# Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors

Paul J. Drongowski
AMD CodeAnalyst Project
Advanced Micro Devices, Inc.
Boston Design Center

16 November 2007

## 1. Introduction

Software applications must use computational resources efficiently in order to deliver best results in a timely manner. This is especially true for time-sensitive applications such as transaction processing, real-time control, multi-media and games. A program profile is a histogram that reflects dynamic program behavior. For example, a profile shows where a program is spending its time. Program profiling helps software developers to meet performance goals by identifying performance bottlenecks and issues. Profiling is most effective when a developer can quickly identify the location and cause of a performance issue.

Instruction-Based Sampling (IBS) is a new profiling technique that provides rich, precise program performance information. IBS is introduced by AMD Family10h processors (AMD Opteron Quad-Core processor "Barcelona.") IBS overcomes the limitations of conventional performance counter sampling. Data collected through performance counter sampling is not precise enough to isolate performance issues to individual instructions. IBS, however, precisely identifies instructions which are not making the best use of the processor pipeline and memory hierarchy. IBS collects a wide range of performance information in a single program run, making it easier to conduct performance testing. The AMD CodeAnalyst performance analysis tool suite supports IBS and correlates the instruction-level IBS information with processes, program modules, functions and source code. IBS in combination with CodeAnalyst helps a developer to find, analyze and ameliorate performance problems.

This technical note is a brief introduction to Instruction-Based Sampling. It shows the kind of information produced by IBS and how that information can be used for performance analysis.

## 2. Matrix multiplication: An example

We will demonstrate the advantages of Instruction-Based Sampling by applying IBS to a matrix multiplication program. Although the example program is small, IBS scales to large applications.

The matrix multiplication program, called "simple_classic," implements the classic, textbook approach to matrix multiplication. This algorithm has well-known memory access performance issues and demonstrates IBS in action.

The full source code for simple_classic is given in Appendix A. The heart of simple_classic is the function *multiply_matrices*, which multiplies two 1000x1000 matrices together and puts the result into a third matrix.

Here is the source code for *multiply_matrices*:

```
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 ;
    }
  }
}
```

C and C++ lay out two-dimensional arrays in row major order; the elements within a row of an array are arranged sequentially in memory. Sequential memory access is advantageous for good performance since data cache locality is improved and hardware-level prefetching can anticipate data access.

Memory access issues in the classic matrix multiplication algorithm arise from non-sequential accesses to one of the operand arrays, in this case, *matrix_b*. The fastest changing array index, *k*, touches a different row of *matrix_b* on each iteration. Since each row of the array is nearly as large as a 4K byte memory page, the long stride through memory causes both data cache (DC) misses and data translation lookaside buffer (DTLB) misses.

We will first measure the memory access behavior of simple_classic using conventional performance counter sampling to show its limitations. Then we will measure the memory behavior of simple_classic using IBS and compare the results.

## 3. Performance counter sampling

AMD processors provide performance monitoring counters (PMC) to measure important hardware events that occur during program execution. The word "program" here refers to any executing software component including the operating system, device drivers, and libraries as well as the application itself. Performance counter sampling uses the PMCs to measure the occurrence of hardware events like retired instructions, DC misses and DTLB misses. Each PMC is configured to measure a particular event. The number of events that can be measured in a single performance test is limited by the number of counters. AMD Family 10h Processors have four PMCs and support performance counter sampling in addition to IBS.

Performance counter sampling is a statistical technique that produces information called an “event sample” after the occurrence of a pre-configured number of events. The instruction address associated with the event sample is the value of the instruction pointer (IP) at the time the sample was taken. This is usually the restart address stored on the stack by the sampling interrupt and is generally not the address of the instruction that caused the triggering event.

It is difficult to relate a hardware event to the instruction that triggered it because the restart address is not the location after the trigger instruction. Contemporary superscalar machines such

as AMD quad-core processors use out-of-order execution to exploit instruction-level parallelism. Up to 72 execution operations may be in-flight at any time. Due to operation reordering and in-order instruction retirement, the sampling interrupt triggered by an execution event may be significantly delayed. The delay is indeterminate and is not fixed. The reporting delay is called "skid." Due to skid, the reported IP value is only in the general neighborhood of the instruction causing the event and may be up to 72 instructions away.

Inaccuracies due to skid accumulate as the program profile is built up. Events that belong to a single instruction are attributed to instructions throughout the neighborhood of the culprit instruction. The ability to isolate a performance issue to any single instruction is lost.

To demonstrate the effects of skid, we used PMC sampling to collect a profile for the example matrix multiplication program. We compiled the simple_classic program with optimizations turned off in order to keep the generated machine code simple and relatively short. The generated code is a fairly literal translation of the three nested loops in *multiply_matrices*. We will concentrate on the innermost loop, since this is the hottest code region in the function.

The following table shows the memory access profile of the innermost loop of *multiply_matrices*. (The complete PMC memory access profile for *multiply_matrices* appears in Appendix B.) The inner loop is implemented by the 16 instructions starting at address 0x401191. The body of the loop reads the array elements from the operand matrices, multiplies the elements together, and adds the product to the running sum. The running sum is stored on the runtime stack at the location specified by [ebp-0Ch].

| Address | Instruction | Ret inst | DC accesses | DC misses | DTLB L1M L2M |
|---|---|---|---|---|---|
| 00401191 | `mov edx,dword ptr [ebp-10h]` | 1291 | 1763 | 7 | 81 |
| 00401194 | `add edx,1` | 1330 | 1551 | 62 | 91 |
| 00401197 | `mov dword ptr [ebp-10h],edx` | 0 | 0 | 0 | 0 |
| 0040119A | `cmp dword ptr [ebp-10h],1000` | 0 | 2 | 0 | 1 |
| 004011A1 | `jge 004011D1` | 1693 | 1353 | 288 | 42 |
| 004011A3 | `mov eax,dword ptr [ebp-4]` | 0 | 0 | 0 | 0 |
| 004011A6 | `imul eax,eax,4000` | 0 | 0 | 0 | 0 |
| 004011AC | `mov ecx,dword ptr [ebp-10h]` | 2196 | 1813 | 45 | 42 |
| 004011AF | `imul ecx,ecx,4000` | 0 | 0 | 0 | 0 |
| 004011B5 | `mov edx,dword ptr [ebp-10h]` | 1818 | 1579 | 10 | 20 |
| 004011B8 | `mov esi,dword ptr [ebp-8]` | 0 | 0 | 0 | 0 |
| 004011BB | `fld dword ptr [eax+edx*4+413FE0h]` | 0 | 0 | 0 | 0 |
| 004011C2 | `fmul dword ptr [ecx+esi*4+7E48E8h]` | 1165 | 1107 | 9 | 218 |
| 004011C9 | `fadd dword ptr [ebp-0Ch]` | 33298 | 22415 | 409 | 2093 |
| 004011CC | `fstp dword ptr [ebp-0Ch]` | 17983 | 8363 | 41 | 256 |
| 004011CF | `jmp 00401191` | 3365 | 4146 | 47 | 276 |

From left to right, the events reported are retired instructions, DC accesses, DC misses and address translations which missed in both the level 1 and level 2 DTLBs. These latter DTLB misses have a high performance penalty.

The instruction at 0x4011C2 reads an individual element from *matrix_b* and multiplies it with the element that was read from *matrix_a*. Since access to *matrix_b* is non-sequential, we would expect this instruction to be the source of DC misses and DTLB misses. However, due to skid and other inaccuracies, the DC misses and DTLB misses are spread across several other

instructions in the code region. Most of the DTLB misses are attributed to the three instructions after the culprit and to the jump instruction at address 0x4011A1. This kind of inaccuracy makes precise attribution of events to the actual culprits impossible in code regions with multiple load/store operations. Lack of precision complicates analysis.

All data reported in this note were collected using AMD CodeAnalyst executing on a quad-core AMD Family 10h processor. Each core provides four PMCs. Without the performance counter multiplexing provided by AMD CodeAnalyst, we would be limited to measuring only four hardware events in a single test run and multiple runs would be required to measure additional events. It is easier and more cost-effective to collect all performance data in a single test run as some performance experiments are difficult to conduct due to long run-time, platform constraints or non-trivial user interaction with the application. Performance counter multiplexing makes it possible to measure many events in one run, but comes at the cost of reduced statistical accuracy.

## 4. Instruction-Based Sampling

Instruction-Based Sampling is a feature introduced in AMD Family 10h processors. Although IBS is a statistical method, the sampling technique delivers precise event information and eliminates inaccuracies due to skid.

The processor pipeline has two main phases: instruction fetch and instruction execution. The fetch phase supplies instruction bytes to the decoder. Decoded AMD64 instructions are executed during the execution phase as discrete operations called “ops.” Since the two phases are decoupled, IBS provides two forms of sampling: fetch sampling and op sampling. IBS fetch sampling provides information about the fetch phase and IBS op sampling provides information about the execution phase.

IBS fetch sampling and IBS op sampling use a similar sampling technique. The IBS hardware selects an operation periodically based on a configurable sampling period. The selected operation is tagged and the operation is monitored as it proceeds through the pipeline. Events caused by the operation are recorded. When the operation completes, the event information and the fetch (or instruction) address associated with the operation are reported to the profiler. Thus, events are precisely attributed to the instruction that caused them. IBS does not impose any overhead on instruction fetch or execution -- everything runs at full speed.

### 4.1 IBS fetch sampling

IBS fetch sampling counts completed fetches and periodically selects a fetch to be tagged and monitored. Several kinds of information are collected:

- The fetch address
- Whether the fetch completed or aborted
- Whether the fetch missed in the instruction cache (IC)
- Whether the fetch missed in the level 1 or level 2 instruction translation lookaside buffer (ITLB)
- The page size of the address translation
- The fetch latency, i.e., cycles from when the fetch was initiated to when the fetch either completed or aborted

This information is collected with every IBS fetch sample and is not restricted by the number of available performance counters.

The table below summarizes the IBS fetch samples that were collected for the *multiply_matrices* function.

| Address | All | Killed | Attempted | Completed | Aborted | IC miss |
|---|---|---|---|---|---|---|
| 401180 | 6 | 0 | 6 | 6 | 0 | 0 |
| 401191 | 7 | 0 | 7 | 7 | 0 | 0 |
| 4011a0 | 4040 | 4 | 4036 | 4036 | 0 | 0 |
| 4011c0 | 3955 | 2 | 3953 | 3953 | 0 | 0 |
| 4011d1 | 5 | 0 | 5 | 5 | 0 | 0 |
| 4011e0 | 4020 | 4019 | 1 | 1 | 0 | 0 |

Each row of the table shows the number of IBS fetch samples collected for an address (the All column) and a breakdown of the events reported by the samples. For example, six IBS fetch samples were taken for the address 0x401180 and all six of those samples were attempted fetches that completed.

Instruction fetch is a speculative activity that anticipates architectural control flow. Fetch operations may be abandoned due to control flow redirection. Some fetch operations are abandoned at a very early stage before address translation. These killed fetches (the third column in the table above) are not useful for analysis and are filtered out by CodeAnalyst. The remaining fetches (column four) are regarded as true fetch attempts. An attempted fetch may either complete and deliver instruction bytes to the decoder (column five), or abort (column six.) Since the matrix multiplication program is so small, it fits entirely in the instruction cache and no IC misses were observed.

The following table shows the IBS fetch results reported by AMD CodeAnalyst for the inner loop of *multiply_matrices*. (The IBS fetch information for the entire function appears in Appendix C.)

| Address | Instruction | All | Killed | Attempted | Completed | Aborted |
|---|---|---|---|---|---|---|
| 00401191 | `mov edx,dword ptr [ebp-10h]` | 7 | 0 | 7 | 7 | 0 |
| 00401194 | `add edx,1` | 0 | 0 | 0 | 0 | 0 |
| 00401197 | `mov dword ptr [ebp-10h],edx` | 0 | 0 | 0 | 0 | 0 |
| 0040119A | `cmp dword ptr [ebp-10h],1000` | 4040 | 4 | 4036 | 4036 | 0 |
| 004011A1 | `jge 004011D1` | 0 | 0 | 0 | 0 | 0 |
| 004011A3 | `mov eax,dword ptr [ebp-4]` | 0 | 0 | 0 | 0 | 0 |
| 004011A6 | `imul eax,eax,4000` | 0 | 0 | 0 | 0 | 0 |
| 004011AC | `mov ecx,dword ptr [ebp-10h]` | 0 | 0 | 0 | 0 | 0 |
| 004011AF | `imul ecx,ecx,4000` | 0 | 0 | 0 | 0 | 0 |
| 004011B5 | `mov edx,dword ptr [ebp-10h]` | 0 | 0 | 0 | 0 | 0 |
| 004011B8 | `mov esi,dword ptr [ebp-8]` | 0 | 0 | 0 | 0 | 0 |
| 004011BB | `fld dword ptr [eax+edx*4+413FE0h]` | 3955 | 2 | 3953 | 3953 | 0 |
| 004011C2 | `fmul dword ptr [ecx+esi*4+7E48E8h]` | 0 | 0 | 0 | 0 | 0 |
| 004011C9 | `fadd dword ptr [ebp-0Ch]` | 0 | 0 | 0 | 0 | 0 |
| 004011CC | `fstp dword ptr [ebp-0Ch]` | 0 | 0 | 0 | 0 | 0 |
| 004011CF | `jmp 00401191` | 0 | 0 | 0 | 0 | 0 |
| 004011D1 | `mov eax,dword ptr [ebp-4]` | 5 | 0 | 5 | 5 | 0 |
| 004011D4 | `imul eax,eax,4000` | 0 | 0 | 0 | 0 | 0 |
| 004011DA | `mov ecx,dword ptr [ebp-8]` | 0 | 0 | 0 | 0 | 0 |
| 004011DD | `mov edx,dword ptr [ebp-0Ch]` | 0 | 0 | 0 | 0 | 0 |
| 004011E0 | `mov dword ptr [eax+ecx*4+0BB51E8h],edx` | 4020 | 4019 | 1 | 1 | 0 |
| 004011E7 | `jmp 0040116F` | 0 | 0 | 0 | 0 | 0 |
| 004011E9 | `jmp 00401150` | 0 | 0 | 0 | 0 | 0 |
| 004011EE | `pop esi` | 0 | 0 | 0 | 0 | 0 |
| 004011EF | `mov esp,ebp` | 0 | 0 | 0 | 0 | 0 |
| 004011F1 | `pop ebp` | 0 | 0 | 0 | 0 | 0 |
| 004011F2 | `ret` | 0 | 0 | 0 | 0 | 0 |

AMD Family 10h processors fetch instruction bytes in 32-byte blocks. The address reported by IBS fetch sampling is either the start of a 32-byte fetch block, is the target of a control transfer or is the fall-through of a conditional branch. AMD64 instructions may straddle the start address of a fetch block. In these cases, CodeAnalyst associates the IBS fetch sample with the instruction that straddles the fetch block.

The large number of killed fetch samples at address 0x4011E0 is due to early speculative prefetch activity. The preceding fetch block contains the jump instruction (address 0x4011CF) that transfers control to the top of the innermost loop. The processor initiates a speculative fetch before receiving the control flow redirection back to the top of the innermost loop. These early speculative fetches were killed before instruction translation lookaside buffer access.

### 4.2 IBS op sampling

AMD Family 10h processors execute AMD64 instructions in discrete execution operations (ops.) IBS op sampling counts processor cycles and periodically selects an op to be tagged and monitored. An IBS op sample is generated when a tagged op retires; a sample is not generated for an aborted (flushed) op. The information reported with an IBS op sample includes:

- The AMD64 instruction address for the op
- The tag-to-retire time (cycles from when the op was tagged to when the op retired)
- The completion-to-retire time (cycles from when the op completed to when the op retired)
- Whether the op implements AMD64 branch semantics (a "branch op")
    - If the branch op was mispredicted
    - If the branch was taken
    - If the branch was a return
    - If the return was mispredicted
- Whether the op was a resync that caused a pipeline flush
- Whether the op performed a load and/or store operation
    - If the operation missed in the data cache
    - If the operation missed in the level 1 or level 2 DTLB
    - The page size of the level 1 or level 2 address translation
    - If the operation caused a misaligned access
    - The DC miss latency (in cycles) if the load operation missed in the data cache
    - The virtual and physical address of the requested memory location
    - If the access was made to local or remote memory

The event information is extensive and would require a large number of performance counters in order to collect equivalent data in a single test run (without performance counter multiplexing.)

The following table shows a breakdown of the IBS op samples for the inner loop of *multiply_matrices*. (A full breakdown for *multiply_matrices* is given in Appendix D.)

| Address | Instruction | All ops | Branch | Load/store |
|---|---|---|---|---|
| 00401191 | `mov edx,dword ptr [ebp-10h]` | 32229 | 0 | 32226 |
| 00401194 | `add edx,1` | 32175 | 0 | 0 |
| 00401197 | `mov dword ptr [ebp-10h],edx` | 32508 | 0 | 32508 |
| 0040119A | `cmp dword ptr [ebp-10h],1000` | 3491 | 0 | 3491 |
| 004011A1 | `jge 004011D1` | 3381 | 3379 | 1 |
| 004011A3 | `mov eax,dword ptr [ebp-4]` | 3416 | 0 | 3416 |
| 004011A6 | `imul eax,eax,4000` | 1361 | 0 | 0 |
| 004011AC | `mov ecx,dword ptr [ebp-10h]` | 2737 | 0 | 2731 |
| 004011AF | `imul ecx,ecx,4000` | 1223 | 0 | 0 |
| 004011B5 | `mov edx,dword ptr [ebp-10h]` | 1295 | 0 | 1290 |
| 004011B8 | `mov esi,dword ptr [ebp-8]` | 1302 | 0 | 1302 |
| 004011BB | `fld dword ptr [eax+edx*4+413FE0h]` | 4127 | 0 | 4127 |
| 004011C2 | `fmul dword ptr [ecx+esi*4+7E48E8h]` | 4091 | 0 | 4090 |
| 004011C9 | `fadd dword ptr [ebp-0Ch]` | 4216 | 0 | 4199 |
| 004011CC | `fstp dword ptr [ebp-0Ch]` | 12226 | 0 | 12207 |
| 004011CF | `jmp 00401191` | 12342 | 12327 | 40 |

The third column is the number of IBS op samples taken at each address. The fourth and fifth columns show the number IBS op samples that were branch ops and/or load/store ops. The jump

instructions at 0x4011A1 and 0x4011CF are correctly identified as branches, illustrating the precision offered by IBS.

The conditional jump at address 0x4011a1 is taken when the loop exit condition is true. 3,379 IBS branch op samples were attributed to this address. Of these samples, only two indicated that the branch was mispredicted. These statistics indicate that the branch was predicted correctly and that branch misprediction is not a performance issue. This kind of analysis is not possible with branch data collected via performance counter sampling since mispredictions or the even the number of times a branch is executed cannot be attributed to a specific branch instruction due to accumulated inaccuracies.

The table below shows IBS op data for load and store operations within the inner loop of *multiply_matrices*.

| Address | Instruction | Load | Store | DC miss | DTLB L1M L2M |
|---|---|---|---|---|---|
| 0x401191 | `mov edx,dword ptr [ebp-10h]` | 32225 | 1 | 8 | 0 |
| 0x401194 | `add edx,1` | 0 | 0 | 0 | 0 |
| 0x401197 | `mov dword ptr [ebp-10h],edx` | 0 | 32508 | 21 | 0 |
| 0x40119a | `cmp dword ptr [ebp-10h],1000` | 3491 | 0 | 2 | 0 |
| 0x4011a1 | `jge 004011D1` | 1 | 0 | 1 | 0 |
| 0x4011a3 | `mov eax,dword ptr [ebp-4]` | 3416 | 0 | 2 | 0 |
| 0x4011a6 | `imul eax,eax,4000` | 0 | 0 | 0 | 0 |
| 0x4011ac | `mov ecx,dword ptr [ebp-10h]` | 2731 | 0 | 6 | 0 |
| 0x4011af | `imul ecx,ecx,4000` | 0 | 0 | 0 | 0 |
| 0x4011b5 | `mov edx,dword ptr [ebp-10h]` | 1290 | 0 | 1 | 1 |
| 0x4011b8 | `mov esi,dword ptr [ebp-8]` | 1302 | 0 | 0 | 0 |
| 0x4011bb | `fld dword ptr [eax+edx*4+413FE0h]` | 4127 | 0 | 51 | 18 |
| 0x4011c2 | `fmul dword ptr [ecx+esi*4+7E48E8h]` | 4090 | 0 | 892 | 3141 |
| 0x4011c9 | `fadd dword ptr [ebp-0Ch]` | 4198 | 1 | 20 | 1 |
| 0x4011cc | `fstp dword ptr [ebp-0Ch]` | 4 | 12203 | 9 | 1 |
| 0x4011cf | `jmp 00401191` | 3 | 37 | 2 | 0 |

Unlike the PMC-based profile, the source of data cache and DTLB misses can be accurately identified as the instruction at address 0x4011C2. IBS lets a software developer identify performance culprits with pin-point precision.

The data cache miss latency for the culprit instruction was 92,465 cycles for the collected samples. This yields an average data cache miss latency of 103.66 processor clock cycles. The large latency is due to the DTLB misses and relatively long read accesses to memory as a result of data cache misses.

## 5. New opportunities for analysis

Instruction-Based Sampling provides rich performance data that lets software developers identify performance culprits precisely. This article describes only the basic features of IBS. Other features offer new opportunities for analysis and optimization:

- IBS op samples report the address of the memory data item accessed by a load or store operation. Coupled with information about local/remote memory access, the address can be used for "data centric" analysis such as analyzing and optimizing data layout on non-uniform memory access (NUMA) platforms.

- Precise IBS information can be used to drive profile-directed compiler optimizations. For example, a compiler could use precise branch taken/not taken information for code straightening.

For more information about Instruction-Based Sampling, please see the Software Optimization Guide for Family 10h Processors (Quad-Core) available at AMD Developer Central.
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf

AMD CodeAnalyst is available for download, also at AMD Developer Central:
CodeAnalyst for Windows: http://developer.amd.com/cawin.jsp
CodeAnalyst for Linux: http://developer.amd.com/calinux.jsp

## 6. Postscript

Acknowledgements:
Many people contributed to the success of IBS. On the hardware side, I would like to thank Ben Sander (architect), Ravi Bhargava and Anasua Bhowmik. I would like to thank the entire AMD CodeAnalyst team especially Lei Yu (team leader), Frank Swehosky, Barry Kasindorf and Tom Evans.

Short bio:
Paul Drongowski is an AMD Senior Member of Technical Staff. He is an engineer in the AMD CodeAnalyst team and has worked on profiling tools and performance analysis for the last 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 University.

## Appendix A: Source code for simple_classic.cpp

```
// simple_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) 2005-2007 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("simple_classic.txt", "w")) == NULL) {
		fprintf(stderr, "Couldn't open result file\n") ;
		perror(argv[0]) ;
		return( EXIT_FAILURE ) ;
	}

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

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

	fclose(result_file) ;

	return( 0 ) ;
}
```

## Appendix B: PMC memory access profile for *multiply_matrices*

| Address | Instruction | Ret inst | DC accesses | DC misses | DTLB L1M L2M |
|---|---|---|---|---|---|
| 00401140 | push ebp | 0 | 0 | 0 | 0 |
| 00401141 | mov ebp,esp | 0 | 0 | 0 | 0 |
| 00401143 | sub esp,10h | 0 | 0 | 0 | 0 |
| 00401146 | push esi | 0 | 0 | 0 | 0 |
| 00401147 | mov dword ptr [ebp-4],0 | 0 | 0 | 0 | 0 |
| 0040114E | jmp 00401159 | 0 | 0 | 0 | 0 |
| 00401150 | mov eax,dword ptr [ebp-4] | 0 | 0 | 0 | 0 |
| 00401153 | add eax,1 | 0 | 0 | 0 | 0 |
| 00401156 | mov dword ptr [ebp-4],eax | 0 | 0 | 0 | 0 |
| 00401159 | cmp dword ptr [ebp-4],1000 | 0 | 0 | 0 | 0 |
| 00401160 | jge 004011EE | 0 | 0 | 0 | 0 |
| 00401166 | mov dword ptr [ebp-8],0 | 0 | 0 | 0 | 0 |
| 0040116D | jmp 00401178 | 0 | 0 | 0 | 0 |
| 0040116F | mov ecx,dword ptr [ebp-8] | 0 | 0 | 0 | 0 |
| 00401172 | add ecx,1 | 0 | 2 | 0 | 0 |
| 00401175 | mov dword ptr [ebp-8],ecx | 0 | 0 | 0 | 0 |
| 00401178 | cmp dword ptr [ebp-8],1000 | 0 | 0 | 0 | 0 |
| 0040117F | jge 004011E9 | 1 | 0 | 0 | 1 |
| 00401181 | mov dword ptr [ebp-0Ch],0 | 0 | 0 | 0 | 0 |
| 00401188 | mov dword ptr [ebp-10h],0 | 0 | 0 | 0 | 0 |
| 0040118F | jmp 0040119A | 2 | 0 | 0 | 0 |
| 00401191 | mov edx,dword ptr [ebp-10h] | 1291 | 1763 | 7 | 81 |
| 00401194 | add edx,1 | 1330 | 1551 | 62 | 91 |
| 00401197 | mov dword ptr [ebp-10h],edx | 0 | 0 | 0 | 0 |
| 0040119A | cmp dword ptr [ebp-10h],1000 | 0 | 2 | 0 | 1 |
| 004011A1 | jge 004011D1 | 1693 | 1353 | 288 | 42 |
| 004011A3 | mov eax,dword ptr [ebp-4] | 0 | 0 | 0 | 0 |
| 004011A6 | imul eax,eax,4000 | 0 | 0 | 0 | 0 |
| 004011AC | mov ecx,dword ptr [ebp-10h] | 2196 | 1813 | 45 | 42 |
| 004011AF | imul ecx,ecx,4000 | 0 | 0 | 0 | 0 |
| 004011B5 | mov edx,dword ptr [ebp-10h] | 1818 | 1579 | 10 | 20 |
| 004011B8 | mov esi,dword ptr [ebp-8] | 0 | 0 | 0 | 0 |
| 004011BB | fld dword ptr [eax+edx*4+413FE0h] | 0 | 0 | 0 | 0 |
| 004011C2 | fmul dword ptr [ecx+esi*4+7E48E8h] | 1165 | 1107 | 9 | 218 |
| 004011C9 | fadd dword ptr [ebp-0Ch] | 33298 | 22415 | 409 | 2093 |
| 004011CC | fstp dword ptr [ebp-0Ch] | 17983 | 8363 | 41 | 256 |
| 004011CF | jmp 00401191 | 3365 | 4146 | 47 | 276 |
| 004011D1 | mov eax,dword ptr [ebp-4] | 0 | 0 | 0 | 0 |
| 004011D4 | imul eax,eax,4000 | 3 | 4 | 0 | 1 |
| 004011DA | mov ecx,dword ptr [ebp-8] | 2 | 0 | 0 | 0 |
| 004011DD | mov edx,dword ptr [ebp-0Ch] | 0 | 0 | 0 | 0 |
| 004011E0 | mov dword ptr [eax+ecx*4+0BB51E8h],edx | 0 | 0 | 0 | 0 |
| 004011E7 | jmp 0040116F | 67 | 94 | 1 | 0 |
| 004011E9 | jmp 00401150 | 0 | 0 | 0 | 0 |
| 004011EE | pop esi | 0 | 0 | 0 | 0 |
| 004011EF | mov esp,ebp | 0 | 0 | 0 | 0 |
| 004011F1 | pop ebp | 0 | 0 | 0 | 0 |
| 004011F2 | ret | 0 | 0 | 0 | 0 |

## Appendix C: IBS fetch profile for *multiply_matrices*

| Address | Instruction | All | Killed | Attempted | Completed | Aborted |
|---|---|---|---|---|---|---|
| 00401140 | push ebp | 0 | 0 | 0 | 0 | 0 |
| 00401141 | mov ebp,esp | 0 | 0 | 0 | 0 | 0 |
| 00401143 | sub esp,10h | 0 | 0 | 0 | 0 | 0 |
| 00401146 | push esi | 0 | 0 | 0 | 0 | 0 |
| 00401147 | mov dword ptr [ebp-4],0 | 0 | 0 | 0 | 0 | 0 |
| 0040114E | jmp 00401159 | 0 | 0 | 0 | 0 | 0 |
| 00401150 | mov eax,dword ptr [ebp-4] | 0 | 0 | 0 | 0 | 0 |
| 00401153 | add eax,1 | 0 | 0 | 0 | 0 | 0 |
| 00401156 | mov dword ptr [ebp-4],eax | 0 | 0 | 0 | 0 | 0 |
| 00401159 | cmp dword ptr [ebp-4],1000 | 0 | 0 | 0 | 0 | 0 |
| 00401160 | jge 004011EE | 0 | 0 | 0 | 0 | 0 |
| 00401166 | mov dword ptr [ebp-8],0 | 0 | 0 | 0 | 0 | 0 |
| 0040116D | jmp 00401178 | 0 | 0 | 0 | 0 | 0 |
| 0040116F | mov ecx,dword ptr [ebp-8] | 0 | 0 | 0 | 0 | 0 |
| 00401172 | add ecx,1 | 0 | 0 | 0 | 0 | 0 |
| 00401175 | mov dword ptr [ebp-8],ecx | 0 | 0 | 0 | 0 | 0 |
| 00401178 | cmp dword ptr [ebp-8],1000 | 0 | 0 | 0 | 0 | 0 |
| 0040117F | jge 004011E9 | 6 | 0 | 6 | 6 | 0 |
| 00401181 | mov dword ptr [ebp-0Ch],0 | 0 | 0 | 0 | 0 | 0 |
| 00401188 | mov dword ptr [ebp-10h],0 | 0 | 0 | 0 | 0 | 0 |
| 0040118F | jmp 0040119A | 0 | 0 | 0 | 0 | 0 |
| 00401191 | mov edx,dword ptr [ebp-10h] | 7 | 0 | 7 | 7 | 0 |
| 00401194 | add edx,1 | 0 | 0 | 0 | 0 | 0 |
| 00401197 | mov dword ptr [ebp-10h],edx | 0 | 0 | 0 | 0 | 0 |
| 0040119A | cmp dword ptr [ebp-10h],1000 | 4040 | 4 | 4036 | 4036 | 0 |
| 004011A1 | jge 004011D1 | 0 | 0 | 0 | 0 | 0 |
| 004011A3 | mov eax,dword ptr [ebp-4] | 0 | 0 | 0 | 0 | 0 |
| 004011A6 | imul eax,eax,4000 | 0 | 0 | 0 | 0 | 0 |
| 004011AC | mov ecx,dword ptr [ebp-10h] | 0 | 0 | 0 | 0 | 0 |
| 004011AF | imul ecx,ecx,4000 | 0 | 0 | 0 | 0 | 0 |
| 004011B5 | mov edx,dword ptr [ebp-10h] | 0 | 0 | 0 | 0 | 0 |
| 004011B8 | mov esi,dword ptr [ebp-8] | 0 | 0 | 0 | 0 | 0 |
| 004011BB | fld dword ptr [eax+edx*4+413FE0h] | 3955 | 2 | 3953 | 3953 | 0 |
| 004011C2 | fmul dword ptr [ecx+esi*4+7E48E8h] | 0 | 0 | 0 | 0 | 0 |
| 004011C9 | fadd dword ptr [ebp-0Ch] | 0 | 0 | 0 | 0 | 0 |
| 004011CC | fstp dword ptr [ebp-0Ch] | 0 | 0 | 0 | 0 | 0 |
| 004011CF | jmp 00401191 | 0 | 0 | 0 | 0 | 0 |
| 004011D1 | mov eax,dword ptr [ebp-4] | 5 | 0 | 5 | 5 | 0 |
| 004011D4 | imul eax,eax,4000 | 0 | 0 | 0 | 0 | 0 |
| 004011DA | mov ecx,dword ptr [ebp-8] | 0 | 0 | 0 | 0 | 0 |
| 004011DD | mov edx,dword ptr [ebp-0Ch] | 0 | 0 | 0 | 0 | 0 |
| 004011E0 | mov dword ptr [eax+ecx*4+0BB51E8h],edx | 4020 | 4019 | 1 | 1 | 0 |
| 004011E7 | jmp 0040116F | 0 | 0 | 0 | 0 | 0 |
| 004011E9 | jmp 00401150 | 0 | 0 | 0 | 0 | 0 |
| 004011EE | pop esi | 0 | 0 | 0 | 0 | 0 |
| 004011EF | mov esp,ebp | 0 | 0 | 0 | 0 | 0 |
| 004011F1 | pop ebp | 0 | 0 | 0 | 0 | 0 |
| 004011F2 | ret | 0 | 0 | 0 | 0 | 0 |

## Appendix D: IBS op profile *multiply_matrices*

| Address | Instruction | IBS all ops | IBS BR | IBS load/store |
|---|---|---|---|---|
| 00401140 | push ebp | 0 | 0 | 0 |
| 00401141 | mov ebp,esp | 0 | 0 | 0 |
| 00401143 | sub esp,10h | 0 | 0 | 0 |
| 00401146 | push esi | 0 | 0 | 0 |
| 00401147 | mov dword ptr [ebp-4],0 | 0 | 0 | 0 |
| 0040114E | jmp 00401159 | 0 | 0 | 0 |
| 00401150 | mov eax,dword ptr [ebp-4] | 0 | 0 | 0 |
| 00401153 | add eax,1 | 0 | 0 | 0 |
| 00401156 | mov dword ptr [ebp-4],eax | 0 | 0 | 0 |
| 00401159 | cmp dword ptr [ebp-4],1000 | 0 | 0 | 0 |
| 00401160 | jge 004011EE | 0 | 0 | 0 |
| 00401166 | mov dword ptr [ebp-8],0 | 0 | 0 | 0 |
| 0040116D | jmp 00401178 | 0 | 0 | 0 |
| 0040116F | mov ecx,dword ptr [ebp-8] | 3 | 0 | 3 |
| 00401172 | add ecx,1 | 4 | 0 | 0 |
| 00401175 | mov dword ptr [ebp-8],ecx | 3 | 0 | 3 |
| 00401178 | cmp dword ptr [ebp-8],1000 | 1 | 0 | 1 |
| 0040117F | jge 004011E9 | 0 | 0 | 0 |
| 00401181 | mov dword ptr [ebp-0Ch],0 | 0 | 0 | 0 |
| 00401188 | mov dword ptr [ebp-10h],0 | 2 | 0 | 2 |
| 0040118F | jmp 0040119A | 7 | 7 | 0 |
| 00401191 | mov edx,dword ptr [ebp-10h] | 32229 | 0 | 32226 |
| 00401194 | add edx,1 | 32175 | 0 | 0 |
| 00401197 | mov dword ptr [ebp-10h],edx | 32508 | 0 | 32508 |
| 0040119A | cmp dword ptr [ebp-10h],1000 | 3491 | 0 | 3491 |
| 004011A1 | jge 004011D1 | 3381 | 3379 | 1 |
| 004011A3 | mov eax,dword ptr [ebp-4] | 3416 | 0 | 3416 |
| 004011A6 | imul eax,eax,4000 | 1361 | 0 | 0 |
| 004011AC | mov ecx,dword ptr [ebp-10h] | 2737 | 0 | 2731 |
| 004011AF | imul ecx,ecx,4000 | 1223 | 0 | 0 |
| 004011B5 | mov edx,dword ptr [ebp-10h] | 1295 | 0 | 1290 |
| 004011B8 | mov esi,dword ptr [ebp-8] | 1302 | 0 | 1302 |
| 004011BB | fld dword ptr [eax+edx*4+413FE0h] | 4127 | 0 | 4127 |
| 004011C2 | fmul dword ptr [ecx+esi*4+7E48E8h] | 4091 | 0 | 4090 |
| 004011C9 | fadd dword ptr [ebp-0Ch] | 4216 | 0 | 4199 |
| 004011CC | fstp dword ptr [ebp-0Ch] | 12226 | 0 | 12207 |
| 004011CF | jmp 00401191 | 12342 | 12327 | 40 |
| 004011D1 | mov eax,dword ptr [ebp-4] | 3 | 0 | 3 |
| 004011D4 | imul eax,eax,4000 | 0 | 0 | 0 |
| 004011DA | mov ecx,dword ptr [ebp-8] | 2 | 0 | 2 |
| 004011DD | mov edx,dword ptr [ebp-0Ch] | 4 | 0 | 4 |
| 004011E0 | mov dword ptr [eax+ecx*4+0BB51E8h],edx | 0 | 0 | 0 |
| 004011E7 | jmp 0040116F | 3 | 3 | 0 |
| 004011E9 | jmp 00401150 | 0 | 0 | 0 |
| 004011EE | pop esi | 0 | 0 | 0 |
| 004011EF | mov esp,ebp | 0 | 0 | 0 |
| 004011F1 | pop ebp | 0 | 0 | 0 |
| 004011F2 | ret | 0 | 0 | 0 |