White Paper

Using Intel® VTune™ Performance Analyzer Events/ Ratios & Optimizing Applications

by: Rama Kishan Malladi

Introduction

When developing, optimizing or tuning applications for performance on Intel architecture, it is important to understand the processor microarchitecture. Having said that, an insight into how the applications are performing on the microarchitecture is gained through performance monitoring. The Intel® VTune™ Performance Analyzer provides an interface to monitor performance of the processor and gain insights into possible performance bottlenecks. In this document, we will illustrate some code samples and measure VTune™ events & ratios for monitoring processor performance.

This document provides an overview of the 45nm Hi-k next generation Intel® Core™ microarchitecture, the details of which can be obtained from the processor manuals. The target audience for this document is primarily VTune™ users who would want more information on how to use the VTune™ tool more effectively for performance monitoring, analysis and code optimizations. This is not intended to be a stand-alone document and it is recommended to be used in conjunction with the processor optimization manual and VTune™ help documentation.

Table of Contents

Assumptions

It is assumed that the reader knows the basic operations in VTune™ of sampling an application, creating a sampling activity, and selecting events and ratios from the GUI. If you would like to familiarize yourself with VTune™, you can review the tutorial shipped with the product. If you want to obtain an evaluation of VTune™ product, you can download the same from the product website http://www.intel.com/software/products.

1. Overview of Intel Processor Architectures

Over the last decade, Intel has released various processors and associated features that enhance application performance with each generation of the release. Some of the features that are relevant to software optimization are mentioned below. These apply to Intel Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo and Intel® Core™2 Quad, Intel® Core™ i7, Pentium® 4, Intel® Xeon®, Pentium® M and IA-32 processors with multi-core architectures.

- SIMD instruction extensions including MMX™ technology, Streaming SIMD Extensions (SSE), SSE2, SSE3, SSSE3, SSE4.1, SSE4.2.
- Microarchitectures that enable executing instructions with high throughput, high speed cache and ability to fetch data with high speed system bus.
- Intel® Extended Memory 64 Technology (Intel® EM64T).
- Intel® processors support Hyper-Threading (HT) Technology.
- Multi-core supported in Intel® Xeon, Core™ Duo, Core™2 Duo, Core™2 Quad, Core™ i7, Intel Pentium® D and Pentium® Extreme Edition processors.

Intel Pentium 4, Pentium D, Extreme Edition processors are based on Intel NetBurst® microarchitecture. The latest Core™2 Duo and Xeon® processors are based on the Intel Core™ microarchitecture which provides additional enhancements for performance and power efficiency as below:

- **Intel Wide Dynamic Execution:** Enables each processor core to fetch, dispatch, execute with high bandwidths and retire up to four instructions per cycle.
- **Intel Smart Cache:** Delivers higher bandwidth from the second level cache to the core, optimal performance and flexibility for single-threaded and multi-threaded applications.
- **Intel Smart Memory Access:** Prefetches data from memory in response to data access patterns and reduces cache-miss exposure of out-of-order execution.
- **Intel Advanced Digital Media Boost:** Improves most 128-bit SIMD instructions with single-cycle throughput and floating-point operations.

2. Application Tuning

As a rule-of-thumb, it is recommended that performance tuning of applications be done in a top-down approach as shown in Figure 1. This approach provides the best gains with minimum required effort to tune the application. Table 1 gives an overview of the Return on Investment at each layer.

As indicated in Figure 1 and Table 1, the approach to performance tuning is to first focus on system level, then application level and then microarchitecture/ processor level tuning. The ROI at each of these levels justifies the approach. In this document we focus on microarchitecture level tuning using Intel VTune™ Performance

Analyzer product and look at usage of events and ratios. We will also briefly discuss system and application level tuning as follows.

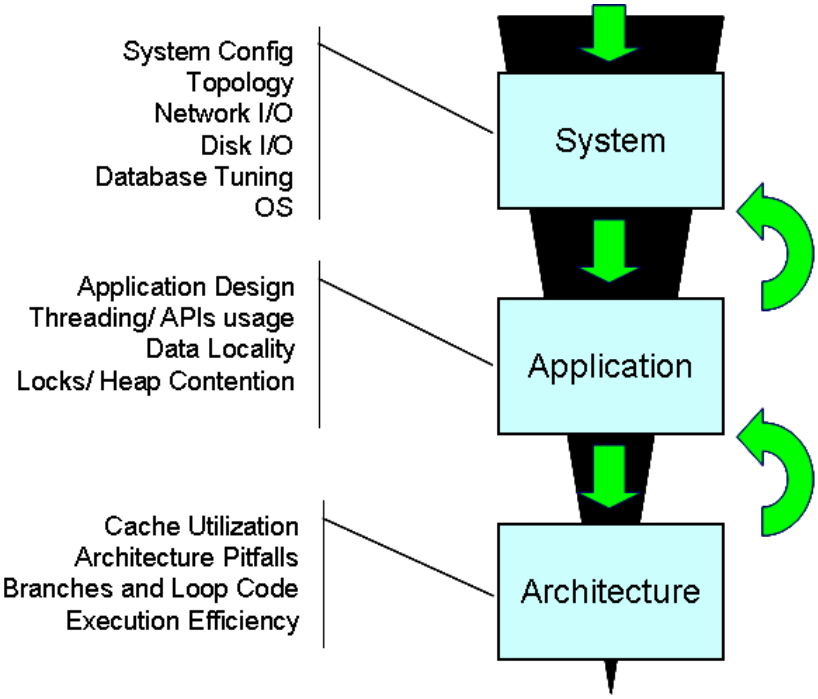


Figure 1: Top-Down Approach for Application Tuning

Tuning Order	Tuning Level	Goals	Key Areas to Investigate	Return on Investment
1	System	Improve how the application interacts with the system.	• Network Problems • Disk Performance • Memory Usage	High
2	Application	Improve the application algorithm.	• Locks, Heap Contention • Threading Algorithm • Build, Compiler Options • Process Parallelism	Medium
3	Microarchitecture	Improve how the application runs on the processor.	• Architecture Coding Pitfalls • Data/ Code Locality (Cache) • Data Alignment	Low

Table 1: Performance Optimization Strategy

A typical question to ask before beginning system level tuning is "what do I know about the sub-system?" This helps you understand on what machine you are executing your workload, the expected performance which further helps you nail down the problem. For example, an I/O intensive workload might get gain in performance from tuning the I/O controller and corresponding I/O devices. Now, to monitor which system

component is being stressed, you would want to use system level analysis tools like Microsoft* Windows* Perfmon* tool. Some of the typical counters that you could monitor in Perfmon* would be memory usage, CPU Utilization, network traffic and disk traffic. A detailed analysis can be done by plotting a throughput curve as indicated in Figure 2 (self-explanatory).

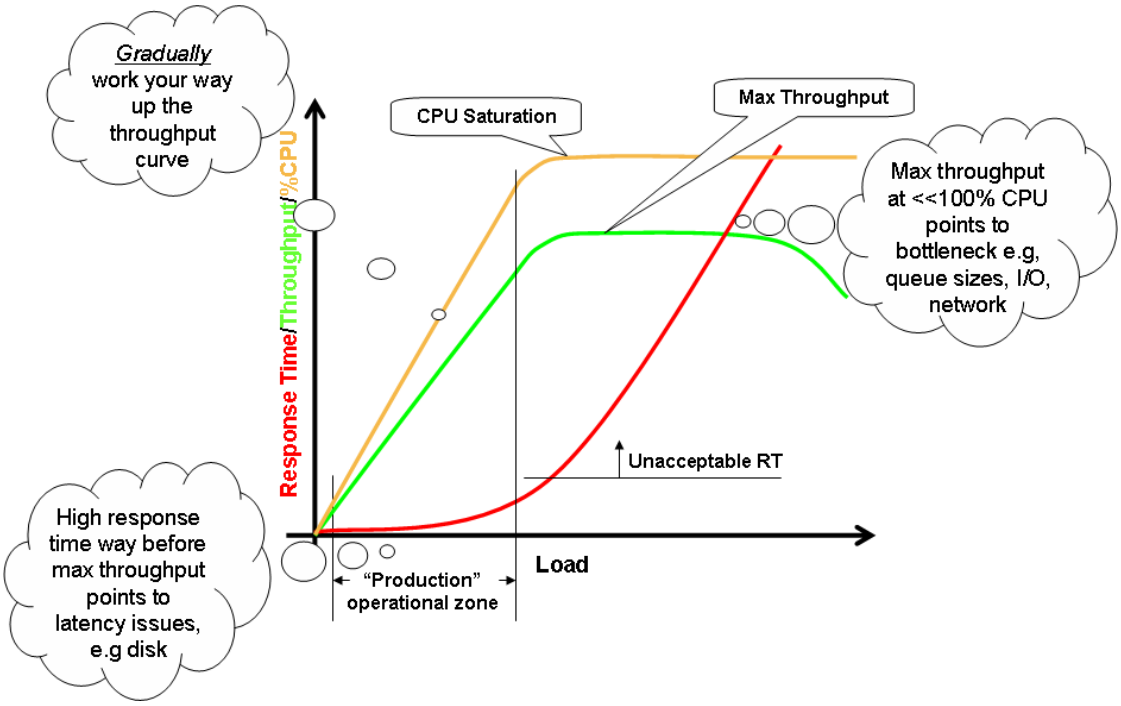


Figure 2: System Level Analysis - Throughput Curve

Now, for application level tuning, in addition to trying the techniques described in Table 1, you could also try the Intel® Software Products available at http://www.intel.com/software/products. These include:

- **Intel® C++ and FORTRAN Compilers:** Generate highly optimized executable code for Intel® 64 and IA-32 processors. Use the compiler optimizations to get the best performance.
- **Intel® VTune™ Performance Analyzer:** Collects & displays Intel architecture specific performance data from system-wide to specific source lines.
- **Intel® Performance Libraries:** Consists of set of software libraries optimized for Intel processors. These include the Intel® Math Kernel Library (Intel® MKL) and the Intel Integrated Performance Primitives (Intel® IPP).
- **Intel® Threading Tools:** These tools help debug and optimize threaded code performance. They are the Intel® Thread Checker and Thread Profiler.
- Intel Performance Tuning Utility & several other tools posted on www.whatif.intel.com.

Note that, tools help you identify and understand application performance issues. After which, you can try a different algorithm/ implementation to mitigate the performance issue. Also, it is recommended to try thread and process parallel implementations for your application to improve multi-core processor utilization.

3. Using VTune™ Events / Ratios

The Intel VTune™ Performance Analyzer is a powerful software-profiling tool available on both Microsoft* Windows* and Linux* OS. VTune™ helps you understand the performance characteristics of your software at all levels: system, application and microarchitecture. The main features of VTune™ are sampling, call graph and counter monitor. In this document we would specifically look at the event based sampling feature of VTune™ and how to choose these VTune™ events and ratios for monitoring performance. For details on the other features and how to use the tool, you would want to review the VTune™ documentation.

While many of us may have used VTune™ to sample an application, we may not have necessarily used the several events and ratios that are available in the tool. The most commonly used events are clockticks and instructions retired which are selected by VTune™ when you create a new activity. In this document we try to address an important and a common question asked by VTune™ users on "Which events and ratios to use?" We will review some sample code examples and collect event ratios and interpret the results. Below are the event ratios that we will review in this document:

1. Cycles per Retired Instruction (CPI)
2. Parallelization Ratio
3. Modified Data Sharing Ratio
4. L2 Cache Miss Impact
5. Branch Misprediction Ratio
6. Bus Utilization Ratio

Event ratios provide useful insight into performance issues. Each processor has its own set of predefined event ratios that it can monitor. You can define event ratios to meet your specific tuning needs. Several other ratios that you could look at are floating point ratios, DTLB miss rate, length changing prefix stall, partial register stall, speculative execution efficiency, power management ratios, software prefetching ratios and several other stall ratios. A complete list is available in the 'Processor Events and Advice' section of VTune™ help documentation.

3.1 VTune™ Configuration

For data collection in this whitepaper, VTune™ was configured to "event based sampling" with calibration selected. The sampling was allowed to be done until the application execution terminates and no delay sampling was used. The application to launch is a console application of the source code shown in the sections to follow. In each of the sampling activities, collect the above 6 event ratios and observe which of the event ratios are high to investigate the performance issue. The code shown will also give you an insight into the performance problems. Note, the code samples are simple and are meant for "driving the point home".

Description of System Under Test (SUT): The SUT was a machine with Intel Core™2 Quad Processor 2.66GHz frequency and 4GB RAM. The processor was built on 45-nm manufacturing process technology. The OS was Microsoft* Windows* XP SP2 and the tool suite was Visual Studio* 2005 (Visual C++ Compiler v8), Intel C++ Compiler v10.1, Intel VTune™ Performance Analyzer v9.0 Update9.

Please note that the SUT is a single socket machine and the event ratios & performance data measured for the code samples are applicable to this machine. These event ratios would have to be modified appropriately for a multi-socket SUT or a machine with different processor architecture.

3.2 Cycles per Retired Instruction (CPI)

CPI is one of the most commonly used ratios. This ratio is also referred to as cycles per instructions: CPI. When you want to determine where to focus your performance tuning effort, CPI is the first ratio to check. High CPI may indicate that instructions require more cycles to execute than they should. In this case there may be opportunities to modify your code to improve efficiency with which instructions are executed in the processor. CPI definition for **45nm Hi-k next generation Intel® Core™ microarchitecture** is as below:

45nm Hi-k next generation Intel® Core™ microarchitecture

CPI Equation: CPU_CLK_UNHALTED.CORE / INST_RETIRED.ANY
Ideal CPI: 0.25

Please note, the event naming convention in 45nm Hi-k next generation Intel® Core™ microarchitecture has changed from that of Intel NetBurst™ microarchitecture. The ideal CPI is 0.25 because the processor can retire 4 instructions per cycle. When optimizing an application, we should try to get as low CPI as feasible.

3.3 Parallelization Ratio

Parallelization ratio can be used to measure the amount of parallel execution in your application. If you are measuring a single-threaded application, this value should be close to 0. If you are measuring for a multi-threaded application, this value should be close to 1. If the value is less than 1 then it means the application suffers from multi-threading inefficiencies, i.e., sequential execution and imbalanced threads.

45nm Hi-k next generation Intel® Core™ microarchitecture

Equation: 1 – (CPU_CLK_UNHALTED.NO_OTHER / CPU_CLK_UNHALTED.BUS)
Ideal: 1

3.4 Modified Data Sharing Ratio

This ratio calculates the amount of modified data sharing between two threads. It can happen, if one thread writes to a cacheline, while other thread reads or writes to the same cacheline (typically at a different offset in the cacheline). Modified data sharing leads to cacheline ping-pong and it is recommended to avoid it.

45nm Hi-k next generation Intel® Core™ microarchitecture

Equation: EXT_SNOOP.ALL_AGENTS.HITM / INST_RETIRED.ANY
Ideal: 0

Figure 3 below shows sample pseudo-code for which we would measure the Parallelization ratio and the Modified Data Sharing ratio. We will look at the values of these ratios and understand the performance issue, fix it and again measure the values to check if the issue is resolved.

In the sample code in Figure 3, we have used the OpenMP# directives that are supported by the Intel C++ compiler. The "#pragma omp parallel for" statement tells the compiler to distribute the "for" loop execution on the multiple processors available on the system. In this sample test, the number of processors was 4.

Execute the code shown in Figure 3 using VTune™ and collect profile for Parallelization and Modified Data Sharing ratios. The profile for this code execution is shown in Table 2, when using the OpenMP "dynamic" scheduling for parallelizing the "for" loop. Table 3 shows the profile collected for the scenario when using the OpenMP "static" scheduling.

www.openmp.org

```
#pragma omp parallel for private(i), shared(s,a) schedule(dynamic)
for(i=0;i<CONST_M;i++)
{
      s[i] = s[i] + a[i];
}
```

Understand the impact of using OpenMP "static" or "dynamic" scheduling

Figure 3: Code demonstrating use of parallelization and modified data sharing ratios

CPU_CLK_UNHALTED. BUS events	CPU_CLK_UNHALTED.NO_ OTHER events	Parallelization Ratio	Clocks per Instructions Retired - CPI	EXT_SNOOP.ALL_ AGENTS.HITM	Modified Data Sharing Ratio
2017200000	0	1.000	4.109	103100000	0.028

Table 2: Parallelization, CPI and modified data sharing ratios when using OpenMP "dynamic" scheduling

CPU_CLK_UNHALTED. BUS events	CPU_CLK_UNHALTED.NO_ OTHER events	Parallelization Ratio	Clocks per Instructions Retired - CPI	EXT_SNOOP.ALL_ AGENTS.HITM	Modified Data Sharing Ratio
43600000	0	1.000	1.273	0	0

Table 3: Parallelization, CPI and modified data sharing ratios when using OpenMP "static" scheduling

As it can be observed from the tables 2 and 3, usage of "dynamic" and "static" scheduling has huge impact on the CPI and Modified Data Sharing ratio. The Parallelization ratio doesn't change significantly because in both "static" and "dynamic" scheduling, the execution is in parallel (so the ratio is = 1). Table 4 shows the profile when running a serial version of the code shown in Figure 3 by setting the environment variable OMP_NUM_THREADS=1 and using "dynamic" scheduling.

CPU_CLK_UNHALTED. BUS events	CPU_CLK_UNHALTED.NO_ OTHER events	Parallelization Ratio	Clocks per Instructions Retired - CPI	EXT_SNOOP.ALL_ AGENTS.HITM	Modified Data Sharing Ratio
12800000	12800000	0.000	1.32	0	0

Table 4: Profile when executing serial version of the application (OMP_NUM_THREADS=1)

The profile in Table 4 clearly indicates that the serial version does not suffer from the data sharing problem which is expected as there is no cache thrashing that happens when executing the serial version. But the parallelization ratio is 0 which demonstrates that the application is running serial as expected when executing the serial version.

Thus the parallelization, CPI and modified data sharing ratios can be used effectively to determine if the execution on a multi-core processor is efficient and how the same can be improved further.

Why do "dynamic" and "static" scheduling clauses in OpenMP give this performance problem?

First, in both these schemes, the loop iterations are spread across multiple processors to execute the body of the "for" loop in parallel. However, with "dynamic" scheduling, the iterations are distributed to threads (in this

case, 1 iteration computation to each thread) as the threads request them. In "static" scheduling the iteration space is divided into chunks (in this case approximately equal in size) and distributed to the threads. Now, how would this cause a performance problem? Clearly, parallelization ratio (=1.0) indicates that processors are busy executing the "for" loop code in parallel when using both "static" and "dynamic" scheduling clauses.

Doing a little more deep-dive on this performance difference, you would realize that when using the "dynamic" scheduling clause, processors executing the "for" loop iterations share the same cacheline when operating on consecutive values in the array "s" (which is getting modified). This leads to cacheline invalidation on every processor when another processor modifies the same cacheline i.e., "false sharing". Correspondingly, you note that CPI is higher and modified data sharing ratio is non-zero for the case when using "dynamic" scheduling.

3.5 L2 Cache Miss Impact

L2 Cache is the second-level cache on the processor and it is generally the cache that is in between the L1 cache and the main RAM memory. The L2 cache is typically much larger than the L1 cache and hence can accommodate many more values as compared to L1. But, equally, the L2 cache latency is higher as compared to L1 and hence a miss from L1 would result in L2 access and some associated latency.

A "miss" means that the value that you are trying to fetch is not available in the location intended (cache or memory) and hence should be fetched from the next level. L2 Cache miss impact is much higher since the "miss" would typically result in access from external RAM memory. Other associated issues with L2 cache are the modified data sharing and the bus bandwidth utilization aspects (both of which are discussed in this paper). Figure 4 below shows a code example for 2D matrix-matrix multiply. Let us measure one of the L2 cache miss events for this code as defined below in VTune™.

45nm Hi-k next generation Intel® Core™ microarchitecture

Event: MEM_LOAD_RETIRED.L2_LINE_MISS
Ideal: 0

```
void multiply_d(double a[][NUM], double b[][NUM], double c[][NUM])
{
        int i,j,k;
        double temp;
        for(i=0;i<NUM;i++) {
          for(j=0;j<NUM;j++) {
                for(k=0;k<NUM;k++) {
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
                  }
                }
        }
}
```

Understand how this code behaves and uses cache and memory bandwidth!

Figure 4: Code example to illustrate L2 cache miss impact

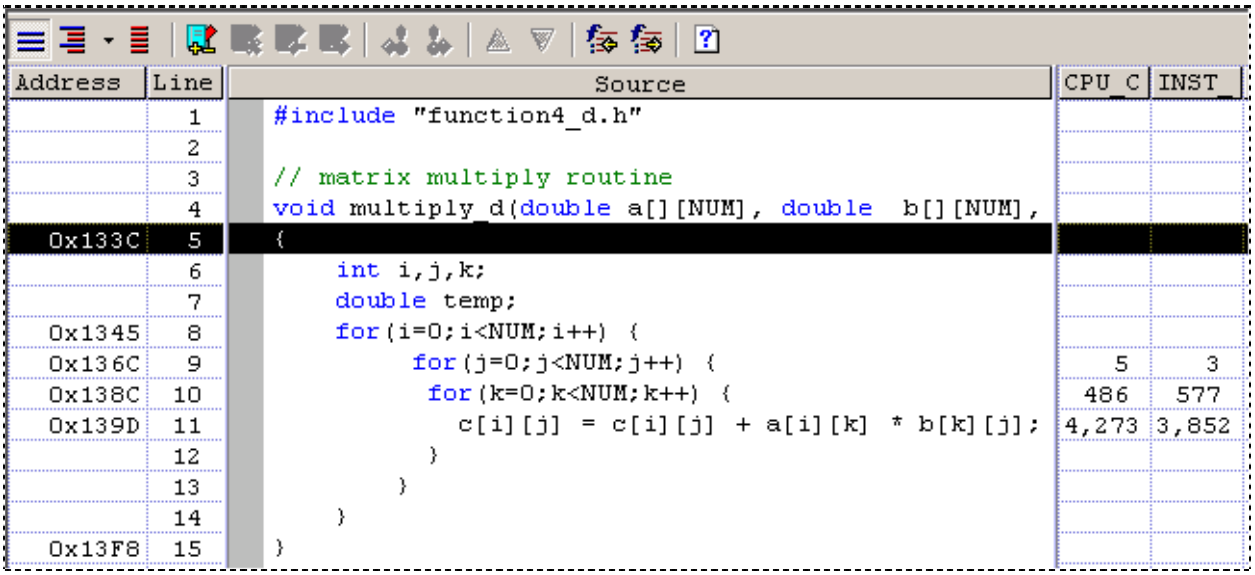
Table 5 shows event #s measured using VTune™ indicating L2 cache miss impact is high (MEM_LOAD_RETIRED event #s are non-zero). Revisiting the code, you would find that the problem is with the access pattern of the array "b[][]" in the innermost loop where we skip in memory trying to fetch values which are not contiguous in memory (matrix elements in a column are not stored contiguously in memory for "C/C++" language).

Figure 5 illustrates the source-level performance analysis view of the 2D matrix-matrix multiply algorithm. This view is supported in VTune™, which allows developers to identify performance issues at various levels and at the end, drill down to the source-level view to pin-point the source that needs the changes!

The performance of the code can be improved by interchanging the inner loop indices "j" and "k" so that the memory access is contiguous and correctness is maintained as illustrated in Figure 6. The new event #s then measured using VTune™ are shown in Table 6. Typically, loop interchanging optimization is automatically done by the compiler (very rarely needing the developer to implement it manually).

CPU_CLK_UNHALTED. CORE events	MEM_LOAD_RETIRED.L2_ LINE_MISS events	Clocks per Instructions Retired - CPI
6072000000	200000	2.702

Table 5: VTune™ performance data illustrating L2 cache miss impact



Address	Line	Source	CPU_C	INST_
	1	#include "function4_d.h"		
	2			
	3	// matrix multiply routine		
	4	void multiply_d(double a[][NUM], double b[][NUM],		
0x133C	5	{		
	6	int i,j,k;		
	7	double temp;		
0x1345	8	for(i=0;i<NUM;i++) {		
0x136C	9	for(j=0;j<NUM;j++) {	5	3
0x138C	10	for(k=0;k<NUM;k++) {	486	577
0x139D	11	c[i][j] = c[i][j] + a[i][k] * b[k][j];	4,273	3,852
	12	}		
	13	}		
	14	}		
0x13F8	15	}		

Figure 5: Source-level view of the 2D matrix-matrix multiply algorithm in VTune™

```
for(i=0;i<NUM;i++) {
  for(k=0;k<NUM;k++) {
        for(j=0;j<NUM;j++) {
              c[i][j] = c[i][j] + a[i][k] * b[k][j];
         }
        }
}
```

Interchanged Loop...optimized!

Figure 6: Loop interchange for optimizing L2 cache access

CPU_CLK_UNHALTED.CORE events	MEM_LOAD_RETIRED.L2_LINE_MISS events	Clocks per Instructions Retired - CPI
1713000000	0	0.757

Table 6: VTune™ performance data illustrating L2 cache miss reduction by loop Interchange optimization

Investigating this performance issue further, you would find that the problem with the original code is that the cachelines are not used effectively (since the access pattern wasn't sequential). While in the optimized case, the access is sequential. The optimized code performs better because the cachelines (which are prefetched) are not evicted. As compared, in the original code cachelines get evicted.

3.6 Branch Misprediction Ratio

Branch prediction logic is a part of the processor core, which is responsible for predicting if a sequence of instructions is more likely to be executed and speculatively stream instructions into the processor pipeline. Branch prediction is very useful when the prediction is correct and the processor pipeline is busy/ full. The performance impact of branch prediction occurs when it is wrong. That is, when "misprediction" occurs.

The impact of misprediction is due to the fact that the pipeline which was speculatively filled by probable instructions has to be flushed and new instructions should be streamed (fetched, decoded). The pseudo-code shown in Figure 7 illustrates how branch misprediction can occur when the "data" vector is random initialized.

45nm Hi-k next generation Intel® Core™ microarchitecture

Equation: BR_INST_RETIRED.MISPRED / BR_INST_RETIRED.ANY
Good: ~0

```
for (i=0; i < ITERS; i++)
{  data[i] = rand()%4;
}


for (j=0; j < REPEAT; j++)
for (i=0; i < ITERS; i++)
{
  if (data[i] == 0)
  { f += (12345.67 + (float)i);
  }
  else if (data[i] == 1)
  { f += (23456.78 + (float)i);
  }
  else if (data[i] == 2)
  { f += (34567.89 + (float)i);
  }
 else if (data[i] == 3)
  { f += (45678.90 + (float)i);
  }
}
```

Understand how this loop behaves when data is random or not!

This example shows data set to random.

Figure 7: Pseudo-code for illustrating branch misprediction.

Table 7 shows the VTune™ profile for the case when misprediction is high. Table 8 shows the ratios when all the elements in the "data" vector are initialized to the same value and misprediction is low.

BR_INST_RETIRED. ANY events	BR_INST_RETIRED. MISPRED events	Clocks per Instructions Retired - CPI	Branch Misprediction Ratio
2125000000	465700000	1.483	0.219152941

Table 7: Branch misprediction ratio is high for a randomly-branching code

BR_INST_RETIRED. ANY events	BR_INST_RETIRED. MISPRED events	Clocks per Instructions Retired - CPI	Branch Misprediction Ratio
1501000000	0	0.462	0

Table 8: Branch misprediction ratio is low when branches are predicted correctly (best case shown)

From the tables 7 and 8, it can be observed that the branch misprediction ratio can be used to identify issues related to execution of "branchy" code in an application.

Using the profile-guided optimization (PGO) feature of the Intel compiler can help reduce branch mispredictions.

3.7 Bus Utilization Ratio

The utilization of system memory bus (FSB) is an important factor that determines application performance. The system memory bus is the connection between the processor and external RAM memory. The bus ratios indicate performance issues because the processor core typically computes faster as compared to the rate at which values can be loaded from the memory into the processor thus causing a stall.

45nm Hi-k next generation Intel® Core™ microarchitecture

Equation: BUS_TRANS_ANY.ALL_AGENTS * 2 / CPU_CLK_UNHALTED.BUS
Ideal: 0
Bad: > 0.6

The impact of bus utilization isn't noticeable when the data accessed is small (may fit into L2 cache). But when the application is executed to check for scalability, the performance may vary significantly with the problem size. This degradation in performance with problem size could very well be due to the memory access pattern and bus utilization (amidst other problems). Figure 8 shows sample code which illustrates how bus utilization ratio manifests itself as the problem size increases.

Notes on Intel® Front-Side Bus (FSB):

To determine if FSB bandwidth is a significant performance issue for an application, it is important to find out where an application lies on the FSB utilization curve. This data can also be used to extrapolate out to faster cores and multi-core. The utilization of the front side bus is a performance concern because latencies of the memory and address FSB will vary with the amount of traffic on the bus. Graphing# the FSB utilization vs. memory latencies shows that memory latencies increase at a rapid rate after ~60% FSB utilization. Latencies then go infinite as the FSB bandwidth passes 70%.

#Intel internal measurements and experiments

```
for(i=0;i<LC;i++)
  {
        w[i] = (int *) malloc(1000000*sizeof(int));
        for(j=0;j<1000000;j++)
         {
            w[i][j]=GRI(1,10000);
         }
  }

 p = (unsigned int *) malloc(1000000*sizeof(unsigned int));

 for(i=0;i<1000000;i++)
  {
    p[i] = GRI(1,10000);
  }
  for(count=0;count<OC;count++)
  {
   mv = 0;
   for(i=0;i<LC;i++)
   {
     temp = 0;
     for(j=0;j<IC;j++)
     {
          temp = temp + p[4*j] * w[i][4*j] + p[4*j+1] * w[i][4*j+1]
                      + p[4*j+2] * w[i][4*j+2]
                      + p[4*j+3] * w[i][4*j+3];
              if (temp > FB) temp = 0;
      }
     if (temp > mv) mv = temp;
   }
  }
```

Understand how this code behaves when running single and multiple threaded versions on a multi-core/ CPU machine!

Figure 8: Code example to illustrate bus utilization and memory access issues

The code example in Figure 8 is a simple matrix-vector dot product and an associated condition within the innermost "for" loop. The problem size is governed by the size of the vector "p" and the matrix "w". The outermost "for" loop is used for repeating the dot-product operation. Though the code shown in Figure 8 is not complete, the idea is to inspect the bus utilization ratio when executing the application.

BUS_TRANS_ANY.ALL_ AGENTS events	CPU_CLK_UNHALTED .BUS events	Bus Utilization Ratio	Clocks per Instructions Retired - CPI
371500000	1423600000	52.19%	0.632

Table 9: Bus utilization ratio indicating performance impact of high memory bus usage

The bus utilization ratio in Table 9 clearly shows that the application is trying to fetch a large amount of data from the memory to the processor for computing the dot product operation. The problem of high bus utilization is a necessary evil for large data processing applications. However, some of the techniques to improve performance would be to use data locality algorithms and fixing the memory access patterns in the application design. Table 10 shows the bus utilization ratio when using a data locality algorithm (data blocking) that helps alleviate the high bus utilization problem.

BUS_TRANS_ANY.ALL_ AGENTS events	CPU_CLK_UNHALTED .BUS events	Bus Utilization Ratio	Clocks per Instructions Retired - CPI
1000000	1240000000	0.16%	0.516

Table 10: Bus utilization ratio decreases when using a data blocking algorithm

The bus utilization problem is even more prominent when executing on multi-core or multi-CPU machine in which the memory bus is shared. The reason is that in a multi-core/ multi-CPU scenario, the utilization increases with the number of processors sharing the bus (assuming all processors are executing the application). Even in such scenarios, using data locality algorithms help alleviate bus utilization problems as indicated in the profiles in Tables 11 thru 14.

BUS_TRANS_ANY.ALL_ AGENTS events	CPU_CLK_UNHALTED .BUS events	Bus Utilization Ratio	Clocks per Instructions Retired - CPI
747800000	2128800000	70.26%	0.965

Table 11: Bus utilization of original algorithm when executing on 2-cores

BUS_TRANS_ANY.ALL_ AGENTS events	CPU_CLK_UNHALTED .BUS events	Bus Utilization Ratio	Clocks per Instructions Retired - CPI
1300000	1208800000	0.22%	0.519

Table 12: Bus utilization of the data blocking algorithm when executing on 2-cores

BUS_TRANS_ANY.ALL_ AGENTS events	CPU_CLK_UNHALTED .BUS events	Bus Utilization Ratio	Clocks per Instructions Retired - CPI
1587000000	4408000000	72.01%	1.916

Table 13: Bus utilization of original algorithm when executing on 4-cores

BUS_TRANS_ANY.ALL_ AGENTS events	CPU_CLK_UNHALTED .BUS events	Bus Utilization Ratio	Clocks per Instructions Retired - CPI
4100000	1116800000	0.73%	0.526

Table 14: Bus utilization of the data blocking algorithm when executing on 4-cores

Tables 11 and 13 indicate the high bus utilization performance problem when executing on 2-cores and 4-cores. As you can note, the CPI almost doubles indicating the performance problem. Tables 12 and 14 show the profile obtained when using a data blocking algorithm. Clearly, data blocking algorithm has superior performance (better CPI).

To conclude, in this section 4, we reviewed various VTune™ performance monitoring events & ratios, their definitions and steps to measure them for samples of C code. Note, this is not a complete list of events & ratios in VTune™. The complete list is specific to processor architecture and changes in the tool support as well. However, the basic idea presented all along is to correlate the performance data with the code executing using the VTune™ tool and eventually get benefit from the processor architecture and the platform in general. You may also want to use tools like Microsoft* Windows* PerfMon*, Linux* VMSTAT, SAR and IOSTAT for analyzing and improving application performance.

4. Optimization Strategies

4.1 Hardware Configurations

- **Processors:** These form the core of the majority of computing. Understand the processor details like Pipelining, Instruction Length, Registers (usage), Function Units, RISC/ CISC/ VLIW.
- **Data Storage:** Though processors form the core, the memory sub-system also plays a significant role in deciding performance of applications. You should be aware of aspects like Caches, Virtual Memory (usage), RAM Memory (availability and usage pattern), I/O Performance Tips (like using buffered I/O) and RAID solutions.
- **Parallel Executions:** Understand parallel models, the hardware infrastructure for parallelism, and how to tune performance on distributed computing systems like Symmetric Multiprocessor system (SMP), NUMA architecture, Clusters, Grids and their associated memory locality issues.

4.2 Other Software Optimizations

- **Compiler:** Use compiler options/ pragmas for optimizing basic blocks, loops (unrolling, stripping, fusion and interchange), inter-procedural within and across files, function inlining.
- **Parallel Processing:** Understand Process, Thread parallelism algorithms, NUMA and MPI.
- **High Performance Libraries:** Use optimized libraries and mathematical kernels for performance.
- **Analysis of Algorithms:** Understand the worst-case, order and computational complexity.
- Inspect **performance counters** using Windows* PerfMon* or Linux IOStat*, SAR*
- Perform **Scalability** analysis of the application and investigate issues.

Conclusions

In this article, we have introduced the concept of software performance analysis and optimization lifecycle. An overview of the **45nm Hi-k next generation Intel® Core™ microarchitecture** was also presented and the performance analysis of the same was discussed further using VTune™ events. We have reviewed the aspects of application tuning at the system level and the application level and specifically focused on monitoring several events available in VTune™ to analyze microarchitecture performance.

Through this article, it is intended that users understand the several VTune™ events and ratios better and also monitor the same for software performance analysis and optimizations. The problem of software optimization is difficult as there is no unique solution to a particular problem and several factors govern it. By investigation and using a set of events on several tools, a precise understanding of the problem can be obtained and a solution can then be designed.

About the Author

Rama Kishan Malladi is an Application Engineer with the Software and Solutions Group at Intel in Bangalore. He works with various software developers and vendors, enabling their applications on the latest Intel platforms by addressing architecture, platform, and performance-related issues. Rama has been working at Intel since 2004 supporting Intel Software Tools, tuning high-performance computing applications on Intel architecture, and resolving performance issues on client/ server applications.

Rama pursued his undergraduate studies in Electronics and Communications Engineering from Osmania University in Hyderabad, India and Master of Science studies in Electrical Engineering from University of Massachusetts, Dartmouth. He can be reached via email at rama.kishan.v.malladi@intel.com.

References

- *Intel® 64 and IA-32 Architectures Software Developer's Manuals* – available at the URL http://www.intel.com/products/processor/manuals/
- *Intel® Software Development Products* http://www.intel.com/software/products
- *Multi-Core Programming Increasing Performance through Software Multithreading* (Intel Press) by Shameem Akhter and Jason Roberts
- *The Software Optimization Cookbook, Second Edition High-Performance Recipes for IA-32 Platforms* (Intel Press) by Richard Gerber, Aart J.C. Bik, Kevin B. Smith and Xinmin Tian
- *VTune™ Performance Analyzer Essentials Measurement and Tuning Techniques for Software Developers* (Intel Press) by James Reinders