PROCESSOR-ORIENTED PERFORMANCE MEASUREMENT TOOL

Martin Schwarzbauer(a), Michael Bogner(b), Franz Wiesinger(c), Andreas Gschwandtner(d)

(a, b, c, d) Upper Austria University of Applied Sciences, Hagenberg Austria,
Hardware/Software Design & Embedded Systems Design

(a)martin.schwarzbauer@fh-hagenberg.at, (b)michael.bogner@fh-hagenberg.at,
(c)franz.wiesinger@fh-hagenberg.at, (d)andreas.gschwandtner@fh-hagenberg.at

ABSTRACT
Nowadays, a lot of powerful and different processors with special techniques for improved performance exist. The clock frequency was doubled within a year in the last decades and the number of cores is increasing continuously. Because commonly available performance measuring tools like Microsoft Windows Task Manager are known not to display the exact load of a processor, it is not possible to compare applications and different implementations regarding to their performance on different processors. A special tool is needed which allows the correct and processor-oriented measuring of the processor's load. This paper describes the technique for measuring the performance of modern processors precisely and shows a sample implementation for an Intel Core 2 Duo processor on the Microsoft Windows operating system. Using the implemented tool it is possible to analyze and compare different applications regarding to their performance. The tests have shown that no application is able to generate a processor load higher than 30%.

Keywords: processor performance, performance measurement tool, performance counter

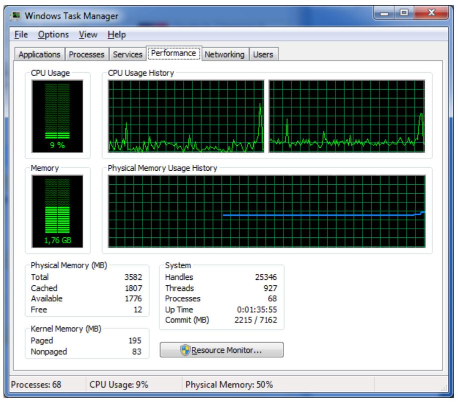
1. INTRODUCTION

Nowadays, a lot of powerful and different processors with different techniques for improved performance exist. The clock rate has been doubled within a year in the last decades and new multi core processors have been developed and sold by the manufacturer. All these techniques improve the performance of the processor and speed up the execution of instructions. But this doesn't mean that the execution of an application also speeds up with the same ratio like the processors do.

It is not possible to utilize the actual processors. A lot of the available performance gets lost and cannot be used for the execution of instructions. The processor itself would be able to execute more instructions but the biggest problem and the bottleneck in current personal computers is the slow access on peripherals.

Processors with doubled clock rates would be able to process nearly twice the number of instructions at the same time but if, for example, data from the hard disk or other peripherals is fetched the processor has to wait. The peripherals like random access memory (RAM) or the hard disk are connected to the processor via bus systems. These bus systems are not as quick as the processor, so it has to wait until the data arrives.

For performance optimization and comparison of different applications a tool is needed to measure the real performance of the processors. There are a lot of tools which are able to display the load of the processor over the time. The most common and widely-used tool is the Microsoft Windows Task Manager (Microsoft Corporation, 2010b) on Microsoft Windows operating systems (Figure 1). There also exist a lot of other possibilities to measure the performance - for example at the high level programming language C# (Microsoft Corporation, 2010).



Physical Memory (MB)	
Total	3582
Cached	1807
Available	1776
Free	12

Kernel Memory (MB)	
Paged	195
Nonpaged	83

System	
Handles	25346
Threads	927
Processes	68
Up Time	0:01:35:55
Commit (MB)	2215 / 7162

Processes: 68 CPU Usage: 9% Physical Memory: 50%

Figure 1: Microsoft Windows Task Manager.

The advantage using these tools is the simplicity in the usage for measuring the performance, but there is a big disadvantage for processor-oriented performance measurements because the measured load is simply incorrect. Available tools show the load of the processor from the point of view of an operating system. This means that wait times – when fetching data from RAM or hard disk – are shown as 100% load of the processor. From this point of view this is correct because without the data from the peripheral the processor couldn't continue its work. But for processor-oriented performance measurements this distores and falsifies the results as the processor is actually idle.

This paper describes a solution to measure the real processor load so that it is possible to evaluate performance optimizations. Using such a measurement tool opens new options for comparing different implementations or compiler options (compile for speed, compile for size) and helps to explore and assemble special performance design pattern for software development to speed up processing simply by design.

2. PERFORMANCE MEASURMENT

The best way to measure the exact load of a processor is to use the processor itself for the performance measurement. The two most common and widely used processors are manufactured by *Intel* and *AMD*. Both implement on their processors options for performance measurement.

Each processor has some specially built in registers in hardware – the so called model specific registers (MSR). The number of available MSRs on Intel processors can be found in (Intel Corporation, 2010, Appendix B). A subset of these MSRs could be used to measure the performance directly in hardware. These registers are called *Performance Counters*. The performance counters could be configured to count specific and processor dependent events in hardware (Dringowski, 2008; Intel Corporation, 2010). Some typical events are *Instructions Retired*, *Instruction per Cycle*, *Level 1 Cache Miss*, and so on. Using the correct event it is possible to calculate the exact load of a processor without the wait times for memory access, stalls and so on.

The advantage of these registers is that they count the configured events in hardware without any overhead in software and impact on the processor’s behaviour. To read or write these MSRs the processor uses an assembler instruction (for example on Intel processors: *rdmsr* and *wrmsr*) which has to be executed in real-address mode or at privilege level 0 (Intel Corporation, 2009). The execution of an instruction at privilege level 0 requires on the two most common operating systems - UNIX and Microsoft Windows - a special driver to execute the instructions and access the MSRs. Figure 2 shows an overview of the concept to access the performance counter registers using this driver.

The available events and registers for performance counting are limited and different for each processor – also within a processor family. On newer processors there are more events available than on older ones and the address of the registers also change. So it is necessary to implement the performance measuring for each processor differently.

On multi core processors it is necessary to measure the performance of each core independent from the others. The manufacturer implements for each core a set of registers which could be used to count different events. For the sake of convenience these registers have the same address on each core. To access the register it is necessary to ensure that the application which reads or writes the register is executed at the core the event should be counted.

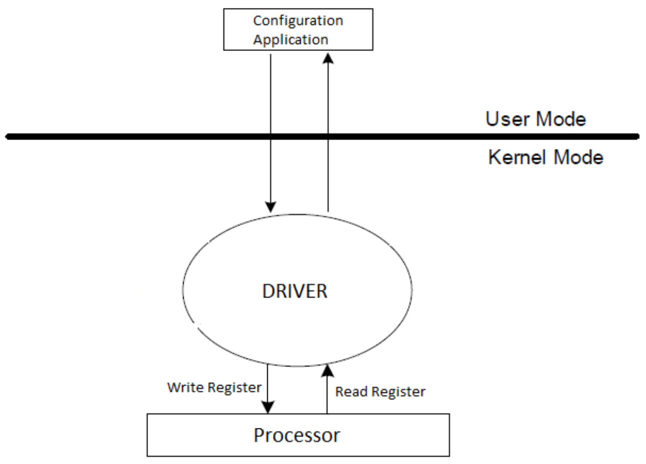


Figure 2: Accessing the performance counter registers using a driver.

The usage of the performance counters always follows the same principle described below:

1. Ensure that execution is running on core X.
2. Set the configuration register to count the specific event.
3. Continue step 1 + 2 for each core and event.
4. Ensure that execution is running on Core X.
5. Read number of counted events from the register.
6. Continue step 4 + 5 for each core.

Intel introduced different MSRs years ago in their processors so it was able to count different processor events in hardware. The configuration and access to these MSRs was processor depended. Since the Pentium 4 processor Intel standardised the access and configuration of the MSRs for performance measurement. This standardisation provides two different versions to configure up to seven defined events (even more on newer processors) for performance measurement. These seven architecture and processor independent events are

1. Unhalted Core Cycles
2. Instructions Retired
3. Unhalted Reference Cycles
4. LLC Reference
5. LLC Misses
6. Branch Instructions Retired
7. Branch Misses Retired

Figure 3 shows the necessary MSRs for configuration and performance measurement in dependency of the two different available versions (Registers marked with * are used for configuration). When any other events except the seven mentioned above should be used, the configuration and usage is described in the different processor's manuals.

When using the event *Instructions Retired* it is possible to calculate the processors' load in percent using (1). Because of the processors ability to execute more than one instruction per cycle the maximum number of instruction per cycles must be known and used for the load calculation in (1).

$$Load\ [\%] = \frac{Instructions\ \ Retired}{Clock\ Cycles * max\ .Instructions\ \ per\ Cycle} \qquad (1)$$

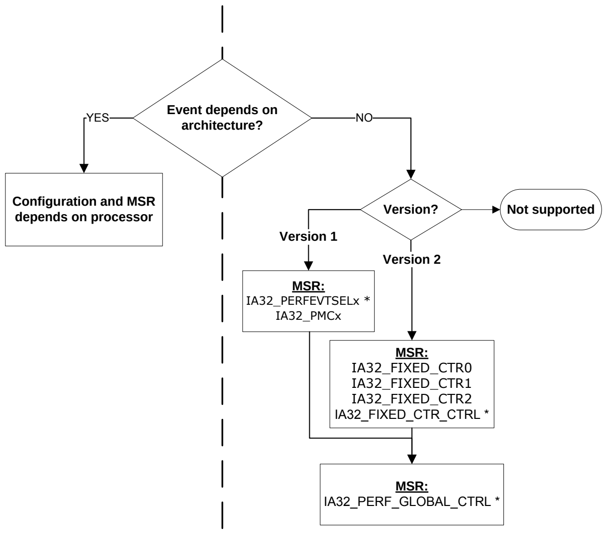


Figure 3: Overview of the available MSRs in dependency of the supported version.

The configuration for counting the *Instructions Retired* event differs by the two supported versions. Using the assembler instruction *cpuid* it is possible to detect which version is supported by the current used processor. Based on the supported version there are different MSRs for configuration and reading the collected performance data. In version 1 each of the seven events mentioned above could be counted – but maximum two events at the same time. Version 2 only supports counting the three following events

1. Instructions Retired (IA32_FIXED_CTR0)
2. Unhalted Core Cycles (IA32_FIXED_CTR1)
3. Unhalted Reference Cycles (IA32_FIXED_CTR2)

Figures 4, 5 and 6 show the different MSRs in detail that are needed for configuration.

MSR IA32_PERFEVTSELx (Figure 4) is used to configure the performance counting mechanism in version 1. The fields Unit Mask and Event Select identify the event which should be counted. USR and OS bit specify if the event should be counted in user mode and/or in operating system mode. Bit EN enables counting the selected event.

The MSR shown in Figure 5 is used to enable each of the three possible events in version 2. Bit PMI defines if an interrupt should be generated when a counter overflow occurs. The bit EN specifies the mode where the events should be counted.

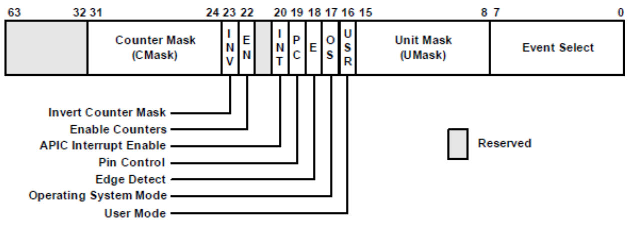
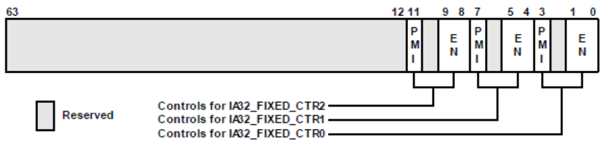


Figure 4: IA32_PERFEVTSELx MSR is used to configure the counting mechanism in version 1 (Intel Corporation, 2010).



63
12 11
9 8 7
5 4 3
1 0
PMI
EN
PMI
EN
PMI
EN
Reserved
Controls for IA32_FIXED_CTR2
Controls for IA32_FIXED_CTR1
Controls for IA32_FIXED_CTR0

Figure 5: IA32_FIXED_CTR_CTRL MSR is used to configure and enable the event counting mechanism in version 2 (Intel Corporation, 2010).

In Figure 6 the global enable MSR is shown. When bit 32, 33 and/or 34 is set, the event counting for version 2 is enabled. Bit 0 and 1 is used to enable performance measurement in version 1.

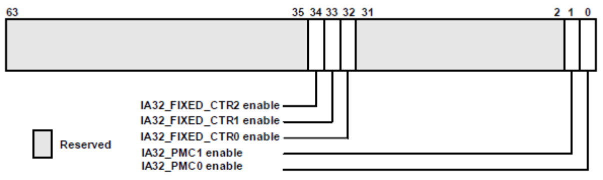


Figure 6: IA32_PERF_GLOBAL_CTRL MSR is used to enable the different events (Intel Corporation, 2010).

To retrieve the number of counted events the MSRs IA32_PMCx in version 1 and IA32_FIXED_CTRx in version 2 must be read.

The following pseudo code shows the usage of the *Instructions Retired* event on an Intel Core 2 Duo processor using version 1.

```
if event supported then
   if version 1 supported then
      switch to core 1;

      // set UMASK and EventSelect to
      // specify the event
      // EN=1: activate counting
      // OS=USR=1: count event in user
      // and kernel mode
      IA32_PERFEVTSEL0= (UMASK=0x00) |
          (EventSelect=0xC0) | (EN=1) |
          (USR=1) | (OS=1);

      switch to core 2;

      // config core 2
      IA32_PERFEVTSEL0= (UMASK=0x00) |
          (EventSelect=0xC0) | (EN=1) |
          (USR=1) | (OS=1);

      //enable counting
      IA32_PERF_GLOBAL_CTRL =
                    (IA32_PMC_0 enable = 1);

            while not terminate then
                sleep(pollintervall);

                // read events counted on core 1
                Switch to core 1;
                Value1 = IA32_PMC0;

                // read events counted on core 2
                Switch to core 2;
                Value2 = IA32_PMC0;
            end while
        end if
    end if
```

With the collected number of occurred events stored in Value1 and Value2 it is possible to calculate the processor's load in percent. When using version 2 the configuration and access to the counted events only differs from version 1 in different MSRs.

Because of the widely-used combination of Intel processor and the operating system *Microsoft Windows* a sample application has been implemented for this platform for the performance measuring on an Intel Core 2 Duo processor.

3. IMPLEMENTATION

In Figure 7 all components required for the realization of a performance analysis tool are shown. The arrows illustrate the interaction between the different components.

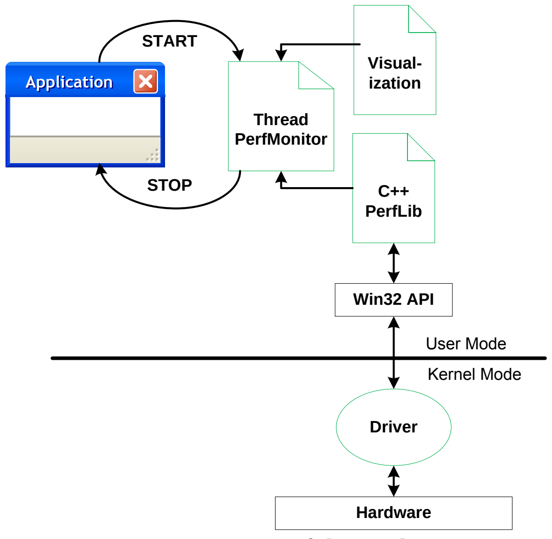


Figure 7: Overview of the implementation.

The C++ class ThreadPerfMonitor configures the performance measurement. Using the methods *start* or *stop* the application can control the time of measurement. The collected performance data will be written into a file in comma-separated-values (CSV) format, so that it is possible to process the values with different applications.

The interaction between the processor's hardware is encapsulated in a C++ class called PerfLib. This class provides methods to write and read the different MSRs using the driver.

As already mentioned a driver is needed to execute the instructions *rdmsr* and *wrmsr* to access the different MSRs. In Figure 8 the usage of the driver is shown for reading a MSR using the *rdmsr* instruction. A Win32 API accesses the driver via Input/Output Controls (IOCTL). Using the IOCTL IOCTL_READMSR the driver dispatches the registered function and executes the *rdmsr* assembler instruction with the given parameters. The read value will be returned to the user mode and could be processed.

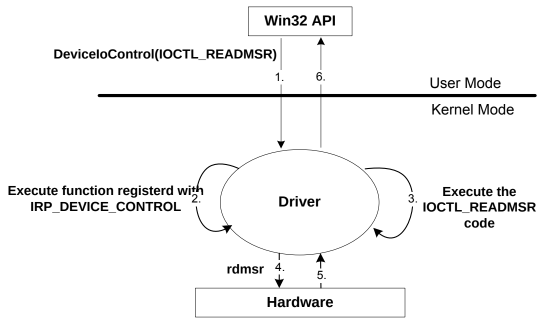

Win32 API
DeviceIoControl(IOCTL_READMSR)
1.
6.
User Mode
Kernel Mode
Execute function registerd with IRP_DEVICE_CONTROL
2.
Driver
3.
Execute the IOCTL_READMSR code
rdmsr
4.
5.
Hardware

Figure 8: Sequence reading a MSR.

This implementation can easily be extended for different processors on the Microsoft Windows platform. Accessing the registers on UNIX, a different driver must be implemented but the assembler instructions and the configuration sequence of these MSRs will be the same. The application uses the *Instructions Retired* event to collect information of the processor so that it is possible to calculate the load in percent.

4. RESULTS

The implemented measurement application is written in C++ and is able to count different events on an Intel Core 2 Duo processor. Currently only this processor is supported but the software can be easily expanded so that any other processor and event can be used for performance measurement. This could be done by specifying the different processor dependent register addresses for reading and writing the registers and the configuration values that have to written to the MSRs. At the moment the application configures the processor to count the event *Instructions Retired*. Using this event, the number of clock cycles, and the maximum count of instructions (on Intel Core 2 Duo: 4; Intel Corporation, 2008), which could be executed at one clock cycle, it is possible to calculate the processor's load in percent (see (1) on page 3).

To visualize the results the application writes the data into a file in CSV format. Using the written values it is possible to visualize them, use it for calculations or any other application can read the values and process them. One of the most common tools to create a graph based on a CSV file is Microsoft Windows Excel (Figure 4 show a graph created with this application).

The realized performance measurement tool could be used as a standalone application to record the load of the processor's cores independent of any other application. Another practice is to use the performance tool explicit in the own application for performance measurement. This could be easily done in applications implemented in C++ because the developer only has to call a Start- and Stop-Routine for the performance measurement. The advantage is that only the period of time is recorded the developer wants to measure.

Figure 9 and Figure 10 show the comparison between the Microsoft Windows Task Manager and the values recorded with the implemented processor-oriented performance measurement tool during the execution of a test application. The Microsoft Windows Task Manager shows a load of 100% on core one. In comparison to this we can see in Figure 10 that the processor only uses effectively 25% of its available resources. This example shows the difference in the performance measurement tools. The test application fetches a lot of data from the hard disk so that the processor has to spend most of its time on waiting. Available tools display the processor's load from the point of view of an operating system. This means that while fetching data from peripherals the operating system cannot continue with its work so the load for the operating system is 100%.

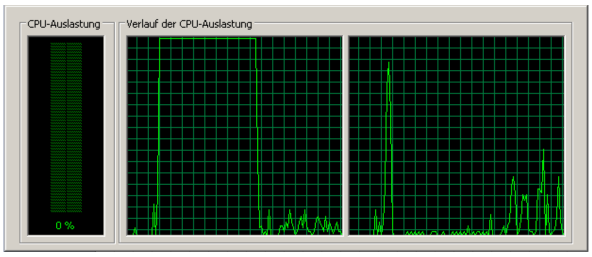


Figure 9: Processor load measured with Microsoft Windows Task Manager.

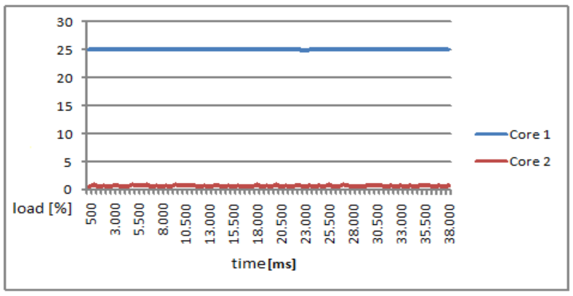


Figure 10: Processor load measured with the processor's performance counters.

The implemented tool shows the exact load of a processor so these results are more significant than the results of common tools. Based on these results it is not advisable to use common tools for performance measurements and testing optimization possibilities.

With the implemented application and their measured results different optimization techniques could be evaluated and compared regarding their performance on the processor's core.

Based on different test applications (e.g. benchmark Prime95, Mersenne Research Inc., 2010) there were nearly always the same results. Modern processors can reach only a maximum load of about 30%. This means that the processor spends about 70% of the available time on waiting for data or simply doing nothing. At this point of view an application has a good performance when it can reach a maximum of about 30% load on common processors – because this is still the maximum reachable load using current computer systems.

To sum up it could be said that the results have shown that modern processors and highly optimized compilers aren't able to create applications that can use the available resource of the processor's core in an adequate way. It is necessary to find different techniques and design pattern for software development to improve application's execution so that the load increases. Using the implemented processor-oriented performance measurement tool it is possible to collect performance data and find software techniques and design pattern and their different impact on the processor's load.

REFERENCES

Dringowski, P.J., 2008. *Basic Performance Measurements for AMD Athlon(TM) 64, AMD Opteron(TM) and AMD Phenom(TM) Processors.* Available from: http://developer.amd.com/Assets/Basic_Performance_Measurements.pdf [accessed 31 March 2010]

Intel Corporation, 2008. *Intel(R) Core(TM)2 Duo Processor. Maximizing Dual-Core Performance Efficiency*. Available from: http://download.intel.com/products/processor/core2duo/mobile_prod_brief.pdf [accessed 5 April 2010]

Intel Corporation, 2009. *Intel(R) 64 and IA-32 Architectures. Software Developer's Manual. Volume 2B: Instruction Set Reference, N-Z.* Available from: http://www.intel.com/design/processor/manuals/253667.pdf [accessed 31 March 2010]

Intel Corporation, 2010. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual. Volume 3B. System Programming Guide, Part2.* Available from: http://developer.intel.com/Assets/PDF/manual/253669.pdf [accessed 17 July 2010]

Mersenne Research Inc., 2010. *Prime95*. Available from: http://mersenne.org/freesoft [accessed 31 March 2010]

Microsoft Corporation, 2010a. *Performance Counters*. Available from: http://msdn.microsoft.com/en-us/library/aa373083(VS.85).aspx [accessed 31 March 2010]

Microsoft Corporation, 2010b. *What is Task Manager?*. Available from: http://windows.microsoft.com/en-US/windows-vista/What-is-Task-Manager [accessed 9 April 2010]