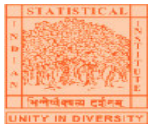


A Study on Performance Monitoring Counters in x86-Architecture

Shibdas Bandyopadhyay
Roll No. MTC0414
M.Tech CS 1st Year
Indian Statistical Institute

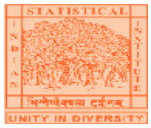
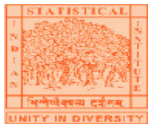


Table of Contents



1. Introduction

Performance Monitoring counters is a valuable tool for measuring performance of a program which can be analyzed to identify the bottlenecks in the program. These counters are present in the most modern processors including Intel Pentium, Pentium Pro, P6, Pentium 4, AMD, Cyrix etc. These counters are hardware registers attached with the processor which measures various programmable events occurring in the processor. They do not require any additional overhead and supports a wide range of events. This document describes these counters for x86 systems, specifically for Intel Pentium Processor as they are most widely used in current personal computers.

The whole document is divided into four sections:-

1. First Sections describes various hardware components associated with the performance counters in Pentium processors and the events that they can measure.

2. Second Section describes an existing implementation which provides an API for accessing various performance counters present in x86-architecture.

3. Third Section describes a proof-of-concept implementation which measure various performance counters when a particular process is run.

4. Fourth Section provides how we can derive some useful performance metrics from the data we get by measuring various events.

2. Hardware Set-up for Performance monitoring

We will describe the performance counters and related hardware components present in the Pentium Architecture. In particular, performance monitoring in P6 family of the processors (including Pentium, Pentium Pro, and Pentium II & III) will be described. Pentium 4 and Xeon family of processors employs more advanced technique for measuring events although the basic methodology is the same.

The P6 family processors provide two 40-bit performance counters, allowing two types of events to be monitored simultaneously. These counters can either count events or measure duration. When counting events, a counter is incremented each time a specified event takes place or a specified number of events take place. When measuring duration, a counter counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level.

The performance-monitoring counters are supported by four MSRs(Model Specific Registers, These registers are specific to a particular Model of the processor and they are not guaranteed to present in the future processors, e.g. registers described here are not present in Pentium 4 / Xeon Processors) : the performance event select MSRs (PerfEvtSel0 and PerfEvtSel1) and the performance counter MSRs (PerfCtr0 and PerfCtr1). These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0. The PerfCtr0 and PerfCtr1 MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction.

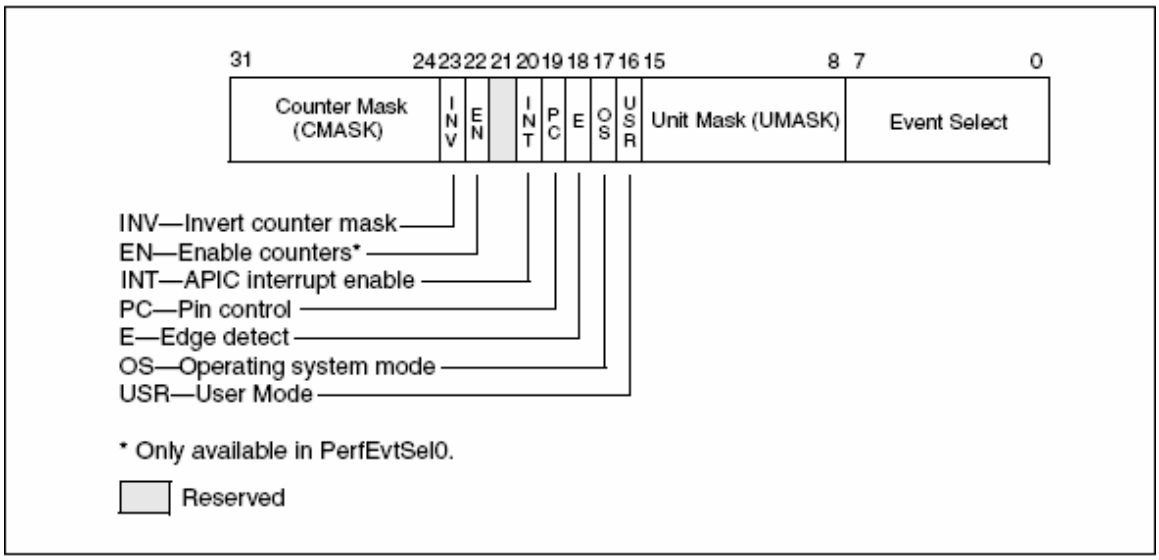
2.1 PerfEvtSel0 and PerfEvtSel1 MSRs

The PerfEvtSel0 and PerfEvtSel1 MSRs control the operation of the performance-monitoring counters, with one register used to set up each counter. They specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. Following figure shows the flags and fields in these MSRs.

Event select field (bits 0 through 7)
Selects the event to be monitored

Unit mask (UMASK) field (bits 8 through 15)
Further qualifies the event selected in the event select field. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states.

USR (user mode) flag (bit 16)
Specifies that events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.



OS (operating system mode) flag (bit 17)
Specifies that events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USR flag.

E (edge detect) flag (bit 18)
Enables (when set) edge detection of events. The processor counts the number of deserted to asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

PC (pin control) flag (bit 19)
When set, the processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.

INT (APIC interrupt enable) flag (bit 20)
When set, the processor generates an exception through its local APIC on counter overflow.

EN (Enable Counters) Flag (bit 22)
This flag is only present in the PerfEvtSel0 MSR. When set, performance counting is enabled in both performance-monitoring counters; when clear, both counters are disabled.

INV (invert) flag (bit 23)
Inverts the result of the counter-mask comparison when set, so that both greater than and less than comparisons can be made.

Counter mask (CMASK) field (bits 24 through 31)
When nonzero, the processor compares this mask to the number of events counted during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented. This mask can be used to count events only if multiple occurrences happen per clock (for example, two or more instructions retired per clock). If the counter-mask field is 0, then the counter is incremented each cycle by the number of events that occurred that cycle.

2.2 PerfCtr0 and PerfCtr1 MSRs

The performance-counter MSRs (PerfCtr0 and PerfCtr1) contain the event or duration counts for the selected events being counted. The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.
The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed. Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

The WRMSR instruction cannot arbitrarily write to the performance-monitoring counter MSRs (PerfCtr0 and PerfCtr1). Instead, the lower-order 32 bits of each MSR may be written with any value and the high-order 8 bits are sign-extended according to the value

of bit 31. This operation allows writing both positive and negative values to the performance counters.

2.3 Starting and Stopping the Performance-Monitoring Counters

The performance-monitoring counters are started by writing valid setup information in the PerfEvtSel0 and/or PerfEvtSel1 MSRs and setting the enable counters flag in the PerfEvtSel0 MSR. If the setup is valid, the counters begin counting following the execution of a WRMSR instruction that sets the enable counter flag. The counters can be stopped by clearing the enable counters flag or by clearing all the bits in the PerfEvtSel0 and PerfEvtSel1 MSRs. Counter 1 alone can be stopped by clearing the PerfEvtSel1 MSR.

2.4 Monitoring Counter Overflow

The P6 family processors provide the option of generating a local APIC interrupt when a performance-monitoring counter overflows. This mechanism is enabled by setting the interrupt enable flag in either the PerfEvtSel0 or the PerfEvtSel1 MSR. The primary use of this option is for statistical performance sampling.

To use this option, the operating system should do the following things on the processor for which performance events are required to be monitored:

- Provide an interrupt vector for handling the counter-overflow interrupt.
- Initialize the APIC PERF local vector entry to enable handling of performance-monitor counter overflow events.
- Provide an entry in the IDT that point to a stub exception handler that returns without executing any instructions.
- Provide an event monitor driver that provides the actual interrupt handler and modifies the reserved IDT entry to point to its interrupt routine.

When interrupted by a counter overflow, the interrupt handler needs to perform the following actions:

- Save the instruction pointer (EIP register), code-segment selector, TSS segment selector, counter values and other relevant information at the time of the interrupt.
- Reset the counter to its initial setting and return from the interrupt.

An event monitor application utility or another application program can read the information collected for analysis of the performance of the profiled application.

2.5 P6 Family Processor Performance-Monitoring Events

The following table lists the events that can be counted with the performance-monitoring counters and read with the RDPMC instruction for the P6 family processors. The unit

column gives the micro architecture or bus unit that produces the event; the event number column gives the hexadecimal number identifying the event; the mnemonic event name column gives the name of the event; the unit mask column gives the unit mask required (if any); the description column describes the event; and the comments column gives additional information about the event.

All of these performance events are model specific for the P6 family processors and are not available in this form in the Pentium 4 processors or the Pentium processors. Some events (such as those added in later generations of the P6 family processors) are only available in specific processors in the P6 family. All performance event encodings not listed in table are reserved and their use will result in undefined counter results.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Data Cache Unit (DCU)	43H	DATA_MEM_REFS	00H	All loads from any memory type. All stores to any memory type. Each part of a split is counted separately. The internal logic counts not only memory loads and stores, but also internal retries. 80-bit floating-point accesses are double counted, since they are decomposed into a 16-bit exponent load and a 64-bit mantissa load. Memory accesses are only counted when they are actually performed (such as a load that gets squashed because a previous cache miss is outstanding to the same address, and which finally gets performed, is only counted once). Does not include I/O accesses, or other nonmemory accesses.	
	45H	DCU_LINES_IN	00H	Total lines allocated in the DCU.	

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	46H	DCU_M_LINES_IN	00H	Number of M state lines allocated in the DCU.	
	47H	DCU_M_LINES_ OUT	00H	Number of M state lines evicted from the DCU. This includes evictions via snoop HITM, intervention or replacement.	
	48H	DCU_MISS_ OUTSTANDING	00H	Weighted number of cycles while a DCU miss is outstanding, incremented by the number of outstanding cache misses at any particular time. Cacheable read requests only are considered. Uncacheable requests are excluded. Read-for-ownerships are counted, as well as line fills, invalidates, and stores.	An access that also misses the L2 is short-changed by 2 cycles (i.e., if counts N cycles, should be N+2 cycles). Subsequent loads to the same cache line will not result in any additional counts. Count value not precise, but still useful.
Instruction Fetch Unit (IFU)	80H	IFU_IFETCH	00H	Number of instruction fetches, both cacheable and noncacheable, including UC fetches.	
	81H	IFU_IFETCH_ MISS	00H	Number of instruction fetch misses. All instruction fetches that do not hit the IFU (i.e., that produce memory requests). Includes UC accesses.	
	85H	ITLB_MISS	00H	Number of ITLB misses.	
	86H	IFU_MEM_STALL	00H	Number of cycles instruction fetch is stalled, for any reason. Includes IFU cache misses, ITLB misses, ITLB faults, and other minor stalls.	

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	87H	ILD_STALL	00H	Number of cycles that the instruction length decoder is stalled.	
L2 Cache[1]	28H	L2_IFETCH	MESI 0FH	Number of L2 instruction fetches. This event indicates that a normal instruction fetch was received by the L2. The count includes only L2 cacheable instruction fetches; it does not include UC instruction fetches. It does not include ITLB miss accesses.	
	29H	L2_LD	MESI 0FH	Number of L2 data loads. This event indicates that a normal, unlocked, load memory access was received by the L2. It includes only L2 cacheable memory accesses; it does not include I/O accesses, other nonmemory accesses, or memory accesses such as UC/WT memory accesses. It does include L2 cacheable TLB miss memory accesses.	

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	2AH	L2_ST	MESI 0FH	Number of L2 data stores. This event indicates that a normal, unlocked, store memory access was received by the L2. Specifically, it indicates that the DCU sent a read-for-ownership request to the L2. It also includes Invalid to Modified requests sent by the DCU to the L2. It includes only L2 cacheable memory accesses; it does not include I/O accesses, other nonmemory accesses, or memory accesses such as UC/WT memory accesses. It includes TLB miss memory accesses.	
	24H	L2_LINES_IN	00H	Number of lines allocated in the L2.	
	26H	L2_LINES_OUT	00H	Number of lines removed from the L2 for any reason.	
	25H	L2_M_LINES_INM	00H	Number of modified lines allocated in the L2.	
	27H	L2_M_LINES_OUTM	00H	Number of modified lines removed from the L2 for any reason.	
	2EH	L2_RQSTS	MESI 0FH	Total number of L2 requests.	
	21H	L2_ADS	00H	Number of L2 address strobes.	
	22H	L2_DBUS_BUSY	00H	Number of cycles during which the L2 cache data bus was busy.	
	23H	L2_DBUS_BUSY_RD	00H	Number of cycles during which the data bus was busy transferring read data from L2 to the processor.	

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
External Bus Logic (EBL)[2]	62H	BUS_DRDY_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which DRDY# is asserted. Utilization of the external system data bus during data transfers.	Unit Mask = 00H counts bus clocks when the processor is driving DRDY#. Unit Mask = 20H counts in processor clocks when any agent is driving DRDY#.
	63H	BUS_LOCK_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which LOCK# is asserted on the external system bus.[3]	Always counts in processor clocks.
	60H	BUS_REQ_ OUTSTANDING	00H (Self)	Number of bus requests outstanding. This counter is incremented by the number of cacheable read bus requests outstanding in any given cycle.	Counts only DCU full-line cacheable reads, not RFOs, writes, instruction fetches, or anything else. Counts "waiting for bus to complete" (last data chunk received).
	65H	BUS_TRAN_BRD	00H (Self) 20H (Any)	Number of burst read transactions.	
	66H	BUS_TRAN_RFO	00H (Self) 20H (Any)	Number of completed read for ownership transactions.	
	67H	BUS_TRANS_WB	00H (Self) 20H (Any)	Number of completed write back transactions.	
	68H	BUS_TRAN_ IFETCH	00H (Self) 20H (Any)	Number of completed instruction fetch transactions.	
	69H	BUS_TRAN_INVAL	00H (Self) 20H (Any)	Number of completed invalidate transactions.	
	6AH	BUS_TRAN_PWR	00H (Self) 20H (Any)	Number of completed partial write transactions.	

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	6BH	BUS_TRANS_P	00H (Self) 20H (Any)	Number of completed partial transactions.	
	6CH	BUS_TRANS_IO	00H (Self) 20H (Any)	Number of completed I/O transactions.	
	6DH	BUS_TRAN_DEF	00H (Self) 20H (Any)	Number of completed deferred transactions.	
	6EH	BUS_TRAN_ BURST	00H (Self) 20H (Any)	Number of completed burst transactions.	
	70H	BUS_TRAN_ANY	00H (Self) 20H (Any)	Number of all completed bus transactions. Address bus utilization can be calculated knowing the minimum address bus occupancy. Includes special cycles, etc.	
	6FH	BUS_TRAN_MEM	00H (Self) 20H (Any)	Number of completed memory transactions.	
	64H	BUS_DATA_RCV	00H (Self)	Number of bus clock cycles during which this processor is receiving data.	
	61H	BUS_BNR_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the BNR# pin.	

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	7AH	BUS_HIT_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HIT# pin.	Includes cycles due to snoop stalls. The event counts correctly, but the BPM*i* pins function as follows based on the setting of the PC bits (bit 19 in the PerfEvtSel0 and PerfEvtSel1 registers): <•>If the core-clock-to-bus-clock ratio is 2:1 or 3:1, and a PC bit is set, the BPM*i*pins will be asserted for a single clock when the counters overflow. <•>If the PC bit is clear, the processor toggles the BPM*i*pins when the counter overflows. <•>If the clock ratio is not 2:1 or 3:1, the BPM*i* pins will not function for these performance-monitoring counter events.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	7BH	BUS_HITM_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HITM# pin.	Includes cycles due to snoop stalls. The event counts correctly, but the BPM*i* pins function as follows based on the setting of the PC bits (bit 19 in the PerfEvtSel0 and PerfEvtSel1 registers): <•>If the core-clock-to-bus-clock ratio is 2:1 or 3:1, and a PC bit is set, the BPM*i* pins will be asserted for a single clock when the counters overflow. <•>If the PC bit is clear, the processor toggles the BPM*i*pins when the counter overflows. <•>If the clock ratio is not 2:1 or 3:1, the BPM*i* pins will not function for these performance-monitoring counter events.
	7EH	BUS_SNOOP_ STALL	00H (Self)	Number of clock cycles during which the bus is snoop stalled.	

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Floating-Point Unit	C1H	FLOPS	00H	Number of computational floating-point operations retired. Excludes floating-point computational operations that cause traps or assists. Includes floating-point computational operations executed by the assist handler. Includes internal sub-operations for complex floating-point instructions like transcendentals. Excludes floating-point loads and stores.	Counter 0 only.
	10H	FP_COMP_OPS_EXE	00H	Number of computational floating-point operations executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This number does not include the number of cycles, but the number of operations. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	Counter 0 only.
	11H	FP_ASSIST	00H	Number of floating-point exception cases handled by microcode.	Counter 1 only. This event includes counts due to speculative execution.
	12H	MUL	00H	Number of multiplies. This count includes integer as well as FP multiplies and is speculative.	Counter 1 only.
	13H	DIV	00H	Number of divides. This count includes integer as well as FP divides and is speculative.	Counter 1 only.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	14H	CYCLES_DIV_ BUSY	00H	Number of cycles during which the divider is busy, and cannot accept new divides. This includes integer and FP divides, FPREM, FPSQRT, etc. and is speculative.	Counter 0 only.
Memory Ordering	03H	LD_BLOCKS	00H	Number of load operations delayed due to store buffer blocks. Includes counts caused by preceding stores whose addresses are unknown, preceding stores whose addresses are known but whose data is unknown, and preceding stores that conflicts with the load but which incompletely overlap the load.	
	04H	SB_DRAINS	00H	Number of store buffer drain cycles. Incremented every cycle the store buffer is draining. Draining is caused by serializing operations like CPUID, synchronizing operations like XCHG, interrupt acknowledgment, as well as other conditions (such as cache flushing).	
	05H	MISALIGN_ MEM_REF	00H	Number of misaligned data memory references. Incremented by 1 every cycle, during which either the processor's load or store pipeline dispatches a misaligned µop. Counting is performed if it is the first or second half, or if it is blocked, squashed, or missed. In this context, misaligned means crossing a 64-bit boundary.	MISALIGN_MEM_ REF is only an approximation to the true number of misaligned memory references. The value returned is roughly proportional to the number of misaligned memory accesses (the size of the problem).

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	07H	EMON_KNI_PREF_DISPATCHED	00H 01H 02H 03H	Number of Streaming SIMD extensions prefetch/weakly-ordered instructions dispatched (speculative prefetches are included in counting): 0: prefetch NTA 1: prefetch T1 2: prefetch T2 3: weakly ordered stores	Counters 0 and 1. Pentium III processor only.
	4BH	EMON_KNI_PREF_MISS	00H 01H 02H 03H	Number of prefetch/weakly-ordered instructions that miss all caches: 0: prefetch NTA 1: prefetch T1 2: prefetch T2 3: weakly ordered stores	Counters 0 and 1. Pentium III processor only.
Instruction Decoding and Retirement	C0H	INST_RETIRED	OOH	Number of instructions retired.	A hardware interrupt received during/after the last iteration of the REP STOS flow causes the counter to undercount by 1 instruction. An SMI received while executing a HLT instruction will cause the performance counter to not count the RSM instruction and undercount by 1.
	C2H	UOPS_RETIRED	00H	Number of μops retired.	
	D0H	INST_DECODED	00H	Number of instructions decoded.	
	D8H	EMON_KNI_INST_RETIRED	00H 01H	Number of Streaming SIMD extensions retired: 0: packed & scalar 1: scalar	Counters 0 and 1. Pentium III processor only.
	D9H	EMON_KNI_COMP_INST_RET	00H 01H	Number of Streaming SIMD extensions computation instructions retired: 0: packed and scalar 1: scalar	Counters 0 and 1. Pentium III processor only.
Interrupts	C8H	HW_INT_RX	00H	Number of hardware interrupts received.	

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	C6H	CYCLES_INT_ MASKED	00H	Number of processor cycles for which interrupts are disabled.	
	C7H	CYCLES_INT_ PENDING_ AND_MASKED	00H	Number of processor cycles for which interrupts are disabled and interrupts are pending.	
Branches	C4H	BR_INST_ RETIRED	00H	Number of branch instructions retired.	
	C5H	BR_MISS_PRED_ RETIRED	00H	Number of mispredicted branches retired.	
	C9H	BR_TAKEN_ RETIRED	00H	Number of taken branches retired.	
	CAH	BR_MISS_PRED_ TAKEN_RET	00H	Number of taken mispredictions branches retired.	
	E0H	BR_INST_ DECODED	00H	Number of branch instructions decoded.	
	E2H	BTB_MISSES	00H	Number of branches for which the BTB did not produce a prediction.	
	E4H	BR_BOGUS	00H	Number of bogus branches.	
	E6H	BACLEARS	00H	Number of times BACLEAR is asserted. This is the number of times that a static branch prediction was made, in which the branch decoder decided to make a branch prediction because the BTB did not.	

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Stalls	A2H	RESOURCE_ STALLS	00H	Incremented by 1 during every cycle for which there is a resource related stall. Includes register renaming buffer entries, memory buffer entries. Does not include stalls due to bus queue full, too many cache misses, etc. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	
	D2H	PARTIAL_RAT_ STALLS	00H	Number of cycles or events for partial stalls. This includes flag partial stalls.	
Segment Register Loads	06H	SEGMENT_REG_ LOADS	00H	Number of segment register loads.	
Clocks	79H	CPU_CLK_ UNHALTED	00H	Number of cycles during which the processor is not halted.	
MMX Unit	B0H	MMX_INSTR_ EXEC	00H	Number of MMX Instructions Executed.	Available in Intel Celeron, Pentium II and Pentium II Xeon processors only. Does not account for MOVQ and MOVD stores from register to memory.
	B1H	MMX_SAT_ INSTR_EXEC	00H	Number of MMX Saturating Instructions Executed.	Available in Pentium II and Pentium III processors only.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	B2H	MMX_UOPS_ EXEC	0FH	Number of MMX µops Executed.	Available in Pentium II and Pentium III processors only.
	B3H	MMX_INSTR_ TYPE_EXEC	01H 02H 04H 08H 10H 20H	MMX packed multiply instructions executed. MMX packed shift instructions executed. MMX pack operation instructions executed. MMX unpack operation instructions executed. MMX packed logical instructions executed. MMX packed arithmetic instructions executed.	Available in Pentium II and Pentium III processors only.
	CCH	FP_MMX_TRANS	00H 01H	Transitions from MMX instruction to floating-point instructions. Transitions from floating-point instructions to MMX instructions.	Available in Pentium II and Pentium III processors only.
	CDH	MMX_ASSIST	00H	Number of MMX Assists (that is, the number of EMMS instructions executed).	Available in Pentium II and Pentium III processors only.
	CEH	MMX_INSTR_RET	00H	Number of MMX Instructions Retired.	Available in Pentium II processors only.
Segment Register Renaming	D4H	SEG_RENAME_ STALLS	01H 02H 04H 08H 0FH	Number of Segment Register Renaming Stalls: Segment register ES Segment register DS Segment register FS Segment register FS Segment registers ES + DS + FS + GS	Available in Pentium II and Pentium III processors only.
	D5H	SEG_REG_ RENAMES	01H 02H 04H 08H 0FH	Number of Segment Register Renames: Segment register ES Segment register DS Segment register FS Segment register FS Segment registers ES + DS + FS + GS	Available in Pentium II and Pentium III processors only.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	D6H	RET_SEG_ RENAMES	00H	Number of segment register rename events retired.	Available in Pentium II and Pentium III processors only.

3. Overview of an existing implementation for accessing performance monitoring counters

There are many high level GUI tools as well as low level API's for accessing performance counters in Intel IA-32 architecture. Intel's Vtune is a GUI tool which performs various analyses on the data collected from performance counters. "lperfex" is a command line tool for Linux which provides counter values when a particular program runs. PAPI is a set of API's for accessing performance counters from applications. We are going to analyze the Linux x86 Performance-Monitoring Counters Driver and Kernel patch developed by Mikael Petterson. Most of the tools on Linux depend on this driver and as it is open-source software, it made us possible to view how the things are done.

This package is named "perfctr" (we will refer to this name afterwards) and its purpose, as described in the "readme" file of this package:-
"This package adds support to the Linux kernel for using the Performance-Monitoring Counters (PMCs) found in many modern x86-class processors. Supported processors are:
- All Intel Pentium processors, i.e. Pentium, Pentium MMX, Pentium Pro, Pentium II, Pentium III, and Pentium 4, including Celeron and Xeon versions.
- The AMD K7 and K8 processor families.
- Cyrix 6x86MX, MII, and III.
- VIA C3 (Cyrix III).
- Centaur WinChip C6/2/3.

Limited support is available for generic x86 processors with a Time-Stamp Counter but no PMCs, such as the AMD K6 family. For these processors, only TSC (Time Stamp Counter)-based cycle-count measurements are possible. However, all high-level facilities implemented by the driver are still available."

Various features of this package as described in the "readme" file:

"Each Linux process has its own set of "virtual" PMCs. That is, to a process the PMCs appear to be private and unrelated to the activities of other processes in the system. The virtual PMCs have 64-bit precision, even though current processors only implement 40 or 48-bit PMCs. Each process also has a virtual Time-Stamp Counter (TSC). On most machines, the virtual PMCs can be sampled entirely in user-space without incurring the overhead of a system call.

A process accesses its virtual PMCs by opening /proc/self/perfctr and issuing system calls on the resulting file descriptor. A user-space library is included which provides a more high-level interface.

The driver also supports global-mode or system-wide PMCs. In this mode, each PMC on each processor can be controlled and read. The PMCs and TSC on active processors are sampled periodically and the accumulated sums have 64-bit precision. Global-mode

PMCs are accessed via the /dev/perfctr device file; the user-space library provides a more high-level interface.

Support for performance-counter overflow interrupts is provided for Intel P4 and P6, and AMD K7 and K8 processors."

The "perfctr" package consists of two parts:-

- A Patch to the Linux Kernel
- A driver for providing access to PMC's.

Let us analyze two parts in details:-

3.1 Linux Kernel Patch

We consider the patch to Linux Kernel Version 2.4.18. Modifications in the kernel code are required to support per-process PMC. PMC's are general purpose registers in the processor and are not associated with the code currently executed by the processor. Modifications are needed in the data structures & context switching routines of the processes to hold per-process values of these counters. The various modifications and their cause are explained:-

- Changes are made to the "credits" file for including the details of this patch. "CONFIG_PERFCTR" option is added to the Kernel Configuration file to provide an option for including the support of PMC's in the kernel. There are various sub-options under this option:-

  a) CONFIG_PERFCTR_DEBUG:- This option enables additional internal consistency checking in the perfctr driver. The scope of these checks is unspecified and may vary between different versions of the driver.

  b) CONFIG_PERFCTR_INIT_TESTS:- This option makes the driver run additional hardware tests during initialisation. This is not necessary for normal operation, but it can be useful during driver development.

  c) CONFIG_PERFCTR_VIRTUAL:- The processor's performance-monitoring counters are special-purpose global registers. This option adds support for virtual per-process performance-monitoring counters which only run when the process to which they belong is executing. This improves the accuracy of performance measurements by reducing "noise" from other processes.

  d) CONFIG_PERFCTR_GLOBAL:- This option adds driver support for global-mode (system-wide) performance-monitoring counters. In this mode, the driver allows each performance-monitoring counter on each

processor to be controlled and read. The driver provides a sampling timer to maintain 64-bit accumulated event counts.

- Changes are made to the ioctl-numbers.txt file to include the ioctl number associated with this driver.

- Changes the made to the Makefile and Config files of the kernel to add commands for compiling the driver.

- Now we will discuss the changes made to the actual kernel source files. Here "linux-2.4.18-perfctr" refers to the top level directory containing the kernel source.

  As we know that APIC (Advanced Programmable Interrupt Controller) generates an interrupt when a counter overflow occurs. Intel provides three types of interrupt descriptors: Task, Interrupt, and Trap Gate Descriptors. Task Gate Descriptors are irrelevant to Linux, but its Interrupt Descriptor Table contains several Interrupt and Trap Gate Descriptors. Linux classifies them as follows, using a slightly different breakdown and terminology from Intel:

  Interrupt gate
  An Intel interrupt gate that cannot be accessed by a User Mode process (the gate's DPL field is equal to 0). All Linux interrupt handlers are activated by means of interrupt gates, and all are restricted to Kernel Mode.

  System gate
  An Intel trap gate that can be accessed by a User Mode process (the gate's DPL field is equal to 3). The four Linux exception handlers associated with the vectors 3, 4, 5, and 128 are activated by means of system gates, so the four assembly language instructions int3, into, bound, and int $0x80 can be issued in User Mode.

  Trap gate
  An Intel trap gate that cannot be accessed by a User Mode process (the gate's DPL field is equal to 0). Most Linux exception handlers are activated by means of trap gates.

  The following architecture-dependent functions are used to insert gates in the IDT (Interrupt Descriptor Table):

  set_intr_gate (n, addr)
  Inserts an interrupt gate in the *n* th IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to addr, which is the address of the interrupt handler. The DPL (Descriptor Privilege Level) field is set to 0.

set_system_gate(n,addr)
Inserts a trap gate in the *n* th IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to addr, which is the address of the exception handler. The DPL field is set to 3.

set_trap_gate(n,addr)
Similar to the previous function, except the DPL field is set to 0.

As an interrupt gate for the Local APIC has to be set, "set_intr_gate" function is used. The modification is done in linux-2.4.18-perfctr/arch/i386/kernel/i8259.c

```
#include <asm/perfctr.h>

#if defined(CONFIG_X86_LOCAL_APIC) && defined(CONFIG_KPERFCTR)
        set_intr_gate(LOCAL_PERFCTR_VECTOR, perfctr_interrupt);
#endif
```

This patch includes two header files in include/Linux and include/asm-i386 both are named "perfctr.h" which contains various data structure and assembly instructions needed for the "perfctr" driver.

The function "perfctr_interrupt" is implemented in include/asm-i386/perfctr.h which does the necessary acknowledgement job and runs an interrupt handler.

- Changes are made to the IRQ(Interrupt Requests) handling functions in linux-2.4.18-perfctr/arch/i386/kernel/irq.c to describe irqs associated with PMC's as follows:-

```
#if defined(CONFIG_X86_LOCAL_APIC) && defined(CONFIG_KPERFCTR)
        p += sprintf(p, "PMC: ");
        for (j = 0; j < smp_num_cpus; j++)
                p += sprintf(p, "%10u ",
                        apic_lvtpc_irqs[cpu_logical_map(j)]);
        p += sprintf(p, "\n");
#endif
```

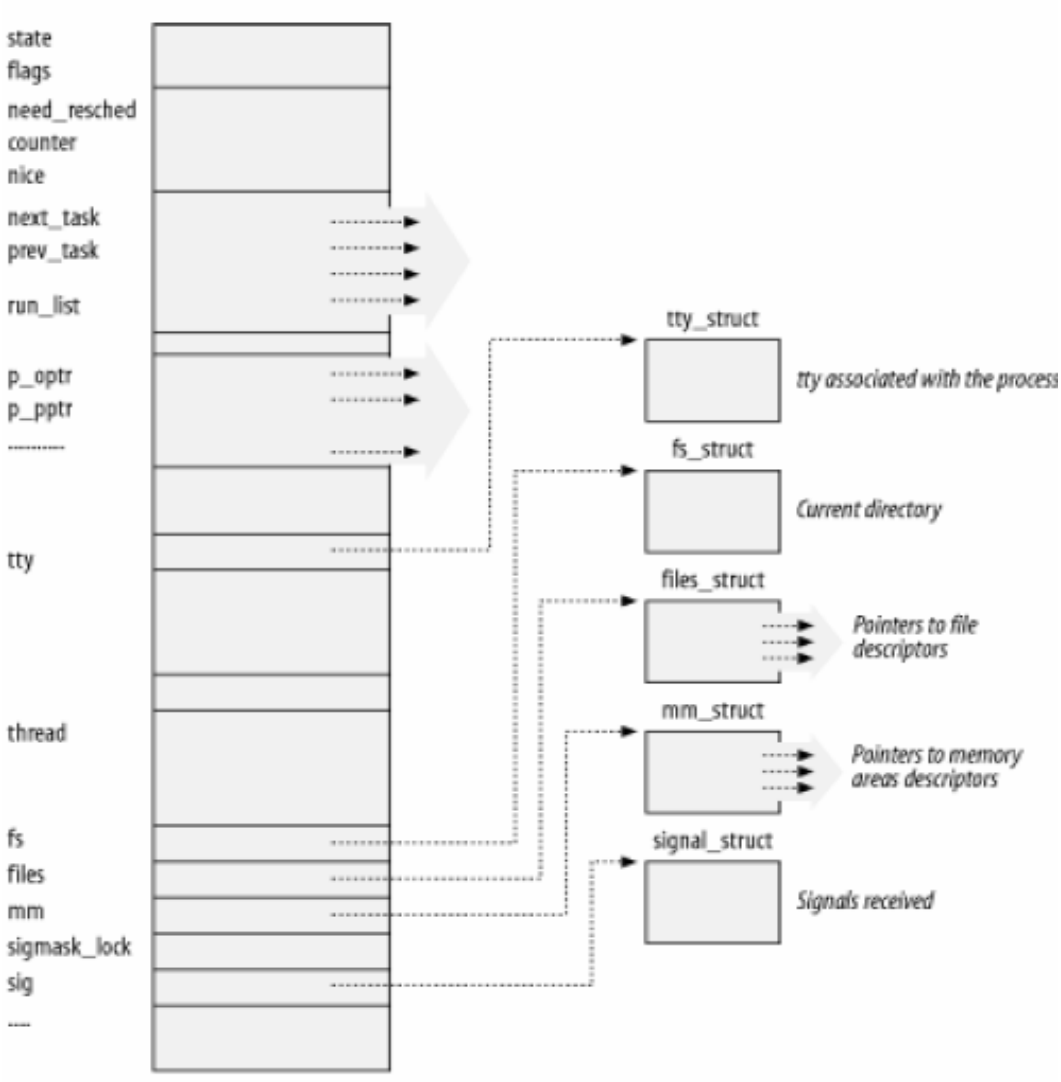
- LOCAL_PERFCTR_VECTOR and non-maskable interrupts are defined in kernel header files linux-2.4.18-perfctr/include/asm-i386/hw_irq.h and linux-2.4.18-perfctr/include/asm-i386/apic.h as:-

/* In unpatched Linux kernel FIRST APIC VECTOR which is available to the drivers starts from 0xed which is now occupied by LOCAL PERFCTR VECTOR and so it is modified to start from 0xee. */

```
#define LOCAL_PERFCTR_VECTOR        0xee
#define FIRST_SYSTEM_VECTOR         0xee
```

extern struct pm_dev *nmi_pmdev;
extern unsigned int nmi_perfctr_msr;

- Changes the made to the process specific files to provide support for per-process PMC's. Let us look at how the process descriptor is stored by the Linux Kernel. It will make us understand how the changes help to implement this feature.
  To manage processes, the kernel must have a clear picture of what each process is doing. It must know, for instance, the process's priority, whether it is running on a CPU or blocked on an event, what address space has been assigned to it, which files it is allowed to address, and so on. This is the role of the *process descriptor* — a task_struct type structure whose fields contain all the information related to a single process. As the repository of so much information, the process descriptor is rather complex. In addition to a large number of fields containing process attributes, the process descriptor contains several pointers to other data structures that, in turn, contain pointers to other structures. Figure describes the Linux process schematically.



At every process switch, the hardware context of the process being replaced must be saved somewhere. Thus, each process descriptor includes a field called thread of type thread_struct, in which the kernel saves the hardware context whenever the process is being switched out.

PMC's values are stored in the thread_struct structure which is defined in linux-2.4.18-perfctr/include/asm-i386/processor.h as follows:-

```
struct vperfctr;                /* opaque; no need to depend on <linux/perfctr.h> */

struct thread_struct {
            unsigned long  esp0;
            unsigned long  eip;
            unsigned long  esp;
            unsigned long  fs;
            unsigned long  gs;
            /* Hardware debugging registers */
            unsigned long  debugreg[8];  /* %%db0-7 debug registers */
           /* fault info */
           unsigned long   cr2, trap_no, error_code;
           /* floating point info */
           union i387_union    i387;
           /* virtual 86 mode info */
           struct vm86_struct  * vm86_info;
           unsigned long       screen_bitmap;
           unsigned long       v86flags, v86mask, saved_esp0;
           /* IO permissions */
           int ioperm;
           unsigned long   io_bitmap[IO_BITMAP_SIZE+1];
           /* performance counters */
           struct vperfctr *perfctr;
};

/* Also for initializing this structure values are filled with zeroes */
#define INIT_THREAD  {                                          \
       0,                                                       \
       0, 0, 0, 0,                                              \
       { [0 ... 7] = 0 },       /* debugging registers */  \
       0, 0, 0,                                                 \
       { { 0, }, },                 /* 387 state */             \
       0,0,0,0,0,                                               \
       0,{~0,}                          /* io permissions */    \
       NULL,                            /* performance counters */ \
}
```

- The patch modifies the process handling routines to include calls to the virtual per-process counter driver routines which do the job of saving and restoring various counter values. This patch adds the following calls to the kernel process handling routines in linux-2.4.18-perfctr/arch/i386/kernel/process.c.

```
void exit_thread(void)
{
    perfctr_exit_thread(&current->thread);
}

/* This the higher level context switching function
    here we store the various PMC counts of the current process and also
    resume with the next process */
void __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
   …
   perfctr_suspend_thread(prev);
   …
   perfctr_resume_thread(next);
}

void copy_segments(struct task_struct *p, struct mm_struct *new_mm)
{
     …
     perfctr_copy_thread(&p->thread);
     …
 }
```

These pertctr function calls are defined in include/linux/perfctr.h which calls the corresponding vperfctr functions defined in driver/virtual.c. From driver/virtual.c the calls go to x86.c where these are defined as follows:-
(These basically stores/restores the counter values to/from the perfctr data structures)

```
void perfctr_cpu_suspend(struct perfctr_cpu_state *state)
{
       unsigned int i, cstatus, nractrs;
       struct perfctr_low_ctrs now;

        #if PERFCTR_INTERRUPT_SUPPORT
        if( perfctr_cstatus_has_ictrs(state->cstatus) )
                perfctr_cpu_isuspend(state);
         #endif
         perfctr_cpu_read_counters(state, &now);
         cstatus = state->cstatus;
```

```
        if( perfctr_cstatus_has_tsc(cstatus) )
                state->sum.tsc += now.tsc - state->start.tsc;
        nractrs = perfctr_cstatus_nractrs(cstatus);
        for(i = 0; i < nractrs; ++i)
                state->sum.pmc[i] += now.pmc[i] - state->start.pmc[i];
        /* perfctr_cpu_disable_rdpmc(); */   /* not for x86 */
}

void perfctr_cpu_resume(struct perfctr_cpu_state *state)
{
        #if PERFCTR_INTERRUPT_SUPPORT
          if( perfctr_cstatus_has_ictrs(state->cstatus) )
                perfctr_cpu_iresume(state);
        #endif
        /* perfctr_cpu_enable_rdpmc(); */  /* not for x86 or global-mode */
        perfctr_cpu_write_control(state);
        perfctr_cpu_read_counters(state, &state->start);
}

void perfctr_cpu_sample(struct perfctr_cpu_state *state)
{
        unsigned int i, cstatus, nractrs;
        struct perfctr_low_ctrs now;

        perfctr_cpu_read_counters(state, &now);
        cstatus = state->cstatus;
        if( perfctr_cstatus_has_tsc(cstatus) ) {
                state->sum.tsc += now.tsc - state->start.tsc;
                state->start.tsc = now.tsc;
        }
        nractrs = perfctr_cstatus_nractrs(cstatus);
        for(i = 0; i < nractrs; ++i) {
             state->sum.pmc[i] += now.pmc[i] - state->start.pmc[i];
             state->start.pmc[i] = now.pmc[i];
        }
}
```

By this way, perfctr driver maintains a virtual count for all the processes running in the system. At any point of time from state->sum.pmc[i], we get the accumulated values for every process.

- Last thing that the patch does is to provide an interface to the /proc file system so that a process might use /proc/self/perfctr to read the values of various performance counters. Changes are made to linux-2.4.18-perfctr/fs/proc/base.c to reflect these things. As the PMC values are associated with the process we insert

this into the directory of the process in the /proc file systems by modifying the following enumerations:-

```
#include <linux/perfctr.h>

enum pid_directory_inos {
        PROC_PID_INO = 2,
        PROC_PID_STATUS,
        PROC_PID_MEM,
        PROC_PID_CWD,
        PROC_PID_ROOT,
        PROC_PID_EXE,
        PROC_PID_FD,
        PROC_PID_ENVIRON,
        PROC_PID_CMDLINE,
        PROC_PID_STAT,
        PROC_PID_STATM,
        PROC_PID_MAPS,
        PROC_PID_CPU,
        PROC_PID_PERFCTR, /* for PMC */
        PROC_PID_MOUNTS,
        PROC_PID_FD_DIR = 0x8000,       /* 0x8000-0xffff */
};

/* and in their entries in inodes */
#define E(type,name,mode) {(type),sizeof(name)-1,(name),(mode)}

static struct pid_entry base_stuff[] = {
  E(PROC_PID_FD,   "fd",              S_IFDIR|S_IRUSR|S_IXUSR),
  E(PROC_PID_ENVIRON,     "environ",     S_IFREG|S_IRUSR),
  E(PROC_PID_STATUS,      "status",      S_IFREG|S_IRUGO),
  E(PROC_PID_CMDLINE,     "cmdline",     S_IFREG|S_IRUGO),
  E(PROC_PID_STAT,    "stat",         S_IFREG|S_IRUGO),
  E(PROC_PID_STATM, "statm",          S_IFREG|S_IRUGO),
#ifdef CONFIG_SMP
  E(PROC_PID_CPU, "cpu",              S_IFREG|S_IRUGO),
#endif
  E(PROC_PID_MAPS,   "maps",              S_IFREG|S_IRUGO),
  E(PROC_PID_MEM,    "mem",               S_IFREG|S_IRUSR|S_IWUSR),
  E(PROC_PID_CWD,    "cwd",           S_IFLNK|S_IRWXUGO),
  E(PROC_PID_ROOT,   "root",          S_IFLNK|S_IRWXUGO),
  E(PROC_PID_EXE, "exe",              S_IFLNK|S_IRWXUGO),
  #ifdef CONFIG_PERFCTR_VIRTUAL
  E(PROC_PID_PERFCTR,     "perfctr",     PERFCTR_PROC_PID_MODE),
  #endif
  E(PROC_PID_MOUNTS,      "mounts",      S_IFREG|S_IRUGO),
```

```
  {0,0,NULL,0}
};

/* Modifies the proc_base_lookup to include the reference to the various file
operations */
static struct dentry *proc_base_lookup(struct inode *dir, struct dentry *dentry)
{

      …
      #ifdef CONFIG_PERFCTR_VIRTUAL
            case PROC_PID_PERFCTR:
               perfctr_set_proc_pid_ops(inode);
               break;
      #endif
      …
}
```

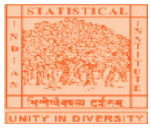
These file operations are mapped into functions in driver/perfctr/virtual.c as follows:-

```
static struct file_operations vperfctr_file_ops = {
                  .owner = THIS_MODULE,
                  .mmap = vperfctr_mmap,
                  .release = vperfctr_release,
                  .ioctl = vperfctr_ioctl,
                   .open = vperfctr_open,
};
```

In vperfctr_open function puts the perfctr structure in the user supplied file structure so that it can access the counters from there. There is some checking whether the process itself is calling the function or its child is calling and allocation of structure is done depending on its creation.

```
static int vperfctr_open(struct inode *inode, struct file *filp)
{
                              struct task_struct *tsk;
                              struct vperfctr *perfctr;
                              int err;

      /* The link from /proc/<pid>/perfctr exists even if the
        hardware detection failed. Disallow open in this case. */
      if( !vperfctr_init_done )
            return -ENODEV;

      /*
       * Allocating a new vperfctr object for the O_CREAT case is
       * done before the self-or-remote-control check.
```

```
* This is because get_empty_vperfctr() may sleep, and in the
* remote control case, the child may have been killed while we
* slept. Instead of dealing with the ugly revalidation issues,
* we allocate ahead of time, and remember to deallocate in case of errors.
* If we only supported 2.4+ kernels, this would be much less of
* an issue, since the task pointer itself remains valid across a sleep thanks to
* get_task_struct().
 perfctr = NULL;
 if( filp->f_flags & O_CREAT ) {
      perfctr = get_empty_vperfctr(); /* may sleep */
      if( IS_ERR(perfctr) )
          return PTR_ERR(perfctr);
 }
 tsk = current;
 if( !proc_pid_inode_denotes_task(inode, tsk) ) { /* remote? */
     tsk = get_task_by_proc_pid_inode(inode);
     err = -ESRCH;
     if( !tsk )
         goto err_perfctr;
     err = ptrace_check_attach(tsk, 0);
     if( err < 0 )
         goto err_tsk;
  }
  if( filp->f_flags & O_CREAT ) {
         err = -EEXIST;
         if( task_thread(tsk)->perfctr )
               goto err_tsk;
               perfctr->owner = tsk;
               task_thread(tsk)->perfctr = perfctr;
   } else {
               perfctr = task_thread(tsk)->perfctr;
               /* In the /proc/pid/perfctr API, there is one user, viz.
                 ioctl PERFCTR_INFO, for which it's ok for perfctr to
                  be NULL. Hence no non-NULL check here. */
   }
   filp->private_data = perfctr; /* put all data to the user */
   if( perfctr )
         atomic_inc(&perfctr->count);
   if( tsk != current )
         put_task_struct(tsk);
         return 0;
   err_tsk:
         if( tsk != current )
             put_task_struct(tsk);
```



```
        err_perfctr:
                if( perfctr )          /* can only occur if filp->f_flags & O_CREAT */
                        put_vperfctr(perfctr);
        return err;
}
```

vperfctr_mmap() and vperfctr_release() do their corresponding my mapping the first page of the memory to the user and releasing the perfctr structure respectively. The virtual also supports ioctls VPERFCTR_STOP, VPERFCTR_UNLINK, VPERFCTR_SAMPLE, VPERFCTR_IRESUME and they get mapped to the corresponding vperfctr_exit(), vperfctr_sample() functions which are described previously.

By providing a /proc interface virtual PMC driver ensures that processes can get the perfctr structure containing PMC values by opening the file /proc/self/perfctr.

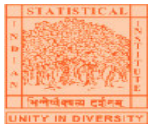
3.2 Global-mode PMC Driver

Global-mode PMC driver emulates a device /dev/perfctr to which users can issue ioctls to obtain values of various PMC's. It defines only a function mapping to ioctl in its file operations structure as follows:-

```
static struct file_operations dev_perfctr_file_ops = {
        .owner = THIS_MODULE,
        .ioctl = dev_perfctr_ioctl,
};
```

List of ioctls that can be sent to this device and their corresponding functions are given by the following function

```
static int dev_perfctr_ioctl(struct inode *inode, struct file *filp,
                             unsigned int cmd, unsigned long arg)
{
        switch( cmd ) {
        case PERFCTR_INFO:
                return sys_perfctr_info((struct perfctr_info*)arg);
        case GPERFCTR_CONTROL:
                return gperfctr_control((struct gperfctr_control*)arg);
        case GPERFCTR_READ:
                return gperfctr_read((struct gperfctr_state*)arg);
        case GPERFCTR_STOP:
                return gperfctr_stop();
        }
        return -EINVAL;
}
```



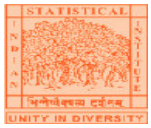
PERFCTR_INFO ioctl returns a structure filled with the information on various counters to the user using copy_to_user function.

GPERFCTR_CONTROL ioctl makes the driver to allocate various perfctr structures and start a timer which will be used to sample the values at periodic intervals.

GPERFCTR_READ ioctl returns a perfctr structure updated with the values of the PMC's.

```
static int gperfctr_read(struct gperfctr_state *arg)
{
        unsigned nrcpus, i;
        struct gperfctr *perfctr;
        struct gperfctr_cpu_state state;

        if( get_user(nrcpus, &arg->nrcpus) )
                return -EFAULT;
        if( nrcpus > smp_num_cpus )
                nrcpus = smp_num_cpus;
        if( sampling_timer.data == 0 )          /* no timer; sample now */
                sample_all_cpus();
        for(i = 0; i < nrcpus; ++i) {
                perfctr = &per_cpu_gperfctr[cpu_logical_map(i)];
                spin_lock(&perfctr->lock);
                state.cpu_control = perfctr->cpu_state.control;
                state.sum = perfctr->cpu_state.sum;
                spin_unlock(&perfctr->lock);
                if( copy_to_user(&arg->cpu_state[i], &state, sizeof state) )
                        return -EFAULT;
        }
        return nr_active_cpus;
}
```

GPERFCTR_STOP ioctls releases the timer and resets various PMC's to their previous values.



4. A proof-of-concept implementation

The implementation is done in Red Hat Linux 8.0, Kernel – 2.4.18 running on a Pentium III 653 MHz. processor. This implementation consists of two parts: -

- A Driver enabling PMC access from User-Mode
- A user Level Program which can measure them.

4.1 PMC Driver

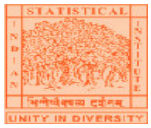
This driver is provides a character device with ioctls for user programs to communicate with it. It defines the open, close and ioclt operations of the device. First in “init_module” it registers the device under the name “pmc”.

```
int init_module(void)
{
 int a;
 if ((pmc_major = register_chrdev(0, "pmc", &pmc_fops))
    == -EBUSY) {
  printk("unable to get major for pmc device\n");
  return -EIO;
 }
 …
}
```

Then it sets the PCE(Performance Counter Enable) bit of the cr4 register to enable “RDPMC” work in any privilege level. Although it is not a good design, it enables user programs to directly access the performance counters.

```
 /*
   In order to use the rdpmc instruction in user mode, we need to set the
   PCE bit of CR4. PCE is 8th bit of cr4, and 256 is 2 << 8
 */
 printk("<4> Setting the PCE bit \n");
 __asm__("movl %%cr4, %0\n\t" /* move CR4 into a */
          "orl $256, %0\n\t"  /* turn on PCE bit in a */
          "movl %0, %%cr4\n" /* copy a back into CR4  */
          : "=r" (a) /* writes to registerized a */
          : /* no inputs */);
```

There are two utility functions pmc_clear() and setCounter which are implemented as follows:-



```
/* pmc_clear clears the PMC specified by counter
 * counter = 0 => perfctr0
 * counter = 1 => perfctr1
 * it uses WRMSR to write the values in the counters
 */
static void pmc_clear(int counter) {
   int counterRegister = PERFCTR0 + counter;

   /* clear the old register */
   __asm__("mov %0, %%ecx\n\t"
           "xor %%edx, %%edx\n\t"
           "xor %%eax, %%eax\n\t"
           "wrmsr\n\t"
           : /* no outputs */
           : "m" (counterRegister)
           : "eax", "ecx", "edx" /* all clobbered */);
}


/* This function writes the value specified by the arg to the counter
 * indicated by counter */

static void setCounter(int counter, unsigned long arg) {
  if ((counter < 0) || (counter > 1)) {
    printk("pmc:setCounter illegal value for counter\n");
  } else {
    int selectionRegister = PERFEVENTSEL0 + counter;
    int counterRegister   = PERFCTR0 + counter;

    pmc_clear(counter);

    /* set the value */
    __asm__("mov %0, %%ecx\n\t"   /* ecx contains the number of the MSR to set */
            "xor %%edx, %%edx\n\t"/* edx contains the high bits to set the MSR to */
            "mov %1, %%eax\n\t"    /* eax contains the log bits to set the MSR to */
            "wrmsr\n\t"
            : /* no outputs */
            : "m" (selectionRegister), "m" (arg)
            : "eax", "ecx", "edx" /* clobbered */);
  }
}
```

As WRMSR instruction and accessing cr4 register can only be done in privilege level 0, they are implemented in the driver portion.

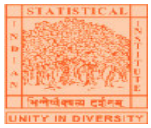
Various ioctl calls are performed as given by the following function:-

```
static int ioctl_pmc(struct inode *inode, struct file *f,
                                unsigned int cmd, unsigned long arg) {

  int a;

  switch (cmd) {
 case 0:
  /* disable counter */
  setCounter(0, 0);

  pmc_clear(0);

  /* set counter 0 */
  setCounter(0, arg);
  break;
 case 1:
  /* disable counter */
  setCounter(1, 0);

  pmc_clear(1);

  /* set counter 1 */
  setCounter(1, arg);
  break;

 default:
  printk("ioctl_pmc: illegal cmd: %d\n", cmd);
  break;
 }
 return 0;
}
```

We are now able to access PMC's using RDPMC instruction once we load the driver in the kernel.

4.2 User Space Program

This program acts like the "perfex"/ "lperfex" program found in SGI IRIX / Linux systems although it does some basic counting of events.



This program is called pmcCount and it can be used in two ways(As Pentium III has only two counters, we can measure at most two events simultaneously):-

pmcCount event1 event2 program args

pmcCount SUM program args

In the first case, it will run the “program” with “args” and at end will give the number of times event1 and event2 has occurred. Each event is associated with a code e.g. DATA_MEM_REFS means the event of accessing memory.

In the second case, after running “program” with “args” a few numbers of times it will measure various events and produce a summary with various performance metrics.

This program at first clears various counters by opening /dev/pmc and issuing ioctls. Then it creates (forks) a child process and runs the program specified by the user. When the process terminates it measures the values of counters using RDPMC instruction and passes to the higher level routine to analyze it.

The “count” function which does the main job is defined as below:-

```
int count(int eventId1, int eventId2, char **args, long long *res1, long long *res2)
{

  int fd;
  int result;


  fd  = open("/dev/pmc", O_RDONLY);

  if (fd < 0) {
          perror("can't open /dev/pmc - is pmc driver installed?");
       return -1;
  }

  result = ioctl(fd, 0, eventId1 | PMC_USER_MASK | PMC_OS_MASK |
PMC_ENABLE_MASK);
  result = ioctl(fd, 1, eventId2 | PMC_USER_MASK | PMC_OS_MASK |
PMC_ENABLE_MASK);

  close(fd);

  /* fork a child process, run the program given by the user, make parent wait for its
     termination */
```

```
 switch(fork()) {
 case -1:
   perror("Can't fork:");
   exit(-1);
 case 0:
   /* in the child */
   execvp(args[0], args);
   perror("can't exec");
   return -1;
   /* NOTREACHED */
   break;
 default:
   /* in the parent */
   {
     int status;
     wait(&status);
     if(status == -1)
        return -1;
     break;
   }
 }

 /* isssue rdpmc instruction to read the counters, write the values to
  * the arguments passed to the function */

 *res1 = rdpmc(0);
 *res2   = rdpmc(1);

 return 0;

}
```

Two utility function rdpmc and rdpmc32 pulls out values from the counters. The function "rdpmc" returns a 64 bit long long value while "rdpmc32" returns a 32 bit integer value. These functions uses the RDPMC instruction to get the values.

```
long long rdpmc(int counter) {
  unsigned eax;
  unsigned edx;
  unsigned long long r;

   __asm__ __volatile__ ("mov %2, %%ecx\n\t"
          "rdpmc\n\t"
          "mov %%eax, %0\n\t"
          "and $255, %%edx\n\t"
          "mov %%edx, %1\n\t"
```

```
          : "=m" (eax), "=m" (edx), "=m" (counter)
          : /* no inputs */
          :  "eax", "ecx", "edx"); /* eax, ecx, edx clobbered */
 r = ((unsigned long long) edx << 32)  | eax;
 return r;
}


int rdpmc32(int counter) {
 unsigned eax;
 __asm__ __volatile__ ("mov %1, %%ecx\n\t"
          "rdpmc\n\t"
          "mov %%eax, %0\n\t"
          : "=m" (eax), "=m" (counter)
          : /* no inputs */
          :  "eax", "ecx", "edx"); /* eax, ecx, edx clobbered */
 return eax;
}
```

For summary operations we measure the following events (Next describes how these events help us to produce some useful performance metrics):-

Events	Event Codes (Pentium III)
Cycles ---	CPU_CLK_UNHALTED
Instructions ---	INST_RETIRED
Loads & Stores --	DATA_MEM_REFS
L1 Misses --	DCU_LINES_IN
L2 Misses --	L2_LINES_IN
Fl. pt. Inst. ---	FLOPS
Branches --	BR_INST_DECODED
Branch Misses --	BR_MISS_PRED_RETIRED
TLB Misses ---	ITLB_MISS
Icache Misses --	IFU_IFETCH_MISS

5. Some Useful Performance Metrics

We can derive some useful performance metrics from the events counted by Performance monitoring counters. Formulas for various performance metrics which can be derived from the events listed above are as follows:-

IPC (Instruction Per Cycle) or CPI = Total Instruction Executed / Total Cycles.

MFLOPS (Millions of Floating Point Operations per second)
= (Total Floating Point Instruction) * CPU Clock (MHz) / Cycles.

L1 Cache Hit Rate = 1 – (L1 Cache Misses) / (Load + Stores)

L2 Cache Hit Rate = 1 – (L2 Cache Misses) / (L1 Cache Misses)

Memory / Flops Ratio = (Loads + Stores) / (Floating Point Instructions Executed)

Branch Rate = (Total Decoded Branch Instruction) / (Total Instruction Executed)

Branch Miss Prediction Rate = (Miss Predicted Braches) / (Total Instruction Executed)

TLB (Translation Look-Aside Buffer) Miss Rate = (TLB Misses) / (Loads + Stores)

L1-TO-L2 Bandwidth used (MB/s) = (L1_Misses * L1 Line size * CPU Clock) / Cycles

L2-TO-Memmory Bandwidth used (MB/s)
= (L2 Misses * L2 Linesize * CPU Clock) / Cycles

As there are about 90 odd events that can be measured with the performance monitoring counters many other performance metrics can be derived from observing these counters.

Our sample implementation provides a summary option (as mentioned earlier) which outputs some of these performance metrics. Here is an output from the pmcCount program which monitors the counters while running "ls" command on current directory.

Command : pmcCount SUM ls

Output:

```
-- These are ls outputs ---
install  Makefile    pmc1.c  pmc.c  pmc.html  pmcTime    pmcTime.c~    pmcTime.o
rdpmc.c  rdpmc.o  testasm.c
it       output_pmc  pmc1.o  pmc.h  pmc.o     pmcTime.c  pmcTime.html  pmcTime.s
rdpmc.h  test     test.c
install  Makefile    pmc1.c  pmc.c  pmc.html  pmcTime    pmcTime.c~    pmcTime.o
rdpmc.c  rdpmc.o  testasm.c
it       output_pmc  pmc1.o  pmc.h  pmc.o     pmcTime.c  pmcTime.html  pmcTime.s
rdpmc.h  test     test.c
install  Makefile    pmc1.c  pmc.c  pmc.html  pmcTime    pmcTime.c~    pmcTime.o
rdpmc.c  rdpmc.o  testasm.c
it       output_pmc  pmc1.o  pmc.h  pmc.o     pmcTime.c  pmcTime.html  pmcTime.s
rdpmc.h  test     test.c
install  Makefile    pmc1.c  pmc.c  pmc.html  pmcTime    pmcTime.c~    pmcTime.o
rdpmc.c  rdpmc.o  testasm.c
```

```
it      output_pmc  pmc1.o  pmc.h  pmc.o    pmcTime.c  pmcTime.html  pmcTime.s
rdpmc.h  test     test.c
install  Makefile   pmc1.c  pmc.c  pmc.html  pmcTime   pmcTime.c~   pmcTime.o
rdpmc.c  rdpmc.o  testasm.c
it      output_pmc  pmc1.o  pmc.h  pmc.o    pmcTime.c  pmcTime.html  pmcTime.s
rdpmc.h  test     test.c

-- These are values of various performance metrics described above --
 Various Performance Metrics
==================================

 FLOPS:                  14
 Instrunctions Per Cycle:  0.559751
 MFLOPS:                 0.000002 x CPU Speed(MHz)
 L1 Cache Hit Rate:      0.985110
 L2 Cacle Hit Rate:      0.039523
 Memory/ Flops Ratio:    209974.071429
 Branch Rate:            0.198983
 Branch miss rate:       0.049764
 TLB Miss Rate           0.002687
```

6. Conclusion

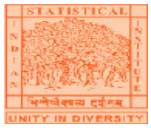
This study shows that various useful performance metrics can be derived from the performance monitoring counters present in most modern processor. It has also discussed an existing implementation as well as a proof-of-concept implementation to access the various counters under Linux Operating System and how performance metrics are calculated from the event counts obtained. The proof-of-concept implementation provided is very limited and so it can be extended in many ways including:-

1. Inclusion of a routine to determine the current CPU speed, so that it can be used along with the other event counts to give some precise performance metrics. (It involves reading /proc/cpuinfo and then parse the result to obtain CPU clock speed).

2. Porting it to the Solaris environment (As Solaris 10 will be open-sourced [according to sun] we look forward to make changes to the Solaris Kernel to implement per-process counters if possible).

3. Average of the values obtained by running the same program several times will give more accurate results.

4. Running various portions of the program separately and measuring their performance will lead to identify the bottlenecks.



These performance metrics helps us to identity the portion of the codes that are consuming more times and running more inefficiently than other part. This will help us to modify the codes in such a way to improve the performance.

7. References

1. Intel IA-32 Architecture Manuals, http://www.developer.intel.com

2. Source Code of "Perfctr" patch by Mikael Pattersson http://www.csd.uu.se/~mikpe/linux/perfctr

3. Understanding the Linux Kernel, 2nd Edition, Daniel P. Bovet, Marco Cesati, O'Reilly.