

An introduction to analysis and optimization with AMD CodeAnalyst™ Performance Analyzer

Paul J. Drongowski
AMD CodeAnalyst Team
Advanced Micro Devices, Inc.
Boston Design Center
8 September 2008

Introduction

This technical note demonstrates how to use the AMD CodeAnalyst™ Performance Analyzer to analyze and improve the performance of a compute-bound program.

The program that we chose for this demonstration is an old classic: matrix multiplication. We'll start with a "textbook" implementation of matrix multiply that has well-known memory access issues. We will measure and analyze its performance using AMD CodeAnalyst. Then, we will improve the performance of the program by changing its memory access pattern.

1. AMD CodeAnalyst

AMD CodeAnalyst is a suite of performance analysis tools for AMD processors. Versions of AMD CodeAnalyst are available for both Microsoft® Windows® and Linux®. AMD CodeAnalyst may be downloaded (free of charge) from [AMD Developer Central](http://developer.amd.com). (Go to <http://developer.amd.com> and click on CPU Tools.) Although we will use AMD CodeAnalyst for Windows in this tech note, engineers and developers can use the same techniques to analyze programs on Linux.

AMD CodeAnalyst performs system-wide profiling and supports the analysis of both user applications and kernel-mode software. It provides five main types of data collection and analysis:

- Time-based profiling (TBP),
- Event-based profiling (EBP),
- Instruction-based sampling (IBS),
- Pipeline simulation (Windows-only feature), and
- Thread profiling (Windows-only feature).

We will look at the first three kinds of analysis in this note. Performance analysis usually begins with time-based profiling to identify the program hot spots that are candidates for optimization. Then we may apply event-based profiling and instruction-based sampling to those hot spots to diagnose performance issues.

We will not discuss step-by-step details of taking measurements and navigating the graphical user interface. The tutorial and help system that are part of the AMD CodeAnalyst distribution cover the user interface in detail.

2. Baseline program

Matrix multiplication is a simple problem and familiar to all of us. Our goal is to multiply two conformable matrices to produce a third matrix, which is the matrix product. Given three matrices represented in the arrays `matrix_a`, `matrix_b`, and `matrix_r`, the classic C language implementation of matrix multiply is:

```
void multiply_matrices()
{
    for (int i = 0 ; i < ROWS ; i++) {
        for (int j = 0 ; j < COLUMNS ; j++) {
            float sum = 0.0 ;
            for (int k = 0 ; k < COLUMNS ; k++) {
                sum = sum + matrix_a[i][k] * matrix_b[k][j] ;
            }
            matrix_r[i][j] = sum ;
        }
    }
}
```

The program assumes that the matrices are square. The arrays `matrix_a` and `matrix_b` contain the two operand matrices. The matrix product is written into the array `matrix_r`. In our example, each array is 1000 by 1000 elements in size (ROWS is 1000 and COLUMNS is 1000). [Appendix A](#) shows the complete baseline program.

The C language stores the individual array elements in row major order. That is, the array elements in a row occupy sequential, adjacent locations in memory. Note that, in the function above, the iteration variable `k` changes most frequently since it is defined within the innermost loop. As `k` changes, it will step through the elements of a row in `matrix_a` and the elements of a column in `matrix_b`. The innermost loop walks through `matrix_a` in small sequential strides and walks through `matrix_b` in big strides. The small strides through `matrix_a` are, of course, more cache memory-friendly than the big strides through `matrix_b`. It's no secret that the classic implementation is inefficient because large strides miss more often in the data cache. The large strides also cause misses in the data translation lookaside buffers (DTLB) – another significant source of slowdown.

Note: Fortran stores individual array elements in column major order (i.e., array elements in a column occupy sequential, adjacent locations in memory.)

The program takes 16.9 seconds to execute on our test platform, a quad-core AMD Opteron™ processor.

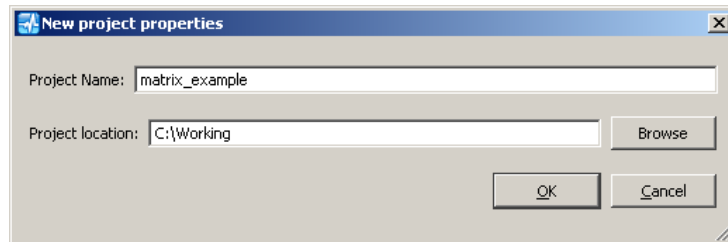
3. Time-based profiling (TBP)

The first step in performance analysis is to identify the hot spots in the program under evaluation. A "hot spot" is a code region where the program spends most of its time. Hot spots are the best candidates for performance improvement. Let's assume that the program has two code regions: one region taking 90% of execution time and the other region taking 10%. If analysis and improvements reduce execution time by 10%, then improving the region taking 90% of the time will yield a 9% overall gain. If the colder region is improved, a 10% reduction will yield only 1% overall. Thus, it's better to concentrate on the most lucrative (hottest) candidates.

Time-based profiling (TBP) is the appropriate tool for the job because it provides a profile showing where the program spends its time and it does this with relatively little measurement overhead. AMD CodeAnalyst uses a timer to periodically interrupt and sample the program. It records the point at which the program was executing. Over time, it collects a large number of samples and accumulates a statistically accurate picture -- a histogram of samples. AMD CodeAnalyst collects more samples for the code regions where the program spends most its time because those regions are more likely to be sampled. The resulting profile (histogram) shows the distribution of samples. The tallest peaks in the distribution are the hottest code regions.

To analyze a program in AMD CodeAnalyst, we first create a new project. A project contains configuration information and organizes performance test runs into sessions; each test run is a session. AMD CodeAnalyst saves performance data collected during a test run in a session. Create a new project by selecting the "File > New" menu item and then filling out the new project properties dialog box (Figure 1). Fields within this dialog box specify the name of the project and the parent directory of the files associate with the project.

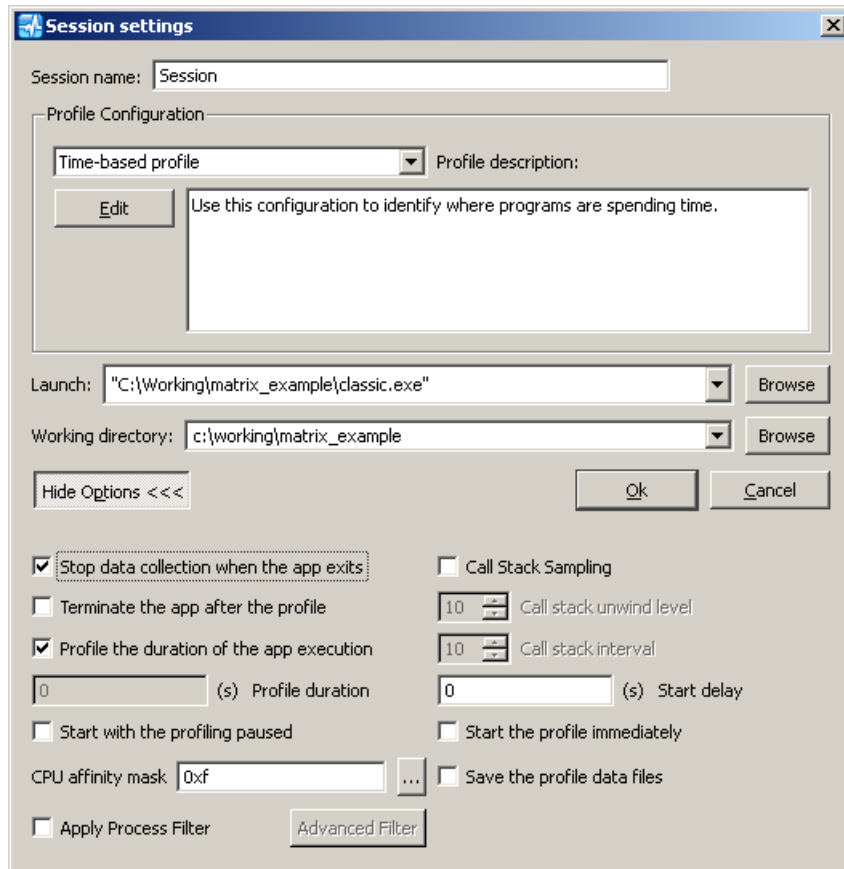
Figure 1 – New project dialog box



After creating a new project, AMD CodeAnalyst displays the session settings dialog box (Figure 2), allowing us to choose the kind of analysis to perform and configure how to launch and monitor the application program. AMD CodeAnalyst provides several pre-defined kinds of analyses. Select "Time-based profile" from the list of profile configurations to collect a time-based profile. Fill in the program to launch and the working directory. If the application takes command line arguments, enter them after the application program name. AMD CodeAnalyst is also able to launch a test script (a shell or "BAT" file), making it easier to set up and execute performance experiments.

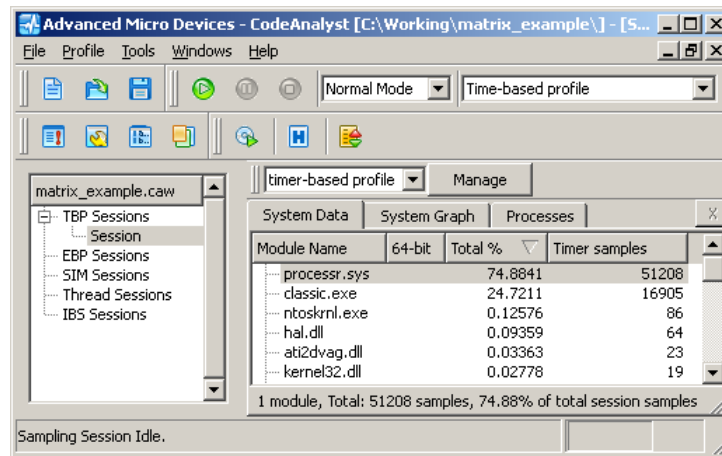
Extended configuration parameters control data collection and performance monitoring. (If necessary, click the "Show options" button to reveal the extended parameters and options.) In this case, select "Stop data collection when the app exits" and "Profile the duration of the app execution," which tell AMD CodeAnalyst to collect performance data only while the application program is running.

Figure 2 – Session settings dialog box



Now that we have configured time-based profiling, AMD CodeAnalyst enables the sampling control buttons in its toolbar. The control buttons resemble the buttons on a CD or DVD player (see the green start button under the “Windows” menu in Figure 3). The buttons start, pause, and stop data collection. Click the start button to begin data collection and to launch the program you plan to monitor.

Figure 3 - TBP System Data tab



After the program terminates, AMD CodeAnalyst adds a new session below “TBP Sessions” in the project management window (Figure 3). It also displays profile information in three tabs, which we will explore in detail in the next few pages:

- System Data shows a module-by-module breakdown of timer samples in table form;
- System Graph shows the same breakdown in chart form; and,
- Processes shows a process-by-process breakdown of timer samples in table form.

The tables and chart provide an overview of the system's performance and let us zoom in to find hot spots and possible performance issues.

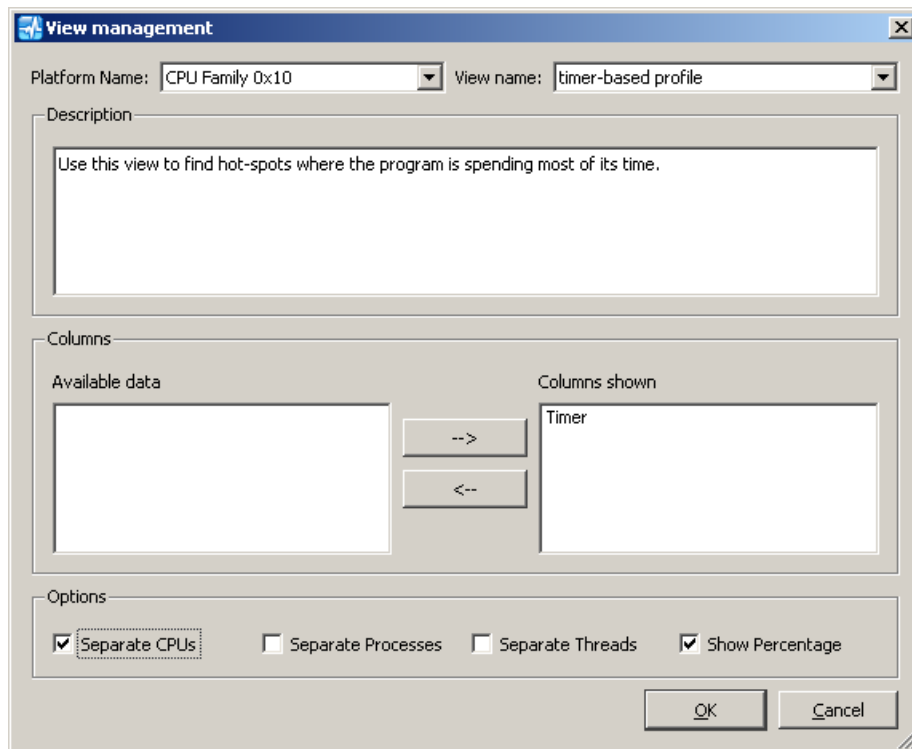
The System Data tab contains a table that shows where the system spent its time during a test run. Since AMD CodeAnalyst performs system-wide sampling, the System Data table shows the number of samples collected for all executable modules that were active during the measurement period such as:

- The matrix multiplication program (`classic.exe`),
- The CPU driver (`processor.sys`),
- The operating system kernel (`ntoskrnl.exe`),
- The Windows Hardware Abstraction Layer (`hal.dll`),
- The ATI display driver (`ati2dvag.dll`), and
- The interface to the Windows operating system kernel (`kernel32.dll`).

We will focus our attention on the matrix multiplication program.

By default, AMD CodeAnalyst uses a one-millisecond sampling period (that is, AMD CodeAnalyst takes a sample every millisecond.) Thus, each reported sample has an approximate weight of one millisecond. Absolute time units (seconds) can be derived from timer samples by multiplying the sample count times the sampling period. The number of timer samples (16,905) for `classic.exe` is consistent with an elapsed execution time of 16.9 seconds. Figure 3 shows that nearly 99.5% of all samples have been attributed to two modules: `classic.exe` and `processor.sys`. The first module is the matrix multiplication program and the second module is the operating system's CPU driver. The CPU driver contains the Windows idle loop that runs when no other program is available to execute on the CPU. The matrix multiplication program is single-threaded and keeps only one core of the quad-core test platform busy.

Figure 4 – View management dialog box



We can explore CPU utilization in more detail by separating the timer samples by core. Click the Manage button to display the view management dialog box (Figure 4). Select the "Separate CPUs" option to display a breakout of timer samples by core (Figure 5).

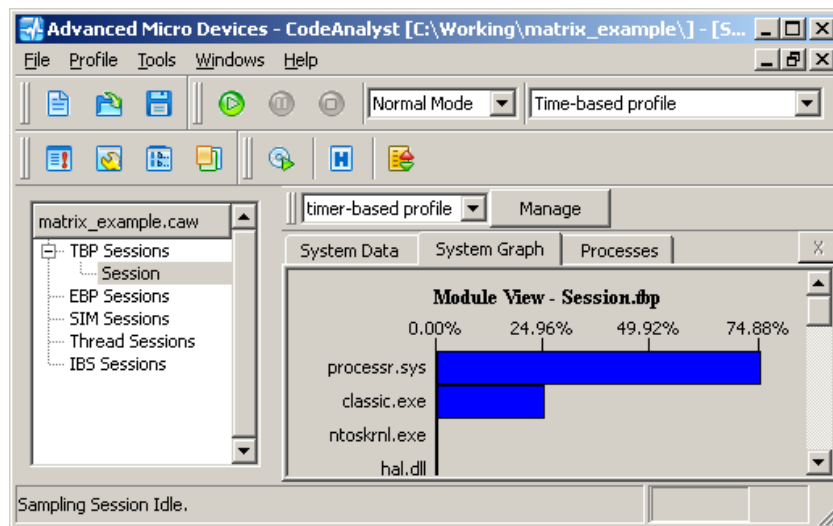
Figure 5 - TBP System Data per individual core

Module Name	64-bit	Total %	Timer samples - C0	Timer samples - C1	Timer samples - C2	Timer samples - C3
processr.sys		74.8841	17024	8505	17057	8622
classic.exe		24.7211	0	8552	0	8353
ntoskrnl.exe		0.12576	36	10	17	23
hal.dll		0.09359	26	12	3	23
ati2dvag.dll		0.03363	3	3	1	16
kernel32.dll		0.02778	1	0	0	18

1 module, Total: 51208 samples, 74.88% of total session samples

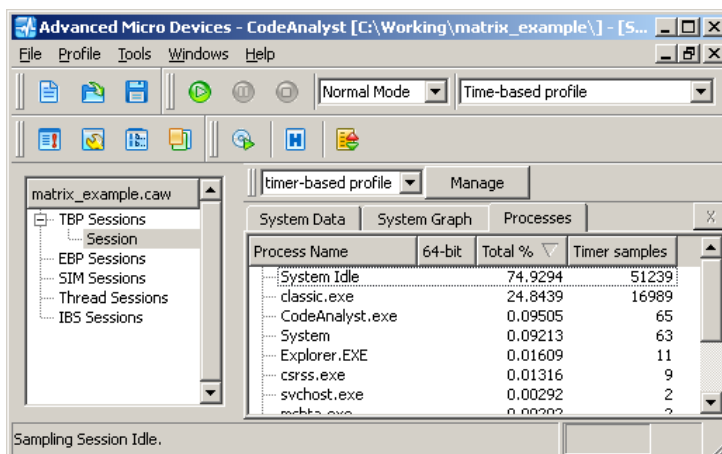
The processor in our test platform is quad-core, so the System Data table displays four columns to show a core-by-core breakdown of timer samples for each module. The matrix multiplication program spent about half of its time executing on the second core (C1) and the remaining half on the fourth core (C3). The idle loop in `processor.sys` takes up most of the rest of the time. The compute equivalent of three cores was wasted while the matrix multiplication program was running. If the application divided the work of the matrix multiplication program into four threads, we might have kept all cores busy and potentially further cut execution time.

Figure 6 - TBP System Graph tab



The System Graph tab in Figure 6 shows the distribution of timer samples graphically, aggregating samples from all four cores. (For the rest of the analyses in this article, we will aggregate measurements across all four cores.)

Figure 7 - TBP Processes tab



The Processes tab (Figure 7) shows the breakdown of timer samples by process. Under the “Total%” column, this table clearly shows the 25-75 percent split between the matrix multiplication program and the system idle process. Only 65 timer samples were attributed to AMD CodeAnalyst itself, reflecting its low collection overhead.

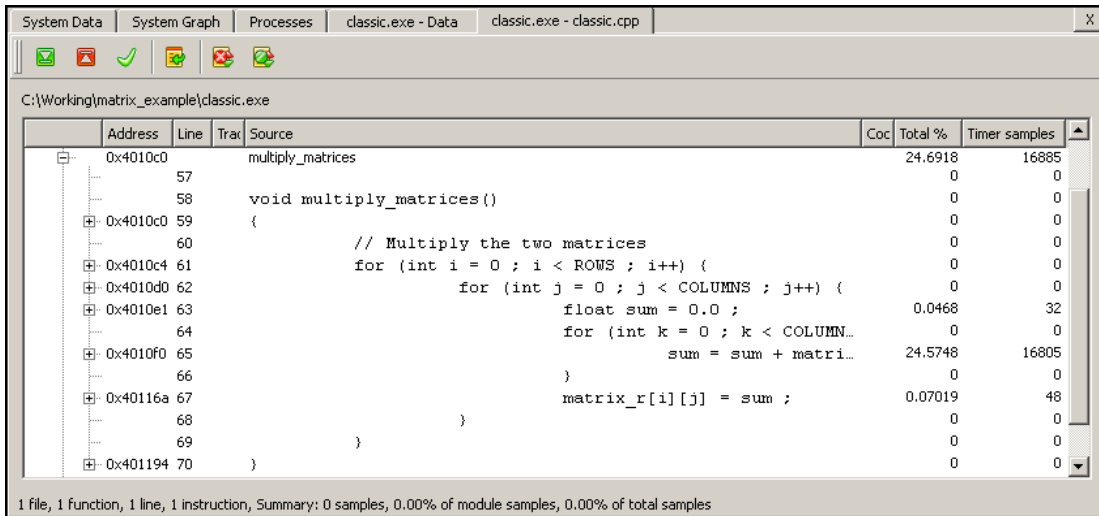
Figure 8 - TBP function view

CS:EIP	Symbol + Offset	64-bit	Total %	Timer samples
0x4010c0	multiply_matrices		24.6918	16885
0x401000	initialize_matrices		0.01755	12
0x4019a0	_getptd		0.00877	6
0x401287	rand		0.00292	2

1 function, 37 instructions, Total: 16885 samples, 99.88% of samples in the m...

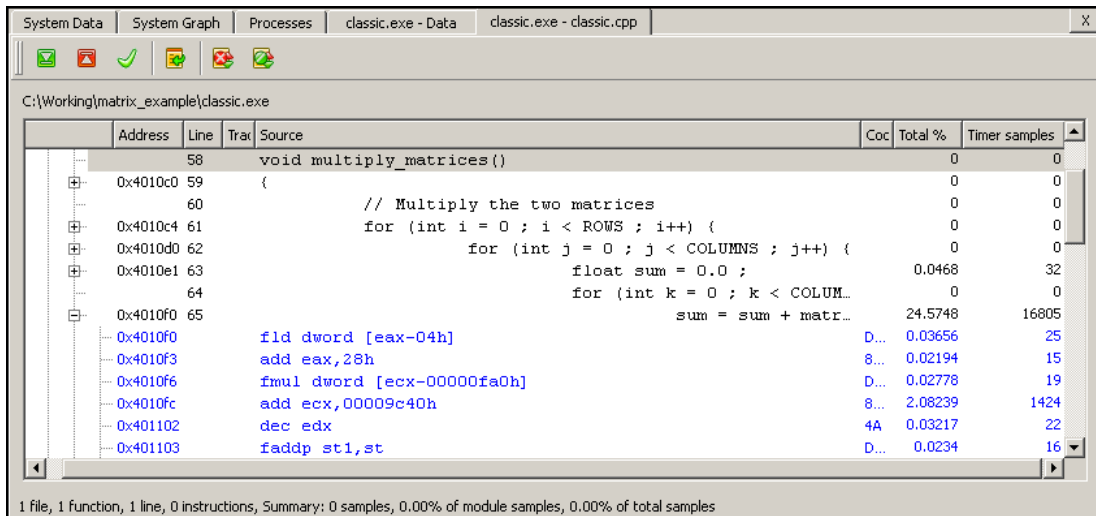
We can drill down to the functions within the matrix multiplication program by double-clicking on the `classic.exe` entry in the System Data table or by double-clicking on the bar representing `classic.exe` in the System Graph. Double-clicking on `classic.exe` in the System Data table produces a table displaying the distribution of timer samples across the functions in the matrix multiplication program (Figure 8). This table shows that the application spent most of the execution time in the function `multiply_matrices`. Double-click on the entry for `multiply_matrices` to drill down into the function.

Figure 9 - Source for `multiply_matrices`



The source tab shows the distribution of timer samples across the C language statements in the function `multiply_matrices` (Figure 9). Most of the samples fall on the statement that is the body of the innermost loop. This statement reads an element from each array, multiplies the two elements, and adds the product to a running sum of products.

Figure 10 - Disassembled instructions for `multiply_matrices`



AMD CodeAnalyst can display both annotated source code and the disassembled instructions associated with the source code (Figure 10.) Users can enable or disable display of assembler code by clicking the "Asm" button (not shown in the figure above).

Users may also show or hide disassembled instructions on a statement-by-statement basis by clicking on the "+" or "-" sign, as appropriate, in the expansion tree to the left of the source language statements. Sometimes it helps to see the kind of code the compiler generates in order to adjust compiler code generation options.

We chose to suppress the code density chart displayed at the top of the source view, allowing AMD CodeAnalyst to display more statements in the source view. We hid the code density chart by selecting the "CodeAnalyst Options" item in the Tools menu to display the CodeAnalyst Options dialog box, then removing the check from the "Show code density chart" item. Enable the code density chart again by checking this item.

4. Event-based profiling (EBP)

Through the simple steps demonstrated in the previous section, we were able to drill down to the hottest code region in the matrix multiplication program quickly. At this point, we could examine the source code or instructions to reason about the algorithm and its data access patterns, and apply one of many standard source-level code improvement techniques.

Often, though, we need more information to determine if a particular code region is suffering a performance issue. AMD processors are equipped with performance monitoring hardware to measure and collect data about the behavior of programs. An "event" is a hardware condition or action caused by a program as it executes. AMD CodeAnalyst samples these events and builds up an event-based profile, a statistically accurate picture of a program's behavior over time.

As mentioned earlier, AMD CodeAnalyst provides several predefined kinds of analyses. We used the "Time-based profile" configuration to collect and analyze a time-based profile. Five predefined profile configurations use event-based profiling for data collection and analysis (Table 1).

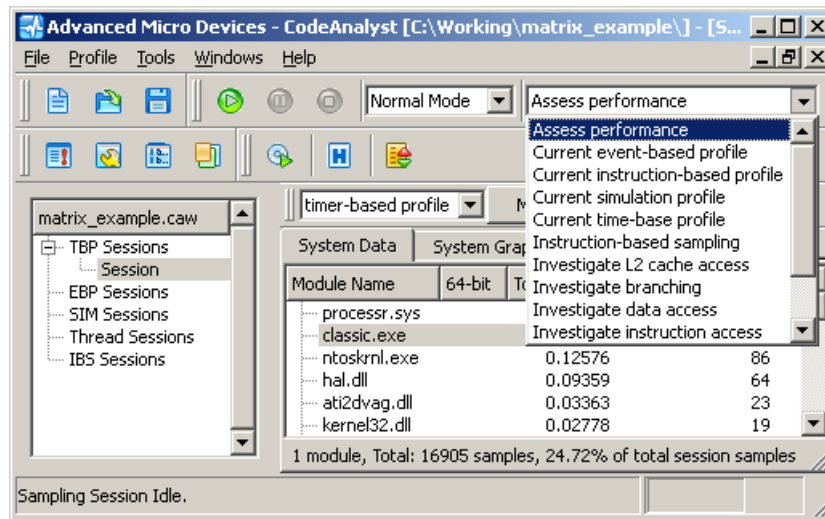
Table 1 – EBP profile configurations

Profile configuration	Purpose
Assess performance	Get an overview of data access/branch behavior
Investigate data access	Analyze data cache and DTLB behavior
Investigate L2 cache access	Analyze L2 cache behavior
Investigate instruction access	Analyze instruction cache and ITLB behavior
Investigate branching	Analyze branch behavior including prediction

Choose the "Assess performance" profile configuration from the drop-down list in the toolbar (Figure 11). This configuration provides an overview of data access and branch behavior, which is good for initial performance triage. It's a good place to start and may indicate an area for in-depth analysis using the other EBP or instruction-based sampling (IBS) profile configurations. You may also choose your own performance events by editing the "Current event-based profile" configuration.

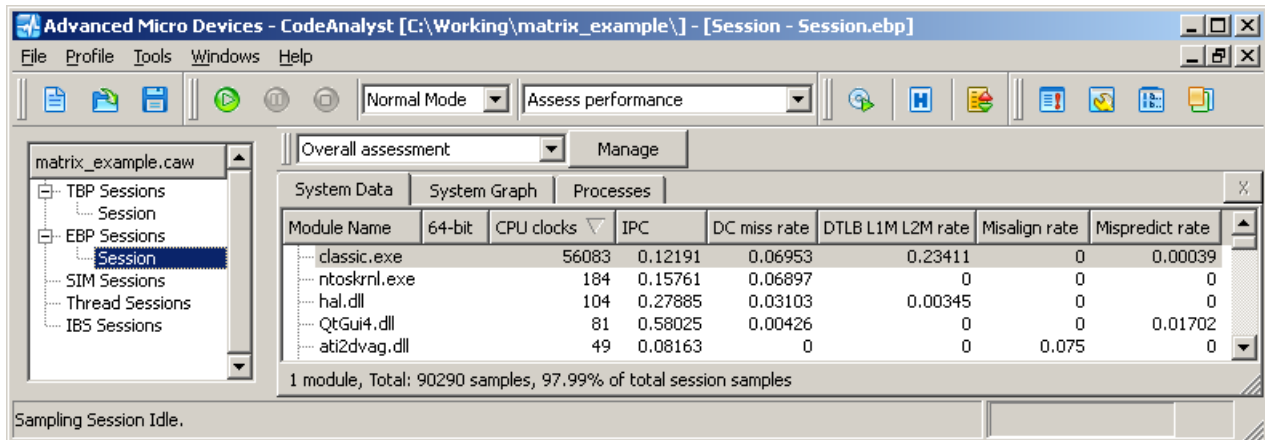
Note: The on-line help documentation discusses the creation of custom profile configurations. Please consult the sections on "Configuration management." If you would like to experiment with this feature, choose the "Configuration management" item from the Tools menu.

Figure 11 - EBP event configuration



Click the start button in the toolbar to launch the matrix multiplication program and begin EBP data collection. AMD CodeAnalyst displays a System Data tab once data collection completes (Figure 12).

Figure 12 - EBP System Data tab



The “Overall assessment” view shows six key indicators of performance. The CPU clocks column contains the number of samples taken for the CPU Clocks Not Halted event. This event is often called “processor cycles” (or just “cycles”) and reflects active CPU usage. The number of cycle samples should correlate with the number of TBP samples we measured earlier. CPU cycles show the hottest, most time-consuming code regions. IPC, “instructions per cycle,” is the number of instructions executed per cycle. IPC is a rough measure of efficiency or instruction-level parallelism (ILP) and indicates how well a program makes overall use of hardware resources.

DC miss rate is the number of level 1 (L1) data cache (DC) misses per retired instruction. This rate indicates how often read and write operations are missing in the primary, high-speed data cache. If a read or write misses, the data must be retrieved from the level 2 (L2) cache. If the access misses in the L2 cache, too, then the data must be retrieved from (optional) L3 cache or primary memory. Data cache hits are better than misses because access time to L2 cache is longer than L1 cache and access time to primary memory is much longer than even L2 cache. The relative penalty between primary memory and D-cache is a factor of 50 or more depending on CPU clock speed, memory speed, etc.

Note: A full description of memory system behavior is beyond the scope of this introductory note.

The DTLB L1M L2M rate indicates how often address translation operations missed in both level 1 and level 2 of the data translation lookaside buffer. The rate is the number of level 1 and level 2 DTLB misses per retired instruction. The DTLB is a hardware unit that assists the translation of virtual addresses into physical memory addresses. The AMD Opteron processor in the test platform has a two-level DTLB. Mapping information is first sought in the L1 DTLB, then the L2 DTLB. If mapping information cannot be found in the L2 DTLB, it must be retrieved from operating system page tables -- a relatively expensive operation. The DTLB L1M L2M rate measures the frequency of the most expensive DTLB misses.

The misalign rate is the number of misaligned data accesses per retired instruction. Access to naturally aligned memory data is generally faster than misaligned access. For example, access to a double-precision floating point value is faster if it naturally aligns to an 8-byte address boundary. The misalign rate measures the frequency of misaligned data access operations.

Finally, the mispredict rate is the number of mispredicted branches per retired instruction. Superscalar processors depend on a steady stream of incoming instructions to keep the execution units busy. Conditional branches change control flow and potentially disrupt the stream of fetched instructions. To keep the stream flowing, processors predict the likely program control flow path and aggressively fetch (and partially execute) instructions along the predicted path. If a prediction is wrong, then the processor must discard work and begin fetching along the correct program path. The mispredict rate indicates how frequently the processor incorrectly predicted the control path.

So, let’s apply these measures to the matrix multiplication program (Figure 12). The `classic.exe` module is the hottest module, with 56,083 cycle samples. An IPC of 0.122 is very poor: barely one-tenth of an instruction is executing each machine cycle. Double-click on `classic.exe` to drill down to the function level.

The function `multiply_matrices` has the most processor cycle samples (Figure 13). Its IPC is 0.120, indicating the presence of a performance bottleneck. The misalign rate and the mispredict rates are both very low. Therefore, these factors are unlikely to be the root cause of the performance issue and we can eliminate them from consideration.

The data cache miss rate is roughly 70.6 misses per thousand (retired) instructions. This is nearly one miss every ten instructions – moderately high and worthy of further investigation. The L1 and L2 DTLB miss rate is 237.8 misses per thousand instructions. An expensive DTLB miss occurred once every five instructions – an unacceptably high rate of occurrence. The long strides through the array `matrix_b` cause the DTLB misses. The likely cause of the data cache misses is poor cache line utilization, a secondary symptom of the long stride issue.

Figure 13 – Function table (Overall assessment view)

CS:EIP	Symbol + Offset	64-bit	CPU clocks	IPC	DC miss rate	DTLB L1M L2M rate	Misalign rate	Mispredict rate
0x4010c0	multiply_matrices		55991	0.12022	0.07063	0.2378	0	0.0004
0x401000	initialize_matrices		42	1.02381	0	0	0	0
0x4019a0	_getptd		30	1.06667	0	0	0	0
0x401287	rand		20	1.55	0	0	0	0

1 function, 37 instructions, Total: 89710 samples, 99.36% of samples in the module, 97.36% of total session samples

Event-based profiling supports drill-down in much the same way as time-based profiling. Double-click on `multiply_matrices` to drill down to the source code for the function and examine the body of the function (Figure 14). Most of the event samples are associated with the assignment statement within the nested loops, which was the hot spot identified through time-based profiling.

Figure 14 – Source for `multiply_matrices`

Source	Coc	CPU clocks	IPC	DC miss rate
void multiply_matrices()			0	0
{			0	0
// Multiply the two matrices			0	0
for (int i = 0 ; i < ROWS ; i++) {			0	0
for (int j = 0 ; j < COLUMNS ; ...			0	0
float sum = 0.0 ;		134	0.08955	0.16667
for (int k = 0 ; k < ...			0	0
sum = sum +...		55728	0.12035	0.07027
}			0	0
matrix_r[i][j] = sum ;		129	0.09302	0.175
}			0	0
}			0	0
}			0	0

Other predefined views are ready for use when the appropriate event data is available. Check the drop-down list of views next to the view management (“Manage”) button (Figure 15). Choose the “Data access assessment” view to get more detailed information about data cache access behavior.

Figure 15 – Change view to “Data access assessment”

-bit	CPU clocks	IPC	DC miss rate	DTLB L1M L2M	Misalign rate	Mispredict rate
55991	0.12022	0.07063	0.2378	0	0	0.0004
42	1.02381	0	0	0	0	0
30	1.06667	0	0	0	0	0
20	1.55	0	0	0	0	0

1 function, 37 instructions, Total: 89710 samples, 99.36% of samples in the module, 97.36% of total session samples

The “Data access assessment” view contains additional rates and ratios that characterize the program’s data access behavior (Figure 16). The DC access rate is the number of read/write operations per retired instruction. It

describes how often the program accesses data through the memory subsystem (beginning with the L1 data cache). The data access rate is 615.8 read/write operations per thousand instructions. The matrix multiplication program is a “memory-intensive” application that frequently accesses memory data. The DC miss ratio is the portion of read/write operations that missed in the L1 data cache. Approximately 11.4% of all memory operations missed in the L1 data cache.

Figure 16 – Function table (Data access assessment view)

CS:EIP	Symbol + Offset	64-bit	Ret inst	DC accesses	DC misses	DC access rate	DC miss rate	DC miss ratio
0x4010c0	multiply_matrices		6731	4145	4754	0.61581	0.07063	0.11469
0x401000	initialize_matrices		43	36	0	0.83721	0	0
0x4019a0	_getptd		32	38	0	1.1875	0	0
0x401287	rand		31	24	0	0.77419	0	0

1 function, 37 instructions, Total: 89710 samples, 99.36% of samples in the module, 97.36% of total session samples

The “All data” view shows the raw number of event samples taken for each event (Figure 17).

Figure 17 – Function table (All data view)

Symbol + Offset	64-bit	DC access	DC misses	Misalign	CPU clocks	Ret inst	Ret branch	Ret misp branch	DTLB L1M L2M
0 multiply_matrices		4145	4754	0	55991	6731	2056	27	16006
0 _getptd		38	0	0	30	32	139	0	0
0 initialize_matrices		36	0	0	42	43	58	0	0
7 rand		24	0	0	20	31	87	0	0

1 function, 37 instructions, Total: 89710 samples, 99.36% of samples in the module, 97.36% of total session samples

5. Instruction-Based Sampling (IBS)

Before discussing how to improve the matrix multiplication program, let's take a quick look at instruction-based sampling. IBS is a hardware-level, precise event-monitoring technique supported by AMD CodeAnalyst. IBS is available on AMD Family 10h processors, which include certain quad-core AMD Opteron and AMD Phenom™ processors.

IBS is an improvement over EBP because it precisely attributes hardware events to the instructions that cause them. The cause and effect relationship between instructions and performance-related behavior is unambiguous, making it easier to identify performance culprits. The following example illustrates the benefits of IBS.

Event-based profiling uses performance monitor counter (PMC) sampling to collect profile data. One or more PMCs are configured to measure a performance event. When the sampling period for the event expires, the current instruction pointer (IP) value is recorded and a sample is stored by software. Ideally, the IP value identifies the instruction that caused the triggering event. Unfortunately, the sampled IP is, at best, in the neighborhood of the actual instruction that caused the triggering event. Imprecise attribution leads to profiles in which event samples are distributed throughout the neighborhood near the actual performance culprits. This makes it difficult to identify culprit instructions directly.

For example, EBP data access events are distributed throughout the body of the inner loop in the function `multiply_matrices` (Figure 18). The `fld` and `fmul` instructions read array elements. Consider the fourth instruction (`add ecx,00009c40h`). This instruction does not read or write memory. However, it has a large number of data cache access and data cache miss event samples. This identifies it as one of the hottest memory access instructions within the loop body. This is clearly a false positive.

Figure 18 – EBP data access events

Source	Code Bytes	Ret inst	DC accesses	DC misses	DC access rate
<code>fld dword [eax-04h]</code>	D9 40 FC	11	78	2	9
<code>add eax,28h</code>	83 C0 28	17	61	4	17
<code>fmul dword [ecx-00000fa0h]</code>	D8 89 60 F0 FF FF	8	66	0	12
<code>add ecx,00009c40h</code>	81 C1 40 9C 00 00	539	4651	479	1521
<code>dec edx</code>	4A	11	60	1	3
<code>faddp st1,st</code>	DE C1	12	76	2	1
<code>fld dword [ecx-00009c40h]</code>	D9 81 C0 63 FF FF	102	544	18	43
<code>fmul dword [eax-28h]</code>	D8 48 D8	648	4686	464	1559
<code>faddp st1,st</code>	DE C1	53	262	15	2
<code>fld dword [eax-24h]</code>	D9 40 DC	173	750	22	64
<code>fmul dword [ecx-00008ca0h]</code>	D8 89 60 73 FF FF	12	74	1	11
<code>faddp st1,st</code>	DE C1	683	4818	420	1552
<code>fld dword [eax-20h]</code>	D9 40 E0	172	746	29	51

IBS periodically selects and tags pipeline operations. IBS remembers the instruction address associated with a tagged operation and monitors the tagged operation. The pipeline hardware records events caused by the operation. When IBS takes a sample, it reports both the events and the instruction address. Thus, IBS precisely associates events with the operations and instructions that caused them.

Figure 19 – IBS load/store operations and events

Source	Code Bytes	IBS load/store	IBS load	IBS store	IBS DC miss	
fld dword [eax-04h]	D9 40 FC	3381	3381	0	49	
add eax,28h	83 C0 28	0	0	0	0	
fmul dword [ecx-00000fa0h]	D8 89 60 F0 FF FF	3476	3476	0	805	
add ecx,00009c40h	81 C1 40 9C 00 00	0	0	0	0	
dec edx	4A	0	0	0	0	
faddp st1,st	DE C1	0	0	0	0	
fld dword [ecx-00009c40h]	D9 81 C0 63 FF FF	3465	3465	0	736	
fmul dword [eax-28h]	D8 48 D8	3503	3503	0	7	
faddp st1,st	DE C1	0	0	0	0	
fld dword [eax-24h]	D9 40 DC	2470	2470	0	43	
fmul dword [ecx-00008ca0h]	D8 89 60 73 FF FF	2329	2329	0	464	
faddp st1,st	DE C1	0	0	0	0	
fld dword [eax-20h]	D9 40 E0	2826	2826	0	5	

IBS monitors a wide range of instruction fetch and execution behavior. Many PMCs would be required to measure the equivalent EBP events. For example, IBS monitors memory load operations, store operations, and L1 data cache misses. Figure 19 shows the IBS profile data for the same instructions as shown in Figure 18. Only memory access instructions are identified by IBS as performing load or store operations. Similarly, data cache misses are correctly associated with those instructions that access memory. Memory-related events are not incorrectly attributed to the fourth instruction (`add ecx,00009c40h`).

IBS complements EBP. We can use EBP to measure the rate at which certain events are occurring, and it is generally good enough to associate events with a hot code region. We can use IBS to further isolate performance issues to specific instructions (operations) within a hot code region, providing additional diagnostic clues.

You can enable IBS by selecting the "Instruction-Based Sampling" item in the drop-down list of profile configurations.

6. Improved matrix multiplication program

As mentioned in the introduction, the cause of poor performance in the textbook implementation of matrix multiplication is its poor data access pattern. In this section, we will change the matrix multiplication program to improve its performance and then take a new set of AMD CodeAnalyst measurements.

Fortunately, each iteration of the matrix multiplication loop is independent of previous iterations when running sums accumulate directly into the result array. This observation lets us interchange the nesting of the two innermost loops, as shown below. Find the full code for the new program in [Appendix B](#).

```
void multiply_matrices()
{
    for (int i = 0 ; i < ROWS ; i++) {
        for (int k = 0 ; k < COLUMNS ; k++) {
            for (int j = 0 ; j < COLUMNS ; j++) {
                matrix_r[i][j] = matrix_r[i][j] + matrix_a[i][k] * matrix_b[k][j] ;
            }
        }
    }
}
```

With this rearrangement, the index variables `j` and `k` change the most frequently. Variables `j` and `k` provide the column indices. All steps through the arrays are now unit strides. The program walks through each array sequentially, and this linear access pattern leads to better data cache behavior. DTLB behavior also improves because misses to the L1 and L2 DTLBs should be less frequent.

Note: Loop interchange cannot be applied to all array-based programs. Dependencies between loop iterations and the particular access pattern (i.e., combination of array indices) may prevent interchange or successful conversion of all strides to the unit stride. More general techniques, such as array blocking, may be required.

Now, let's run the program again and take measurements to see if performance has improved. The matrix multiplication programs contain some simple code to measure elapsed execution time. Execution time decreased from 16.9 seconds to 1.7 seconds -- a substantial improvement. The TBP measurements reflect the decrease in execution time. The number of timer samples decreased from 16,905 timer samples to 1,623 timer samples. Table 2 summarizes the results.

Table 2 – Comparison of baseline vs. improved matrix multiplication program

Measurement	Baseline	Improved
Elapsed time (sec)	16.9	1.7
IPC	0.12022	1.60784
DC access rate	0.61581	0.73122
DC miss rate	0.07063	0.00071
DC miss ratio	0.11469	0.00097
DTLB request rate	0.61581	0.73122
DTLB L1M L2M miss rate	0.23780	0.00019
DTLB L1M L2M miss ratio	0.38615	0.00025
Misalign rate	0.00000	0.00000
Branch rate	0.03055	0.02603
Mispredict rate	0.00040	0.00024
Mispredict ratio	0.01313	0.00934

Comparing event data from the two different programs reveals that our changes had the desired effect on data cache and DTLB behavior.

CPU cycles decreased in proportion to elapsed execution time. The rate of data cache misses and the rate of L1 and L2 DTLB misses decreased substantially. The IPC ratio improved from 0.12 to 1.61. The new code obtained these gains even though the number of retired x86 instructions actually *increased* (6,731 samples vs. 8,536 samples). Memory-related program behavior has a strong effect on overall performance.

Summary

We demonstrated how to use AMD CodeAnalyst to analyze a memory-intensive application program. We also showed how to improve the program and how to use AMD CodeAnalyst to assess performance improvements.

Can the performance of the program be improved still further? Here are a few other experiments to try:

Data blocking: Data blocking subdivides array computations into smaller tiles or blocks that fit into cache memory. Although there is usually a small penalty to create the blocks, substantial performance gains are possible through better cache memory behavior.

Flag mining: In the example above, we just accepted and used the default optimization level offered by the Microsoft Visual Studio compiler. Better performance may be possible at higher levels of optimization and through using SSE/SSE2 instructions.

Loop unrolling and software pipelining: Loop unrolling expands a loop body into multiple copies, thereby reducing the loop overhead. Loop unrolling is often combined with software pipelining to overlap memory access with computation. Software pipelining takes advantage of the number of registers provided by the AMD64 architecture.

Data prefetching: Explicit data prefetching gives the processor hints about the memory data items needed next. The memory subsystem will start to bring those items into cache, making them available when needed.

Multithreading: If you can separate a computation into independent parts, you can assign those parts to separate CPU cores for execution. Many hands make short work. Multithreading takes direct advantage of the multi-core revolution sparked by AMD.

Now you're ready to use AMD CodeAnalyst Performance Analyzer.

Acknowledgments

I thank my colleagues Anton Chernoff, Frank Swehosky, and Suravee Suthikulpanit for reviewing this article. I also thank the whole AMD CodeAnalyst team for their dedication to our product.

Biography

Paul Drongowski is a Senior Member of Technical Staff at AMD. He is a member of the AMD CodeAnalyst Performance Analyzer development 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.

Appendix A - Baseline matrix multiplication program

[Download code sample from AMD Developer Central](#)

```
// classic.cpp : "Textbook" implementation of matrix multiply

// Author: Paul J. Drongowski
// Address: Boston Design Center
//          Advanced Micro Devices, Inc.
//          Boxborough, MA 01719
// Date:    20 October 2005
//
// Copyright (c) 2005 Advanced Micro Devices, Inc.

// The purpose of this program is to demonstrate measurement
// and analysis of program performance using AMD CodeAnalyst(tm).
// All engineers are familiar with simple matrix multiplication,
// so this example should be easy to understand.
//
// This implementation of matrix multiplication is a direct
// translation of the "classic" textbook formula for matrix multiply.
// Performance of the classic implementation is affected by an
// inefficient data access pattern, which we should be able to
// identify using CodeAnalyst(TM).

#include <cstdlib>
#include <cstdio>
#include <ctime>

static const int ROWS = 1000 ; // Number of rows in each matrix
static const int COLUMNS = 1000 ; // Number of columns in each matrix

float matrix_a[ROWS][COLUMNS] ; // Left matrix operand
float matrix_b[ROWS][COLUMNS] ; // Right matrix operand
float matrix_r[ROWS][COLUMNS] ; // Matrix result

FILE *result_file ;

void initialize_matrices()
{
    // Define initial contents of the matrices
    for (int i = 0 ; i < ROWS ; i++) {
        for (int j = 0 ; j < COLUMNS ; j++) {
            matrix_a[i][j] = (float) rand() / RAND_MAX ;
            matrix_b[i][j] = (float) rand() / RAND_MAX ;
            matrix_r[i][j] = 0.0 ;
        }
    }
}

void print_result()
{
    // Print the result matrix
    for (int i = 0 ; i < ROWS ; i++) {
        for (int j = 0 ; j < COLUMNS ; j++) {
            fprintf(result_file, "%6.4f ", matrix_r[i][j]) ;
        }
        fprintf(result_file, "\n") ;
    }
}
```

```

void multiply_matrices()
{
    // Multiply the two matrices
    for (int i = 0 ; i < ROWS ; i++) {
        for (int j = 0 ; j < COLUMNS ; j++) {
            float sum = 0.0 ;
            for (int k = 0 ; k < COLUMNS ; k++) {
                sum = sum + matrix_a[i][k] * matrix_b[k][j] ;
            }
            matrix_r[i][j] = sum ;
        }
    }
}

void print_elapsed_time()
{
    double elapsed ;
    double resolution ;

    // Obtain and display elapsed execution time
    elapsed = (double) clock() / CLK_TCK ;
    resolution = 1.0 / CLK_TCK ;

    fprintf(result_file,
        "Elapsed time: %8.4f sec (%6.4f sec resolution)\n",
        elapsed, resolution) ;
}

int main(int argc, char* argv[])
{
    if ((result_file = fopen("classic.txt", "w")) == NULL) {
        fprintf(stderr, "Couldn't open result file\n") ;
        perror("classic") ;
        return( EXIT_FAILURE ) ;
    }

    fprintf(result_file, "Classic matrix multiplication\n") ;

    initialize_matrices() ;
    multiply_matrices() ;
    print_elapsed_time() ;

    fclose(result_file) ;

    return( 0 ) ;
}

```

Appendix B - Improved matrix multiplication program

[Download code sample from AMD Developer Central](#)

```
// interchange.cpp : Matrix multiply with loop nest interchange

// Author: Paul J. Drongowski
// Address: Boston Design Center
//          Advanced Micro Devices, Inc.
//          Boxborough, MA 01719
// Version: 1.1
// Date:    26 October 2005
//
// Copyright (c) 2005 Advanced Micro Devices, Inc.

// The purpose of this program is to demonstrate measurement
// and analysis of program performance using AMD CodeAnalyst(tm).
// All engineers are familiar with simple matrix multiplication,
// so this example should be easy to understand.
//
// This implementation of matrix multiplication interchanges
// the nesting of the innermost loops in order to improve the
// programs data access pattern. The improved pattern steps
// sequentially through array elements in memory with a small
// stride. Data cache and data translation lookaside buffer (DTLB)
// misses should be reduced.

#include <cstdlib>
#include <cstdio>
#include <ctime>

static const int N = 1000 ;    // Dimensions
static const int K = 1000 ;
static const int M = 1000 ;

float matrix_a[N][K] ;    // Left matrix operand
float matrix_b[K][M] ;    // Right matrix operand
float matrix_r[N][M] ;    // Matrix result

FILE *result_file ;

void initialize_matrices()
{
    // Define initial contents of the matrices
    for (int i = 0 ; i < N ; i++) {
        for (int j = 0 ; j < K ; j++) {
            matrix_a[i][j] = (float) rand() / RAND_MAX ;
        }
    }
    for (int i = 0 ; i < K ; i++) {
        for (int j = 0 ; j < M ; j++) {
            matrix_b[i][j] = (float) rand() / RAND_MAX ;
        }
    }
    for (int i = 0 ; i < N ; i++) {
        for (int j = 0 ; j < M ; j++) {
            matrix_r[i][j] = 0.0 ;
        }
    }
}
```

```

void print_result()
{
    // Print the result matrix
    for (int i = 0 ; i < N ; i++) {
        for (int j = 0 ; j < M ; j++) {
            fprintf(result_file, "%6.4f ", matrix_r[i][j]) ;
        }
        fprintf(result_file, "\n") ;
    }
}

void multiply_matrices()
{
    // Multiply the two matrices
    //
    // Please note that the nesting of the innermost
    // loops has been changed. The index variables j
    // and k change the most frequently and the access
    // pattern through the operand matrices is sequential
    // using a small stride (one.) This changes improves
    // access to memory data through the data cache. Data
    // translation lookaside buffer (DTLB) behavior is
    // also improved.
    for (int i = 0 ; i < N ; i++) {
        for (int k = 0 ; k < K ; k++) {
            for (int j = 0 ; j < M ; j++) {
                matrix_r[i][j] = matrix_r[i][j] +
                    matrix_a[i][k] * matrix_b[k][j] ;
            }
        }
    }
}

void print_elapsed_time()
{
    double elapsed ;
    double resolution ;

    // Obtain and display elapsed execution time
    elapsed = (double) clock() / CLK_TCK ;
    resolution = 1.0 / CLK_TCK ;

    fprintf(result_file,
        "Elapsed time: %8.4f sec (%6.4f sec resolution)\n",
        elapsed, resolution) ;
}

int main(int argc, char* argv[])
{
    if ((result_file = fopen("interchange.txt", "w")) == NULL) {
        fprintf(stderr, "Couldn't open result file\n") ;
        perror("interchange") ;
        return( EXIT_FAILURE ) ;
    }

    fprintf(result_file, "Matrix multiplication with loop nest interchange\n") ;

    initialize_matrices() ;
    multiply_matrices() ;
    print_elapsed_time() ;
    fclose(result_file) ;
    return( 0 ) ;
}

```