



Zen and the Art of Performance Monitoring

Michael Chynoweth - Sr. Principal Engineer Intel Corporation

Contributors: Joe Olivas, Patrick Konsor, Rajshree Chabukswar, Seth Abraham, Stas Bratanov



Agenda

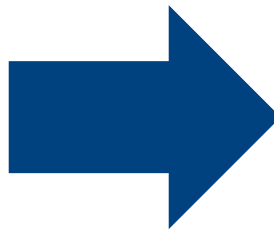
- End in Mind
 - Show some of the innovations we have in performance monitoring
 - Demonstrate how those advancements resolve problem with examples
- Topic1: Delivering definitive paths to debug across all architectures
 - Top Down and how it helps avoid pitfalls
- Topic2: Determining paths of execution, call stacks and timing
- Topic3: Large amounts of data, small timeframe, small perturbation or blind spots

Topic1: Delivering Definitive Paths to Debug Power and Performance

- The Lines Between Segments is Blurring
 - Customers ask for training on all segments Quark®, Atom®, Core® and Xeon®
- Problem = Performance monitoring unit features are designed by experts
- Without a clear optimization path, our customers will get on “tangents”
 - Example: Customer was concentrating on memory ordering “Nukes” as part of their performance analysis (Using Top Down we found Nukes were 0.3% of execution)



Memory Ordering
“Nukes”



Memory Ordering
“Fluffy, Harmless, Rainbow-Colored Bunnies”

Customers Should Be Pointed by Our Methodology and Tools Exactly Where to Look

Defining One Starting Point For Core, Uncore and Power Across All SoCs

Examples:

Core = Top Down

UnCore = Memory

Power = Energy MSRs

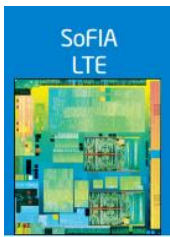
Allow technologies that
can scale down to
Quark

Start of single run to debug
At high level

One run, to get higher level
and allow for debugging
further



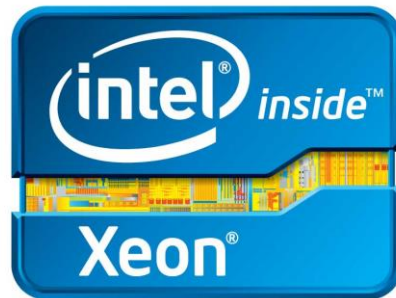
Quark®



Atom®



Core®



Xeon Phi®

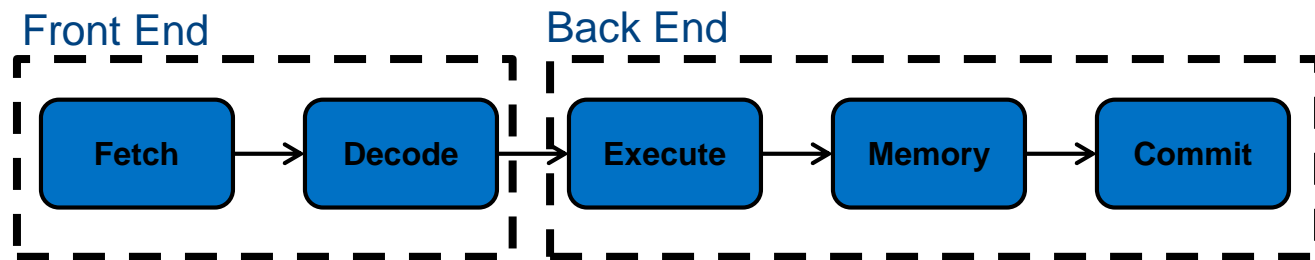


Start Using Scalable, Consistent and Converged Methodologies



Consistent Methodologies to Avoid Tangents:

Example = Top Down Methodology for Debugging CPU Bottlenecks

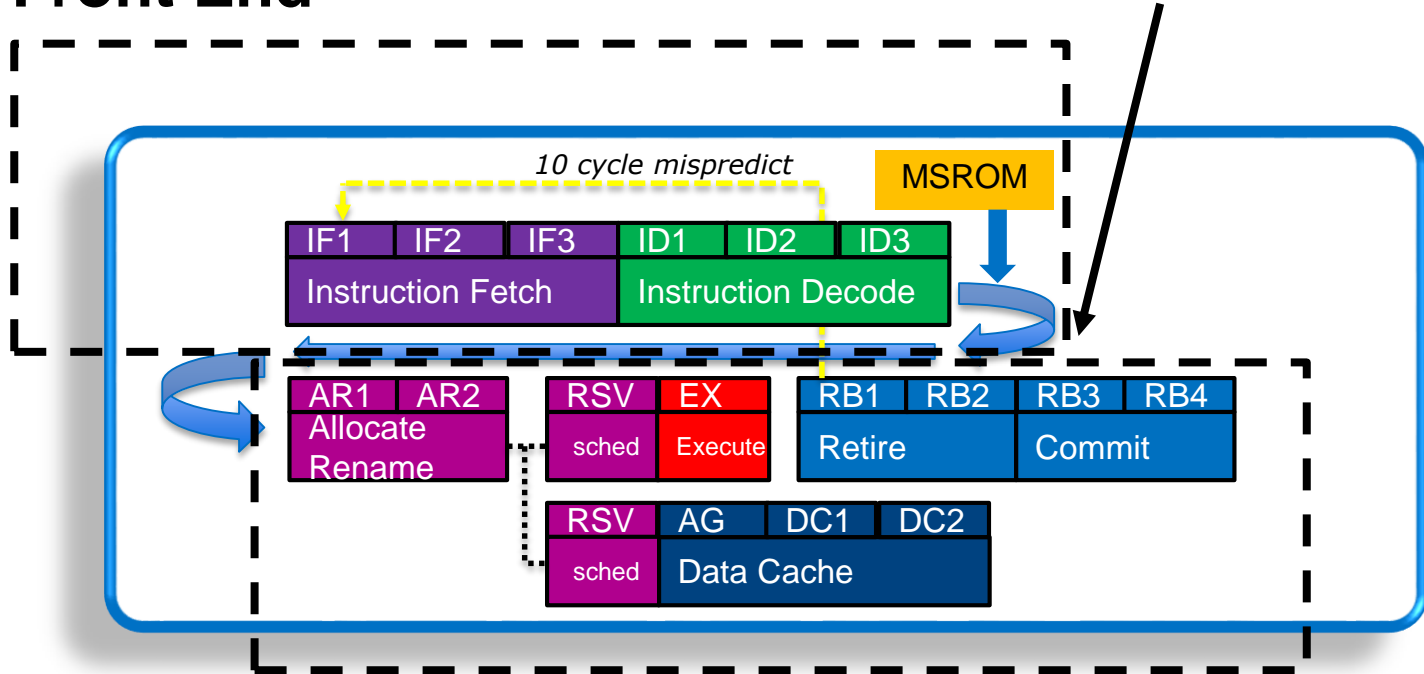


Top Down breaks the pipeline into 4 categories

- Front End Bound = Bound in Instruction Fetch -> Decode (Instruction Cache, ITLB)
- Back End Bound = Bound in Execute -> Commit (Example = Execute, load latency)
- Bad Speculation = When pipeline incorrectly predicts execution (Example branch mispredict memory ordering nuke)
- Retiring = Pipeline is retiring uops

Event NO_ALLOC_CYCLES.NOT_DELIVERED counts when BackEnd requests UOPs and FrontEnd Cannot Deliver

Front End



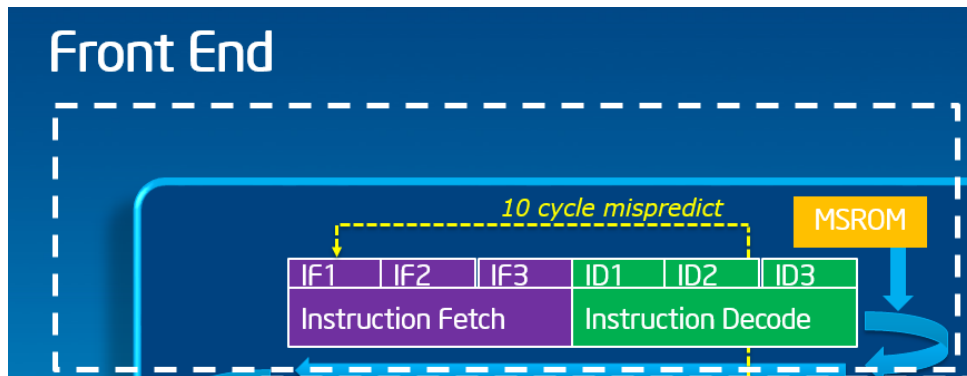
Back End

Why Do We Use Top Down to Drive Looking at Other Events?

Stats	BayTrail	Calculation
Cycles Per Instruction (CPI)	2.9	$\text{CPU_CLK_UNHALTED.CORE} / \text{INST_RETIRED.ANY}$
Front End Bound Cost	0.1%	$\text{NO_ALLOC_CYCLES.NOT_DELIVERED} * 1 / \text{CPU_CLK_UNHALTED.CORE}$
Microcode Sequencer Entry Cost	57.0%	$\text{MS_DECODED.MS_ENTRY} * 5 / \text{CPU_CLK_UNHALTED.CORE}$
MSUOPS/UOP_RETIRED	65%	$\text{UOPS_RETIRED.MS} / \text{UOPS_RETIRED.ALL}$

Microcode Sequencer Entry Cost is 57% of all cycles?! Should I raise the alarm?

Hint



When Would the Microcode Sequencer Matter?


Stats	BayTrail	Minimized Baytrail	Calculation
Cycles Per Instruction (CPI)	2.9	4.6	$CPU_CLK_UNHALTED.CORE/INST_RETIRED.ANY$
Front End Bound Cost	0.1%	63.6%	$NO_ALLOC_CYCLES.NOT_DELIVERED*1/CPU_CLK_UNHALTED.CORE$
Microcode Sequencer Entry Cost	57.0%	41.4%	$MS_DECODED.MS_ENTRY*5/CPU_CLK_UNHALTED.CORE$

Looking at just events is dangerous.
Even though MS Entry cost is 57% of all cycles we know that it is not impacting performance!

FE 64% of all cycles!
2x MS issues explain ~97% FE bottleneck

When Would the Microcode Sequencer Matter?

Stats	BayTrail	Minimized Baytrail	Calculation
Cycles Per Instruction (CPI)	2.9	4.6	$CPU_CLK_UNHALTED.CORE / INST_RETIRED.ANY$
Front End Bound Cost	0.1%	63.6%	$NO_ALLOC_CYCLES.NOT_DELIVERED * 1 / CPU_CLK_UNHALTED.CORE$
Microcode Sequencer 1/2 Speed Cost	0.0%	20.5%	$UOPS_RETIRED.MS / (2 * CPU_CLK_UNHALTED.CORE)$
Microcode Sequencer Entry Cost	57.0%	41.4%	$MS_DECODED.MS_ENTRY * 5 / CPU_CLK_UNHALTED.CORE$



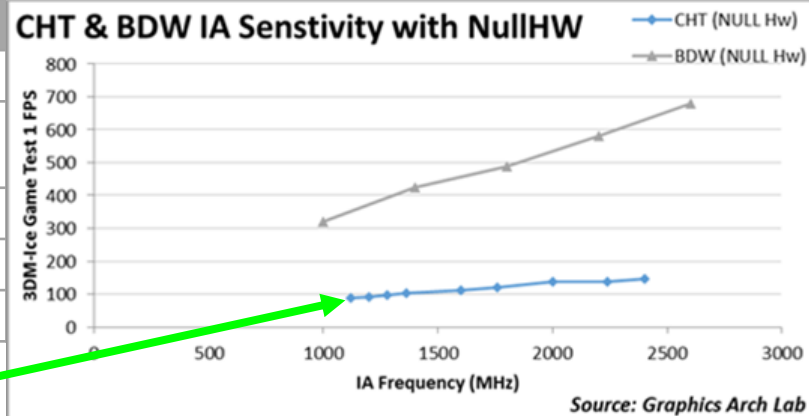
Now we have explained ~62% of all front end bound cycles explained

Top Down Let Us Identify Items We Did Not Understand

Where Is It Going?: Would Like to Better Tag Indirect Impacts with Top Down

Atom GFX 640 MHz	1360	1600	2000	2400	2560	Entire Data Set
Atom (GFX Enabled) FPS	96.06	106.21	120.35	132.4	137.5	1.43
Perfect Frequency Scaling		1.18	1.25	1.20	1.07	1.88
Actual Scaling		1.11	1.13	1.10	1.04	1.43
Frequency_Dep%		60%	53%	50%	58%	49%
Non_Frequency_Dep%		40%	47%	50%	42%	51%

CHT & BDW IA Sensivity with NullHW



Latency to Memory Causing Problems

Module_Name	HotThread%	CPI	OBSERVATIONS	Issue_Summary
Benchmark.exe: Benchmark.exe	65.80%	2.89	Retiring=19.44:FrontEnd=23.3: BackEnd&BadSpec=57.27:	L2_MISS=19%_D:ICACHEMISSES=10%_D
Benchmark.exe: igd10iumd32.dll	13.11%	4.15	Retiring=13.07:FrontEnd=64.55:Bac kEnd&BadSpec=22.37:	ICACHEMISSES=34%_D:ITLB_MISSES=8%:L2_ MISS=10%_D:

Benchmark binary has a large data cache footprint

Graphics Driver has a large instruction cache footprint

Graphics Driver and Benchmark Binary Battle Over Instruction + Data Real Estate



What Are The Last Branch Records?

	63	62	61	60:48	47:16	15:0
LBR_FROM_IP	SIGN_EXT (bit 47)				LBR FROM address	
LBR_TO_IP	SIGN_EXT (bit 47)				LBR TO address	
LBR_INFO	MISPRED	IN_TX	TSX_ABORTED	Reserved		cycle-count (*)

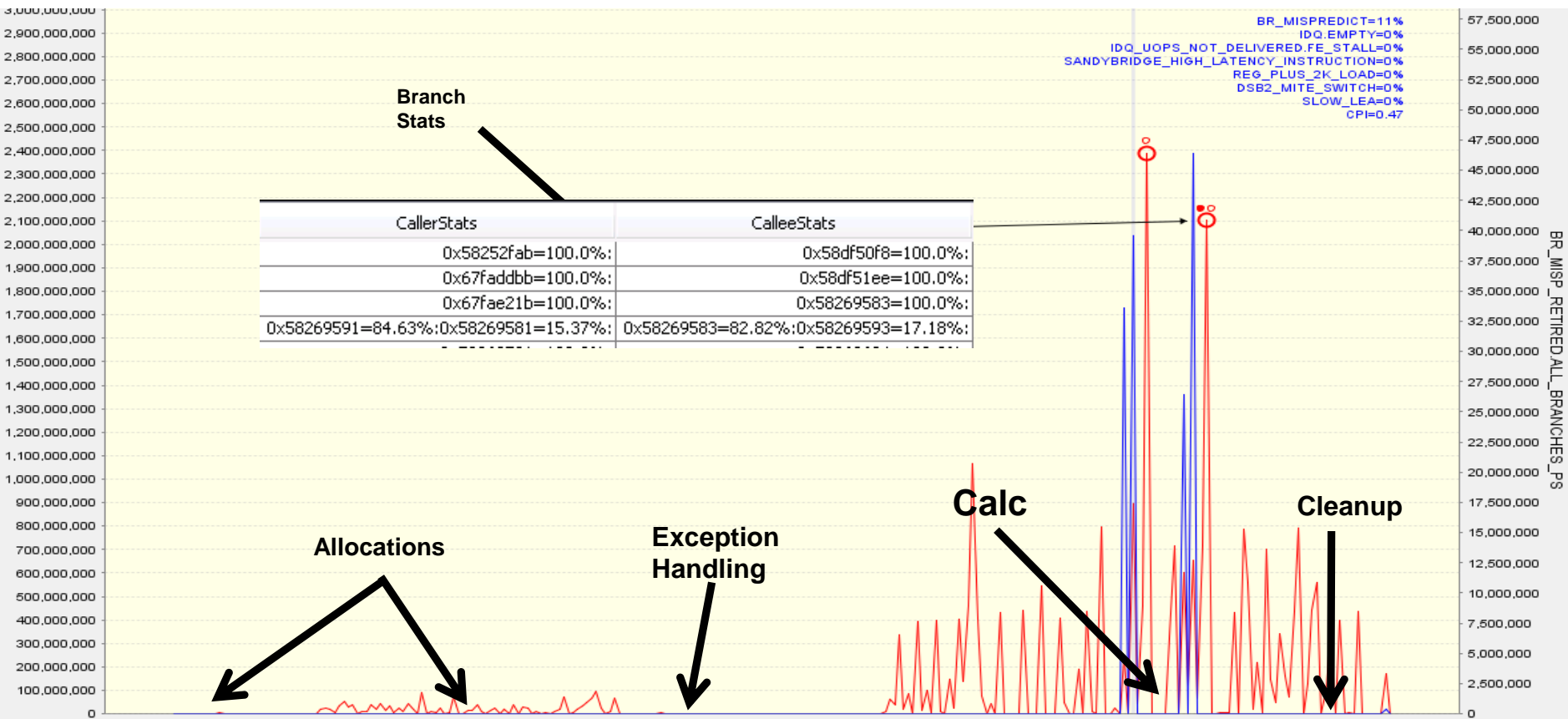
LBR Overview:

- LBRs dynamically track the last N taken branches:
 - N can now traverse from 8 to 32 taken branches
 - LBRs can be filtered for types of branches
- How are they used today?
 - Use them to recreate paths of execution
 - Assist in obtaining basic block hit counts
 - Used to weight cost of all
 - Paths of execution (function, branch, module)
 - Compilers are starting to use them for profile guided feedback
 - Example = AutoFDO
- Most Recent
 - Call stacks to any point of interest with LBR call stack
 - Cycle count



Pay attention, this one is brand new

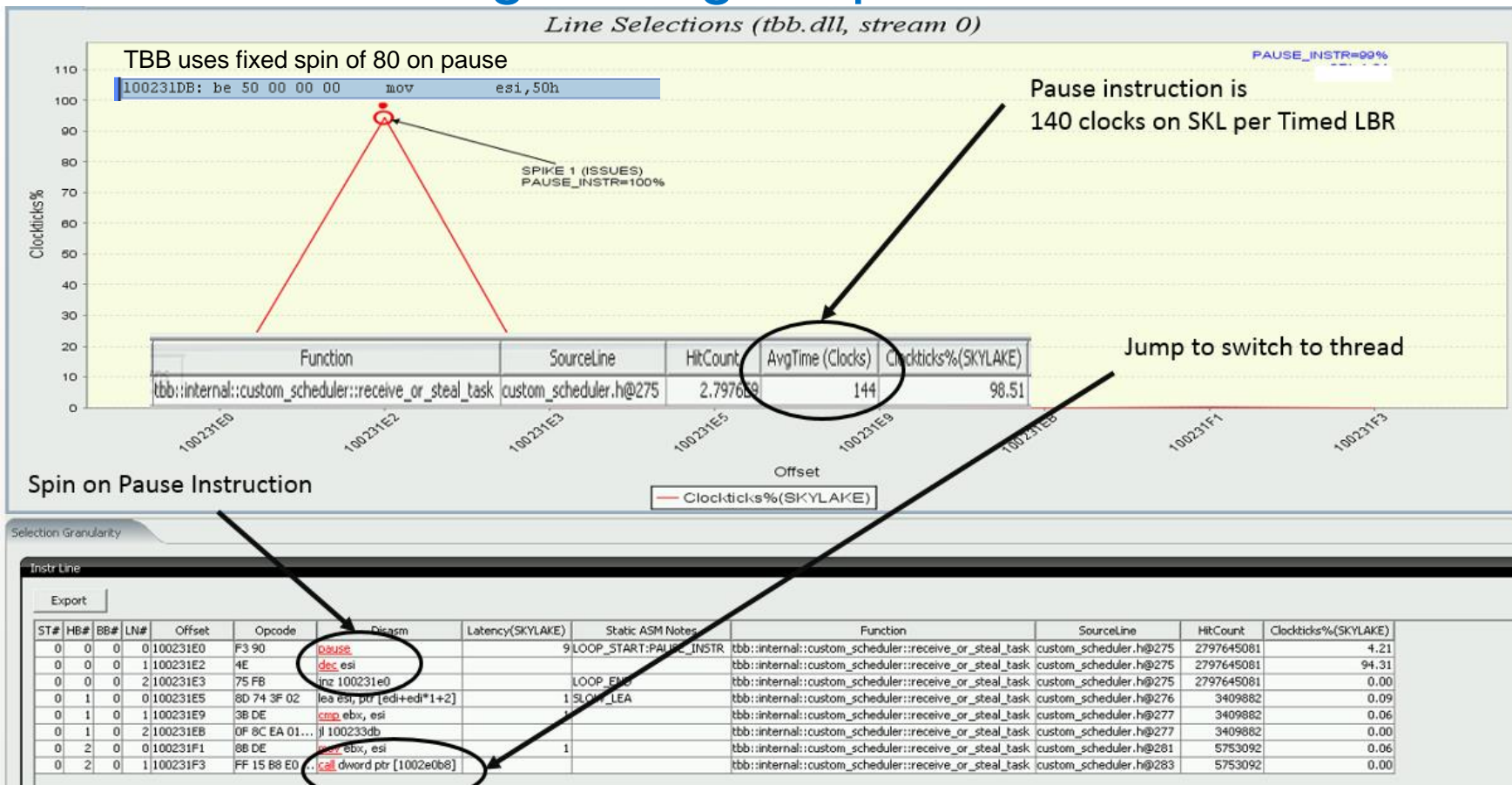
LBR is Utilized to Recreate Hot Path of Execution



LBR Allows Visibility of Complete Transaction



How Does Adding Timing Help?



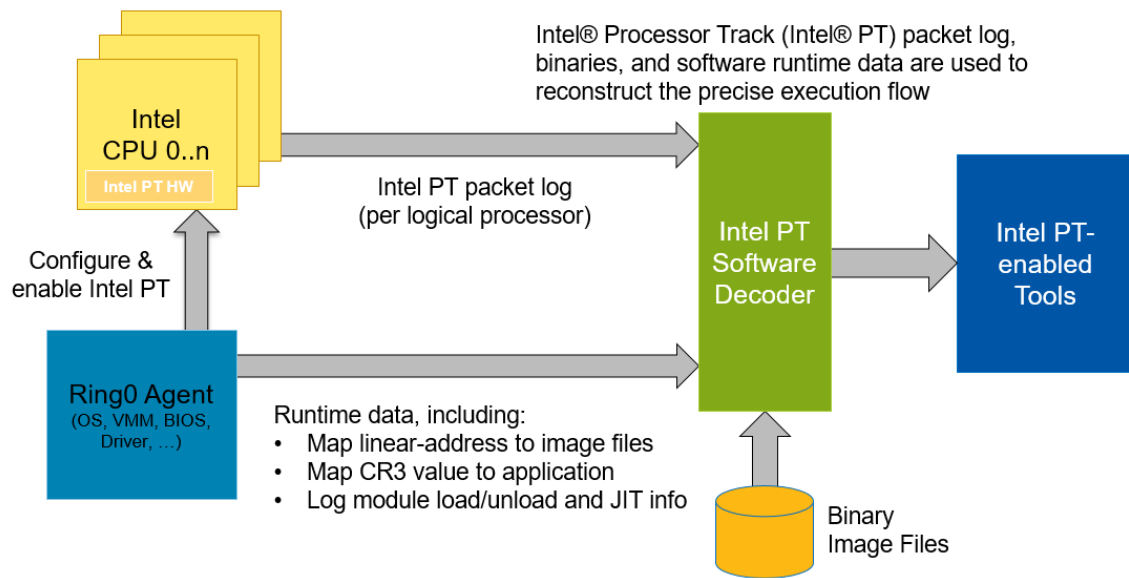
What is Intel® Processor Trace?

Intel® Processor Trace (Intel® PT) is a hardware feature that logs information about software execution with minimal impact to system execution

Supports control flow tracing with <5% overhead

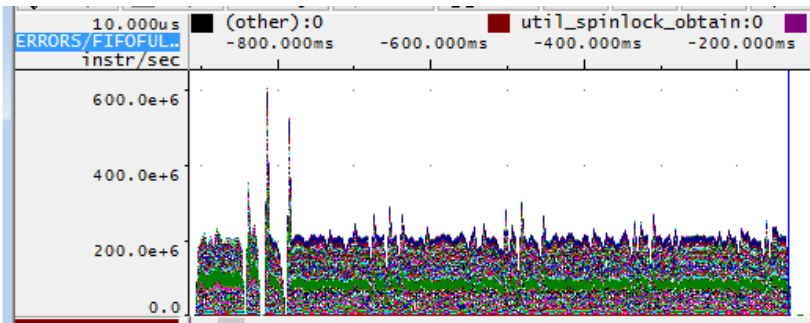
- Decoder can determine exact flow of software execution from trace log

Can store both cycle count and timestamp information



Intel Processor Trace is Delivering New Capabilities

Zooming at Microsecond Granularities



Locking Debug

Determining Contention on a Lock:

RETRY_LOCK = 10564

GOT_LOCK = 43542

$10564 / (43542 + 10564) = 0.1952$ (or 20% contended)

B::Trace.FindAll , Address_retry_protection					
	run	address	cycle	data	symbol
10564	run	address	cycle	data	symbol
12433600	41	00000000000000000000000000000000	28062288	0x00000000	C:\Program Files\...

B::Trace.FindAll , Address V.RANGE("_already_owned")					
	run	address	cycle	data	symbol
43542	run	address	cycle	data	symbol

Exceptions Hurting Performance

Interrupt showing up immediately after a conditional jump in memset call

Aha! It is a Device Not Available Due to touching XMM registers

Traverses tons of code

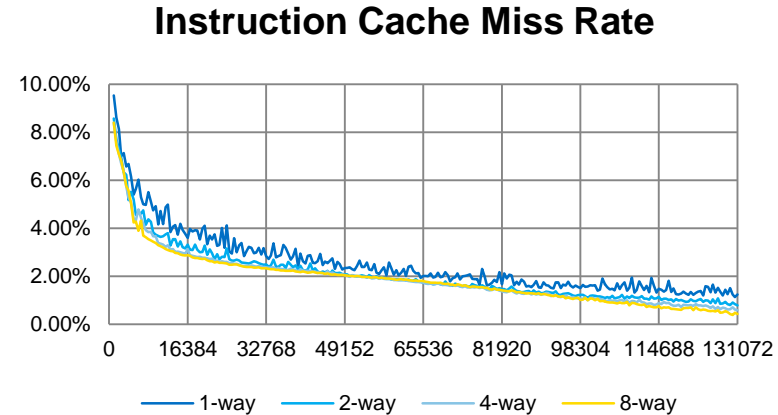
```
cmp     ecx, 10x20
jnb     0x38056700
interrupt
NP:000000003805D218 ptrace
call    0x3805B510 ; _tx_thread
cmp     dword ptr [0xFFF1F004], +0x0;
je      0x3805B52A ; _tx_thread
inc     dword ptr [0xFFF1F004]; dword
cmp     dword ptr [0xFFF1F000], +0x0;
```

```
pop     esp, +0x4
iretd
NP:0000000038056700 ptrace
movd    xmm0, eax
pshufd  xmm0, xmm0, 0x0
test    edx, 0x0F
```

RTIT Was Up and Running in Weeks...Because It is a Trace-Based Technology

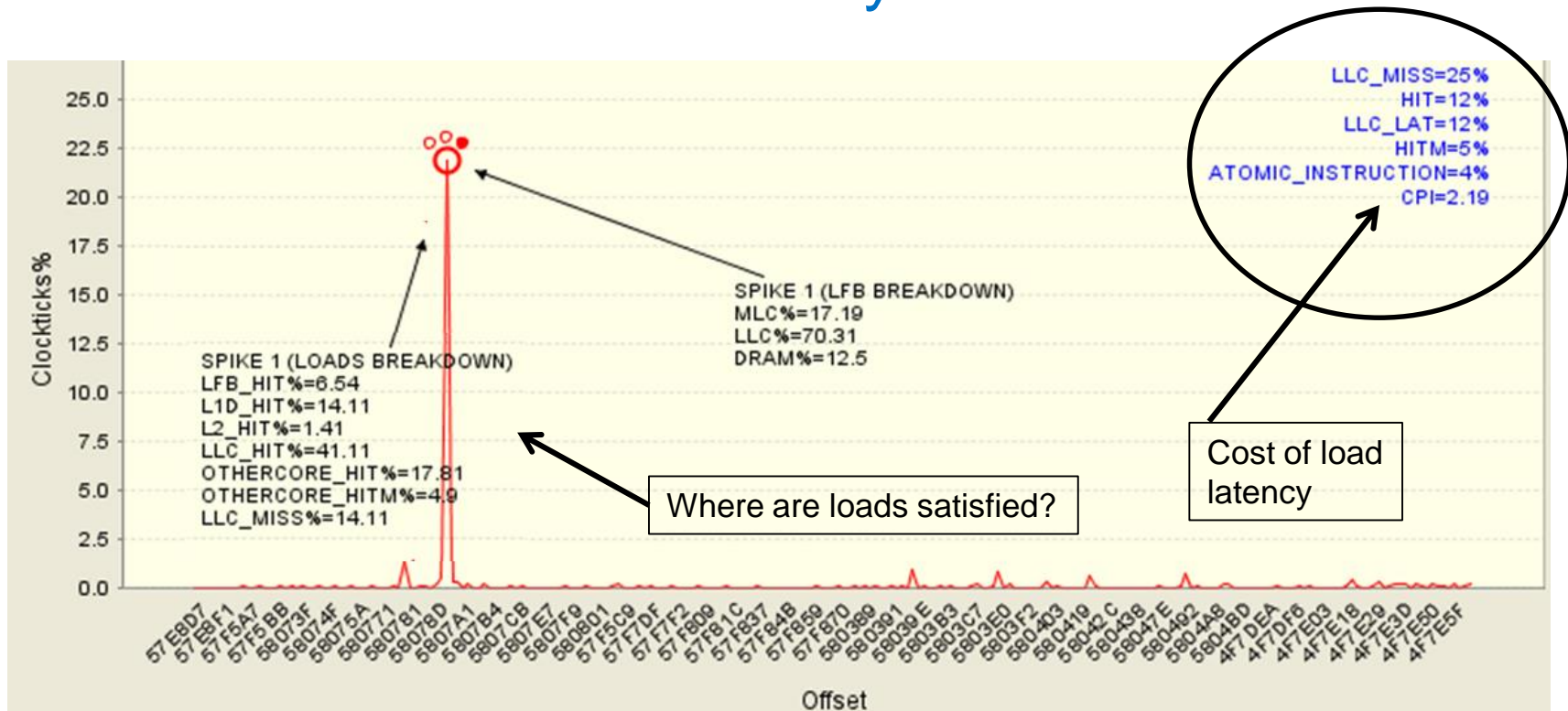
SoC Sizing using Modeling with Instruction Traces

- Model Icache, ITLB, and pre-decode in software, with a range of sizes and configs
- Simulate over traces from target workloads
 - Instruction traces quick to capture
 - And (relatively) quick to simulate
- Enables easy estimation of cache behavior for a given workload
 - Accurate within a few percentage points for Icache, ITLB, and pre-decode
- Enables evaluation of different cache configs across a range of workloads



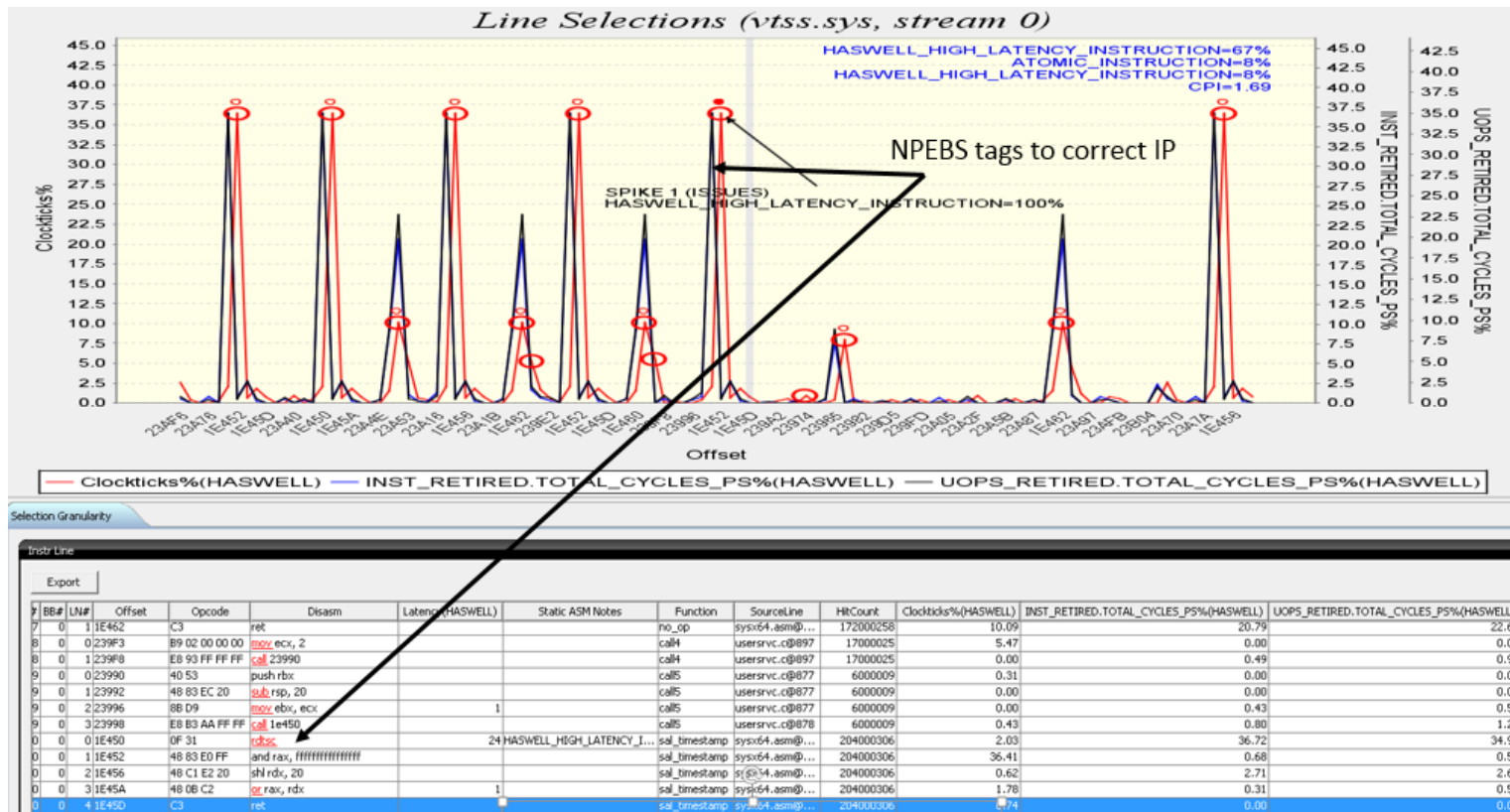
Intel Processor Trace Allows Modeling of Instruction Cache

Precise Events Are Incredibly Useful



Precise Events Collect Eventing IP, Registers, Data Linear Address (some) and Do NOT Require and Performance Monitoring Interrupt to Collect

Utilizing PEBS Triggering on Non-Precise Events



Capability to Collect PEBS on Non-Precise Events Allows For Less Overhead,
Better IP Tagging and Works When Interrupts Masked

Conclusions

- Shifting toward definitive ways to debug performance
 - Need tools help to ensure this is all automated and to help innovate
- LBRs are now complemented with timing
 - Get exact timing to nanosecond granularity
- Intel Processor Trace is Augmenting LBRs
 - Being used for advanced debug
- Precise Event Based Sampling is being utilized on non-precise events
 - Allows for an extremely cheap methodology to collect events without performance monitoring interrupts and allows for better tagging of issues