

Vtune Performance Analyzer

About VTune

- VTune Performance Analyzer provides information about execution of your application
- It analyzes without having to recompile your application or linking it with any library
- Helpful in hunting Hotspots in your code
- Useful in tuning your application to maximize the performance

Salient functions of VTune

- Sampling Wizard
- Call Graph Analysis – critical path
- Call List View – time spent by each function , self time, wait time
- Optimization Reports

Sampling Wizard

- The Sampling Summary provides data on the five of the most active functions in the system during data collection.
- It provides details of events that occurred during the runtime of the execution
- Highlights Hotspots in the code.
- Shows per-Core data which helps in finding if any core was under-utilized.

Most Active Functions In Your Application

(Sampling Hotspot Summary by Process)

As your application was running VTune(TM) Performance Analyzer took a periodic sample to see which function was executing. Improving the performance of the most active functions will create the biggest improvement in overall performance.

Function Name (click to view the source)	Percentage of the Process "prog_icc.exe"	Module (click to view the function list)
proc3	98,76 %	prog_icc.exe
fill_array	0,27 %	prog_icc.exe
proc21	0,05 %	prog_icc.exe
proc22	0,04 %	prog_icc.exe
<prog_icc.exe unresolved addresses>	0,02 %	prog_icc.exe
All other functions	0,86 %	View All Modules

Total elapsed time: 34,02 seconds
All other processes consumed 3,16 % of the whole system ([Why is this important?](#))

View [All Processes](#) and their functions
Command executed: /home/richer/dev/vtune/prog_icc.exe

Learn more:

- [Improving performance with compiler optimization switches](#)
- [What causes a large number in the "Percentage of Process" column?](#)
- [How to customize data collection](#)



Addr	Li Nu	Pen & War...	Source	CPU_CLK_UNHALTED.COR
0x52C	56		void proc3(TYPE *c, TYPE *a, TYPE *b, int k) {	0.00%
0x52C	56		proc3: pushl %ebp	0.00%
0x52D	56		movlr %esp, %ebp	
0x52F	56		subl \$0xc, %esp	
	57		int j;	
	58			
0x532	59		for (j=0;j<SIZE2;++j) {	13.36%
0x532	59		movl \$0x0, -4(%ebp)	0.01%
0x539	59		movlr -4(%ebp), %eax	
0x53C	59		cmpl \$0x7d0, %eax	
0x541	59		jge proc3+0x52 (0x804857e)	
0x571	59		incl -4(%ebp)	6.25%
0x574	59		movlr -4(%ebp), %eax	0.98%
0x577	59		cmpl \$0x7d0, %eax	6.00%
0x57C	59		jnge proc3+0x17 (0x8048543)	0.13%
0x543	60		c[j]=a[j]*k+b[j];	86.21%
0x543	60		proc3+0x17: movlr -4(%ebp), %eax	5.77%
0x546	60		movlr 0xc(%ebp), %edx	
0x549	60		flds (%edx, %eax, 4)	0.36%
0x54C	60		movlr 0x14(%ebp), %eax	4.62%
0x54F	60		movl %eax, -12(%ebp)	5.28%
0x552	60		fildl -12(%ebp)	0.03%
0x555	60		fstns -12(%ebp)	6.50%

RVA	Siz	Name	CPU_CLK_UNHALTED.CORE
		...	86.21%
0x52C	0x54	proc3	99.62%
0x580	0x60	proc21	0.05%

Sampling Events

- CPU_CLK_UNHALTED.CORE - Gives the number of clock cycles when the core was not in HALT state
- Instructions Retired – Number of instruction that retire execution
-

Determining Efficiency

- Methods to determine efficiency
 - 1) % Execution stalled
 - 2) Changes in CPI
 - 3) Code Examination

% Execution Stalled

- Helps you understand how efficiently your app is using the processors
- $$\left(\frac{\mu\text{OPS_EXECUTED.CORE_STALL_CYCLES}}{\mu\text{OPS_EXECUTED.CORE_ACTIVE_CYCLES} + \mu\text{OPS_EXECUTED.CORE_STALL_CYCLES}} \right) * 100$$
- Less than 10% stall cycles is good – focus on code reduction
- 10%-50% for client apps – worth investigating stall reduction
- 50%-80% for server apps - worth investigating stall reduction

Changes in CPI

- Another measure of efficiency when comparing 2 runs
- $$\text{CPI} = \text{CPU_CLK_UNHALTED.THREAD} / \text{INST_RETIRED.ANY_P}$$
- Performing compilation time optimizations could reduce the CPI and possibly improve performance

Possible causes of inefficiency

- Cache misses
- NUMA distribution
- Branch Mispredicts
- Front End Stalls
- Address Aliasing
- Long Latency Instructions and Exceptions
- DTLB Misses
- Server/HPC Issues

Cache Misses

- Within a hotspot , estimate the % of cycles due to long latency data access
- For 3rd Level Misses :
$$\left(\frac{(\text{MEM_LOAD_RETIRED.LLC_MISS} * 180)}{\text{CPU_CLK_UNHALTED.THREAD}} \right) * 100$$
- For 2nd level misses :
$$\left(\frac{(((\text{MEM_LOAD_RETIRED.LLC_UNSHARED_HIT} * 35) + (\text{MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM} * 74))}{\text{CPU_CLK_UNHALTED.THREAD}} \right) * 100$$
- If percentage is significant (>20%) , consider reducing cache misses

NUMA Distribution

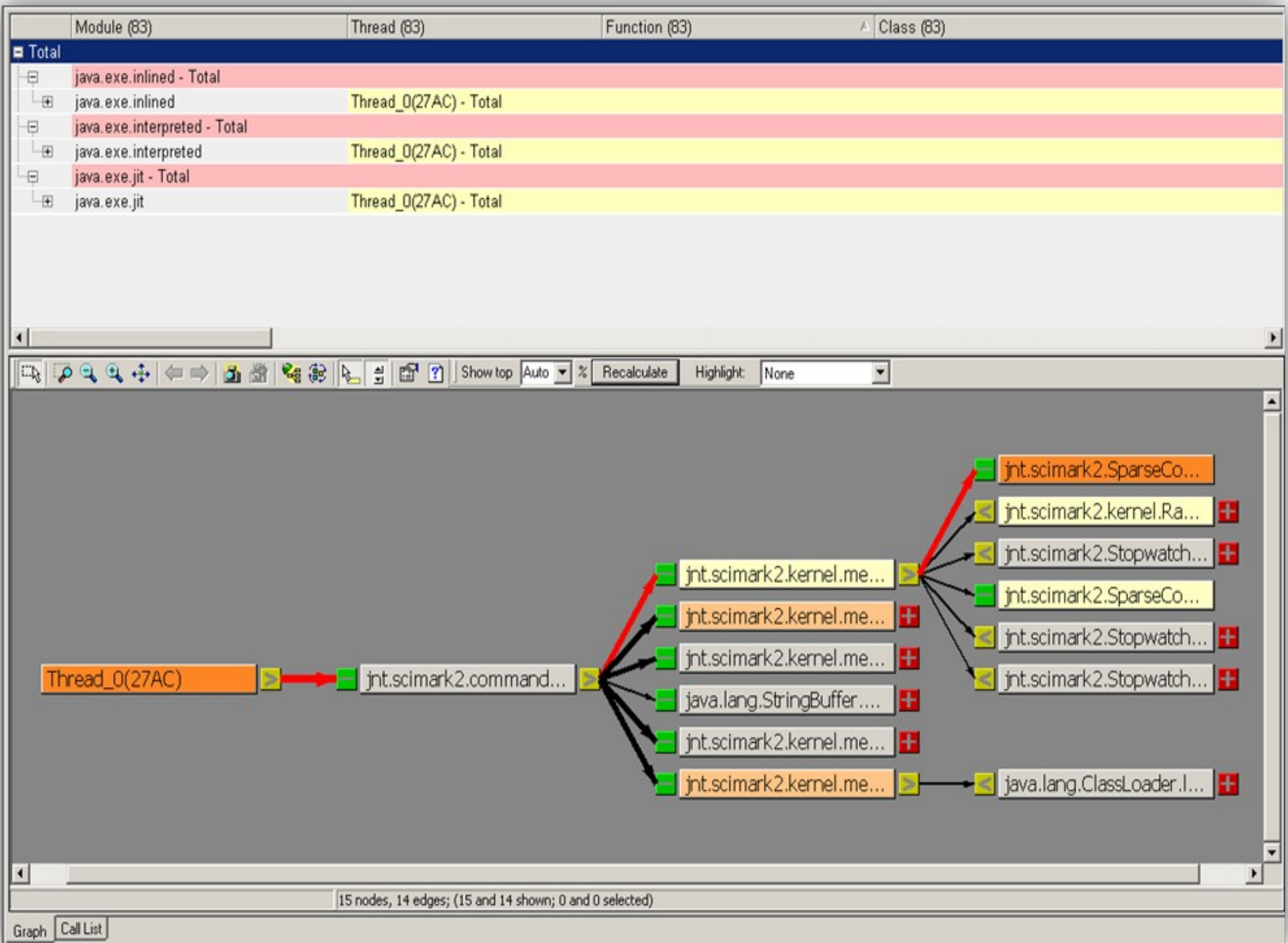
- Remote memory accesses add latency and raise CPI of application
- Determine NUMA distribution for your workload
- %Remote access =
$$\frac{\text{OFFCORE_RESPONSE_0.DATA_IN.REMOTE_DRAM}}{(\text{OFFCORE_RESPONSE_0.DATA_IN.LOCAL_DRAM} + \text{OFFCORE_RESPONSE_0.DATA_IN.REMOTE_DRAM})} * 100$$
- If percentage is significant (>20%) , consider strategies for improving NUMA access : use a NUMA-aware memory allocator, privatize variables. System tuning : ensure memory is balanced across nodes.

Using Compilation Time Report

- Using the Intel Compiler , we can generate compilation reports
- Optimization report indicates if a loop was not vectorized. Steps could be taken to remove the dependencies which were stopping a loop from being vectorized

Call Graph Analysis

- Call Graph determines calling sequences and graphically displays the critical path.
- Shows the context of bottleneck



▪

Thank you

References

- VTune manual to optimize Intel i7 Core Processor
- Intel Vtune Manuals