7th Central and Eastern European Software Engineering Conference in Russia - CEE-SECR 2011

October 31 – November 3, Moscow

Three Dimensions of Software Analysis: Performance, Power, Parallelism

Stanislav Bratanov

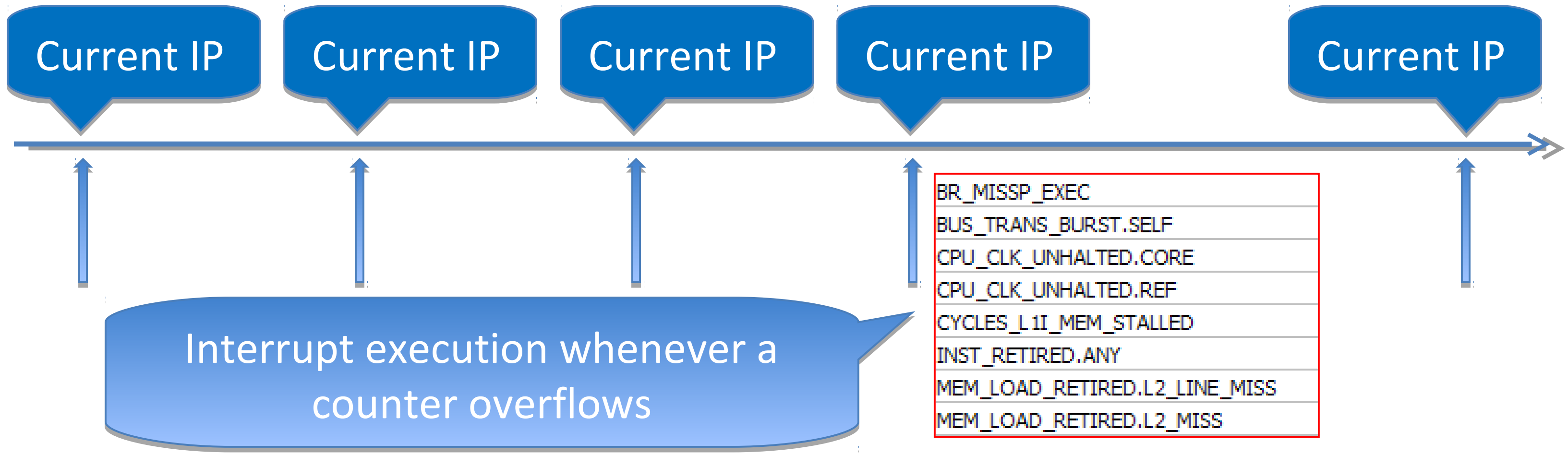
Intel Corporation

Agenda

- Evolution of performance analysis methods
- Statistical Call Tree Model
- Correlated parallel software analysis
- Application-centric power analysis
- Tools to use

Current Technology: Event Based Sampling

- Fetch addresses (IP) correlated with HW events
- Part of Intel® VTune™ for more than a decade
- Reliable, with a good balance between precision and performance
  - Typical overhead of 5%



Event Based Sampling: Misleading

Was it beeping all the time?

About **80%** of time was consumed by the OS kernel

PMU Call Stack	Module	CPU_CLK_UNHALTED…	CPU…
Total		0	100.0%
HalMakeBeep	hal.dll	26966000000	70.1%
KiFastCallEntry	ntkrnlpa.exe	1498000000	3.9%
KiFastSystemCallRet	ntdll.dll	1154000000	3.0%
KiIdleLoop	ntkrnlpa.exe	598000000	1.6%
hook_annotation	tpsstool.dll	564000000	1.5%
RtlEnterCriticalSection	ntdll.dll	340000000	0.9%
RtlLeaveCriticalSection	ntdll.dll	296000000	0.8%

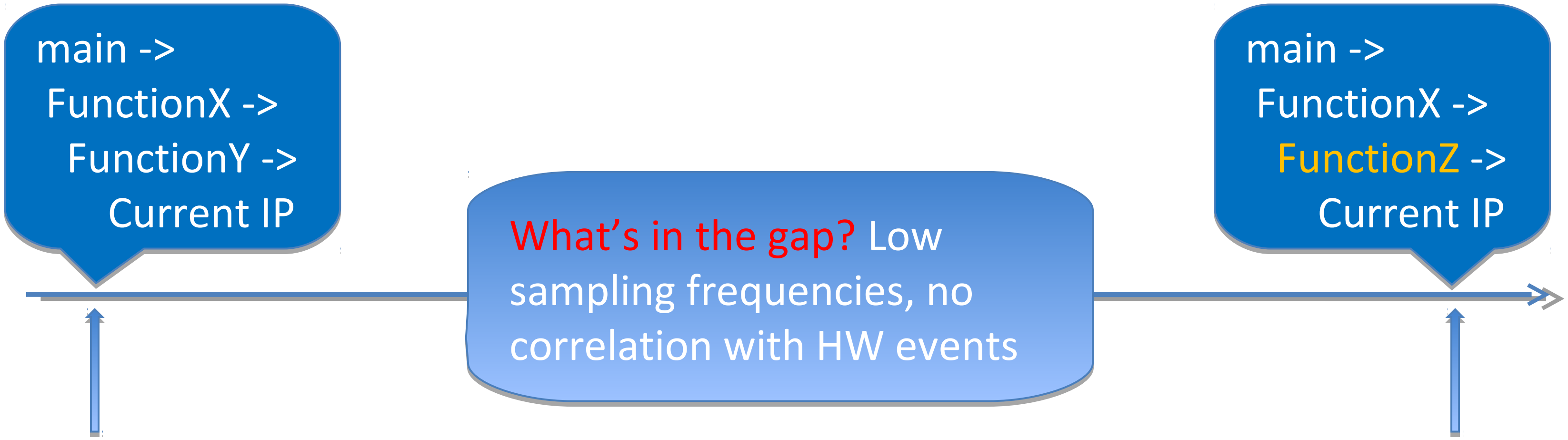
Conclusion: Our code's near perfect. Need to buy a better OS

Our code (*tpsstool*) is just **1.5%**

Even the most popular methods are not perfect

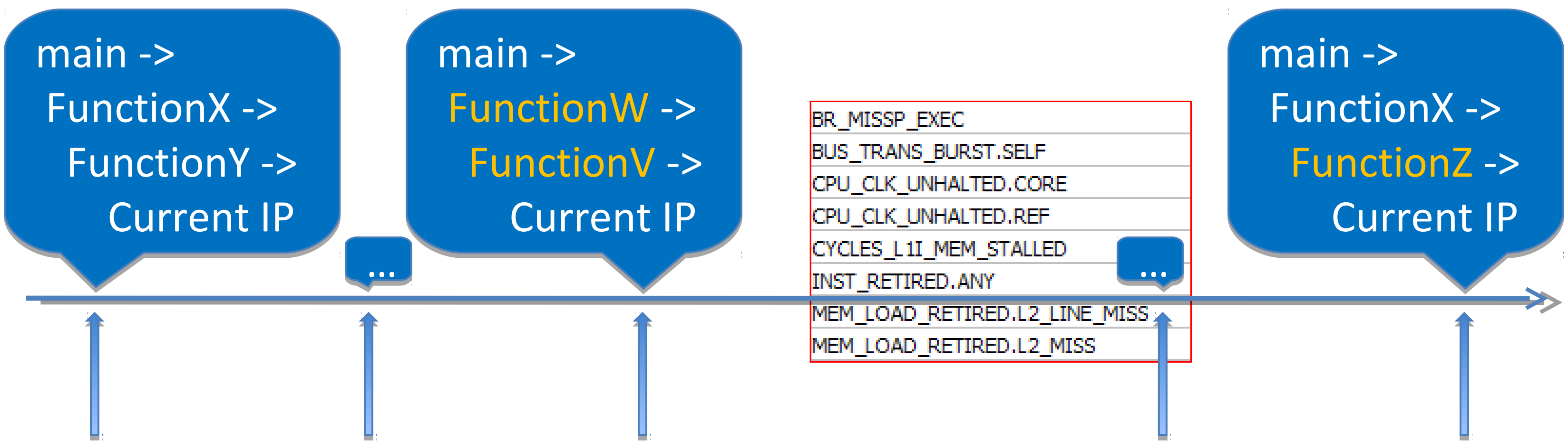
Statistical Call Graph: Hardware Agnostic

- Introduced in Intel® PTU, then in Intel® Parallel Amplifier
- Unique in stack quality and performance
  - typical overhead under 10%
  - but not connected with HW events
  - sampling frequency is too low



EBS + SCG = A Natural Improvement

- Experimental feature of Intel® VTune™ Amplifier XE
  - Employed user-mode SCG technology in kernel mode
  - Overhead is just a few % greater than that of EBS



Statistical Call Tree Model: A Problem Solver

All OS kernel hotspots are combined into a single node (kernel entry)

Result: **9x performance increase** after changing our timing algorithm!

Function - Parent Call Stack	CPU_CLK_UNHALTED.CORE_P ...
⊞ KiFastSystemCallRet	87.0%
⊞ hook_annotation	1.7%
⊞ RtlLeaveCriticalSection	0.9%
⊞ RtlEnterCriticalSection	0.9%

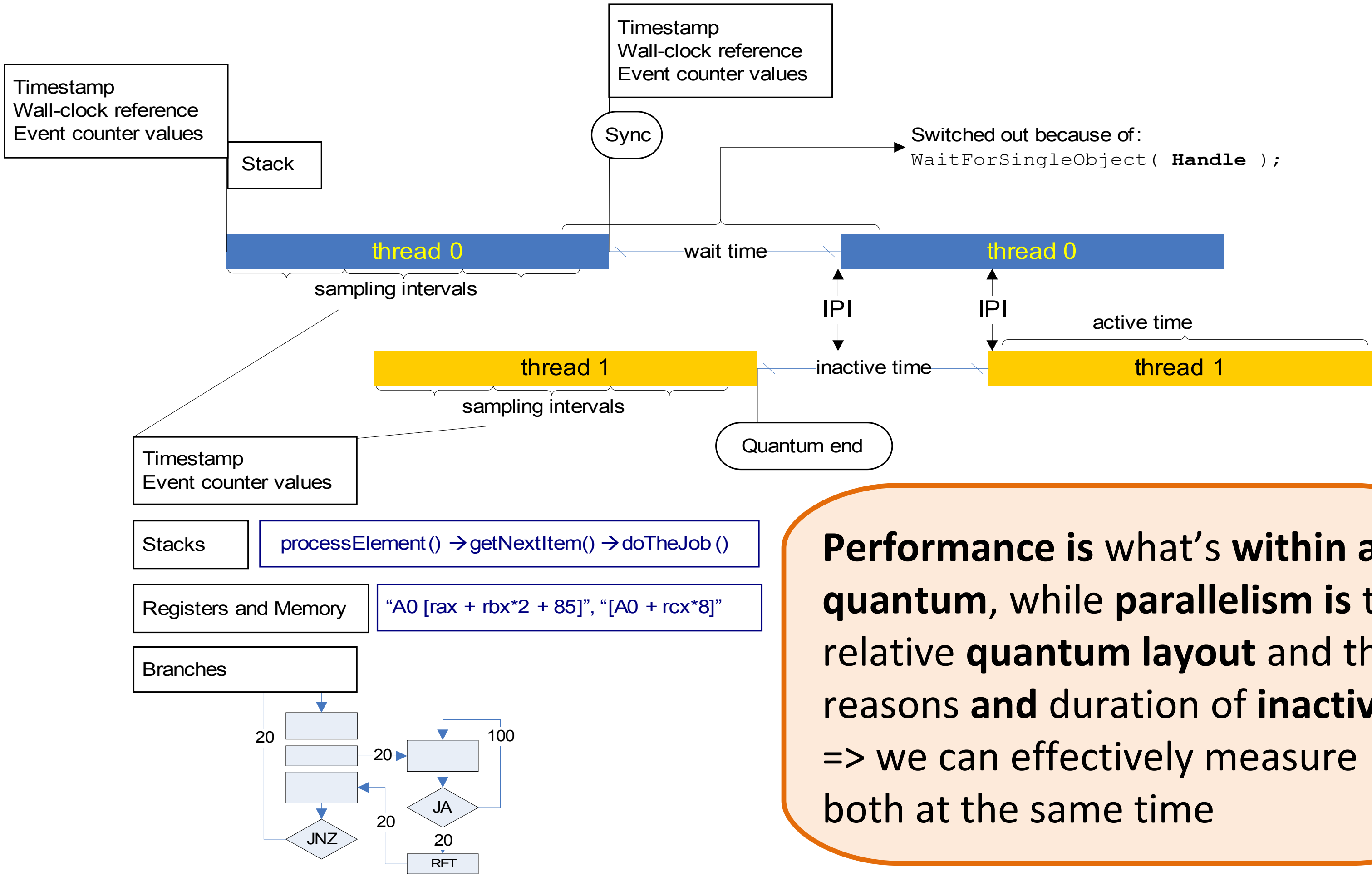
Function - Parent Call Stack	CPU_CLK ...
⊟ KiFastSystemCallRet	87.0%
⊟ ↖ ZwQueryPerformanceCounter ← QueryPerformanceCounter ← sal_system_time	86.6%
⊟ ↖ hook_annotation	85.0%

The majority of the hotspots were called on behalf of our code (*hook_annotation*)…

…primarily, to request the absolute time

It is a step forward in performance analysis

Parallelism: What's Going on in the System?



Performance is what's **within a quantum**, while **parallelism is** the relative **quantum layout** and the reasons **and** duration of **inactivity** => we can effectively measure both at the same time

Everything is interconnected

Parallel Metrics on the Tree: Overview

- Performance and parallelism metrics join naturally

Primary hotspot

Synchronization hotspot (wait-spot)

HW events (e.g., clocks)

Thread contention

OS impact

Time lost on waits

Scheduled off CPU

/Functio... stack	CPU_CLK_UNH… CORE by H/W …	Synchronization Context Switches …	Preemption Context Switches by H/W …	Wait Time by H/W Context	Inactive Time by H/W Context
⊟ quantize_lines_xrpow	3,556,035,917	0	233	0	57,516,084
⊟ ↖ quantize_lines_xr…w_ISO ← count_bits	3,556,035,917	0	233	0	57,516,084
↖ trancate_sma…pectrums ← iteration_loop ← lam	3,180,425,513	0	209	0	51,279,756
↖ bin_search_S…epSize ← trancate_smallspectrums	375,610,404	0	24	0	6,236,328
⊟ KiFastSystemCallRet	3,089,180,223	6,222	244	11,894,526,708	444,816,840
⊞ ↖ NtWaitForSingleObject ← WaitForSingleObjectEx ←	2,993,182,443	5,730	218	11,817,527,328	435,746,772
⊞ ↖ ZwRequestWaitReplyPort ← CsrClientCallServer	62,794,255	454	15	25,823,520	461,976
⊞ ↖ ZwSetEvent ← SetEvent	20,723,639	32	6	50,760,216	8,319,132

We **lost almost half of potential performance** on contention: clocks wasted on contention are comparable with the time of useful work

Major contention on a WaitForSingleObject

Its SetEvent counterpart is not that contended

Fixing Parallel Performance Issues

Primary computation hotspot called in parallel from an OpenMP region

Function / Call Stack	CPU_CLK_UNH… THREAD by H/…	Synchronization Context Switches …
InterpolatorKic<unsigned char,float>::InterpolateN	585,912,744	23
InterpolatorKic<unsigned char,unsigned char>::Do ← _kmp_invoke_microtask ← _kmpc_invoke_	585,912,744	23
_kmpc_invoke_task_func ← _kmp_launch_worker ← BaseThreadInitThunk ← RtlUserThreadSt	523,897,299	23
_kmp_fork_call ← _kmpc_fork_call ← InterpolatorKic<unsigned char,unsigned char>::Do ← C	62,015,445	0
KImage::getData	181,995,760	8
NtWaitForSingleObject	173,576,490	17,315
WaitForSingleObjectEx ← _kmp_launch_monitor ← BaseThreadInitThunk ← RtlUserThreadStart	173,576,490	17,315

Explanation: Excessive OMP barriers because of processing a picture by blocks of lines and parallelizing each block separately:

```
for(i = 0; i < block_no; i++)
{
    #pragma omp parallel for
    for(j = 0; j < lines_in_block; j++)
    {
        /// do processing
    } /// implicit barrier causing contention and overhead
}
```

To do: Use nowait clause or apply parallel_for to the entire picture and use dynamic work scheduling

Major contention on a WaitForSingleObject inside OpenMP that cost us ~**30% of performance loss** (clockticks of the wait / clockticks of the hotspot)

Fixing Parallel Performance Issues

The relative cost of contention on Sleep() is low

Function / Call Stack	CPU_CLK_UNHALT... THREAD by H/W C...	[illegible]ization Contex[illegible]tch ...
⊟NtDelayExecution	286,509,847	26,997
⊞ SleepEx ← _kmp_invoke_microtask ← _kmp_wait_yield_4 ← _kmp_acquire_lock	286,509,847	26,997
⊟NtWaitForSingleObject	248,806,404	7,565
⊟ WaitForSingleObjectEx	241,934,524	7,541
⊟ _kmp_launch_worker	206,421,435	5,982
_kmp_wait_sleep ← _kmpc_invoke_task_func ← _kmp_launch_worker ←	85,531,355	3,023
⊞ _kmp_get_reduce_method ← _kmpc_invoke_task_func	120,890,080	2,959
_kmp_launch_monitor ← BaseThreadInitThunk ← RtlUserThreadStart	22,055,633	1,496
⊞ QThreadStorageData::finish(void * ptr64 * ptr64) ← QMutex::lock(void)	13,457,45[illegible]	63
⊞ RtlDeNormalizeProcessParams ← RtlDeNormalizeProcessParams	6,871,[illegible]	24
⊞InterpolatorPixel<unsigned char>::Do	15,302,5[illegible]	273

Decreased contention and negative performance impact (**down to ~1%**) by:
using a single parallel_for and dynamic work scheduling

The cost of parallelism is uncovered

Power Metrics on the Tree

- Energy and Power registers:
  - we sample them @ context switches and PMIs
    - on Intel® micro-architecture code named Sandy Bridge
    - to use Coulomb counter on Intel® Atom™ processors later

Call tree

Energy (Joules) consumed by each function/call path on a processor core

Energy consumed by the graphics

Energy consumed by the entire processor package

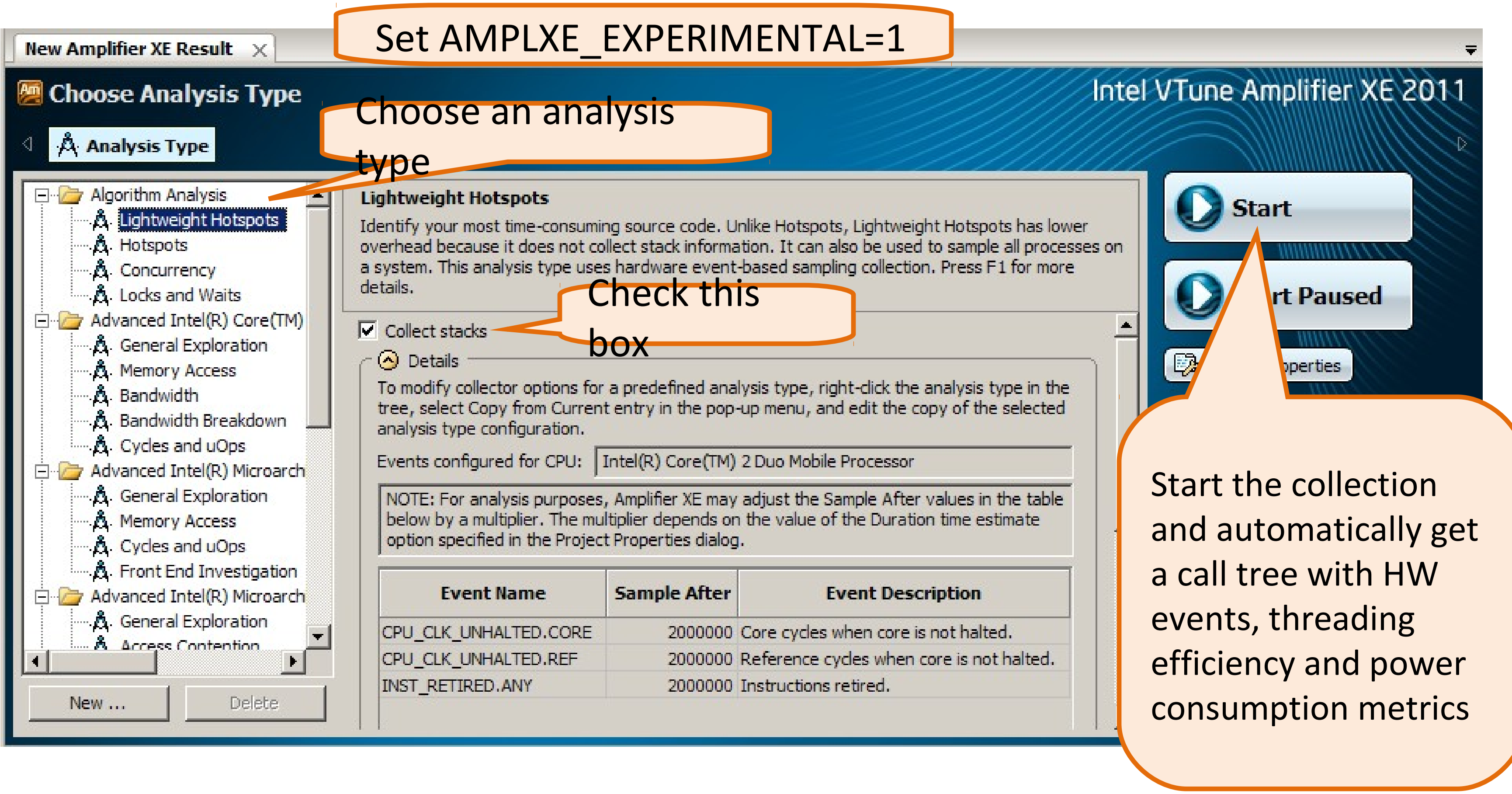
Function / Call Stack	Energy Core by H/W Context	Energy GFX by H/W Context	Energy Pack by H/W Context	CPU_CLK_UNHALTED. THREAD by H/W Co ...	Synchronization Context Switches ...
⊞NtWaitForSingleObject	2,468,224	14,976	2,548,816	173,576,490	17,315
⊟NtDelayExecution	2,103,696	10,176	2,286,704	111,482,970	5,023
⊞ ↖ SleepEx ← _kmp_invoke_microtask ← _kmp_wa	2,103,696	10,176	2,286,704	111,482,970	5,023
⊞NtWaitForMultipleObjects	219,888	1,152	238,000	21,501,501	113
⊞InterpolatorKic<unsigned char,float>::InterpolateN	10,379,264	45,888	11,038,960	585,912,744	23

Execution time

Thread contention

One model fits all

How Can I Enjoy All This Goodness Today?



Easy to pick up for both new and existing users

Summary

- Even the most popular methods are not perfect
- Event-based call tree is yet another advancement in software analysis:
  - it brings to light the software execution logic
  - helps reveal performance inefficiencies and the cost of parallelism
  - correlates performance/power/parallelism metrics
  - is a natural extension of existing methods
- Available as an experimental feature of Intel® VTune™ Amplifier XE: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe