USING EVENT BASED SAMPLING TO UNDERSTAND PERFORMANCE ON INTEL® ARCHITECTURE PROCESSORS

Larry Meadows, Intel Corporation

ATPESC, St Charles, IL

August 7, 2018

Outline

What is Event Based Sampling

System-wide Performance

Core Performance

Vtune usage on Theta: https://www.alcf.anl.gov/user-guides/vtune-xc40

WHAT IS EVENT-BASED SAMPLING

Statistical Profiling with Event Based Sampling

The hardware contains programmable counters to count events (e.g., Instructions Retired, HW Thread Clocks, Cache Misses).

One or more counters are programmed to interrupt after some threshold number of a particular event (sample-after value).

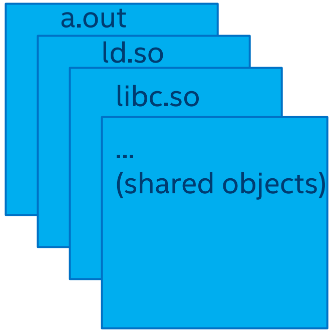
At each interrupt the Instruction Pointer (IP) is recorded, along with the HW Context for which the counter overflowed.

Symbolic information in the executable(s) is used to map IP to some meaningful source or object code entity: instruction, line number, loop, function, shared object.

The profiler creates a statistical profile with aggregated counts by bucket item.

Profiler Data Sources

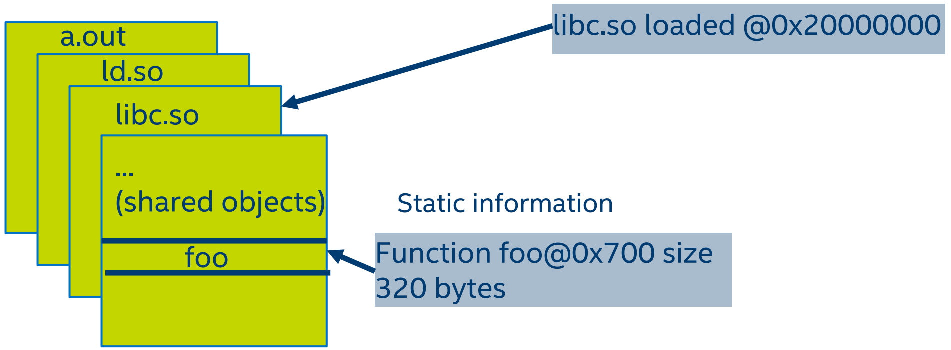
Executable Program



Symbolic Information

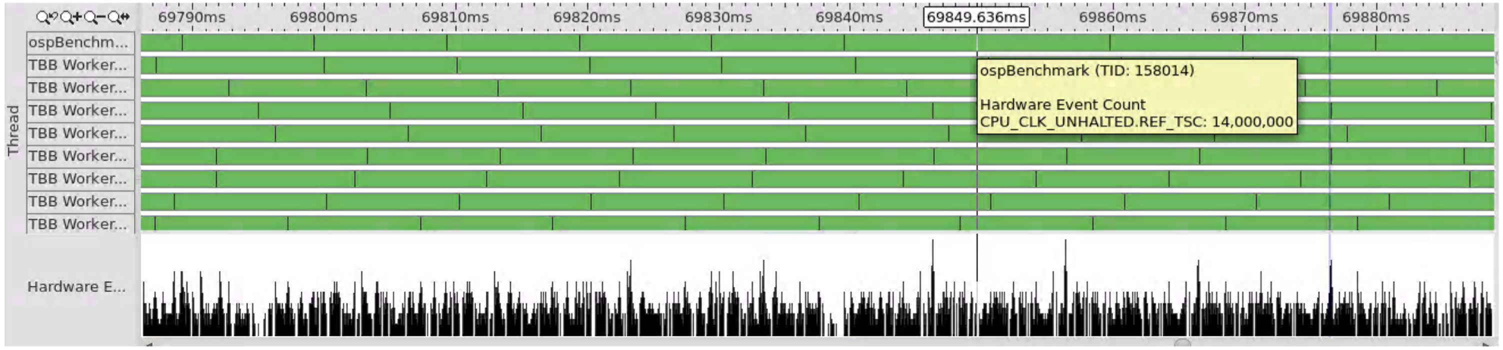
Dynamic information

Static information



Sample Database Over Time

Hardware event interrupts after N occurrences (N=14,000,000 here)

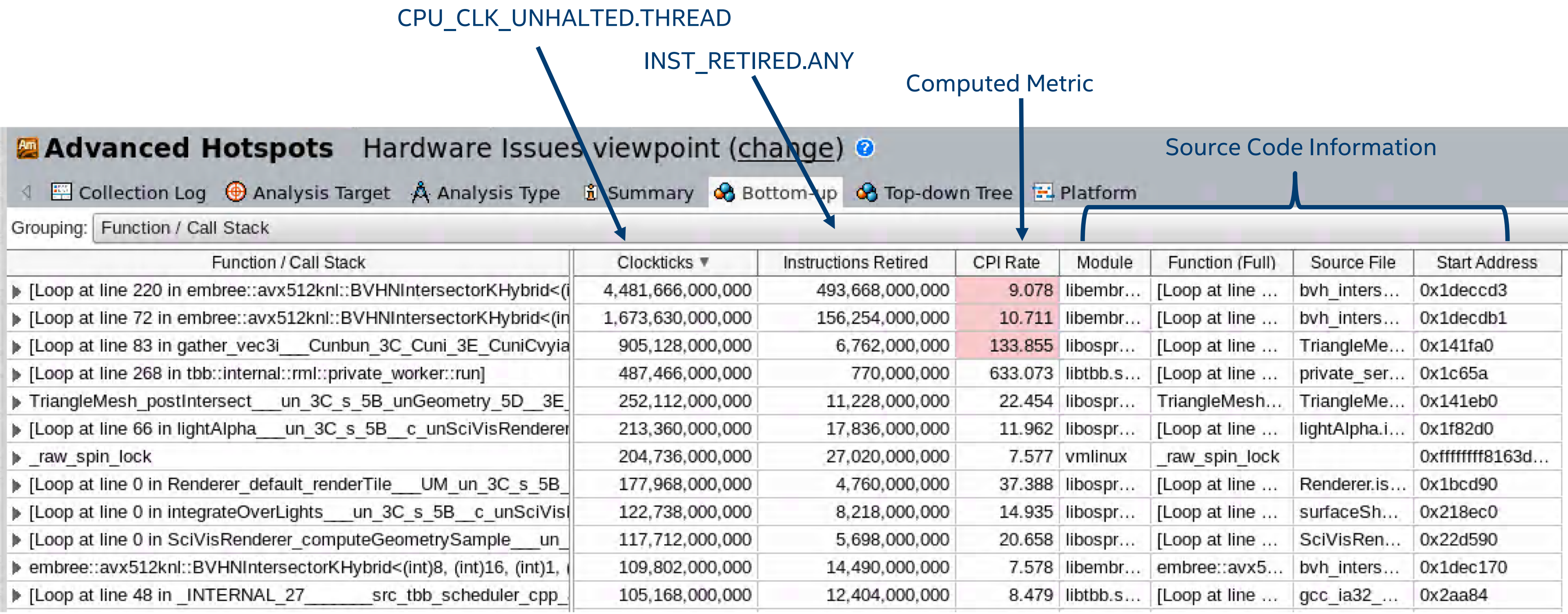


ospBenchmark (TID: 158014)

Hardware Event Count
CPU_CLK_UNHALTED.REF_TSC: 14,000,000

Each sample includes:
instruction pointer
hardware context (thread)
currently running thread ID
Hardware timestamp

Example Profile



Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate	Module	Function (Full)	Source File	Start Address
▶ [Loop at line 220 in embree::avx512knl::BVHNIntersectorKHybrid<(i	4,481,666,000,000	493,668,000,000	9.078	libembr…	[Loop at line …	bvh_inters…	0x1deccd3
▶ [Loop at line 72 in embree::avx512knl::BVHNIntersectorKHybrid<(in	1,673,630,000,000	156,254,000,000	10.711	libembr…	[Loop at line …	bvh_inters…	0x1decdb1
▶ [Loop at line 83 in gather_vec3i___Cunbun_3C_Cuni_3E_CuniCvyia	905,128,000,000	6,762,000,000	133.855	libospr…	[Loop at line …	TriangleMe…	0x141fa0
▶ [Loop at line 268 in tbb::internal::rml::private_worker::run]	487,466,000,000	770,000,000	633.073	libtbb.s…	[Loop at line …	private_ser…	0x1c65a
▶ TriangleMesh_postIntersect___un_3C_s_5B_unGeometry_5D__3E_	252,112,000,000	11,228,000,000	22.454	libospr…	TriangleMesh…	TriangleMe…	0x141eb0
▶ [Loop at line 66 in lightAlpha___un_3C_s_5B__c_unSciVisRenderer	213,360,000,000	17,836,000,000	11.962	libospr…	[Loop at line …	lightAlpha.i…	0x1f82d0
▶ _raw_spin_lock	204,736,000,000	27,020,000,000	7.577	vmlinux	_raw_spin_lock		0xffffffff8163d…
▶ [Loop at line 0 in Renderer_default_renderTile___UM_un_3C_s_5B_	177,968,000,000	4,760,000,000	37.388	libospr…	[Loop at line …	Renderer.is…	0x1bcd90
▶ [Loop at line 0 in integrateOverLights___un_3C_s_5B__c_unSciVisI	122,738,000,000	8,218,000,000	14.935	libospr…	[Loop at line …	surfaceSh…	0x218ec0
▶ [Loop at line 0 in SciVisRenderer_computeGeometrySample___un_	117,712,000,000	5,698,000,000	20.658	libospr…	[Loop at line …	SciVisRen…	0x22d590
▶ embree::avx512knl::BVHNIntersectorKHybrid<(int)8, (int)16, (int)1, (	109,802,000,000	14,490,000,000	7.578	libembr…	embree::avx5…	bvh_inters…	0x1dec170
▶ [Loop at line 48 in _INTERNAL_27_______src_tbb_scheduler_cpp_	105,168,000,000	12,404,000,000	8.479	libtbb.s…	[Loop at line …	gcc_ia32_…	0x2aa84

Steps to get a Profile

Include –g (and -O) when compiling to get source-code mapping information:

```
icpc –O –g myprog.cc –o myprog
```

Collect with the command-line tool:

```
amplxe-cl –collect advanced-hotspots –r myresult ./myprog
```

Analyze with the GUI:

```
amplxe-gui myresult
```

Or with a command-line report

```
amplxe-cl –report hotspots –r myresult
```

Dimensionality of Profiling Analysis

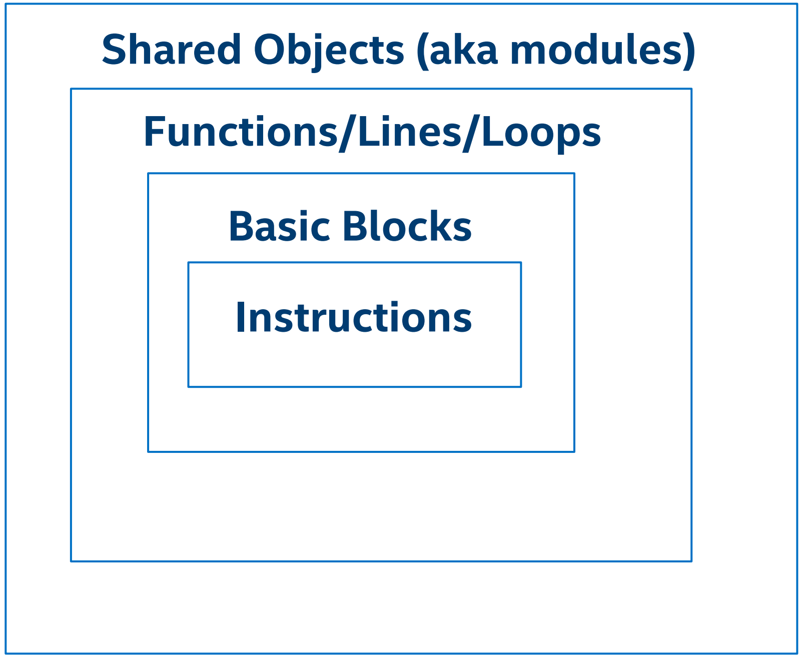
Four dimensions to consider when analyzing an application profile:

- Code Entities
- Software Execution Entities
- Hardware Execution Entities
- Time

The first three have a natural nested structure imposed by the programming model, execution model, and hardware. Time does not have a natural structure, but annotation can impose structure on time.

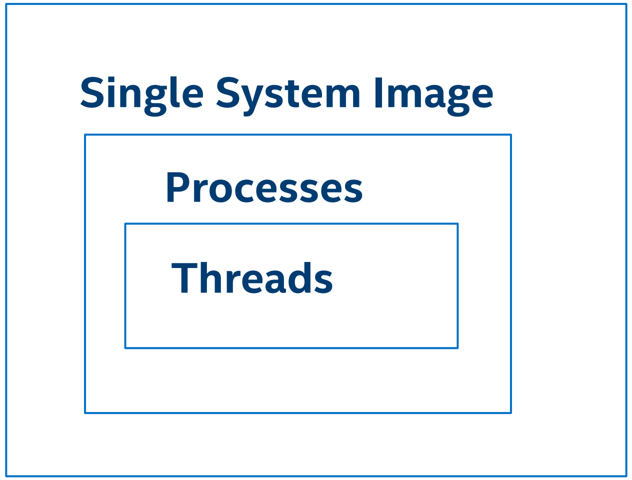
Code Entities

- Compiler generates instructions organized into functions and source lines exposed by symbolic information in object files.
- Linker generates shared objects (or executables) by combining object files.
- Basic Blocks and Loops are usually determined by static object code analysis.



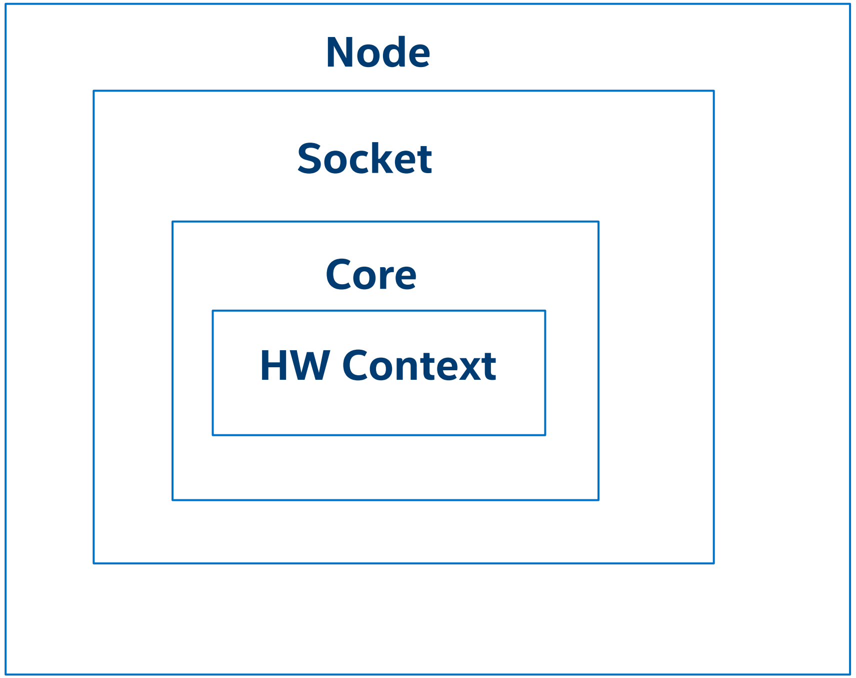
Software Execution Entities

- A thread is an execution context containing an instruction pointer (IP) and associated context.
- A process contains one or more threads.
- A single system image is a shared address space that contains multiple processes.



Hardware Execution Entities

- A HW Context is the hardware entity that executes instructions. It contains an IP, registers, and associated state.
- A core consists of one or more HW contexts that share resources. Instructions from more than one HW context can be in flight at the same time.
- A socket consists of multiple cores with a cache-coherent interconnect and a memory interface. It may also contain other resources (e.g. L3 cache, PCI-E interface).
- A node is a cache-coherent single address space consisting of one or more sockets and possibly other resources.



Time

Time can be measured in two ways:

- CPU time. This is the time for which a given thread was executing. This is a thread-specific quantity.
- Elapsed time. This is traditional wallclock (e.g., stopwatch) time.
- If CPU time < Elapsed time then the thread was not executing for the entire interval.

Note that we can measure elapsed time on a given thread by reading a wallclock-equivalent timer (RDTSC) on that thread (thread-specific markers).

Time has no natural structure, but in any given application, there will be a natural division into phases (e.g., OpenMP parallel regions). Dynamic instrumentation (tracing) can be used to segment a profile into these phases.

A Word on Quantization

Typical sample-after (overflow) values are 1e6 to 10e6 events.

The CPU can do a lot of things in 10 million cycles; sampling implicitly assumes that it was doing the same thing the whole time.

Be very careful about drawing conclusions obtained by zooming in too closely.

Multiplexing events onto HW event counters exacerbates this effect.

Corollary: be very careful about interpreting profile counts for individual instructions and even source lines.

SYSTEM-WIDE PERFORMANCE

Understanding 4D Data

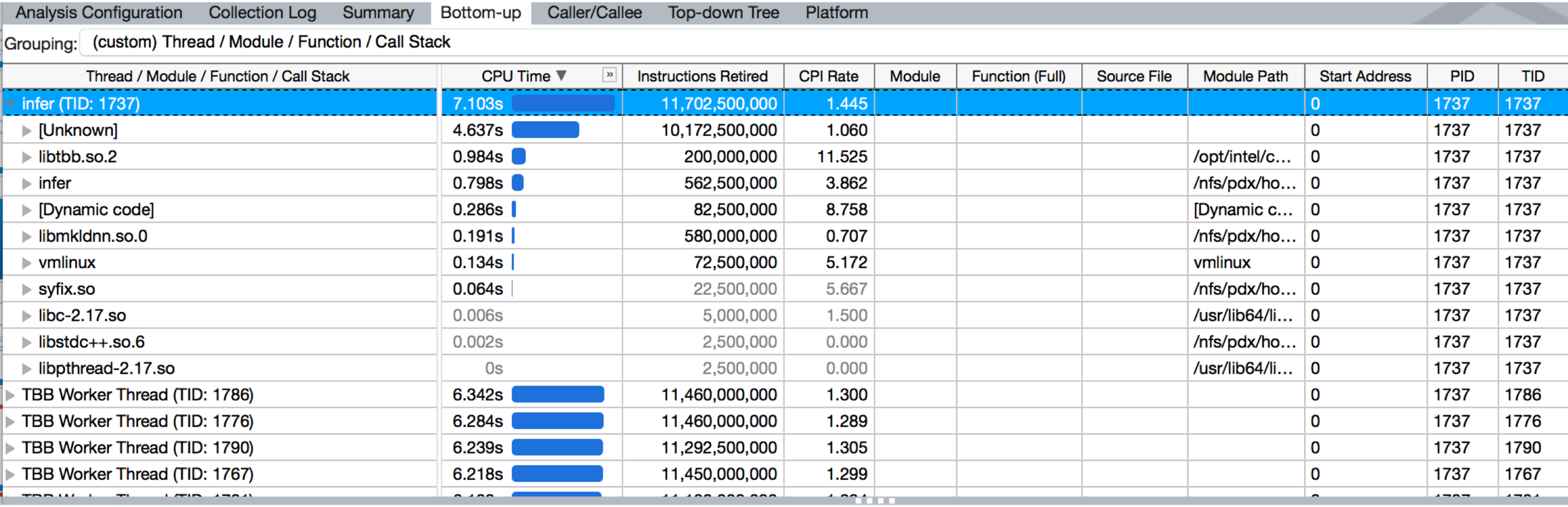
Typical flat profile summarizes data by reducing over Time, HW Execution Entities, and SW Execution Entities and grouping by function.

Vtune Amplifier generalizes this by providing hierarchical groupings, e.g., Thread/Module/Function reduces over Time and groups first by Thread, then by Module, then by Function within the module. You can explore these in the GUI.

We can also generate 2D views using Pivot Tables by selecting a row dimension and a column dimension, deciding on grouping attributes for each, and reducing over the remaining dimensions. For example:

- Pivot by function is a function-level profile
- Pivot function by thread exposes load imbalance
- Pivot process by cpuid exposes multiple processes on same HW thread
- Pivot thread by cpuid exposes thread migration

Vtune Hierarchical Grouping



Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: (custom) Thread / Module / Function / Call Stack

Thread / Module / Function / Call Stack	CPU Time ▼	Instructions Retired	CPI Rate	Module	Function (Full)	Source File	Module Path	Start Address	PID	TID
infer (TID: 1737)	7.103s	11,702,500,000	1.445					0	1737	1737
[Unknown]	4.637s	10,172,500,000	1.060					0	1737	1737
libtbb.so.2	0.984s	200,000,000	11.525				/opt/intel/c…	0	1737	1737
infer	0.798s	562,500,000	3.862				/nfs/pdx/ho…	0	1737	1737
[Dynamic code]	0.286s	82,500,000	8.758				[Dynamic c…	0	1737	1737
libmkldnn.so.0	0.191s	580,000,000	0.707				/nfs/pdx/ho…	0	1737	1737
vmlinux	0.134s	72,500,000	5.172				vmlinux	0	1737	1737
syfix.so	0.064s	22,500,000	5.667				/nfs/pdx/ho…	0	1737	1737
libc-2.17.so	0.006s	5,000,000	1.500				/usr/lib64/li…	0	1737	1737
libstdc++.so.6	0.002s	2,500,000	0.000				/nfs/pdx/ho…	0	1737	1737
libpthread-2.17.so	0s	2,500,000	0.000				/usr/lib64/li…	0	1737	1737
TBB Worker Thread (TID: 1786)	6.342s	11,460,000,000	1.300					0	1737	1786
TBB Worker Thread (TID: 1776)	6.284s	11,460,000,000	1.289					0	1737	1776
TBB Worker Thread (TID: 1790)	6.239s	11,292,500,000	1.305					0	1737	1790
TBB Worker Thread (TID: 1767)	6.218s	11,450,000,000	1.299					0	1737	1767

A Little Pandas

Generate the report:

```
amplxe-cl -report hw-events -r $1 -group-by=cpuid,core,thread,function \
    -format=csv -csv-delimiter=tab -inline-mode=off
```

Create a pandas data frame:

```
import pandas as pd
df = pd.read_csv(file, sep='\t', index_col=False)
```

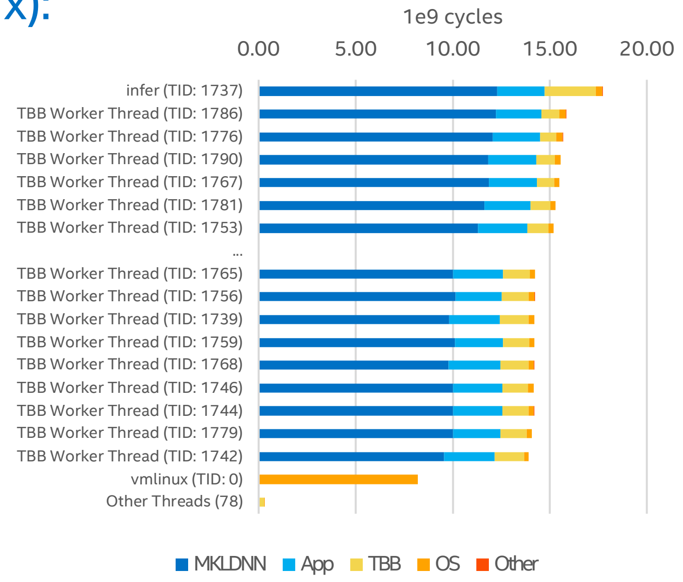
Generate a pivot table:

```
p = pd.pivot_table(df,
        values='Hardware Event Count:CPU_CLK_UNHALTED.REF_TSC',
        index='Thread',
        columns='Module',
        aggfunc=np.sum,
        fill_value=0)
```

Custom Grouping of Code Entities

Modules are often too coarse grained, functions too fine-grained. Use pattern matching and add a column to the dataframe to summarize more usefully (e.g., application, MKL, threading overhead, linux):

Thread	MKLDNN	App	TBB	OS	Other
infer (TID: 1737)	12.28	2.47	2.62	0.34	0.02
TBB Worker Thread (TID: 1786)	12.22	2.36	0.93	0.31	0.00
TBB Worker Thread (TID: 1776)	12.07	2.44	0.84	0.33	0.00
TBB Worker Thread (TID: 1790)	11.83	2.47	0.95	0.30	0.00
TBB Worker Thread (TID: 1767)	11.86	2.49	0.88	0.27	0.00
TBB Worker Thread (TID: 1781)	11.62	2.39	1.01	0.28	0.00
TBB Worker Thread (TID: 1753)	11.29	2.56	1.09	0.25	0.00
…					
TBB Worker Thread (TID: 1765)	10.01	2.60	1.38	0.28	0.00
TBB Worker Thread (TID: 1756)	10.14	2.37	1.40	0.31	0.00
TBB Worker Thread (TID: 1739)	9.82	2.60	1.50	0.29	0.00
TBB Worker Thread (TID: 1759)	10.12	2.45	1.35	0.27	0.00
TBB Worker Thread (TID: 1768)	9.78	2.68	1.45	0.27	0.00
TBB Worker Thread (TID: 1746)	10.02	2.53	1.34	0.30	0.00
TBB Worker Thread (TID: 1744)	10.02	2.54	1.37	0.24	0.00
TBB Worker Thread (TID: 1779)	10.01	2.44	1.36	0.27	0.00
TBB Worker Thread (TID: 1742)	9.56	2.61	1.51	0.24	0.00
vmlinux (TID: 0)	0.00	0.00	0.00	8.19	0.00
Other Threads (78)	0.00	0.00	0.30	0.04	0.00



Adding Structure to the Time Dimension

Add instrumentation to the code to record time stamps for ‘interesting’ intervals, e.g. OpenMP parallel regions (Vtune Amplifier has those built in), user-defined tasks, solver calls, CNN convolutions. Vtune Amplifier can import this data, see:

https://software.intel.com/en-us/vtune-amplifier-help-external-data-import

Example: Use MKLDNN_VERBOSE to print log for each operation with start and end TSC, convert with python script, and import into existing result:

```
MKLDNN_VERBOSE=2 amplxe-cl –collect advanced-hotspots sh run.sh >Log

python parse.py Log

amplxe-cl -r r000ah –import mkldnn-hostname-ortce-skl3.csv
```

Example: Metrics by MKLDNN call

```
amplxe-cl -report hw-events -group-by frame-domain -r r000ah/ -format csv -csv-delimiter comma -q >report.csv
```

Frame Domain	INST_RETIRED.ANY	CPU_CLK_UNHALTED.THREAD	CPU_CLK_UNHALTED.REF_TSC	CPI	Turbo	Frame Time	Frame Count
ECF19	178,825,000,000	132,496,100,000	142,772,500,000	0.741	0.93	1.76	100
ECF20	93,478,900,000	65,727,100,000	71,442,600,000	0.703	0.92	0.88	100
ECF17	69,250,700,000	53,900,500,000	58,666,100,000	0.778	0.92	0.72	100
ECF1	52,451,500,000	43,752,900,000	47,913,600,000	0.834	0.91	0.59	100
ECF13	14,280,700,000	33,037,200,000	32,062,000,000	2.313	1.03	0.39	100
ECF18	36,864,400,000	28,664,900,000	31,595,100,000	0.778	0.91	0.39	100
ECF15	28,975,400,000	28,853,500,000	31,146,600,000	0.996	0.93	0.39	100
ECF0	32,296,600,000	26,969,800,000	29,014,500,000	0.835	0.93	0.36	100
ECF14	9,887,700,000	31,700,900,000	28,267,000,000	3.206	1.12	0.35	100
ECF21	36,406,700,000	25,711,700,000	28,135,900,000	0.706	0.91	0.35	100
ECF7	6,216,900,000	23,379,500,000	21,318,700,000	3.761	1.10	0.26	100
ECF16	18,377,000,000	17,894,000,000	19,499,400,000	0.974	0.92	0.24	100
ECF3	17,381,100,000	16,721,000,000	18,160,800,000	0.962	0.92	0.22	100
EPF2	1,285,700,000	17,544,400,000	15,957,400,000	13.646	1.10	0.20	100
ECF11	4,123,900,000	9,832,500,000	10,228,100,000	2.384	0.96	0.13	100
ECF12	3,454,600,000	8,995,300,000	9,434,600,000	2.604	0.95	0.12	100
ECF5	7,316,300,000	8,038,500,000	8,574,400,000	1.099	0.94	0.11	100
ECF9	1,973,400,000	7,659,000,000	7,516,400,000	3.881	1.02	0.09	100
EPF4	471,500,000	5,556,800,000	5,437,200,000	11.785	1.02	0.07	100
EPF6	167,900,000	2,279,300,000	2,343,700,000	13.575	0.97	0.03	100
EPF8	85,100,000	1,087,900,000	975,200,000	12.784	1.12	0.01	100
EPF10	41,400,000	409,400,000	409,400,000	9.889	1.00	0.01	100
[No frame domain - Outside any frame]	33,846,800,000	66,950,700,000	60,122,000,000	1.978	1.11		

CORE PERFORMANCE

Microarchitecture Review

Common elements between KNL and Intel® Xeon® processors (SKX/BDW):

- Multithreaded cores with private L1 caches connected by a mesh

Key differences:

- Two KNL cores share one L2, SKX/BDW cores have private L2
- SKX/BDW has an L3 cache, KNL does not
- SKX/BDW systems are multi-socket, KNL is single-socket
- SKX/BDW can retire 4 IPC, KNL can retire 2 IPC
- KNL MCDRAM has much higher bandwidth than SKX/BDW DDR
- BDW does not have AVX-512

Processor Performance

Standard metric is retired Instructions Per Cycle (IPC)

- KNL max is 2 IPC per core. SKX/BDW max is 4 IPC per core.
- Vtune amplifier displays the reciprocal CPI
- Computed as $\frac{\sum_{hwthreads} instructions}{\sum_{hwthreads} cycles}$
  - *instructions* == `INST_RETIRED.ANY` *cycles* == `CPU_CLK_UNHALTED.THREAD`

Note: IPC per core depends on how many HW threads per core (nHT) are running:

$IPC_{core} = IPC_{thread}$*nHT

$CPI_{core} = CPI_{thread}$/nHT

Vtune Amplifier always displays CPI_{thread}

This is why all scaling graphs should be in terms of cores, not threads, and should show the number of threads per core used.

Common Impediments to Perfect IPC

Bandwidth Bound

- Beware of BW bound code that doesn’t need to be (e.g., fails to optimize for reuse in L2 cache)
- Lack of vectorization, lack of streaming stores, missing SW prefetches may limit BW

Other Impediments

- Code may be BW bound from L1 or L2 cache as well as from memory.
- Micro-architectural decisions may limit IPC (e.g., some VPU instructions issue on only one port)
- Real dependences may limit IPC (one instruction needs to wait for another’s result)
- Code may be front-end bound (instruction fetch and decode, branch prediction)

Vtune Amplifier Pre-defined Collection Types

```
-collect advanced-hotspots
```

CPI, Turbo frequency

```
-collect memory-access
```

Memory bandwidth, L2 hit/miss information

```
-collect general-exploration
```

Top-down cycle accounting

General exploration multiplexes many events onto the hardware counters. This can lead to inaccurate results especially on short-running program segments. It is more useful on SKX/BDW than on KNL.

Finding Bottlenecks

Two general methods:

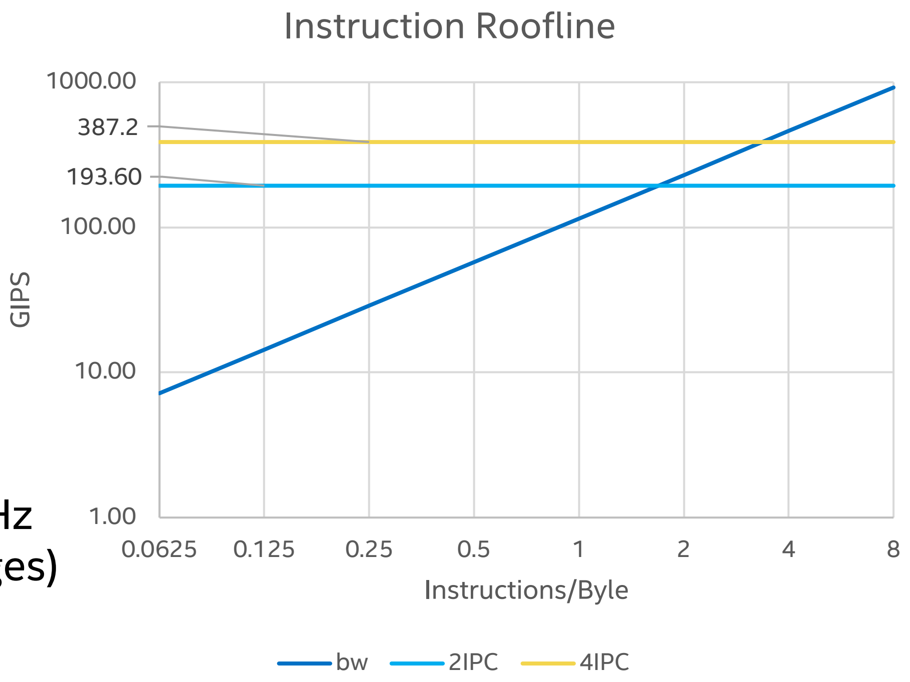
Top-down breaks execution into 4 domains: FE Bound, Bad Speculation, BE Bound, and Retiring.

https://www.researchgate.net/publication/269302126_A_Top-Down_method_for_performance_analysis_and_counters_architecture

Bottleneck analysis looks for common bottlenecks, typically some kind of bandwidth, and compares performance against expected peak.

Instruction Roofline Guides BW Analysis

Generalization of roofline model using instructions rather than flops



E5-2699 v4 @ 2.20GHz
2 socket 22 cores/socket
BW = 115GB/Sec
Peak GIPS: IPC*44*2.2 GHz
(ignores frequency changes)

Top-Down Analysis Categories

FE Bound: Decoder can’t deliver uops fast enough. Usually I$ or really big codes.

Bad Speculation: uops that are speculatively executed but the result is never used.

Retiring: What you want! Normal retirement of uops

BE Bound: Everything else. Stalls due to data unavailable and no more OOO

Vtune Amplifier general exploration has good documentation on this. The GUI tooltips give the actual names of the events.

Top-Down Analysis Example

Metric	Formula
FE Bound	IDQ_UOPS_NOT_DELIVERED.CORE / (4*(CPU_CLK_UNHALTED.THREAD/2))
Bad Speculation	(UOPS_ISSUED.ANY-UOPS_RETIRED.RETIRE_SLOTS+4*INT_MISC.RECOVERY_CYCLES) / (4*CPU_CLK_UNHALTED.THREAD/2)
Retiring	UOPS_RETIRED.RETIRE_SLOTS / (4*(CPU_CLK_UNHALTED.THREAD/2))
BE Bound	1 - (FE Bound + Bad Speculation + Retiring)

Vtune Command:

```
amplxe-cl -collect-with runsa \
-knob event-config=\
CPU_CLK_UNHALTED.REF_TSC,CPU_CLK_UNHALTED.THREAD,\
INST_RETIRED.ANY,\
IDQ_UOPS_NOT_DELIVERED.CORE,INT_MISC.RECOVERY_CYCLES,\
UOPS_RETIRED.RETIRE_SLOTS,UOPS_ISSUED.ANY \
<command>
```

Metrics for KNL

Metric	Formula	Ideal
L2 input BW	64*(L2_REQUESTS.MISS + L2_PREFETCHER.ALLOC_XQ) / elapsed_time	380 GB/sec
L2 Hit Rate	MEM_UOPS_RETIRED.L2_HIT_LOADS / (MEM_UOPS_RETIRED.L2_HIT_LOADS+MEM_UOPS_RETIRED.L2_MISS_LOADS)	100%
L1 input lines	(2*L2_REQUESTS.MISS + 3* L2_PREFETCHER.ALLOC_XQ + L2_REQUESTS.REFERENCE) / CPU_CLK_UNHALTED.THREAD	0.46
L1 Hit Rate	(MEM_UOPS_RETIRED.ALL_LOADS - MEM_UOPS_RETIRED.L1_MISS_LOADS) / MEM_UOPS_RETIRED.ALL_LOADS	100%
SIMD operations/clock	(UOPS_RETIRED.SCALAR_SIMD+UOPS_RETIRED.PACKED_SIMD) / CPU_CLK_UNHALTED.THREAD	2
FE Bound	NO_ALLOC_CYCLES.NOT_DELIVERED / CPU_CLK_UNHALTED.THREAD	<5%
Branch Mispredict Bound	NO_ALLOC_CYCLES.MISPREDICTS / CPU_CLK_UNHALTED.THREAD	<5%
Frequency Ratio	CPU_CLK_UNHALTED.THREAD/CPU_CLK_UNHALTED.REF_TSC	varies with SKU

BACKUP

Performance Analysis and Improvement

1. **Measure** the performance of the application running on the system
2. **Estimate** the peak performance of the application
3. If performance is near enough to peak, you are done
4. **Rewrite** the code (and possibly algorithm) and go to step 1

Here we cover #1 and #2

Measuring Performance

Instrumentation

- Modify the code to read timers or counters around interesting code regions
- Summarize the data and print it

Sampling

- Program timers or counters to interrupt the program periodically
- Use the address of the interrupted instruction to build a statistical profile

We primarily discuss sampling but include information on instrumentation techniques

Hardware+OS View of Program Execution

Each HW thread executes instructions from a single OS process at a time

The process executing on a given thread may change (voluntary or involuntary context switch)

For full understanding of machine performance must see all instructions executed on all HW threads

`amplxe-cl –analyze-system` or

`perf –a`

Otherwise you may miss important events perturbing the execution of your program.

Two Different Ways to Collect and View a Profile Using Hardware Counters

Intel® Vtune™ Amplifier

```
amplxe-cl –collect advanced-hotspots –r myresult ./a.out
amplxe-gui myresult   # GUI
amplxe-cl –report hotspots –r myresult  # CLI
```

Linux perf

```
perf record –e cycles –e ref-cycles –e instructions ./a.out
perf report    # GUI or CLI
perf script    # Raw data
```

Why use Vtune Amplifier?

Linux perf has higher overhead than the driver used by Vtune Amplifier

- `perf -a` has lower overhead but requires root permissions or `perf_event_paranoid=0`

Vtune Amplifier has superior GUI and report generation

- See all events on one grid
- Powerful grouping and filtering capabilities
- Command-line reports can use all grouping and filtering capabilities

We primarily focus on Vtune Amplifier with some discussion of perf

Other Tools

Several third party tools can provide useful information:

HPCToolkit (Rice); Tau (ParaTools); MAP (ARM); PAPI (UTK); others…

Linux kernel tracing (ftrace) for detailed kernel-level performance analysis.

Linux statistics (via top, ps, sysfs and procfs psuedo-files) for detailed node-level performance analysis like memory usage, overactive daemons, and I/O performance.

perf stat for a quick check on hardware event counts and metrics.

Don’t forget the shell’s time command!

ANALYZING PARALLELISM

Shared Memory Parallelism Review

Multiple hardware threads cooperate to solve a problem

Use a parallel language like OpenMP or OpenCL, or

Use a parallel library like TBB

Underlying implementation uses pthreads on linux:

- Pthreads are essentially linux processes sharing an address space
- Linux scheduling decides which hardware threads run which pthreads
- It may be necessary to affinitize (bind) pthreads to specific hardware threads to avoid linux scheduler issues
- Normally we assume that we own some or all of the cores on the machine

Parallel Scaling

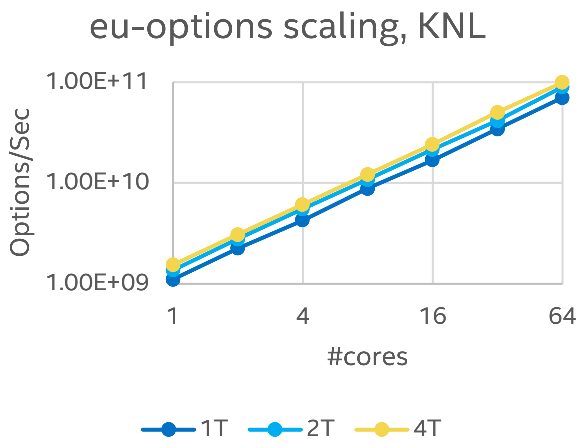
Goal is linear scaling as cores are added:

If one core takes time P, then N cores should take time P/N

Using multiple threads per core can help to hide latency but does not add resources:

2 threads on only 1 core is very different than 2 threads on 2 separate cores

Plot scaling as shown:



Impediments to Parallel Scaling

Algorithmic

- Load Imbalance
- Serial code
- Synchronization

Architectural:

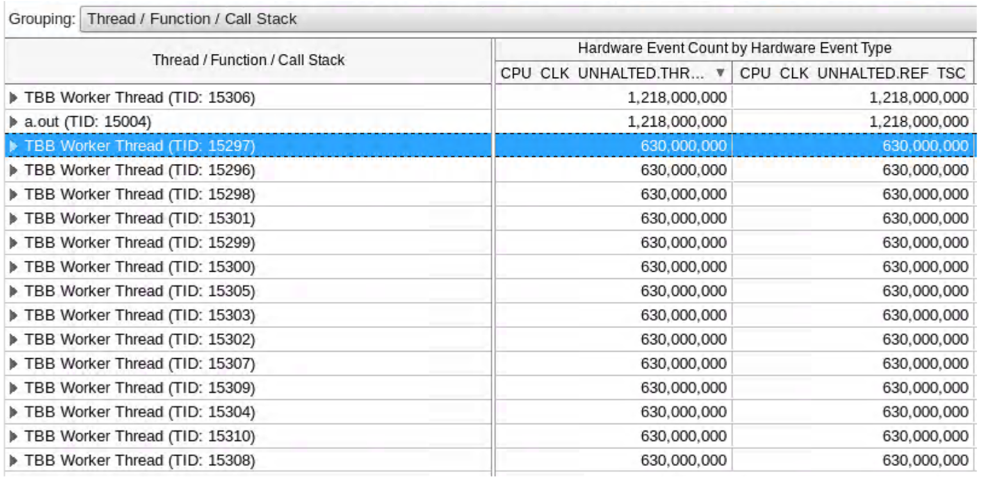
- Cache contention
- Mesh bandwidth
- Memory bandwidth
- NUMA issues

Identifying Load Imbalance

Performance is typically determined by the slowest thread

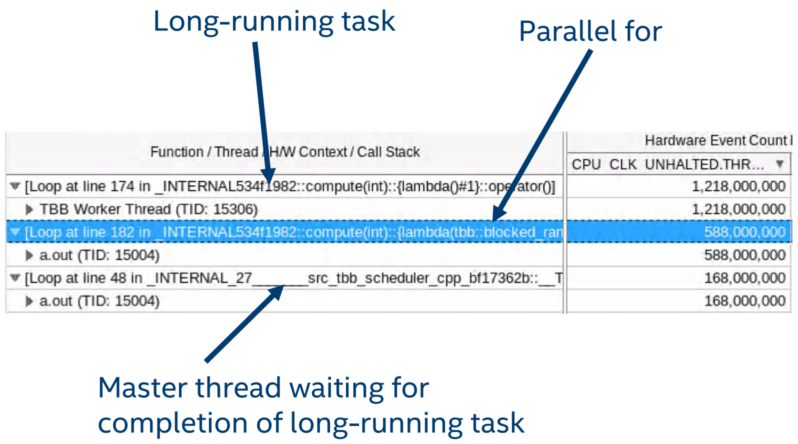
Serial code is an extreme case of load imbalance (only one thread)

Key insight: look at clocks executed per thread.



Grouping: Thread / Function / Call Stack

Thread / Function / Call Stack	Hardware Event Count by Hardware Event Type	
	CPU_CLK_UNHALTED.THR...	CPU_CLK_UNHALTED.REF_TSC
▶ TBB Worker Thread (TID: 15306)	1,218,000,000	1,218,000,000
▶ a.out (TID: 15004)	1,218,000,000	1,218,000,000
▶ TBB Worker Thread (TID: 15297)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15296)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15298)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15301)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15299)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15300)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15305)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15303)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15302)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15307)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15309)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15304)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15310)	630,000,000	630,000,000
▶ TBB Worker Thread (TID: 15308)	630,000,000	630,000,000



Function / Thread / H/W Context / Call Stack	Hardware Event Count
	CPU_CLK_UNHALTED.THR...
▼ [Loop at line 174 in _INTERNAL534f1982::compute(int)::{lambda()#1}::operator()]	1,218,000,000
▶ TBB Worker Thread (TID: 15306)	1,218,000,000
▼ [Loop at line 182 in _INTERNAL534f1982::compute(int)::{lambda(tbb::blocked_ran	588,000,000
▶ a.out (TID: 15004)	588,000,000
▼ [Loop at line 48 in _INTERNAL_27______src_tbb_scheduler_cpp_bf17362b::__T	168,000,000
▶ a.out (TID: 15004)	168,000,000

Synchronization

On Intel® Architecture Processors synchronization is always through memory.

Synchronization implies transfers from cache on one core to cache on another.

Implicit synchronization occurs because of the programming model, e.g., starting and ending a TBB task.

Explicit synchronization occurs due to a programmer action, e.g., an atomic operation.

False synchronization (aka false sharing) occurs when two cores access the same cache line but different parts of that cache line.

Note that cache-to-cache transfers can occur even with read-only data; this is not synchronization but still can have performance implications.

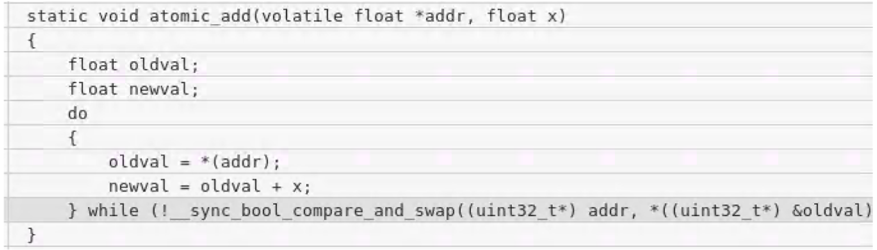
Identifying Synchronization

Significant time in runtime shared objects (modules, e.g. libtbb.so)

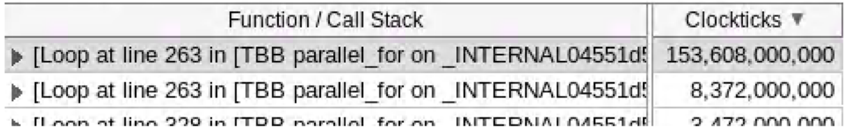
- But that might also be due to load imbalance (waiting for work)

Significant time in or near atomic operations or compare-and-set loops

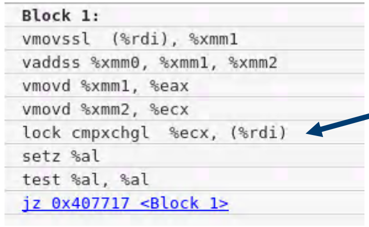
- May be necessary to look at assembly



```
static void atomic_add(volatile float *addr, float x)
{
    float oldval;
    float newval;
    do
    {
        oldval = *(addr);
        newval = oldval + x;
    } while (!__sync_bool_compare_and_swap((uint32_t*) addr, *((uint32_t*) &oldval)
}
```



Function / Call Stack	Clockticks ▼
▶ [Loop at line 263 in [TBB parallel_for on _INTERNAL04551d5	153,608,000,000
▶ [Loop at line 263 in [TBB parallel_for on _INTERNAL04551d5	8,372,000,000
▶ [Loop at line 228 in [TBB parallel_for on _INTERNAL04551d5	3,472,000,000



```
Block 1:
vmovssl  (%rdi), %xmm1
vaddss %xmm0, %xmm1, %xmm2
vmovd %xmm1, %eax
vmovd %xmm2, %ecx
lock cmpxchgl  %ecx, (%rdi)
setz %al
test %al, %al
jz 0x407717 <Block 1>
```

lock prefix is an atomic operation

Bandwidth (KNL)

Memory bandwidth: on Intel® Xeon Phi™ 7250

- DDR: ~90 GB/sec, can saturate with ~10 cores
- MCDRAM: ~490 GB/sec, can saturate with ~64 cores

MCDRAM is asymmetric, read-only BW is ~380GB/sec, write-only is ~200GB/sec

Mesh bandwidth: includes cache-to-cache transfers and memory transfers

- Rule of thumb: ~380 GB/sec max read BW
- We can measure mesh read BW with per-thread hardware events:

```
64*(L2_REQUESTS.MISS + L2_PREFETCHER.ALLOC_XQ) / elapsed_time
```

Bandwidth (Intel® Xeon® processors)

Each socket has its own DDR

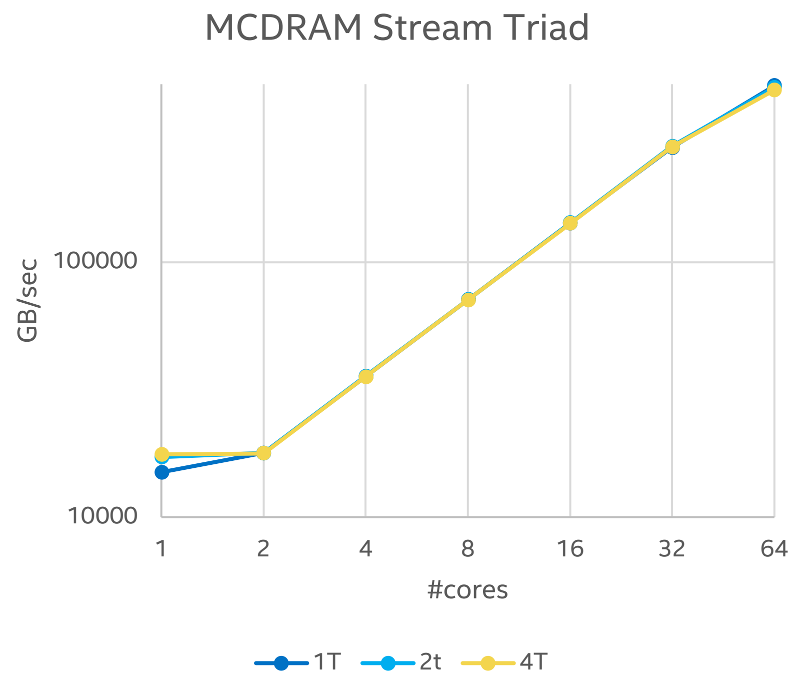
Cross-socket traffic is through QPI/UPI and has much less bandwidth that DDR (NUMA effects)

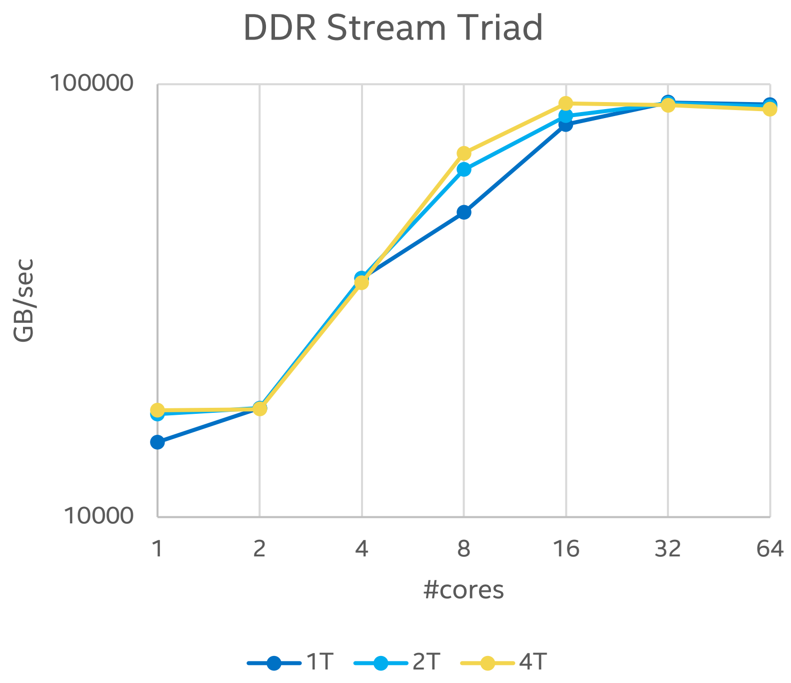
Memory BW is dependent on number of DIMMS and memory frequency. On Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz with 2400MHz DDR4:

Stream Triad 113GB/sec (56GB/sec/socket)

SKX gets ~105GB/sec/socket

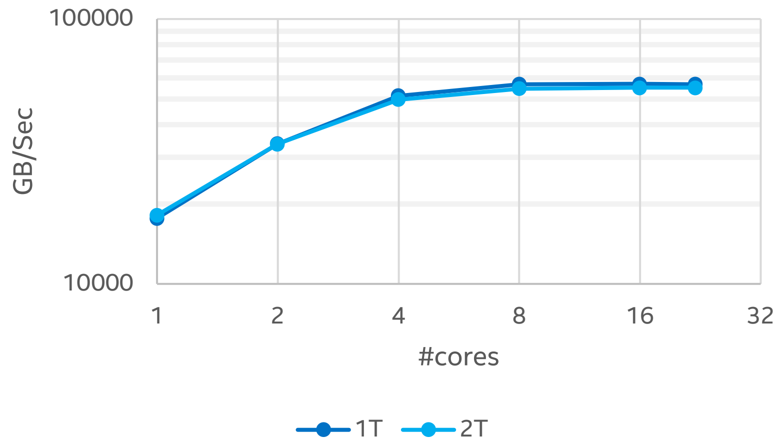
KNL STREAM Benchmark Scaling



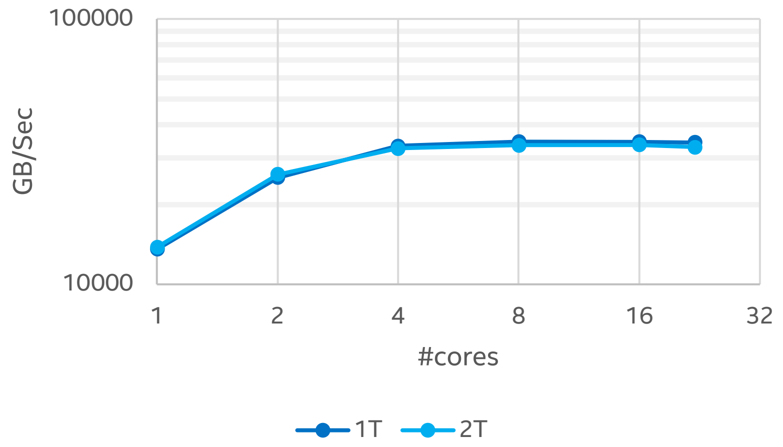


BDW Bandwidth Scaling

One socket, local memory

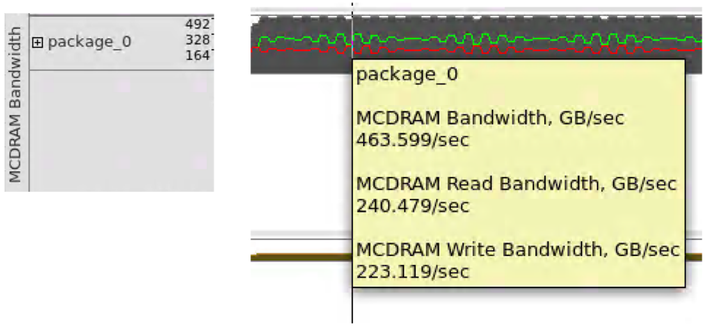


One socket, remote memory



Vtune Amplifier Data, KNL

```
amplxe-cl –collect memory-access
```



package_0

MCDRAM Bandwidth, GB/sec
463.599/sec

MCDRAM Read Bandwidth, GB/sec
240.479/sec

MCDRAM Write Bandwidth, GB/sec
223.119/sec

`-collect memory-access` uses uncore counters at the memory controllers. The data can't be localized to a particular code segment.

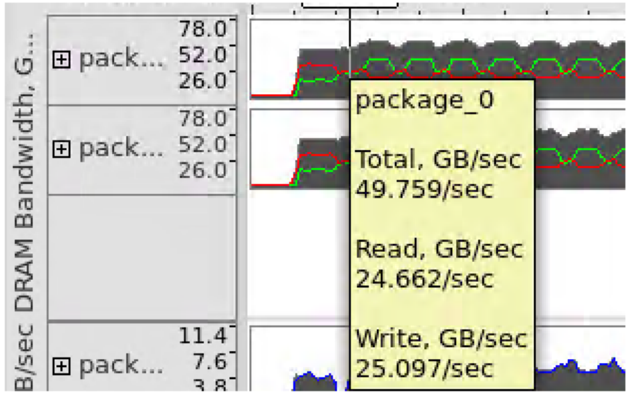
Using core hardware events: `amplxe-cl –collect-with runsa –knob event-config=...`

CPU_CLK_UNHALTED.THREAD	28,798,043,197
CPU_CLK_UNHALTED.REF_TSC	28,802,043,203
L2_REQUESTS.MISS	144,602,169
L2_PREFETCHER.ALLOC_XQ	1,531,807,219
Lines in	1,676,409,388
Elapsed Time	0.321451375
Read BW (GB/sec)	334

Core hardware events can be localized to specific code segments but can measure only read bandwidth and include cache-to-cache transfers. This data is for the STREAM Triad loop only.

Vtune Amplifier Data, BDW

```
amplxe-cl -collect memory-access
```



package_0

Total, GB/sec
49.759/sec

Read, GB/sec
24.662/sec

Write, GB/sec
25.097/sec

Note two packages (sockets)
Data from uncore counters

Using core hardware events: `amplxe-cl -collect-with runsa -knob event-config=...`

CPU_CLK_UNHALTED.THREAD	156,602,234,903
CPU_CLK_UNHALTED.REF_TSC	132,510,198,765
L2_LINES_IN.ALL	1,668,750,061
Time (s)	1.368907012
L2 input BW	78.01845046

L2 input bandwidth for stream triad

Command Line Reports and Pivot Tables

Generate a report with all useful columns:

```
  amplxe-cl -report hw-events -r $1 \
    –group-by=process,package,cpuid,core,thread,function \
        -format=csv -csv-delimiter=comma -inline-mode=off
```

Pivot, summarizing by addition, to see relationships. Examples:

Pivot by function is a function-level profile

Pivot function by thread exposes load imbalance

Pivot process by cpuid exposes multiple processes on same HW thread

Pivot thread by cpuid exposes thread migration

Pivot Table Example

(part of a) Pivot Table of core by process
Entries are CPU_CLK_UNHALTED.REF_TSC (elapsed time)

	eu-options-AVX2	md127_raid10	vmlinux	md127_resync
core_0	2,380,400,000		4,400,000	
core_1	2,292,400,000		6,600,000	
core_10	2,285,800,000		24,200,000	
core_2	2,292,400,000		0	
core_20	2,013,000,000	268,400,000	0	
core_4	2,292,400,000	147,400,000	6,600,000	
Grand Total	99,723,800,000	415,800,000	143,000,000	116,600,000

Note processes md127_* and vmlinux running on same cores as application

Legal Disclaimers

Legal Disclaimers

Optimization Notice