Basic Performance Measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ Processors

Paul J. Drongowski
AMD CodeAnalyst™ Performance Analyzer Development Team
Advanced Micro Devices, Inc.
Boston Design Center

25 September 2008

1. Introduction.

Good performance is crucial to many applications. Program performance tuning is a multifaceted activity grounded in measurement and analysis. Measurement provides an objective basis for assessment and comparison of the performance aspects of program design and implementation.

AMD processors offer extensive features to enable performance measurement. These features use hardware counters to measure performance-related events caused by user- or kernel-level software. Event counts help a systems engineer or software developer to identify the likely cause of a performance issue.

The purpose of this technical note is to describe a collection of basic measurements that engineers and developers can take using the performance monitoring features of Athlon™ 64, AMD Opteron™, and AMD Phenom™ processors. Section 2 recommends online resources describing processor-specific microarchitecture, performance events, and performance analysis tools. Section 3 provides background information about the measurement technique called "performance counter sampling." Section 4 describes the performance measurements. Section 5 illustrates the use of a few common measurements.

2. Resources.

A number of on-line resources are available to assist engineers with performance measurement, analysis, and improvement. Find these resources, and many more, at AMD Developer Central:

http://developer.amd.com

Performance measurement hardware and events are described in the "BIOS and Kernel Developer's Guide," or BKDG. The BKDG is the main resource for details about hardware events and what they measure. These details include important information about conditions affecting event counts (*i.e.*, what is (and is not) included in an event count). The BKDG also describes how to configure the "unit masks" associated with each event to narrow measurement to more specific hardware conditions. AMD publishes a version of the BKDG for each processor family. Find information about performance events supported by quad-core AMD Opteron and AMD Phenom processors in the "BIOS and Kernel Developer's Guide for AMD Family 10h Processors," Publication #31116:

http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/31116.pdf

Performance guidelines and tips for AMD64 processors are in the "Software Optimization Guide" (SWOG). The software optimization guide is a resource for engineers and developers who wish to tune their programs to the processor microarchitecture to achieve the best performance. The guide contains a brief introduction to processor microarchitecture and tips and coding techniques all performance engineers and software developers will find useful. Like the BKDG, AMD publishes a software optimization guide for each processor family. Find information about code tuning for quad-core AMD Opteron and AMD Phenom processors in the “Software and Optimization Guide for AMD Family 10h Processors,” Publication #40546:

http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf

There are many techniques for program performance tuning on AMD Athlon 64 and AMD Opteron Processors in Michael Wall’s article "Performance Optimization of 64-bit Windows® Applications for AMD Athlon 64 and AMD Opteron Processors using Microsoft® Visual Studio 2005." Although this paper addresses Microsoft Visual Studio 2005 specifically, many of the techniques are generic and employable when working with other compilers or operating systems.

http://developer.amd.com/pages/101120051_18.aspx

The Developer Tools page at AMD Developer Central lists a wide range of tools -- from compilers to libraries to simulators -- that can improve the performance of programs on AMD processors. The list includes several profiling tools to measure and analyze the performance and behavior of application programs.

http://developer.amd.com/tools/Pages/default.aspx

AMD provides its own profiling tool, AMD CodeAnalyst™ Performance Analyzer, through AMD Developer Central. AMD CodeAnalyst is available for both the Windows and Linux® operating systems. AMD CodeAnalyst for Linux is open source, making it an attractive, low-cost option for university education as well as industrial-strength analysis. Links to AMD CodeAnalyst for Windows and CodeAnalyst for Linux are:

http://developer.amd.com/tools/codeanalystwindows/Pages/default.aspx
http://developer.amd.com/tools/codeanalystlinux/Pages/default.aspx

A brief introduction to AMD CodeAnalyst is in the white paper, "An introduction to analysis and optimization with AMD CodeAnalyst," by P.J. Drongowski. This paper uses AMD CodeAnalyst in Section 5 to illustrate the use of event-based performance measurements. Please note, however, that engineers and developers can use the performance measurements described in this paper with other profiling tools like OProfile, the Performance API (PAPI), or SunStudio on Solaris™.

http://developer.amd.com/pages/111820052_9.aspx

3. Performance counter sampling.

The AMD Athlon 64, AMD Opteron, and AMD Phenom processors provide four performance counters to measure the hardware events caused by application programs and system software. When you use a profiling tool like AMD CodeAnalyst, it configures these counters to measure specific hardware events like the number of CPU clocks, the number of instructions retired, data cache misses, and so forth. Configuration information consists of an event select value and a unit mask value. The event select value specifies which hardware event to measure. The unit mask modifies the effect of the event select value in certain cases and narrows measurement to a specific subset of the hardware conditions constituting an event.

There are two main approaches to event counting: caliper mode and performance counter sampling. The caliper approach reads the event count before and after a performance-critical region of code. The before-count is subtracted from the after-count to yield the number of events that occurred between the before and after points. (PAPI is a tool that readily supports this model.) This approach, which acts like a caliper, measures the number of events, but does not indicate how the events are distributed across the code region. Caliper-style measurement cannot isolate a performance issue to a single instruction or source-level operation. This style of measurement often requires a change to the program source code to take measurements at key points.

In performance counter sampling, the event counter is preloaded with a threshold or limit count. The performance measurement hardware counts events until reaching the threshold and then causes an interrupt. The interrupt service routine (ISR) records the counter "overflow" as a sample that includes the type of event that occurred, the process ID of the program that was executing at the time of the interrupt, the thread ID of the executing thread, and the instruction pointer (IP). The profiling tool processes the samples and builds up a statistical histogram -- a profile -- of how events of a particular type are distributed across the source lines and/or instructions in the application program.

The threshold or limit count is often called the "sampling period" because a sample is taken periodically based on the magnitude of the preloaded limit. If the sampling period for a particular event is 5,000, then it will generate one sample for each 5,000 occurrences of that event. Thus, developers can use the sampling period as a scale factor or weight to convert a sample count into an estimate of the raw number of events that occurred. Scaling and weight are important because **it is only meaningful to compare sample or event counts that have the same weight or to perform arithmetic on sample or event counts that have the same weight.**

Engineers and developers must scale and normalize sample counts with different sampling periods with respect to each other before comparison or computation. **The formulas used throughout this paper assume appropriate normalization of sample counts.**

Choosing the sampling period is a trade-off among collection overhead (the time spent collecting samples), the intrusive effects of measurement, and the desired resolution and accuracy of statistical results. A smaller sampling period means taking samples more frequently. When you use a smaller sampling period:

- The statistical result is more accurate because more samples are taken in the same time period.
- Resolution is higher because the time between samples is shorter.
- Collection overhead is higher because more time is spent processing a larger number of samples in the same time period.
- Intrusion is higher due to the pollution of caches, translation lookaside buffers (TLBs), and branch history tables. The overall memory trace will be different as well.

It is common practice to use a smaller sampling period for less frequent events like cache misses and to use a larger sampling period for high-frequency events like retired instructions or CPU clocks (processor cycles). We suggest a sampling period of 50,000 for low-frequency events and a sampling period of 500,000 for high-frequency events.

Before turning to the measurements themselves, we need to explain the effects of skid on profiles. Ideally, the instruction pointer recorded with a sample is the IP of the instruction that actually caused the event. However, the IP recorded when a performance counter overflow causes an interrupt is the address of the instruction to be started after returning from the sampling interrupt. Highly parallel and out-of-order machines might execute many instructions between the time the hardware event occurs and delivery of the interrupt. Thus, attribution of the event "skids" to the instruction specified by the restart IP, not the instruction that caused the event. If the degree of skid varies then, over time, events are incorrectly attributed to instructions executed after the actual culprit. In the case of the AMD Family 10h processor, as many as 72 operations may be in flight and skid may be substantial.

When analyzing results collected through performance counter sampling, we need to recognize that event attribution to individual instructions is imprecise. Measurements computed over a region of frequently executed instructions are generally valid. Therefore, it is usually meaningful to discuss instructions per cycle (IPC) for a frequently executed loop or function, but not for an individual instruction.

4. Event-based performance measurements.

This section describes performance measurements we find most useful in practice. They should help performance engineers and software developers identify the most common performance issues. Apply them to event data collected using either caliper-like measurements or performance counter sampling.

Table 1 summarizes the performance measurements. We classify the measurements into these major categories:

- **Efficiency** measurements gauge overall performance of programs, functions, and critical code regions like inner loops. A low IPC figure or low memory bandwidth may indicate the presence of an underlying performance issue.

- **Memory access** measurements assess the use of instruction and data caches. Temporal locality and spatial locality are important to good program performance. Instruction cache misses disrupt the flow of instructions into the processor pipeline. Data cache misses slow the flow of data into the pipeline. The result, in either case, is a stalled pipeline that must wait for instructions or data to arrive. Stalls degrade performance. Employ cache-friendly algorithms and code tuning techniques in performance-critical code.

- **Address translation** measurements appraise the use of instruction and data translation lookaside buffers (TLBs). TLBs provide fast access to the page mapping information that translates virtual memory addresses to physical addresses. Concerns about temporal and spatial locality also apply to TLB access and use.

- **Control transfer** measurements deal with the processor's ability to predict the target address of a conditional branch, indirect branch, or subroutine return. Processors depend on a steady flow of work to keep functional units busy. They accomplish this by predicting the outcome of branches and subroutine returns -- instructions that break the flow of work to do. Based on the prediction, work starts speculatively in the functional units and memory subsystem. If the prediction is wrong, the speculative work must be discarded and the flow of instructions must be restarted. This is expensive.

- The final category addresses certain **special cases** that occur in practice. Avoid access to unaligned data because access to naturally aligned data is faster. Scientific programmers often want to study the number and flow of floating point (FP) operations, including the number of FP exceptions that interrupt the flow of FP operations in the pipeline.

Many of the measurements are expressed as a rate such as the number of L1 data cache misses per retired instruction. A rate helps us to judge the severity of a problem better than a raw event count. For example, 1,000 cache misses per 1,000,000 retired instructions is not a problem, but 1,000 cache misses per 10,000 retired instructions is a major problem. At that rate, a cache miss occurs every 10 retired instructions!

Table 1: Summary of performance measurements.

Category	**Subsystem**	**Measurements**
Efficiency (Section 4.2)	CPU (Section 4.2.1)	`Instructions per cycle (IPC)`
		`Cycles per instruction (CPI)`
	Memory (Section 4.2.2)	`Read data bandwidth`
		`Write data bandwidth`
		`DRAM bandwidth`
Memory access (Section 4.3)	L1, L2, L3 caches	`Cache request rate`
		`Cache miss rate`
		`Cache miss ratio`
Address translation (Section 4.4)	DTLB and ITLB	`L1 TLB request rate`
		`L1 TLB miss rate`
		`L1 TLB miss ratio`
		`L2 TLB request rate`
		`L2 TLB miss rate`
		`L2 TLB miss ratio`
Control transfer (Section 4.5)	Branches (Section 4.5.1)	`Branch rate`
		`Branch misprediction rate`
		`Branch misprediction ratio`
		`Branch taken rate`
		`Branch taken ratio`
		`Instructions per branch`
	Near return (Section 4.5.2)	`Near return rate`
		`Return stack miss rate`
		`Return stack misprediction ratio`
		`Instructions per call`
Special cases (Section 4.6)	Memory (Section 4.6.1)	`Misaligned access rate`
		`Misaligned access ratio`
	Floating point (Section 4.6.2)	`FPU op rate`
		`FP/MMX rate`
		`FLOPS rate`
		`Overall FP exception rate`
		`FP exception rate`

4.1. Format and terminology.

This paper breaks down the description of each event-based performance measurement into four sections:

- **Applicability:** when to apply a set of performance measurements.
- **Collection:** what events to collect to compute the measurements.
- **Formulas:** how to compute derived measurements from the collected events.
- **Interpretation:** how to interpret the results.

All formulas assume normalization of event sample counts such that all quantities have a comparable unit weight.

This article uses the term “system memory” when describing certain performance events and measurements. Memory is organized into multiple levels, with the L1 data and instruction caches at the lowest level and dynamic random access memory (DRAM) at the highest level. Each core has its own L1 data and instruction caches and a unified L2 cache. When a read or write request cannot be satisfied by the L1 or L2 caches belonging to a core, the request is sent to the higher levels of the memory subsystem through the system interface provided by the Northbridge. The request may be satisfied:

- By an L1 or L2 cache in a different core within the same local multi-core processor,
- By the optional shared L3 cache within the same local processor,
- By local DRAM,
- By an L1, L2, or optional L3 cache in a different, remote processor, or
- By remote DRAM.

“System memory” collectively refers to the data sources accessed via the system interface. Local data sources are generally preferred because remote data sources must be accessed over one or more HyperTransport™ technology links.

4.2. Efficiency measurements.

Event-based performance measurements in this category help determine if there is an underlying performance issue that needs further investigation.

4.2.1. Instructions per cycle (IPC).

Applicability. Instructions per cycle is a general measure of computational efficiency. IPC indicates the number of completed instructions per CPU clock cycle. Engineers and developers often use IPC as a measure of instruction-level parallelism (ILP) -- the number of operations that can be performed in parallel. IPC may be computed for a program, module, function, or code region. Due to skid, IPC for individual instructions is rarely accurate. Cycles per instruction (CPI) is the reciprocal of IPC.

IPC is usually measured before and after tuning. Measurements taken before tuning establish a baseline. Measurements taken after tuning show the overall effect of code or algorithmic changes on performance.

Collection. Computation of IPC requires collection of only two basic events:

Event select	Unit mask	Event abbreviation	Event
`0x76`	`N/A`	`CPU_clocks`	`CPU Clocks Not Halted`
`0xC0`	`N/A`	`Ret_instructions`	`Retired Instructions`

CPU clocks and retired instructions are both regarded as high-frequency instructions. We recommend a sampling period of 500,000 for each event. No special unit mask configuration (`N/A`) is required and the unit mask values should be set to `0x00`.

The CPU run state affects the CPU Clocks Not Halted event. Operating systems handle idling in one of two ways:

- By temporarily halting the CPU until there is work to do, or
- By executing an idle loop.

In the first case, the CPU clock halts. CPU clock events are not counted and the resulting CPI and IPC figures directly reflect the behavior of the workload. In the second case, the CPU clock continues to run as the idle loop executes. CPI and IPC figures then include the effects of the idle loop. The idle loop usually has good CPI and IPC, which produces optimistic system-level values. The effects of the idle loop must be isolated and mitigated to state the correct CPI and IPC for the workload.

Formulas. IPC is simply the ratio of instructions to CPU clock cycles.

```
IPC = Ret_instructions / CPU_clocks
```

The inverse of this ratio, cycles per instruction, is sometimes used as well.

```
CPI = CPU_clocks / Ret_instructions
```

Interpretation. Higher values of IPC indicate that more useful work (successfully completed retired instructions) is being performed in a given time unit (a CPU clock cycle). Low values of IPC indicate the presence of some performance-inhibiting factors such as poor temporal or spatial locality, mispredicted branches, use of unaligned data, or floating point exceptions.

In addition to IPC, the distribution of CPU clock cycles and retired instructions across program regions is itself informative. The distribution of CPU clock cycles shows those parts of the program taking the most time while the distribution of retired instructions show the most frequently executed code regions and paths in the program. Identification of a hot spot (a frequently executed or time-consuming code region) allows analysis and tuning efforts to focus on the hot spot with a higher potential payoff for the investment.

4.2.2. Memory bandwidth.

Applicability. These measurements help assess memory bandwidth utilization. Apply bandwidth measurements when the application moves a large amount of data between memory and the processor. Scientific and engineering applications, for example, are often memory-intensive and operate on large data structures, creating a high volume of data traffic.

Collection. Collect data for events in this table to estimate actual memory bandwidth:

Event select	Unit mask	Event abbreviation	Event
0x76	N/A	CPU_clocks	CPU Clocks Not Halted
0x6C	0x07	System_read	System Read Responses by Coherency State
0x6D	0x01	System_write	Quadwords Written to System* Octwords Written to System*
0xE0	FAMDEP	DRAM_accesses	DRAM Accesses

* The event with select value `0x6D` has the name Octwords Written to System on AMD Family 10h processors. The event has the name Quadwords Written to System on other processors.

All these events are regarded as high-frequency events. We recommend a sampling period of 500,000 for all four events.

The memory controller unit measures DRAM accesses. Certain performance counter configuration constraints may need enforcement on multi-core systems (see the BKDG for configuration details).

The unit mask value for the DRAM Accesses event is processor family-dependent (FAMDEP). Certain implementations of AMD Family 10h processors have two memory controllers; other processors have one memory controller (see the appropriate, processor-specific BKDG for more information). Each memory controller counts the number of read and write requests it fulfills. A separate performance counter must be configured for each memory controller. Measure these events on processors with two memory controllers (`DCT0` and `DCT1`):

Event select	Unit mask	Event abbreviation	Event
0xE0	0x07	DRAM_accesses_0	DRAM Accesses [DCT0]
0xE0	0x38	DRAM_accesses_1	DRAM Accesses [DCT1]

Measure this event on processors with one memory controller:

Event select	Unit mask	Event abbreviation	Event
0xE0	0x07	DRAM_accesses	DRAM Accesses

Formulas. First, estimate the number of bytes transferred. AMD Family 10h processors perform 16-byte write transfers and these formulas apply:

```
Read_bytes_transferred = (System_read * Period) * 64
Write_bytes_transferred = (System_write * Period) * 16
```

```
DRAM_bytes_transferred = (DRAM_accesses * Period) * 64
```

For AMD Family 10h processors with two memory controllers, the total number of DRAM access events is the number of access requests fulfilled by both memory controllers:

```
DRAM_accesses = DRAM_accesses_0 + DRAM_accesses_1
```

Other processors perform 8-byte write transfers; instead, use these formulas:

```
Read_bytes_transferred = (System_read * Period) * 64
Write_bytes_transferred = (System_write * Period) * 8
DRAM_bytes_transferred = (DRAM_accesses * Period) * 64
```

Always remember to normalize event samples by the sampling period before performing computations on event data. For clarity, normalization by the sampling period is explicit in these formulas. The three immediately preceding formulas assume a 64-bit DRAM granularity.

Bandwidth is the number of bytes transferred per second:

```
Read data bandwidth (B/s) = Read_bytes_transferred / Seconds
Write data bandwidth (B/s) = Write_bytes_transferred / Seconds
DRAM bandwidth (B/s) = DRAM_bytes_transferred / Seconds
```

We use the CPU Clocks Not Halted event to measure elapsed time. This formula gives elapsed time:

```
Seconds = (CPU_clocks * Period) / Clock_frequency

    Where Period is the sampling period, and
          Clock_frequency is the platform CPU clock frequency
```

Measuring elapsed time in this way is convenient, but it has certain drawbacks. As the event name indicates, CPU clock events are not counted when the CPU clock is halted. The operating system may halt the clock when the system is idle. The operating system may also adjust the clock frequency. Both of these techniques reduce power. However, they affect the accuracy of the CPU Clocks Not Halted event as a time reference. We recommend using a reliable time reference such as the operating system clock or time-stamp counter (TSC).

Interpretation. These measures show the amount of system memory bandwidth used for reading and writing data. The communication capacity of the test platform's memory subsystem and DRAM controller(s) limits bandwidth. Please see platform specifications or use a benchmark program such as Stream to determine maximum attainable bandwidth:

http://www.cs.virginia.edu/stream/

If actual memory bandwidth is much less than potential sustainable bandwidth, then developers should take steps to improve the memory system behavior of the application. These steps include improving the data access pattern, prefetching, software pipelining, and use of streaming stores (non-temporal MOVNTPS and MOVNTQ instructions).

The cores in a multi-core processor share the on-chip memory controllers. Thus, the cores split the available DRAM bandwidth.

4.3. Memory access.

4.3.1. Data cache misses and miss ratio.

Applicability. Good cache behavior is important to good performance. Data caches favor programs that exhibit good spatial and temporal locality when accessing data. Always measure data cache behavior.

Collection. Compute data cache miss statistics from these four events:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0x40	N/A	DC_accesses	Data Cache Accesses
0x42	0x1E	DC_refills_L2	Data Cache Refills from L2
0x43	0x1E	DC_refills_sys	Data Cache Refills from System

We suggest a sampling period of 500,000 for retired instruction events and data cache accesses. We suggest a sampling period of 50,000 for data cache refills. Remember to normalize the results before using the formulas in this section.

Formulas. Behavior at a given level in the memory hierarchy can often be characterized by studying the inflow of data (refills) into the level. The number of data cache misses is equal to the number of refill operations performed to satisfy data cache misses. There are two sources of refill operations for the L1 data cache: the unified level 2 (L2) cache and system memory. Thus, the number of data cache misses is equal to the sum of the refill operations from L2 cache and system memory:

```
DC_misses = DC_refills_L2 + DC_refills_sys
```

There are three derived measurements:

```
Data cache request rate = DC_accesses / Ret_instructions
Data cache miss rate = DC_Misses / Ret_instructions
Data cache miss ratio = DC_Misses / DC_accessess
```

A simplified data collection scheme measures these events:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0x40	N/A	DC_accesses	Data Cache Accesses
0x41	N/A	DC_misses	Data Cache Misses

This set of events measures data cache misses directly. However, the Data Cache Misses event is subject to variation due to streaming store activity (see the BKDG for more information). Use the simplified form as an estimate when conserving performance counters at runtime. The refill method is preferred when greater accuracy is required.

Interpretation. The data cache request rate shows the frequency of L1 data cache requests for a given set of retired instructions. The cache miss rate indicates the frequency of miss operations for the same set of retired instructions. The miss rate should be much less than the access rate.

The data cache miss ratio shows the portion of data accesses that missed in the L1 data cache. The data cache miss ratio should be as low as possible.

Refill operations from system memory are more expensive than refills from L2 cache (90 or more cycles from DRAM versus 12 cycles from L2 cache, depending on processor implementation). The ratio of refills from system memory to data cache misses,

```
  DC_refills_sys / DC_misses
```

shows the portion of data cache misses that went to system memory for resolution. If one instruction executes per cycle, then an access to DRAM is worth 90 lost instructions (worst case with no overlap of memory operations and computation). Avoid refills from system memory by choosing cache-friendly algorithms and coding techniques.

There are three categories of cache misses:

1. **Compulsory misses** occur on first reference to a data item.

2. **Capacity misses** occur when the working set exceeds the cache capacity.

3. **Conflict misses** occur when a data item is referenced after the cache line containing the item was evicted.

Data prefetching can help reduce compulsory misses. Reduce capacity misses by decreasing the size of the program working set or through improved data layout (moving data items closer together). Eliminate conflict misses by relocating data structures to memory locations that do not map to the same cache lines.

Caches favor programs with regular sequential access through memory. Algorithm and program redesign may be needed to improve the memory access pattern (*e.g.*, to use small sequential strides through memory). Changing data layout to place widely separated data items into the same cache line may improve cache efficiency.

4.3.2. Instruction cache misses and miss ratio.

Applicability. Superscalar microprocessors depend on a steady flow of instructions into the pipeline. The instruction cache holds the most recently fetched x86 instructions and is able to provide those instructions to the pipeline when needed quickly. Engineers and developers can obtain good performance when critical code fits in the instruction cache and is reused, as in the case of a frequently executed inner loop or a group of subroutines that call each other frequently.

Collection. Compute instruction cache measurements using these four events:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0x80	N/A	IC_fetches	Instruction Cache Fetches
0x82	N/A	IC_refills_L2	Instr Cache Refills from L2
0x83	N/A	IC_refills_sys	Instr Cache Refills from System

We suggest a sampling period of 500,000 for retired instructions and instruction cache fetches. We recommend a sampling period of 50,000 for instruction cache refills. Remember to normalize the results before using the formulas in this section.

Developers may use the Instruction Cache Misses event (select `0x81`) in place of the two refill events. The sum of the refill events should be roughly equal to the Instruction Cache Misses event. This alternative approach conserves performance counters at runtime.

Formulas. The number of instruction cache misses is equal to the sum of instruction cache refill requests to the unified L2 cache and system memory:

```
IC_misses = IC_refills_L2 + IC_refills_sys
```

A refill operation must satisfy every L1 instruction cache miss.

There are three derived measurements:

```
Instruction cache request rate = IC_accesses / Ret_instructions
Instruction cache miss rate = IC_Misses / Ret_instructions
Instruction cache miss ratio = IC_Misses / IC_accesses
```

Interpretation. The instruction cache request rate indicates the number of instruction cache accesses made for a set of retired instructions. The instruction cache miss rate shows how frequently an instruction cache miss occurred for the same group of retired instructions. The instruction cache miss ratio indicates the portion of instruction cache assesses that caused a miss, thereby resulting in a refill from L2 cache or system memory.

Low instruction cache miss rates and ratios are desirable.

As in the case of the L1 data cache, refills from system memory are more expensive than refills from L2 cache. Avoid refills from system memory.

Reduce instruction cache misses by improving code layout. Frequently executed, related code regions should be placed together to improve spatial locality. This may involve rearranging subroutines to move related subroutines near each other and to place infrequently executed subroutines farther away. For example, place exception-handling routines farther away because exceptions are infrequent. Software developers should also check to see if procedure inlining or loop unrolling has produced code that no longer fits into the L1 instruction cache (64 Kbyte maximum capacity).

4.3.3. Level 2 (L2) cache misses and miss ratio.

The L2 cache is a unified cache containing both instructions and data. On AMD Athlon 64, AMD Opteron, and AMD Phenom processors, the L1 caches and the L2 cache use an "exclusive" line management scheme. A line of cache data is in either the L1 or L2 cache, but not both caches at the same time. The exclusive line management scheme makes better use of cache space than an "inclusive" scheme that may have two redundant copies of the same line in the caches.

As mentioned earlier, a refill from the L2 cache or from system memory satisfies an L1 cache miss. When a line of data is retrieved from system memory, it is written into the L1 cache, but not the L2 cache. This maintains exclusivity. The evicted L1 cache line -- the cache line replaced with a new block of bytes from memory -- is written into the L2 cache, maintaining exclusivity. The evicted cache line is sometimes called a "victim."

There are two different methods for obtaining L2 cache memory measurements. The direct method uses fewer events and is easier to collect and compute. The indirect method is more accurate, but requires the collection of more events and involves a few more computations. Sections 4.3.3.1 and 4.3.3.2 describe both methods.

4.3.3.1. L2 cache measurements: Direct method

Applicability. Use this method to measure L2 cache behavior when less accuracy is required. All event data can be collected in a single run.

Collection. Collect these events to measure L2 behavior:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0x7D	0x07	L2_requests	Requests to L2 Cache
0x7E	0x07	L2_misses	L2 Cache Misses
0x7F	0x03	L2_fill_write	L2 Fill / Writeback

We suggest a sampling period of 500,000 for the retired instructions event and a sampling period of 50,000 for the L2 cache events.

Formulas. These are derived measurements for L2 cache performance:

```
L2 request rate = (L2_requests + L2_fill_write) / Ret_instructions
L2 miss rate = L2_misses / Ret_instructions
L2 miss ratio = L2_misses / (L2_requests + L2_fill_write)
```

Interpretation. The events used to compute these measures include extra requests due to retries associated with address or resource conflicts. In some cases, the extra requests can dominate the event counts, but are not a direct indication of performance impact. The events used in the indirect method better reflect actual cache line movement.

4.3.3.2. L2 cache measurements: Indirect method

Applicability. Use this method to measure L2 cache behavior when requiring greatest accuracy or a breakout of L2 cache activity. Multiple program runs are required to collect all of the needed event data when measuring only four performance events per run.

Collection. Collect these events to measure behavior of the unified L2 cache using the indirect method:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0x40	N/A	DC_accesses	Data Cache Accesses
0x42	0x1E	DC_refills_L2	Data Cache Refills from L2
0x43	0x1E	DC_refills_sys	Data Cache Refills from System
0x80	N/A	IC_fetches	Instruction Cache Fetches
0x82	N/A	IC_refills_L2	Instr Cache Refills from L2
0x83	N/A	IC_refills_sys	Instr Cache Refills from System

0x7D	0x04	L2_requests_TLB	Requests to L2 Cache [TLB fill]
0x7E	0x04	L2_misses_TLB	L2 Cache Misses [TLB fill]

Formulas. There are three sources of requests to the L2 cache: the L1 data cache, the L1 instruction cache, and the Page Table Walker that handles TLB misses. An L2 cache request is made when an access misses in the L1 data cache, when an access misses in the L1 instruction cache, or when the Page Table Walker issues a request:

```
IC_misses = IC_refills_L2 + IC_refills_sys
DC_misses = DC_refills_L2 + DC_refills_sys
L2_requests = IC_misses + DC_misses + L2_requests_TLB
```

This formula gives the L2 cache request rate:

```
L2 request rate = L2_requests / Ret_instructions
```

Misses in the L2 cache occur when a miss in the L1 instruction cache is refilled from system memory, when a miss in the L1 data cache is refilled from system memory, or when a Page Table Walker request misses:

```
L2_misses = IC_refills_sys + DC_refills_sys + L2_misses_TLB
```

The L2 cache miss rate and miss ratio computations are:

```
L2 miss rate = L2_misses / Ret_instructions
L2 miss ratio = L2_misses / L2_requests
```

Compute the proportion of unified L2 cache requests from the L1 instruction cache, L1 data cache, and Page Table Walker with these formulas:

```
L2 instruction fraction = IC_misses / L2_requests
L2 data fraction = DC_misses / L2_requests
L2 page table fraction = L2_requests_TLB / L2_requests
```

Interpretation. Data or instructions are written to the unified L2 cache when evicted from their respective L1 caches. If data or instructions are not found in L2 cache to satisfy a miss in an L1 cache, the performance engineer must consider if the miss is compulsory (first access to a data item) or a capacity or conflict miss. Given the exclusive nature of the L1 and L2 caches, a capacity miss may indicate poor temporal locality such that a data item (or its neighbors) is reused too long after its last use.

4.3.4. L3 cache.

Applicability. Level 3 (L3) cache is available on certain implementations of AMD Family 10h processors (see the appropriate, processor-specific BKDG for details). The L3 cache is a non-inclusive victim cache holding cache lines evicted from the L2 cache. All cores in a multi-core processor dynamically share the L3 cache; its dynamic allocation scheme provides efficient data sharing between cores.

Collection. Use these events to measure L3 cache behavior:

Event select	Unit mask	Event abbreviation	Event
0x0C0	N/A	Ret_instructions	Retired Instructions

0x4E0	0xF7	L3_requests	Read Requests to L3 Cache
0x4E1	0xF7	L3_misses	L3 Cache Misses

We recommend a sampling period of 500,000 for the Retired Instructions event, and a sampling period of 50,000 for the L3 cache events.

Use the unit mask associated with the L3 cache events to measure L3 cache requests and misses by cache coherency state (exclusive, shared, and modified states) and by core. An L3 event breakdown by core can identify one or more specific cores with a high request or miss rate (see the BKDG for additional configuration information).

Formulas. Derive L3 cache measurements using these formulas:

```
L3 request rate = L3_requests / Ret_Instructions
L3 miss rate = L3_misses / Ret_Instructions
L3 miss ratio = L3_misses / L3_requests
```

Interpretation. A high L3 cache miss rate indicates poor spatial and/or temporal locality. Since the L3 is a unified cache, it contains both instructions and data, either of which may be accessed with poor locality. Developers may need to reorganize the program and its data in memory to change the code or data layout to obtain better cache behavior (a lower miss rate). Investigate and modify data access patterns to favor reuse and small sequential strides.

4.4. Address translation.

Translation lookaside buffers help the processor translate virtual addresses to physical addresses. They hold the most recently used page mapping information in fast, chip-resident memory to accelerate address translation.

The processor microarchitecture provides separate TLBs for instructions and data. Each TLB is a two-level structure with a level 1 (L1) TLB and a larger, level 2 (L2) TLB (see the appropriate processor-specific Software Optimization Guide for more details). Each TLB entry contains page mapping information for a limited range of virtual addresses. Since the capacity of each TLB level is also limited, there are only so many pages that can be touched directly without incurring a TLB miss.

When an L1 TLB miss occurs, page mapping information is sought in the corresponding L2 TLB. If found, the entry is written to the L1 TLB. If not found, the Page Table Walker is invoked to find the mapping information in the memory-resident page tables. A refill from the L2 TLB is much faster than a refill from cache or system memory (two cycles versus 90 cycles or more in the worst case when page information is read from DRAM).

TLB behavior favors programs with good spatial and temporal locality and with a small virtual memory working set.

4.4.1. Data TLB misses and miss ratio.

Applicability. Take data TLB (DTLB) measurements for programs that operate on large data sets (like scientific applications involving large arrays) and programs with a non-uniform ("random") data access pattern.

Collection. Measure DTLB behavior using these events:

Event select	Unit mask	Event abbreviation	Event
`0xC0`	`N/A`	`Ret_instructions`	`Retired Instructions`
`0x40`	`N/A`	`DC_accesses`	`Data Cache Accesses`
`0x45`	`FAMDEP`	`DTLB_L1M_L2H`	`L1 DTLB Miss and L2 DTLB Hit`
`0x46`	`FAMDEP`	`DTLB_L1M_L2M`	`L1 DTLB Miss and L2 DTLB Miss`

The unit masks for the DTLB miss events (select values `0x45` and `0x46`) are processor family dependent (FAMDEP.). On AMD Family 10h processors, the unit mask values should be `0x07`. Otherwise, the unit mask should be `0x00`.

We suggest a sampling period of 500,000 for retired instructions and data cache assesses. We recommend a sampling period of 50,000 for the DTLB miss events.

Formulas. The DTLB-related events separate out L1 DTLB misses that are satisfied by the L2 DTLB and cache/system memory. The L1 DTLB request rate is equal to the L1 data cache request rate since all virtual memory data addresses must be translated:

```
  L1 DTLB request rate = DC_accesses / Ret_instructions
```

The derived L1 DTLB miss measurements are:

```
  L1 DTLB miss rate = (DTLB_L1M_L2H + DTLB_L1M_L2M) / Ret_instructions
  L1 DTLB miss ratio = (DTLB_L1M_L2H + DTLB_L1M_L2M) / DC_accesses
```

The L2 DTLB request rate is equal to the L1 DTLB miss rate since all L1 DTLB misses must be sent to the L2 DTLB:

```
  L2 DTLB request rate = (DTLB_L1M_L2H+DTLB_L1M_L2M) / Ret_instructions
```

The derived L2 DTLB miss measurements are:

```
  L2 DTLB miss rate = DTLB_L1M_L2M / Ret_instructions
  L2 DTLB miss ratio = DTLB_L1M_L2M / (DTLB_L1M_L2H + DTLB_L1M_L2M)
```

Interpretation. Address translations that miss in the L1 DTLB and hit in the L2 DTLB are less severe than translations that miss both levels. If a translation misses both the L1 DTLB and the L2 DTLB, page information must be retrieved from either cache or system memory. This penalty is severe.

The L2 DTLB miss rate indicates how frequently these kinds of misses occur during program execution. Consider a simple four-instruction loop that walks sequentially through a one-dimensional array of 4-byte integers. Given a page size of 4 KBytes, it will take 1,024 iterations to walk through a single page of data in the array. Assuming that a DTLB miss occurs when touching the next page of data, then the DTLB miss rate is one DTLB miss per 4,096 retired instructions, or 0.00024. Sometimes the inverse of this rate -- 4,096 retired instructions per DTLB miss -- is easier to understand: a DTLB miss rate of 0.01 (100 retired instructions per DTLB miss) is clearly too high.

Algorithms and coding techniques that improve data cache behavior generally improve DTLB behavior as well. Techniques that reduce the size of a program's virtual memory working set (*e.g.*, packing frequently accessed data into a few pages) should help reduce the occurrence of

DTLB misses. Large pages (*e.g.*, 2 MByte pages) may also improve the utilization of DTLB entries and reduce DTLB misses.

4.4.2. Instruction TLB misses and miss ratio.

Applicability. Instruction TLB (ITLB) misses may be a problem in large application programs that make calls between widely scattered subroutines, as in the case of procedures distributed among many different modules in memory.

Collection. Collect data for these events to measure ITLB behavior:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0x80	N/A	IC_fetches	Instruction Cache Fetches
0x84	N/A	ITLB_L1M_L2H	L1 ITLB Miss and L2 ITLB Hit
0x85	FAMDEP	ITLB_L1M_L2M	L1 ITLB Miss and L2 ITLB Miss

The unit mask for event select `0x85` is family dependent. For AMD Family 10h processors, the unit mask should be `0x03`. Otherwise, the unit mask should be `0x00`.

We suggest a sampling period of 500,000 for retired instructions and instruction cache fetches. We recommend a sampling period of 50,000 for the ITLB events.

Formulas. The L1 ITLB request rate is equal to the instruction cache request rate since the L1 ITLB must be consulted whenever an instruction is requested:

```
  L1 ITLB request rate = IC_fetches / Ret_instructions
```

Derived measurements for the L1 ITLB are:

```
  L1 ITLB miss rate = (ITLB_L1M_L2H + ITLB_L1M_L2M) / Ret_instructions
  L1 ITLB miss ratio = (ITLB_L1M_L2H + ITLB_L1M_L2M) / IC_fetches
```

The L2 ITLB request rate is equal to the L1 ITLB miss rate since all L1 ITLB misses must be passed to the L2 ITLB:

```
  L2 ITLB request rate = (ITLB_L1M_L2H+ITLB_L1M_L2M) / Ret_instructions
```

Derived measurements for the L2 ITLB are:

```
  L2 ITLB miss rate = ITLB_L1M_L2M / Ret_instructions
  L2 ITLB miss ratio = ITLB_L1M_L2M / (ITLB_L1M_L2H + ITLB_L1M_L2M)
```

Interpretation. Misses in the L2 ITLB pay a higher penalty than misses in the L1 ITLB since page information must be retrieved from either cache or system memory.

Developers can reduce ITLB misses by decreasing the size of the program's virtual memory working set. Place subroutines that call each other frequently in the same page or within only a few pages. Place infrequently executed code, like exception handlers, on separate pages, away from frequently executed instructions.

4.5. Control transfer.

Part of maintaining the flow of instructions into the processing pipeline is the ability to handle potentially disruptive changes in control flow. AMD Athlon 64, AMD Opteron, and AMD Phenom processors provide logic to predict both the outcome and target address of conditional branch instructions. They also predict the target address of indirect branches and near returns. Instructions are issued speculatively based on these predictions. When the predictions are wrong, speculative work -- instructions on the "wrong path" -- must be discarded and the pipeline must be restarted with instructions on the correct control path. Recovery is expensive in terms of work discarded, wasted resources, and lost cycles while the pipeline is flushed and redirected.

4.5.1. Branches.

Applicability. Conditional branch mispredictions may be a significant issue in code with a lot of decision-making logic. Conditional branches may be mispredicted when the likelihood of choosing the true or false path is random or near a 50-50 split. The branch prediction hardware cannot "learn" a pattern and branches are not predicted correctly.

Collection. Collect the events in this table to measure branch prediction performance:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0xC2	N/A	Branches	Retired Branch Instructions
0xC3	N/A	Mispred_branches	Retired Mispredicted Branch Instructions
0xC4	N/A	Taken_branches	Retired Taken Branch Instructions

Formulas. Here are branch-related derived measurements:

```
Branch rate = Branches / Ret_instructions
Branch misprediction rate = Mispred_branches / Ret_instructions
Branch misprediction ratio = Mispred_branches / Branches
```

Compute the rate at which branches are taken and the ratio of the number of instructions per branch using these formulas:

```
Branch taken rate = Taken_branches / Ret_instructions
Branch taken ratio = Taken_branches / Branches
Instructions per branch = Ret_instructions / Branches
```

Interpretation. The branch rate is a measure of the relative frequency of branches in executed code. The branch rate will be relatively high in applications with a lot of decision-making logic. The branch misprediction rate indicates how often mispredictions are made for a given set of retired instructions. Ideally, this rate should be low. The reciprocal of this ratio -- retired instructions per mispredicted branch -- is sometimes easier to understand and apply. The branch misprediction ratio indicates how often branch instructions are mispredicted. This ratio should be as small as possible.

Ideally, one would like to compute the branch misprediction ratio and branch taken ratio for individual conditional branch instructions. This would allow a developer to identify specific, poorly predicted branches and improve their predictability. Unfortunately, skid does not allow precise attribution of branch events to individual instructions.

The instructions per branch ratio indicates the number of straight-line instructions executed before encountering a potentially disruptive branch. The microarchitecture favors code with a high instructions-per-branch ratio. Developers can put more work through the pipeline before redirecting the pipeline to a different control path. Procedure inlining and loop unrolling may increase the number of instructions per branch.

Improve performance by replacing branches with straight-line code wherever possible. This increases the instructions per branch ratio, too. Engineers and developers can sometimes eliminate conditional branches using conditional move instructions.

Reduce branch mispredictions by increasing the likelihood of taking one direction of a branch over the other. Even a small increase in the likelihood (for example, from 49% to 51%) may be enough to increase prediction and performance significantly.

4.5.2. Near return.

Applicability. Near return mispredictions should be measured in programs that make frequent subroutine calls, that call a large number of different subroutines, or that use recursion extensively.

Collection. Collect the events in this table to measure return prediction:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0xC8	N/A	Near_returns	Retired Near Returns
0xC9	N/A	Mispred_near_ret	Retired Near Returns Mispredicted

Formulas. Derived measurements for mispredicted near returns are:

```
Near return rate = Near_returns / Ret_instructions
Return stack miss rate = Mispred_near_ret / Ret_instructions
Return stack misprediction ratio = Mispred_near_ret / Near_returns
```

Estimate the number of instructions per subroutine call using this formula:

```
Instructions per call = Ret_instructions / Near_returns
```

Interpretation. The microarchitecture uses a Return Address Stack to predict the target address of a near return. Return addresses are pushed onto the stack when a call is made and are popped to predict the destination of a return. When a return address is pushed and the stack is full, the stack overflows and discards the oldest entry. AMD Family 10h processors provide a 24-entry Return Address Stack; the previous generation of AMD Athlon 64 and AMD Opteron processors uses a 12-entry stack.

The return stack miss rate shows how often returns are mispredicted for a given set of retired instructions. Use the rate to judge the severity of return stack mispredictions. The return stack misprediction ratio indicates how often near returns are mispredicted.

Returns are mispredicted when the return stack is empty (stack underflow) or when there is an imbalance between calls and returns. Compilers sometimes remove returns through intraprocedural analysis. For example, if the compiler determines that a leaf procedure always returns up through a call chain to a particular ancestral procedure, then the compiler can generate a return directly to the ancestor. Return stack underflow may also occur with deep recursion.

Further investigate return stack hits and overflows by measuring the events in this table:

Event select	Unit mask	Event abbreviation	Event
0x88	N/A	Ret_stk_hits	Return Stack Hits
0x89	N/A	Ret_stk_overflow	Return Stack Overflows

4.6. Special cases.

Measurements in this section deal with specific practical situations, including those that occur in scientific and engineering applications.

4.6.1. Unaligned data access.

Applicability. Most compilers align common types of data structures. Unaligned access to data items may occur in programs using dynamically allocated memory, pointer arithmetic, or packed data. Problems with unaligned data access are more likely in code written for and ported from another machine architecture.

Collection. Measure the number and severity of misaligned data accesses using these events:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0x40	N/A	DC_accesses	Data Cache Accesses
0x47	N/A	Misalign_access	Misaligned Accesses

Formulas. Developers can assess the performance impact of misaligned data accesses using these derived measurements:

```
Misaligned access rate = Misalign_access / Ret_instructions
Misaligned access ratio = Misalign_access / DC_accesses
```

Interpretation. Developers can safely ignore the performance impact of infrequent misaligned accesses. If misaligned accesses are occurring within a hot spot, developers should eliminate them.

4.6.2. Floating point.

4.6.2.1. Floating point operations.

Applicability. Use these measurements to gauge the proportion (density) of floating point operations in an application program.

Collection. Measure the occurrence of floating point operations and instructions with these events:

Event select	Unit mask	Event abbreviation	Event
`0xC0`	`N/A`	`Ret_instructions`	`Retired Instructions`
`0x00`	`0x07`	`Dispatched_FP`	`Dispatched FPU Operations`
`0xCB`	`FAMDEP`	`Ret_MMX_FP`	`Retired MMX/FP Instructions`

Use the unit mask for the Dispatched FPU Operations event to select add, multiply, or store operations specifically. The unit mask for the Retired MMX/FP Instructions event (select `0xCB`) is family-dependent (FAMDEP). On AMD Family 10h processors, the unit mask should be `0x07`. Otherwise, the unit mask should be `0x0F`. This unit mask can specifically select x87 instructions, MMX/3DNow!™ instructions, or SSE/SSE2 instructions, allowing analysis to focus on specific types of operations or instructions (see the appropriate, processor-specific BKDG for further configuration details).

These events include non-numeric operations and cannot estimate the floating operations per second (FLOPS) rate of a program.

Formulas. Derived measurements for FP-related instructions are:

```
FPU op rate = Dispatched_FP / Ret_instructions
FP/MMX rate = Ret_MMX_FP / Ret_instructions
```

Interpretation. Assess the degree of floating point computation performed by an application using the FPU operation rate. Packed SSE family instructions offer higher performance than scalar SSE instructions or x87 floating point instructions. Choose compiler options to generate (packed) SSE instructions for floating point arithmetic.

4.6.2.2. FLOPS rate.

Applicability. The FLOPS rate is a common measure of floating point throughput; the high-performance computing (HPC) community often uses it.

Collection. AMD Family 10h processors provide a direct means to measure floating point operations. Compute the FLOPS rate using the events in this table:

Event select	Unit mask	Event abbreviation	Event
`0x76`	`N/A`	`CPU_clocks`	`CPU Clocks Not Halted`
`0x03`	`0x47`	`SSE_SP_FLOPS`	`Retired SSE Operations [Single precision]`
`0x03`	`0x78`	`SSE_DP_FLOPS`	`Retired SSE Operations [Double precision]`

The unit mask values for the Retired SSE Operations event selects between single- and double-precision floating point operations as performed by SSE instructions. The unit masks can also be used to break down floating operations by three categories: add/subtract, multiply, and divide/square root (see the BKDG for AMD Family 10h processors for further configuration details).

CPU Clocks Not Halted is a high-frequency event and we recommend a sampling period of 500,000 for this event. We suggest an initial sampling period of 50,000 for the Retired SSE Operations event. If the density of floating point operations is relatively high, increase the sampling period to 500,000.

Formulas. These formulas give the single- and double-precision FLOPS rates:

```
Single precision FLOPS rate = SSE_SP_FLOPS / Seconds
Double precision FLOPS rate = SSE_DP_FLOPS / Seconds
```

We estimate the elapsed time using the CPU Clocks Not Halted event:

```
Seconds = (CPU_clocks * Period) / Clock_frequency

    Where Period is the sampling period, and
          Clock_frequency is the platform CPU clock frequency
```

Take care when measuring time with this event. The CPU run state and clock frequency throttling affect the CPU Clocks Not Halted event. We recommend using a more reliable time reference, such as the operating system clock or TSC.

Interpretation. The FLOPS rate is a raw measure of floating point throughput and indicates the number of floating operations performed within a specific period of time. The FLOPS rate may vary across program phases and the program's ability to stream FP data from memory into the CPU affects it. A low FLOPS rate may indicate a problem with data layout, access pattern, or read latency, especially when the actual, measured memory bandwidth is low.

4.6.2.3. Floating point exceptions.

Applicability. Floating point exceptions occur to properly round a denormal or when an SSE operation has been supplied data of the wrong type. Only measure floating point exceptions if an application is performing a large number of floating point computations and, of course, ignore them in integer-only applications.

Collection. Collect these events to detect the occurrence and severity of floating point exceptions:

Event select	Unit mask	Event abbreviation	Event
0xC0	N/A	Ret_instructions	Retired Instructions
0xCB	FAMDEP	Ret_MMX_FP	Retired MMX/FP Instructions
0xDB	0x0F	FPU_exceptions	FPU Exceptions

The unit mask for the FPU Exceptions event can isolate specific types of floating point exceptions. The unit mask for the Retired MMX/FP Instructions event (select `0xCB`) is family dependent. On AMD Family 10h processors, the unit mask should be `0x07`. Otherwise, the unit mask should be `0x0F` (see the BKDG for configuration details).

Formulas. These measurements help determine the relative frequency of floating point exceptions:

```
Overall FP exception rate = FPU_exceptions / Ret_instructions
FP exception rate = FPU_exceptions / Ret_MMX_FP
```

Interpretation. Microcode handles recovery from a floating point exception. This causes poor floating point performance; if the rate of occurrence is high, the developer should identify and correct the root cause.

5. Example measurements.

This section demonstrates the use of a few selected performance measurements to analyze the behavior of an application.

In the white paper titled "An introduction to analysis and optimization with AMD CodeAnalyst," we analyzed the behavior of two versions of a short, easy-to-comprehend matrix multiplication program:

http://developer.amd.com/pages/111820052_9.aspx

We will use the same example programs here. The algorithm in the first version of the program uses the classic, textbook implementation of matrix multiplication. The second version of the program interchanges the order of the loop nest to improve the memory access pattern. The second version of the program runs much faster than the textbook implementation (2.1 seconds versus 12.6 seconds).

We used AMD CodeAnalyst to collect the event data needed to compute efficiency, level 1 (L1) data cache, and DTLB measurements. We collected event data on an AMD Athlon 64 processor using AMD CodeAnalyst, which displays event profile data at the process, module, function, source, and instruction levels. We used the events and computations described in Section 4 to compute derived measurements at the function level. The event counts and measurements reported in Subsections 5.1-5.4 of this paper reflect the behavior of the function "multiply_matrices" in the sample programs.

5.1. IPC measurements.

The ratio of instructions per cycle is the most basic measurement of computational efficiency. It indicates the degree to which the hardware is able to exploit instruction level parallelism in a program.

To compute IPC, we collected data for Retired Instructions and CPU Clocks Not Halted (processor cycles) (see Section 4.2.1 for configuration information and other details). This table shows the event data collected for the multiply_matrices() function for both the "classic" and "improved" versions of the program:

Event abbreviation	Classic matrix multiply	Improved matrix multiply
`CPU_clocks`	`506,251 samples`	`80,977 samples`
`Ret_instructions`	`68,183 samples`	`88,124 samples`

The number of processor cycles is much higher for the slower textbook implementation, as expected considering the longer execution time of that version of the program.

After applying the formulas in Section 4.2.1, we obtain the comparison of relative performance shown in this table:

Measurement	Classic matrix multiply	Improved matrix multiply
Elapsed time (sec)	12.546	2.140
IPC ratio	0.135	1.088
CPI ratio	7.425	0.919

If we were analyzing the classic program for the first time, its low IPC would have indicated the presence of a performance issue requiring investigation. The short stride access pattern implemented in the improved version of the matrix multiplication program dramatically improves the ratio of instructions per cycle (by a factor of eight).

5.2. Memory bandwidth measurements.

Memory bandwidth (discussed in Section 4.2.2) shows how effectively data is transferred to and from memory. The test platform was a 2.2 GHz AMD Athlon 64 with a single memory controller, using the same sampling period (50,000) for all four events collected by AMD CodeAnalyst: CPU Clocks Not Halted (processor cycles), System Read Responses, Quadwords Written to System, and DRAM Accesses:

Event abbreviation	Classic matrix multiply	Improved matrix multiply
CPU_clocks	505,137 samples	88,120 samples
System_read	1,266 samples	1,256 samples
System_write	169 samples	48 samples
DRAM_accesses	1,292 samples	1,262 samples

The number of read operation samples is roughly the same in both programs. The smaller number of processor cycle samples reflects the shorter runtime of the improved matrix multiplication program. With a smaller number of processor cycle samples, the bandwidth for the improved matrix multiplication is better than the classic textbook version of the program:

Measurement	Classic matrix multiply	Improved matrix multiply
Elapsed time (sec)	12.7960	2.1240
Seconds	11.4804	2.0027
Read bandwidth (MB/sec)	352.8797	2006.8907
Write bandwidth (MB/sec)	5.8883	9.5871
DRAM bandwidth (MB/sec)	360.1268	2016.4778

5.3. L1 data cache measurements.

We know the classic textbook implementation of matrix multiplication to have a poor memory access pattern. The long stride through one of the argument arrays incurs a large number of data cache (DC) and data translation lookaside buffer (DTLB) misses. Both kinds of misses delay computation while reading data from the memory subsystem.

Section 4.3.1 discusses L1 data cache analysis. We collected event data for Retired Instructions, Data Cache Accesses, Data Cache Refills From L2 Cache and Data Cache Refills From System:

Event abbreviation	Classic matrix multiply	Improved matrix multiply
`Ret_instructions`	`68,184 samples`	`88,153 samples`
`DC_accesses`	`401,893 samples`	`602,380 samples`
`DC_refills_L2`	`45,533 samples`	`11,713 samples`
`DC_refills_sys`	`12,526 samples`	`870 samples`

Approximately 4.6 times as many data cache misses occurred when running the classic matrix multiplication program compared to the improved version, even though the improved version actually accessed memory more often. These derived measurements reflect the higher request rate:

Measurement	Classic matrix multiply	Improved matrix multiply
`Elapsed time (sec)`	`12.859`	`3.469`
`DC request rate`	`0.589`	`0.683`
`DC miss rate`	`0.085`	`0.014`
`DC miss ratio`	`0.144`	`0.021`

An L1 data cache miss occurred every 11.8 (1/0.085) instructions in the classic version, while an L1 data cache miss occurred every 71.5 (1/0.014) instructions when running the improved version.

5.4. DTLB measurements.

To measure DTLB performance, we collected sample data for Retired Instructions, Data Cache Accesses, L1 DTLB Miss and L2 DTLB Hit, and L1 DTLB and L2 DTLB Miss events (see Section 4.4.1).

Event abbreviation	Classic matrix multiply	Improved matrix multiply
`Ret_instructions`	`68,180 samples`	`88,150 samples`
`DC_accesses`	`402,415 samples`	`602,298 samples`
`DTLB_L1M_L2H`	`59,532 samples`	`53 samples`
`DTLB_L1M_L2M`	`157,529 samples`	`175 samples`

In their paper titled "On Reducing TLB Misses in Matrix Multiplication," Kazushige Goto and Robert van de Geijn assert that TLB misses are the limiting factor in fast matrix multiplication. The event data supports their claim.

We derived these measurements from the event data:

Measurement	Classic matrix multiply	Improved matrix multiply

Elapsed time (sec)	13.2340	3.4370
L1 DTLB request rate	0.5902	0.6833
L1 DTLB miss rate	0.3184	0.0003
L1 DTLB miss ratio	0.5394	0.0004
L2 DTLB request rate	0.3184	0.0003
L2 DTLB miss rate	0.2310	0.0002
L2 DTLB miss ratio	0.7257	0.7675

The L1 DTLB request rate is higher for the improved version since it performs more memory access operations than the classic version. For the textbook program, an L1 DTLB miss occurs every 3.1 instructions and an L2 DTLB miss occurs every 4.3 instructions -- clearly unacceptable. The improved matrix multiplication program executes at least 3,300 instructions per DTLB miss.

6.0 Acknowledgments.

I thank Dave Christie for his clear description and explanation of the performance measurement events. I also thank Anoop Iyer, whose work is the source of many of the measurements described in this paper. Special thanks go to Anton Chernoff and Keith Lowery, who reviewed this article.

Paul Drongowski is a Senior Member of Technical Staff at AMD. He is a member of the AMD CodeAnalyst team and has worked on profiling tools and performance analysis for more than ten years. In addition to industrial experience, he has taught computer architecture, software development, and VLSI design at Case Western Reserve University, Tufts, and Princeton.

Paul J. Drongowski
AMD CodeAnalyst™ Performance Analyzer Development Team
Advanced Micro Devices, Inc.
Boston Design Center