS_ISSUED.ANY- UOPS_RETIRED.RETIRE_SLOTS is the mispredicted path length

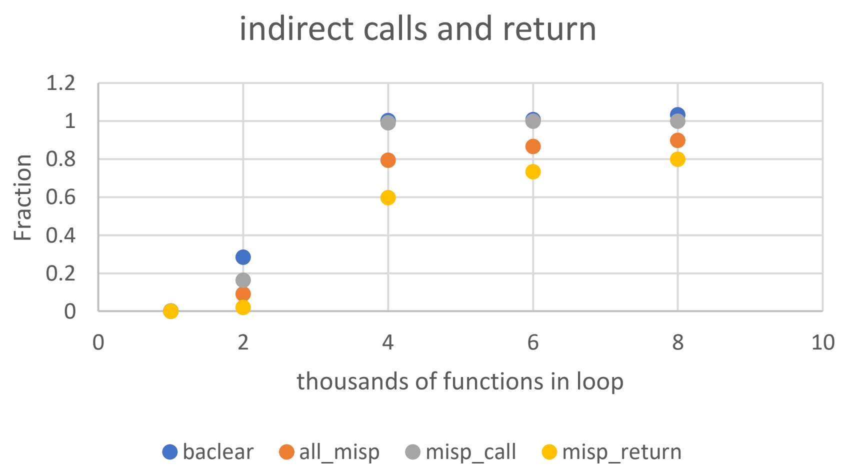
UOPS_ISSUED.ANY:c1/ UOPS_ISSUED.ANY = cycles/issued uop

Branch Target Buffer and indirect branch misprediction

- Indirect branches can mispredict the target if the BTB is overwhelmed by the sheer number of branches in a large binary







Branch Target Buffer and indirect branch misprediction

- Indirect branches can cause a high fraction of branch mispredictions and L3 instruction fetching (https://research.google/pubs/pub48320/)
- The data just shown would suggest the use of a single common BTB.
- There might be benefit in having separate BTBs for branches that mispredict if there is no BTB target
  - Ie indirect branches and returns

Calling convention issues/ call overhead

- X86 calling conventions require arguments be put in particular registers
  - This can require instructions to push current values to the stack and replace them with call arguments
  - Calling a function in a shared object/DLL will result in trampoline branches
  - On arriving in the function, registers must be made available for the functions work. More registers pushed to the stack
  - Refactoring results in function argument validity being tested after entry
    - If this fails all the above work must be undone, a branch to recovery code and a return
- A call can easily result in an overhead of 10 -15 instructions
- Small functions can be costly
  - measured by instructions_retired.any/br_inst_retired.any

Deep call stacks

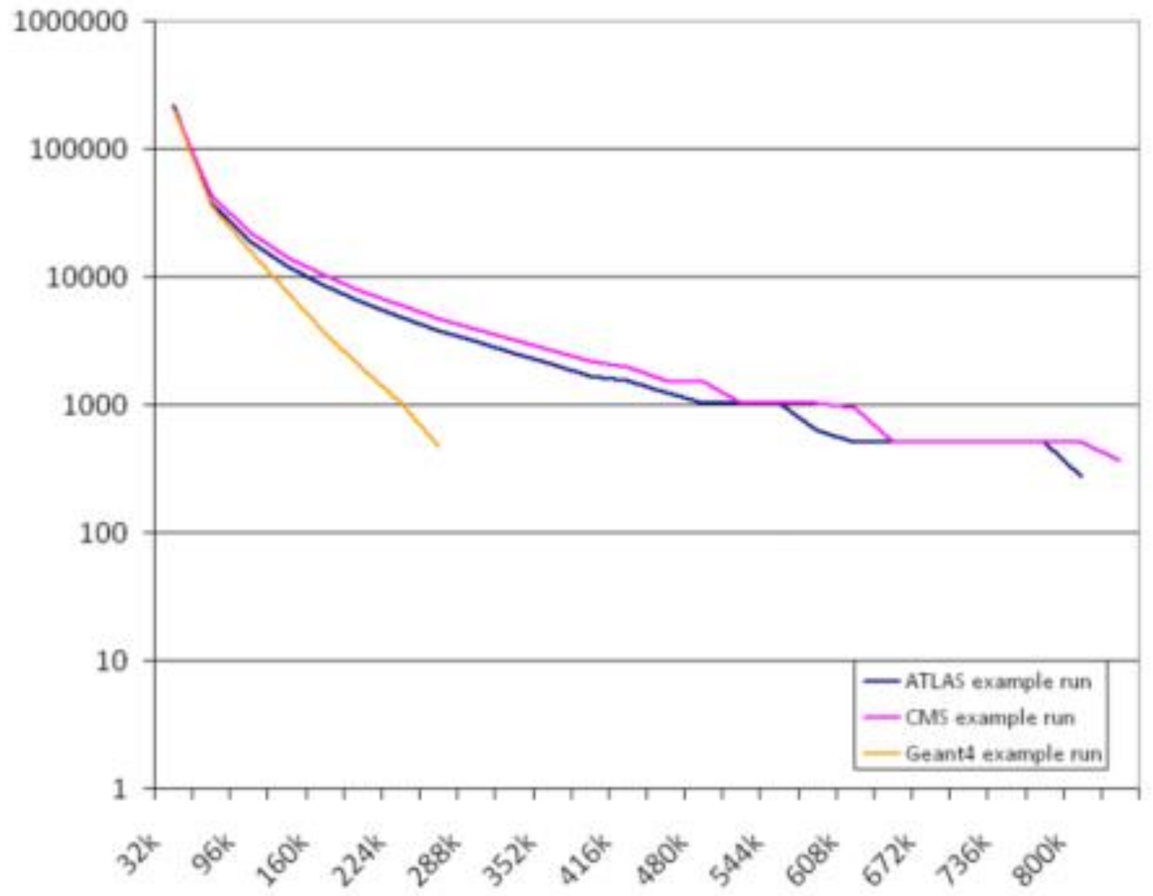
- Return instructions are indirect branches
- Deep call stacks can result in
  - mispredicted returns
  - Larger expense in restoring state
- Can be evaluated with br_misp_retired.near_return
  - If it is available

Profiling for HW evaluation

- Counter overflow driven interrupt sampling can be done with all the events discussed.
- However, server codes tend to have large active binary cores
  - A 100 MB binary can easily have 5 MB of active core
  - To sum to 75% of cycle interrupt samples likely requires 500-1000 functions
  - Hottest functions are likely malloc and free
    - Recall the google “data center tax”
- Sampling on instructions retired (or better LBRs) can yield basic block execution count
- Easier is simply counting instruction cacheline hit frequency
  - Yielding active binary size

Active binary size

How big are the CERN programs



Cacheline access frequency evaluated by sorting cachelines by their accesses

Thus a binary working set size measurement

Micro-kernels establish penalties and validate performance events

- To be useful for performance analysis, events must be able to be assigned a penalty
    - Otherwise it cannot be used to decompose cycle usage
- Do not assume you know what an event counts. You are almost certainly wrong
    - Documentation is always more vague than it appears
- Consider: 0xA0 L3_CACHE_RD L3 cache read
    - What does that include?
    - Ld, rfo, hw prefetch, sw prefetch, speculative + retired, secondary misses
    - If anything but retiring demand loads is included.
    - It is basically useless as it cannot be assigned a penalty

Micro-kernels establish penalties and validate performance events

- Data access latencies are established with linked list walker
  - While(count < limit){count++;p=*p;}
  - Need to worry about how to deal with hw prefetchers and paging
- Triads make good BW tests
  - w/wo RFO or streaming store
- Branch prediction, BTB capacity and ifetch are a bit trickier
- https://github.com/David-Levinthal/gooda/tree/master/gooda-analyzer/kernels
- Kernels produce precisely predicted counts for perf events
  - Validate that they count what they should
  - Harder to prove they do not count what they shouldn't
  - Run all events of interest against all kernels to look for errors

Micro-kernels establish penalties and validate performance events

```
levinth@ubuntu18-2:~/gooda/gooda-analyzer/kernels$ ls
```

arm	**conditional_jmp**	**pl_transition**
bb_exec	**dram_core_scan.sh**	**power**
blocked_load	**event_validation_instructions.txt**	**prefetch**
br_multi_core	**fourby.sh**	**README**
c2c_core_scan.sh	**gather**	**snoop_test**
c2c_core_scan_ht.sh	**inst_bw**	**sources_att**
call_tests	**latency_noarch**	**uncond_jmp_rdm**
callchain	**mem_bw**	**unconditional_jmp**
cond_br_str	**mem_latency**	**validate**

```
levinth@ubuntu18-2:~/gooda/gooda-analyzer/kernels$ ls arm
cond_br_str  matrix_mult  mem_bw  mem_latency  uncond_jmp_rdm
```

Gooda has lots of documentation

```
levinth@ubuntu18-2:~/gooda/gooda-analyzer/docs$ ls
cycle_accounting_and_gooda.pdf             Micro-architecture.pdf
CycleAccountingandPerformanceAnalysis.pdf  old_stuff
Driving_the_Gooda_visualizer.pdf           useful_links.txt
GoodaInternals.pdf
levinth@ubuntu18-2:~/gooda/gooda-analyzer/docs$ ls old_stuff/
17775.pdf                                  kernel_cycles.pdf
17776.pdf                                  Loops.pdf
18027.pdf                                  NHM_NUMA.pdf
29672_itanium.pdf                          Perf_analysis_Nehalem_intro.pdf
HPC_Perf_analysis_Xeon_5500_5600_intro.pdf
performance_analysis_guide.pdf
```