Test programs for measuring clock cycles and performance monitoring

By Agner Fog. Copenhagen University College of Engineering.
Copyright © 2000 - 2014. GPL v. 3. Last updated 2014-10-02.

Contents

1 Introduction

The file `testp.zip` contains a series of test programs for optimizing and testing small pieces of code on x86 and x86-64 family microprocessors. These test programs are intended for testing small pieces of critical code, not for profiling a whole program. Note that these test programs are intended for experts. You must have experience with low-level programming and command line utilities to use these test programs. The test programs are provided for free with no warranty or support.

The test programs use the Time Stamp Counter to count how many clock cycles a piece of code takes and the Performance Monitor Counters to count instructions, micro-operations, cache misses, branch mispredictions and other events relevant to optimization purposes.

The following test packages are included:

Filename	Operating system	Mode	Language	Time stamp counter	Program monitor counters	Micro-proces-sor	Use
PMCTest.zip	Windows, Linux	32, 64	C++, asm	X	X	Intel, AMD, VIA	Advanced testing
old/TSCUni.zip	Windows, Linux, BSD	32, 64	C++	X	-	Intel, AMD, VIA	Simple timing
old/TSCWin.zip	Windows	32, 64	asm	X	-	Intel, AMD, VIA	Simple timing
old/PMCDOS.zip	DOS	16, 32, 64	asm	X	X	Intel	16 bit or privileged mode

The preferred test program is PMCTest. This program supports Windows and Linux, 32 and 64 bit mode, Gnu and Microsoft C++ compilers, MASM, NASM and YASM assemblers, Intel, AMD and VIA processors, and multithreading. The test programs under `old` are not up to date.

Other files included:

testp.pdf	This file
DriverSrcWin.zip	Source code for Windows driver used by `PMCTest.zip`.
DriverSrcLinux.zip	Source code for Linux driver used by `PMCTest.zip`.
TestScripts.zip	Scripts to run multiple tests with `PMCTest.zip`
vs2008_project_files.zip	Project files for running under MS Visual Studio 2008
vs2010_project_files.zip	Project files for running under MS Visual Studio 2010
vs2012_project_files.zip	Project files for running under MS Visual Studio 2012

2 Instructions for running PMCTest under Windows:

The PMCTest program works under 32-bit and 64-bit Windows.

If you are using a 64-bit Windows system, then you have to press F8 during boot and select "Disable Driver Signature Enforcement". Run the IDE or test program as administrator. Alternatively, set USE_PERFORMANCE_COUNTERS to zero if you don't need the performance monitor counts.

Unpack `PMCTest.zip` into a separate folder. If you are using Microsoft Visual Studio then unpack `vs2012_project_files.zip`. If the test code is in C or C++ then use the `c3264` project files for 32 bit or 64 bit mode. If the test code is in assembly language (MASM syntax) then use the `a32` or `a64` project file for 32 bit and 64 bit mode, respectively.

If you are using another development environment then make a project that contains `PMCTestA.cpp` and one of the `PMCTestB` files. If you are using assembly in NASM/YASM syntax then use the `*.nasm` source files instead of the `*.asm` files.

Place the driver files `MSRDriver32.sys` and `MSRDriver64.sys` in the same folder as the compiled test program.

Insert the piece of code to test at the place `### Test code start ###` in `PMCTestB.*`. Other places where you may insert constants or initializations are marked with `###`.

You can choose between different PMC counters. Insert the desired counter id's at `CounterTypesDesired` in the `PMCTestB.*` file. The available counters are listed at the bottom of `PMCTestA.cpp`. You may add your own counter definitions.

When you run the compiled test program you will get a list of TSC clock counts and PMC counts for the test code you have inserted.

3 Instructions for running PMCTest under Linux:

The PMCTest program works under 32-bit and 64-bit Linux. It is recommended to use 64-bit Ubuntu.

3.1 Necessary installations

Install a universal version of the Gnu C/C++ compiler with the command:

```
sudo apt-get install g++-multilib
```

Install the NASM assembler with the command:

```
sudo apt-get install nasm
```

This may not give you the latest version of NASM. If you want to test the newest instructions then you may need the latest release candidate or the latest development snapshot from www.nasm.us. Get the source file `nasm-x.xx.xx.tar.gz` and unpack it with

```
tar -xzf nasm*.tar.gz
```

Then go to the extracted folder and install it with the commands:

```
./configure
make
sudo make install
```

You may use the YASM assembler instead of NASM. These two assemblers are mutually compatible. Get the newest version of YASM from http://yasm.tortall.net/Download.html and install it in the same way as explained above for NASM.

You also need to install the PMCTest driver in the following way:

Unpack `DriverSrcLinux.zip` to a separate folder. In that folder, do:

```
chmod a+x *.sh
make
sudo ./install.sh
```

Every time the computer has been restarted, you have to reinstall the driver with the command:

```
sudo ./install.sh
```

3.2 Inserting the code to test

The project consists of `PMCTestA.cpp` and `PMCTestB.*`. The `PMCTestB.*` file contains the code to test. It is available in C++ and assembly, MASM and NASM syntax, 32-bit and 64-bit mode.

Insert the piece of code to test at the place marked `### Test code start ###` in `PMCTestB.*`. Other places where you may insert constants or initializations are marked with `###`.

You can choose between different PMC counters. Insert the desired counter id's at `CounterTypesDesired` in the `PMCTestB.*` file. The available counters are listed at the bottom of `PMCTestA.cpp`. You may add your own counter definitions.

When you run the compiled test program you will get a list of TSC clock counts and PMC counts for the test code you have inserted.

Make all script files executable with the command:

```
chmod a+x *.sh
```

To compile and run the test program use `c32.sh` or `c64.sh` for C++ code, 32-bit and 64-bit mode, or `a32.sh` or `a64.sh` for assembly code, using the NASM assembler.

4 Testing inside another program

Sometimes it is difficult to isolate the piece of code you want to test from its context. The PMCTest program has an option to set up the performance monitor counters and leave them on for another program to use. You can then add instructions inside your own program to read the counters before and after the piece of code you want to test. The difference between the two counter values gives the count for the code under test.

To turn on the counters, run the `pmctest` program with the command line parameter `startcounters`. To turn off the counters again, use the command line parameter `stopcounters`. You can override the counter definitions in the B file by adding counter definition numbers on the command line, e.g.:

```
pmctest startcounters 1 9 100 311
```

This will set up four counters for core clock cycles (Intel only), instructions, micro-operations and level-1 data cache misses. The output may look like this:

```
Enabled 4 counters in each of 8 CPU cores

PMC number:    Counter name:
0x40000001     Core cyc
0x40000000     Instruct
0x00000000     Uops
```

```
     0x00000001      L1D Miss
```

Then you can read program monitor counter number `0x40000001` in your program to get the core clock cycles.

In Linux: Compile the `pmctest` program as described above, and run it with the desired command line parameters. Make sure the driver is installed. Wait a few seconds before running the program that uses the counters.

In 64-bit Windows: Compile the `pmctest` program or use the precompiled `pmctest.exe`. Make sure the driver file `MSRDriver64.sys` is in the same directory. Turn off driver signature enforcement (Press F8 during boot) and run `pmctest.exe` as administrator. Example script files `startcounters.bat` and `stopcounters.bat` are provided for convenience.

The selected program monitor counters will be set up and enabled in all accessible CPU cores for a maximum of 64 threads. The counters will usually stay on until the computer is reset or goes into sleep mode. You should not leave the counters on for long periods of time if you are running in a multiuser environment.

This example shows how to use the counters inside your own program:

```c
#include "timingtest.h"                 // functions for reading counters

int main()
{
    const int numtests = 10;         // number of test runs
    const int pmc_num = 0x40000001;  // program monitor counter number
    int clockcounts[numtests];       // list of clock counts
    int pmccounts[numtests];         // list of PMC counts
    int clock1, clock2, pmc1, pmc2;  // counter values before and after each test
    int i;                           // loop counter

    for (i = 0; i < numtests; i++) {   // loop for repeated tests
        serialize();                   // prevent out-of-order execution
        clock1 = (int)readtsc();       // read clock
        pmc1   = (int)readpmc(pmc_num);// read PMC

        FunctionToTest();              // your code to test

        serialize();                   // serialize again
        clock2 = (int)readtsc();       // read again after test
        pmc2   = (int)readpmc(pmc_num);
        clockcounts[i] = clock2-clock1;// store differences in list
        pmccounts[i]   = pmc2-pmc1;
    }

    // print out results
    printf("\nTest results\nclock counts   PMC counts");
    for (i = 0; i < numtests; i++) {
        printf("\n%10i   %10i", clockcounts[i], pmccounts[i]);
    }
    printf("\n");

    return 0;
}
```

In this example, we are running the code to test 10 times in order to see if the counts are stable. The first count is usually higher than the rest due to cache misses and branch mispredictions.

Obviously, you should have a switch in your program to turn off testing in the final version. Note that the program will raise an exception if trying to read a PMC counter when the counters are not enabled. There is no way to test if the PMC counters are enabled other than trying to read a counter and catch the possible exception.

5 Test scripts

The file `TestScripts.zip` contains a lot of test scripts that I have used for testing microprocessors and measure latencies, throughputs, etc. of all instructions.

5.1 Running the test scripts under Linux

The test scripts work best under Linux. Use 64-bit Ubuntu and make the necessary installations as described in chapter 3.1 above. Install the latest version of NASM or YASM.

Unpack `TestScripts.zip` to a separate folder.

If you want to use the YASM assembler instead of NASM then modify the file `vars.sh` to specify which assembler to use.

Make script files executable with

        chmod a+x *.sh

Initialize the scripts by running

        ./init.sh

Now you can run the different test scripts. I keep modifying the scripts and making new scripts as new generations of microprocessors have new features to test. Here is a contemporary list of test scripts:

Test scripts for measuring latency, throughput, execution units or execution ports for x86 and x86-64 application instructions	
Script file	**Purpose**
int.sh1	Integer instructions for general purpose registers
32bitinstr.sh1	Instructions that are not available in 64-bit mode
pushpop.sh1	Push and pop instructions
shift.sh1	Shift and rotate instructions
mul.sh1	Integer multiplication instructions
div.sh1	Integer division instructions
misc_int.sh1	Miscellaneous integer instructions
strings.sh1	Integer string instructions and vector string instructions
new_int_instr.sh1	Integer instructions for SSE4.2 and later instruction sets
amd.sh1	Instructions that are only available on AMD processors
x87.sh1	x87 style floating point instructions
convers.sh1	General purpose and vector instructions where source and destination use different register types or memory
convers2.sh1	AVX instructions where source and destination use different register types or memory
ivec.sh1	Integer vector instructions
fvec.sh1	Floating point vector instructions including AVX instructions
misc_vect.sh1	Miscellaneous vector instructions
fma.sh1	FMA3 and FMA4 instructions
avx2.sh1	AVX2 and later vector instructions

`exec_domain.sh1`	Measuring data transfer delays between instructions that possibly use different execution units

The results of the `*.sh1` scripts will appear in the folder named `results1`. Most instructions are ordered according to the test method they require rather than by any logical criteria. Use a text search tool to find the results for a specific instruction.

Test scripts for other purposes	
Script file	**Purpose**
`warmup_fp.sh2`	Test if the floating point unit has a warm up effect
`decode_nops.sh2`	Test instruction fetch and decode rate with different types of long NOPs
`decode_double.sh2`	Test decoder throughput for instructions generating multiple µops (Needs adjustment for each new processor type).
`length_chg_prefix.sh2`	Test if decoding is delayed by length-changing prefixes
`instruct_boundaries.sh2`	Test if instruction boundaries are marked in instruction cache
`loop_size.sh2`	Test loop buffer size and µop cache size
`loop_buffer.sh2`	Test loop buffer size
`jmp.sh2`	Test throughput of jump instructions
`branch.sh2`	Test branch prediction
`branch_x.sh2`	Test branch prediction
`fused_branch.sh2`	Test if branches can be fused with arithmetic instructions
`ucache_misprediction.sh2`	Test if branch prediction depends on µop cache
`ucache_double_entries.sh2`	Test if some instructions take up two entries in µop cache
`returnstack.sh2`	Measure return stack buffer size
`stack_sync_uops.sh2`	Test if stack synchronization µops are inserted when stack pointer is accessed directly
`store_forwarding.sh2`	Test the efficiency of store-to-load forwarding
`partial_register_stall.sh2`	Test if access to partial registers or partial flags leads to stalls or extra µops
`mov_rename.sh2`	Test if register moves can be eliminated by renaming
`xor.sh2`	Test if XOR, SUB and other instructions are independent on previous value if both registers are the same
`out_of_order.sh2`	Measure the depth of out-of-order execution
`read_write_bandwidth.sh2`	Test the bandwidth of L1 data cache read and write
`cache_throughput.sh2`	Test the throughput of each cache level
`cache_latency.sh2`	Measure the latency of each cache level
`cache_banks.sh2`	Test for cache bank contentions
`testmemcpyalign.sh2`	Test the efficiency of memory copying with different alignments
`memcpy.sh2`	Test which memory copying method is fastest
`denormal.sh2`	Measure penalties for floating point underflow, overflow and denormal numbers
`AVX_states.sh2`	Measure penalty of switching between different states of the YMM register file
`resource_sharing.sh2`	Test which resources are shared between multiple threads
`daxpy.sh2`	Benchmark: standard DAXPY algorithm
`taylor.sh2`	Benchmark: Taylor series expansion
`vars.sh`	Called only by the other scripts. See below
`init.sh`	Initialization of files etc. before running test scripts
`allsh1.sh`	Run all tests in the .sh1 category
`allsh2.sh`	Run all tests in the .sh2 category

alltests.sh	Initialize and run all tests
pack_results.sh	Pack all test results into a zip file

The results of the `*.sh2` scripts will appear in the folder named `results2`.

Several other files used by the test scripts are listed below.

5.2 Running the test scripts under Windows

Most of the test scripts can run under 64-bit Windows using Cygwin, but tests that require 32-bit mode cannot, because Cygwin cannot make 32-bit and 64-bit code from the same prompt.

Install the 64-bit version of Cygwin (www.cygwin.com). Install the NASM assembler, or YASM. Reboot the machine, press F8 during boot and select "Disable Driver Signature Enforcement".

Make a subfolder under your Cygwin home, e.g. `C:\cygwin64\home\YourName\testp`. Unpack `PMCTest.zip` and `TestScripts.zip` to this folder. Run the Cygwin64 terminal *as administrator*. In the terminal, go to the folder where you put the test scripts. Run `./init.sh` as described above. If this works, then you may run all the test scripts.

5.3 Interpreting the test results

The listing below is an example of output from the script `fvec.sh1` for the MULPS instruction on an Intel Haswell processor:

```
Latency: mulps r128,r128

Processor 0
    Clock    Core cyc   Instruct   uops RAT      Uops     uop p0   Mov elim
    448709     499990     102027     101041     101027      49992          0
    448732     499990     102027     101048     101027      49992          0
    448729     499990     102027     101044     101027      49986          0
    448732     499990     102027     101049     101027      49987          0
    448732     499990     102027     101030     101027      49994          0

Throughput: mulps r128,r128

Processor 0
    Clock    Core cyc   Instruct       Uops     uop p0     uop p1     uop p2
     44843      50008     102027     101027      50002      50001          0
     44869      50007     102027     101027      49999      50001          0
     44869      50009     102027     101027      50000      50001          0
     44869      50010     102027     101027      50003      50001          0
     44866      50006     102027     101027      50000      50001          0
Processor 0
    Clock    Core cyc   Instruct       Uops     uop p3     uop p4     uop p5
     44892      50012     102027     101027          0          0          0
     44889      50008     102027     101027          0          0          0
     44889      50009     102027     101027          0          0          0
     44892      50010     102027     101027          0          0          0
     44892      50011     102027     101027          0          0          0
Processor 0
    Clock    Core cyc   Instruct       Uops     uop p6     uop p7   uops RAT
     44869      50011     102027     101027       1003          0     101046
     44869      50009     102027     101027       1010          0     101065
     44843      50010     102027     101027       1013          0     101067
     44869      50009     102027     101027       1002          0     101040
     44846      50012     102027     101027        999          0     101040
```

Clock is the clock count measured with the `RDTSC` instruction. *Core cyc* is the clock count measured with the "core clock cycles" counter. The CPU can change its core frequency dynamically in order to save power. The count on the global clock sometimes changes while the core clock count is unchanged. We will use the Core clock cycle count as our unit of measurement in order to get consistent results. AMD and VIA processors do not have a core clock counter. On these processors we are warming up the CPU by keeping it busy for some time in order to get the maximum clock frequency. It may be necessary to turn off any power-saving features in the BIOS setup in order to get consistent clock counts.

The latency measurements are made with a chain of instructions where the output of one instruction is input for the next, e.g.

```
MULPS XMM0, XMM1
MULPS XMM1, XMM0
MULPS XMM0, XMM1
MULPS XMM1, XMM0
 ...
```

We can see that 100,000 instructions take 500,000 clocks. Thus, the latency for this instruction is 5 clocks.

The throughput measurements are made with a sequence of instructions where each instruction does not have to wait for the previous one to finish, e.g.

```
MULPS XMM1, XMM0
MULPS XMM2, XMM0
MULPS XMM3, XMM0
MULPS XMM4, XMM0
 ...
```

Here, 100,000 instructions take 50,000 clocks. Thus, the throughput for this instruction is two instructions per clock cycle. The reciprocal throughput is 1/2 = 0.5 clock cycles per instruction.

The throughput test is repeated three times with different counters because it is not possible to have all the counters in one test. Of the 100,000 instructions, we can see that half of them are going to execution port p0 and half of them to port p1. This means that there are two execution units that can handle floating point multiplications, and this explains the throughput of two instructions per clock cycle. The execution units are pipelined so that they can start a new calculation every clock cycle even though the previous calculations are not finished yet.

We have a sequence of 100 `MULPS` instructions inside a loop that repeats 100 or 1000 times in order to get 10,000 or 100,000 operations. The extra 2000 instructions in the example are loop overhead. These 2000 instructions end up as 1000 fused branch instructions at port p6 in this example.

5.4 How the scripts work

The test scripts listed in chapter 5 involve the use of the following files:

File name	Purpose
PMCTestA.cpp	Main C++ file which loads the drivers, sets up the counters, and prints out the results. Needs to be linked together with one of the 'B' files.
PMCTest.h MSRDriver.h PMCTestWin.h	C++ header files. Shared between the main test program and the drivers.

`PMCTestLinux.h`
`intrin1.h`
`MSRdrvL.h`	

`cpugetinfo.cpp`	Program to get CPU-specific information about CPU brand, model, cache size, supported instruction sets, etc. Compiled by `init.sh`, called by `init.sh`, `vars.sh` and by some test scripts.

`shufflelist.cpp`
`testmemcpy.cpp`
`testmemcpyalign.cpp`
	Additional code used by specific test scripts only.

`TemplateB32.nasm` `TemplateB64.nasm`	Assembly language template implementing the test code, 32- or 64-bit mode. Specific code may be inserted at relevant points through macros defined in `*.inc` files.
`memcpy64.nasm` `testmemcpyal.nasm`	Additional code used by specific test scripts only.
`vars.sh`	Called only by the test scripts and by `init.sh`. Defines the following CPU-specific variables in a bash script: `CPUbrand`, `ifamily`, `imodel`, `blockports`, `PMCs`, `PMClist`, `BranchPMCs`, `LoopPMCs`, `CachePMCs`.
`countertypes.inc`	Produced by `init.sh`. Assembly code included by `TemplateB??.nasm`. Defines default PMC counters for the specific CPU. Also defines CPU-specific variables: `CPUbrand`, `ifamily`, `imodel`, `USEAVX`.
`*.inc`	Assembly language include files with macros to insert into `TemplateB??.nasm`. There may be specific `*.inc` files for each test script.
`cpuinfo.txt`	List of instruction sets supported by current processor. Produced by `init.sh`. May be modified manually if there are new instruction sets not detected by `cpugetinfo.cpp` or certain instruction sets you want to avoid.
`results1/info.txt`	Information about the processor, OS version, compiler version, etc. Produced by `init.sh`.
`results1/*.txt`	Results of `*.sh1` test scripts.
`results2/*.txt`	Results of `*.sh2` test scripts.
`a32.o` `a64.o`	This is `PMCTestA.cpp` compiled to an object file, 32- and 64-bit mode. Produced by `init.sh`.
`b32.o` `b64.o`	This is the assembly language templates and include files compiled into object files, 32- and 64-bit mode. Recompiled and overwritten at each test.
`c64.o`	Any additional code needed by specific tests compiled into object file. Recompiled and overwritten at relevant tests.
`x`	Compiled and linked test program. Recompiled and overwritten at each test.

The test scripts use three different programming languages. Files with extension `.sh`, `.sh1` and `.sh2` use Bash scripting language. Files with extension `.cpp` use C++ language. Files with extension `.nasm` use assembly language in the NASM/YASM syntax dialect. (Files with extension `.asm` use assembly language in the MASM syntax dialect. They are not used by the scripted tests).

Each test is made by compiling and linking together two parts, A and B (occasionally three). The A part is `PMCTestA.cpp` which loads the drivers, sets up the counters, and prints out the results. The B part is made from the assembly language template `TemplateB32.nasm` or `TemplateB64.nasm`. It is possible to insert specific initializations, instructions and test code by the use of macros. Single-line macros may be specified at the NASM command line with the `-D` option. Multi-line macros are defined in included files, named `*.inc`, which are inserted at the NASM command line with the option `-P`.

The two parts are compiled, assembled and linked together into an executable test program with the name `x`. This executable program outputs the results which are redirected into a text file. Each test script typically makes and executes many `x` test programs. Each new `x` file overwrites the previous one.

The test scripts may specify which performance monitor counters to use for a specific test. Some useful lists of performance monitor counters are defined by `vars.sh` for the specific CPU that the test is running on:

Counter list	Use
PMCs	Default counter list. Counts instructions, µops etc.
PMClist	List of counter lists to count µops at each execution port. Needs multiple test runs if there are too many counters to test at once.
BranchPMCs	Counts branch mispredictions
LoopPMCs	Counts instructions from loop buffer, µop cache and code cache if relevant for current CPU
CachePMCs	Counts cache misses

Some processors have no counters for detecting which execution unit or execution port is used. Instead it is possible to insert an extra instruction that is known to use a specific execution port in order to test if that execution port is needed by the instruction under test. The variable `blockports`, defined in `vars.sh`, defines a list of ports to block, one by one. The extra instructions to insert are defined in the include file `lt.inc`.

A few of the test scripts are generating extra columns in the output for various calculated results. The file `PMCTestA.cpp` defines two optional extra output columns. The column "Ratio" can contain one of the other counts multiplied by an arbitrary factor, or a ratio of two of the other counts. The column "Extra out" can contain any value calculated in the B module after the test. The headings of these columns can be changed in the B module. See the file `strings.inc` for an example.

6 How to modify or update the test programs

Each new family of microprocessors has new features that may require new test methods or other modifications of the test program and the test scripts.

The performance monitor counters for different CPU families are all defined in `CounterDefinitions[]` in `PMCTestA.cpp`. You may add new entries to this table if you need a counter that is not in this table, and assign a vacant id number to it. The id number is the reference used in the B module. If a new processor family has different counters from the previous families then you may add a new family name to `EProcFamily` in `PMCTest.h` and a detection of this processor family in `CCounters::GetProcessorFamily()` in `PMCTestA.cpp`.

New processor families often have new instruction sets. Each instruction set is detected by a certain bit in the output of the `CPUID` instruction. You can add the specific CPUID bit and the name for the new instruction set in `isets[]` in `cpugetinfo.cpp`. This will enable the test scripts to detect the availability of this instruction set. Then you can add tests for the new instructions to one of the existing test scripts or make a new test script for these instructions. See e.g. `new_int_instr.sh1` for examples of how to make tests that are executed only if a particular instruction set is available.

7 Frequently Asked Questions

7.1 What is this test package for?

I made these test programs as research tools for measuring instruction timings, throughput, micro-operations and various bottlenecks in x86 family microprocessors. My results are published in a series of optimization manuals at www.agner.org/optimize. I am making my test tools public so that others can reproduce my tests, make further tests, and optimize small critical pieces of code.

7.2 Can I use this as a benchmark test?

This is not like an ordinary benchmark test. It can give answers to detailed questions like, what is the latency of floating point multiplication. But it does not measure the overall performance of big programs where different pieces of code go in and out of the caches all the time. One advantage of this test program is that it is open source and you know exactly what it is measuring. Closed source benchmark programs are often suspected of being biased or measuring the wrong things.

7.3 Can I use this test program as a profiler?

No. These test programs are not suitable for testing a whole program and finding where the most critical time-consuming parts (hot spots) are. You need a profiler for that. Once you have identified a hot spot with a profiler, you can isolate this little critical part of the code and test it with the test program in this package. The test program may help you study details of how the critical piece of code is performing and how it can be improved.

7.4 Which microprocessors are supported?

All x86 processors from Intel, AMD and VIA.

7.5 Which operating systems are supported?

The test program can run under Windows and Linux, 32- and 64-bit versions. The automated test scripts currently work only under Linux.

If you are using PMCTest in 64-bit Windows then you have to press F8 during boot and select "Disable Driver Signature Enforcement". Run the IDE or test program as administrator.

7.6 Why do the scripts only run under Linux?

I am doing most of my test by remote access. I cannot afford to buy every new micro-processor model on the market, but friendly IT journalists give me remote access to their test machines. Operating a Windows PC remotely is very troublesome because you can hardly do anything without a mouse under Windows, and a remotely operated mouse is moving erratically with unpredictable delays so that you don't know where you are clicking. In Linux you can do everything from a command line, and no extra equipment is needed for remote operation. Furthermore, Linux has a powerful scripting language that makes it easier to automate the tests. It may be possible to run the scripts under Windows by using Mingw, but this has not been tested.

7.7 Why are the drivers needed?

Setting up the performance monitor counters requires privileged access in the CPU. This is only possible from a device driver.

7.8 Which compiler can I use?

Microsoft's C++ compiler comes with MS Visual Studio or MS Windows Driver Kit. Gnu C++ compiler comes with standard Linux distributions. Intel's C++ compiler has good inline assembly features and may be used instead, though this has not been fully tested.

7.9 Should I compile for debug or release mode?

If the code you are testing is written in C++ or inline assembly in a C++ file then you should turn off debug support in your compiler or select release mode in your project IDE and enable the desired optimization options. The debugging options will interfere with the measurements and give too high clock counts.

The debugging information does not interfere with pure assembly code. If the code you are testing is written in an assembly file then you may turn on debugging options. Depending on which debugger you are using, it may be possible to single-step through the assembly code and see the register values etc.

7.10 Which assembler can I use?

For `.asm` files under 32 bit Windows, use ML or JWASM.
For `.asm` files under 64 bit Windows, use ML64 or JWASM.
For `.nasm` files under Windows or Linux, use NASM or YASM.
For C++ with inline assembly under 32 bit Windows, use Microsoft, Intel or Digital Mars compiler.
For C++ with inline assembly under 64 bit Windows, use Intel C++ compiler.
For C++ with inline assembly in MASM syntax under Linux, use Intel C++ compiler.
For C++ with inline assembly in Intel syntax under Linux, use Gnu C++ and insert appropriate syntax directives (`.intel_syntax noprefix` / `.att_syntax prefix`)
For C++ with inline assembly in AT&T syntax under Linux, use Gnu C++.

7.11 Should I write the test code in C++ or assembly?

If you have written the test code in C++ then the compiler may optimize away parts of your code if the result is not used, or it may move invariant parts of your code outside the test loop. This may give misleading results.

If your purpose is to optimize a piece of C++ code then of course you have to write it in C++. You may have to make some variables global or volatile in order to prevent the compiler from making optimizations that interfere with your measurements. You may have to store the results of your calculations in a global variable or a function return value in order to prevent the compiler from optimizing away a result that is not used. You may put the test code in a separate `.cpp` file and call it from within the test loop.

If you want to test single instructions or very small pieces of code then you will get the most accurate measurements with assembly.

7.12 How does the time stamp counter work?

The time stamp counter is a 64 bit counter inside the microprocessor chip that is incremented on every tick of the microprocessor clock. For example, if the microprocessor is running at 2 GHz then each clock count is 0.5 ns (nanosecond). Measuring clock counts is much more accurate than measuring milliseconds. If you are testing the same code on two different microprocessors of the same type but different clock frequencies, then you can expect to get the same counts if you use the time stamp counter, but not if you measure

milliseconds. Newer Intel processors with count with a resolution equal to the clock multiplier, which can vary. Use the counter for "core clock cycles" as the reference to get consistent results.

Some of the test programs use only the lower 32 bits of the 64 bit counter.

The test programs use the serializing instruction CPUID before and after reading the time stamp counter in order to prevent out-of-order execution to interfere with the measurements.

7.13 How do the performance monitor counters work?

The performance monitor counters are implemented inside the microprocessor chip for test purposes. They are not used during normal program execution. These counters can be set up to count various events such as cache misses or branch mispredictions. You need privileged access to set up the performance monitor counter, but the counters can be read in non-privileged code when enabled. The performance monitor counters do not interfere with clock measurements.

The PMCTest programs use a kernel mode driver `MSRDriver32.sys` or `MSRDriver64.sys` under Windows or `MSRdrv` under Linux to get privileged access to set up the counters. The driver is loaded the first time the test program starts.

The PMCDOS test program runs entirely in privileged mode. Therefore, it does not need a special driver. But this test program is obsolete and can only run with old operating systems.

7.14 What do I need the performance monitor counters for?

These counters are useful for counting instructions, micro-operations, cache misses, branch mispredictions and other events that are useful for testing program performance.

If the critical part of a program has many cache misses then you may improve the performance by rearranging data so that variables that are used together are also stored together. If the critical part of a program has many branch mispredictions then you may improve the performance by limiting the number of branches or making them more predictable.

7.15 Why are there three versions of the driver?

`MSRDriver32.sys` is used by PMCTest in 32-bit Windows systems.
`MSRDriver64.sys` is used by PMCTest in 64-bit Windows systems, even if the test program runs in 32 bit mode.
`MSRdrv` is used by PMCTest in Linux systems, both 32 and 64 bit.

7.16 If the drivers do not work

Under Windows:
Make sure the driver files `MSRDriver32.sys` and `MSRDriver64.sys` are in the path. The 64 bit driver `MSRDriver64.sys` is needed under 64-bit Windows, even if running in 32-bit mode.
If running under 64-bit Windows: Press F8 during system boot and select "Disable Driver Signature Enforcement". Run as administrator.
Ignore the popup message "Windows requires a digitally signed driver".

Under Linux:
Make sure you have installed the driver as described in chapter 3.1. You have to reinstall the driver every time the computer has been restarted, using:

```
        sudo ./install.sh
```

7.17 Error compiling .cpp file

Make sure the character set is ASCII, not Unicode. In Visual C++ set:
        Project → Properties → General → Character set → Not Set.
Make sure you compile for console mode, unmanaged code.
Do not enable precompiled headers. Comment out `#include <intrin.h>` in the `.cpp` files if
the compiler does not support intrinsic functions.

7.18 I get the error message: "Cannot open intrin.h"

Comment out the line `#define USE_INTRINSICS` in the `.cpp` files to disable the use of intrinsic
functions. The program will use inline assembly instead. Alternatively, use the included
`intrin1.h` file for Microsoft compilers.

7.19 I get the error message: No matching counter definition found

The counter definitions are CPU-specific. Change the counter types under
`CounterTypesDesired` in the B module. The available counter types are defined in
`PMCTestA.cpp`.

7.20 How can I define new performance monitor counter events?

You may add new event specifications for the appropriate microprocessor family in the
`CounterDefinitions[]` table at the bottom of `PMCTestA.cpp`.

See IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming
Guide and AMD BIOS and Kernel Developer's Guide for specification of the available PMC
events. The PMC events for VIA processors are currently undocumented.

7.21 Why is the first clock count higher than the rest?

Because the code and data are unlikely to be cached at the first test run, and branches are
more likely to be mispredicted. The first count represents the time it takes to execute the
test code when code and data are not in the level-1 cache. The subsequent counts
represent the time it takes when everything is in the level-1 cache.

Some of the later counts may occasionally be higher in the event of interrupts occurring
during the test or reorganization of a trace cache.

7.22 How accurate are the counts?

The clock counts for pure assembly code are usually accurate to +/- 4 clocks. If you want to
measure a single instruction accurately, then repeat the instruction many times. Example
(MASM syntax):

```
REPT 100
    SHR  EAX, 5
ENDM
```

This will produce a hundred `SHR EAX,5` instructions. If the measured clock count for 100
instructions is 396 then you know that the `SHR` instruction takes exactly 4 clock cycles. The
resolution of the clock counter is equal to the clock multiplier in newer Intel processors. For
example, if the clock multiplier is 11 then all clock counts will be divisible by 11. It is more

accurate to use the counter for "Core clock cycles" rather than the time stamp counter as reference on Intel processors.

If you want higher counts without exceeding the code cache size then include the above code in a loop, for example:

```
MOV ECX, 10000
ALIGN 16
L:
    REPT 100
        SHR  EAX, 5
    ENDM

    DEC  ECX
    JNZ  L
```

7.23 Why do I get negative counts?

The test programs make a reference measurement without the test code and then a similar measurement with the test code. The reference count is then subtracted from the test count. The reference count may be higher than the test count because of random fluctuations, and this will give a negative result.

7.24 Where can I get the compilers?

Microsoft Visual Studio can be bought from software dealers. You may use the free express edition of visual studio.

Microsoft 32 bit C++ compiler is included with Microsoft Windows Driver Kit (WDK) or Driver Development Kit (DDK) which you can download from www.microsoft.com.

Intel C++ compiler 32/64 bit for Windows or Linux can be bought or obtained as a time limited trial version from www.intel.com.

Digital Mars C++ compiler can be downloaded for free from www.digitalmars.com.

Gnu C++ compiler can be installed in Linux by using normal installation procedures. It can be installed in Windows as part of the Mingw or Cygwin package.

7.25 Where can I get the assemblers?

The Microsoft assemblers are included in some versions of Microsoft Visual Studio or Microsoft Windows Driver Kit or Microsoft Driver Development Kit.

The Microsoft Macro Assembler reference manual is found at msdn.microsoft.com.

The NASM, YASM and JWASM assemblers are freeware and support several different operating systems. This may be a useful alternative to the Microsoft assembler.

For inline assembly in 32 bit Windows you may use Microsoft, Intel or Digital Mars C++ compiler.

For inline assembly in 64 bit Windows you can only use Intel C++ compiler.

For inline assembly in 32 bit and 64 bit Linux you can use Intel C++ compiler if you prefer MASM syntax. For AT&T syntax you can also use the Gnu C++ compiler.

7.26 Can I run multiple threads?

Yes. The `PMCTest.zip` package allows multi-threaded testing. The results are less reliable than single-threaded tests. Set the define `NUM_THREADS` in the B module to the desired number of threads. All threads will run the same code simultaneously. It is useful to set the number of threads to 3. If the CPU can run two threads in each core then the test program will put two of the three threads in the same core and one in a separate core. If the threads running together are contending for a limited resource then they will run slower than the thread running alone. You will also get delays if multiple threads are writing to the same cache line.

7.27 How can the latency of memory read and write instructions be measured?

It can't be measured without some kind of physical probe at the memory chips. We can measure the latency of a memory write followed by a read from the same address. A modern CPU will use store forwarding where possible, so that we are actually measuring the latency of store forwarding rather than the latency of cache access. You may define the read latency as the latency of pointer chasing, i.e. walking through a linked list of pointers to pointers (repeated `MOV RAX,[RAX]`). The apparent write latency can then be defined as the store forwarding latency minus the pointer chasing latency. While these values are fictive, they are useful for calculating the total latency of a long dependency chain.

7.28 How can the latency be measured of an instruction that uses different types of registers for input and output?

This is usually impossible. You cannot measure the latency of an instruction that has one type of registers as input and another type of registers as output, e.g. `MOVD EAX,XMM0`, because you cannot make a dependency chain with this instruction only. It is only possible to measure the combined latency of such an instruction and another instruction that converts the other way, e.g. `MOVD EAX,XMM0 / MOVD XMM0,EAX`. The combined latency can then be divided arbitrarily between the two instructions. Sometimes you may make a sensible choice about how much of the total latency to assign to each of these two instructions, based on the number of micro-operations used by each instruction, by making a three-way chain with a memory intermediate, or by using technical documentation from the CPU vendor. However, if something is impossible to measure then it is also irrelevant for calculating the performance of a piece of software. Only the combined latency is relevant.

8 License

These test programs are published under the Gnu General Public License version 3. See www.gnu.org/licenses/#GPL.