Temporal accuracy and modern high performance processors: A case study using Pentium Pro[*]

Krishnan K. Kailas Bao Trinh Ashok K. Agrawala
Institute for Advanced Computer Studies
Systems Design and Analysis Group
Department of Computer Science
University of Maryland, College Park
{*krish, bao, agrawala*}*@cs.umd.edu*

Technical Report

Abstract

Real-time systems must be able to ensure temporally determinate execution of real-time tasks at run-time. By *temporal accuracy*, we refer to the timing accuracy with which the execution of a task can be started at a predetermined time. Temporally determinate execution of tasks on modern high performance processors is becoming more and more difficult because of the techniques used by these processors to boost their average performance. This report describes the experiments we have conducted to measure the temporal accuracy that can be achieved with the Pentium Pro processor. We present the results of these experiments and analyze these results to highlight the limitations of temporally determinate execution of programs on modern high performance processor architectures.

[*]This work is supported in part by ARPA under contract N66001-95-C-8619 and USAF F30602-96-1-0329. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, U.S. Air Force or the U.S. Government.

1 Introduction

The ability to execute a set of instructions with predictable execution time is an important criteria for programs with hard real-time requirements. Hard real-time systems are characterized by the presence of stringent timing constraints on the computations carried out by the system. These timing constraints are often expressed as the start times and deadlines of tasks, as well as the temporal relationships among the tasks. The goal of a real-time scheduling scheme is to create a feasible schedule such that the timing requirements of all the tasks in the system are satisfied. The run-time system must be able to ensure, with high temporal accuracy, a temporally determinate execution of real-time tasks based on such a schedule. By *temporal accuracy*, we refer to the timing accuracy with which the execution of a task can be started at a predetermined time.

Clearly, when we consider such temporal determinacy at the application task level, operating systems introduce significant variability. Real-time operating systems typically give fast context switch time and priority based resource management. These are not sufficient for assuring temporal accuracy. Moreover, modern high performance processor architectures make use of a number of techniques to boost the average performance, such as multiple execution units, deep pipelines, dynamic branch prediction and speculative execution, register renaming, on-chip cache, etc. These techniques make the temporally determinate execution of tasks on such processors very difficult and thereby making the system less predictable[7, 8].

In this report, we present the results of experiments we have carried out to study the temporal accuracy of an off-the-shelf modern high performance processor. Specifically, our goal was to construct a run-time mechanism to achieve temporally accurate execution of programs on such processors. We choose the Intel Pentium Pro processor, one of the fast processors in extensive use today, for our experiments.

The rest of this report is organized as follows. In section 2 we provide a brief introduction to the microarchitecture of Pentium Pro processor. In section 3 we explain our experimental setup and the methodology adopted. In section 4 and 5 we present the results of the two sets of experiments we have conducted. We discuss the results of our experiments and how these results will be used for implementing the next release of Maruti hard real-time operating system in section 6. Finally, we conclude this report with some comments in section 7.

2 Pentium Pro Processor Microarchitecture

The Pentium Pro processor has a superscalar architecture with 12 stage pipeline. The processor has an instruction pool coupled with three independent units, viz. the Fetch/Decode unit, the Dispatch/Execute unit and the Retire unit as shown in Figure-1. A user program is executed by the Pentium Pro processor as follows (for a detailed description see [2, 5]). The user program instruction stream is fetched from the instruction cache and decoded into a series of micro-operations (μops) by the Fetch/Decode unit. Pre-fetching of instructions is speculative, based on a dynamic branch prediction scheme. The Dispatch/Execute unit speculatively executes the μops in the instruction pool based on the data dependencies and resource availability, and then temporarily stores the results. The Retire unit selects, in the original program order, the μops of the instructions that have completed execution for retirement and commits the results. Dynamic register renaming technique is used to facilitate out-of-order execution of μops. All the above techniques make it hard to predict the execution time of an instruction. Moreover, mispredicted branches, interrupts, breakpoints, traps and faults can cause some or all of the speculative state to be flushed by the processor [1], thereby adding more unpredictability.

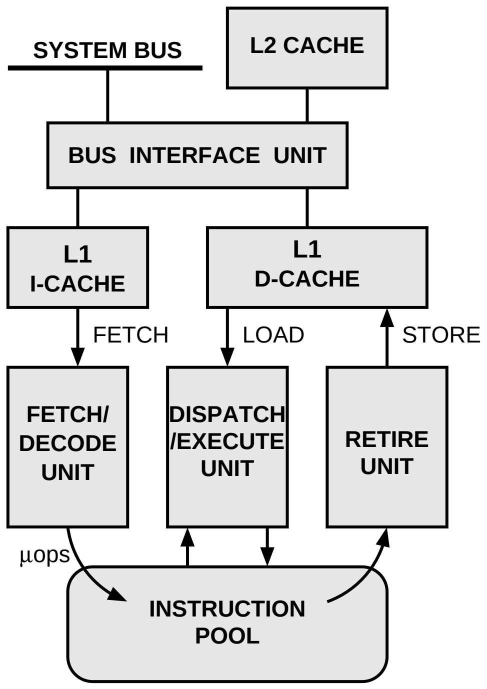

Figure 1: Pentium Pro schematic (adapted from Pentium Pro Family Developer's Manual [2])

The Pentium Pro architecture offers two interesting timing mechanisms – a pollable 64-bit time register called the Time Stamp Counter(TSC) and a 32-bit programmable timer. The TSC register is incremented at the processor's clock speed and can be accessed with either one of these two instruction, RDTSC (Read TSC) or RDMSR. The TSC can also be reset/preset by executing a WRMSR instruction. The programmable timer is a part of the on-chip interrupt controller called the Advanced Programmable Interrupt Controller (APIC). The time base for the APIC timer is derived from the processor's bus clock through a user programmable pre-scaler (divide configuration register). The timer can be programmed either in one-shot mode to generate an interrupt on terminal count, or in periodic mode to generate interrupts at regular intervals. Both of these timing mechanisms are also available on Pentium processor.

3 Experimental Setup and Methodology

In this section, we describe the hardware platform on which our experiments were carried out and the details of the experiments.

3.1 Experimental Setup

Our experiments were conducted on a 200 MHz Intel Pentium Pro processor based personal computer (PC). The PC we used has a SuperMicro P6DNE motherboard with the Intel 440FX chipset and 32 MB RAM.

The 200MHz Pentium Pro processor we used has separate 8KB non-blocking L1 caches for data and instruction, and a 256KB integrated non-blocking L2 cache. The experiments were conducted with the cache enabled as well as disabled to study the effect of variability introduced by cache hits and misses. In the latter case, accesses to the cache were prevented by disabling the on-chip cache, in addition to invalidating the cache, disabling the MTRR registers, and setting the default memory type to “uncached”.

Our test programs were run on the bare machine, which was re-booted before each run. A modified version of the Maruti kernel [6] was used to gain exclusive access to the processor and run our test code. There was absolutely no operating system or any other program running concurrently when the test programs were executed. All the hardware interrupts were also disabled during the experiments. Only the APIC timer was enabled in the set of experiments using the APIC timer.

3.2 Methodology

The basic methodology we used was to write programs that attempt to achieve some pre-defined timing target and use the processor's own TSC to measure how close we got to that target. In general, the goal is to be able to specify the time when an event is to take place, execute a program to signal the arrival of that time, and then perform that event. In our experiments, the first instruction of the event is a RDTSC instruction and its result is compared to the target time. The temporal accuracy is given by the difference between the time measured by RDTSC instruction and the target time. We studied the variability in the temporal accuracy because if the difference is a constant, it can be used to make adjustments to achieve the desired target value. We have used two approaches to calculate the temporal accuracy that can be achieved on Pentium Pro – one making use of TSC, and the other making use of both the TSC and the APIC timer.

4 Time Stamp Counter based experiments

This is a straightforward approach to measure the temporal accuracy that can be achieved on Pentium Pro processor. In this set of experiments, we calculate the target time by adding an integer offset value to the current value of the TSC and then wait in a tight loop reading the TSC repeatedly and comparing it with the target time. When the specified target time arrives, the loop falls through, and the first instruction executed after exiting the loop is a RDTSC instruction as shown in Figure-2. The TSC value read by second RDTSC instruction is used for computing the temporal accuracy as explained above. Instead of adding the same constant offset to current TSC value, we have varied the offset over a wide range of values in the actual experiments to study the effect of variations in target time on temporal accuracy. Each set of experiments was repeated many times to assure repeatability.

Figure-3 shows a portion of the data collected during the experiments with cache enabled. It was observed that the variation of temporal accuracy obtained for increasing target TSC interval follows a regular repetitive pattern with a periodicity of 32 clock cycles[1]. Figure-4 shows the variation in temporal accuracy when the same experiment was repeated with cache disabled. It is interesting to note the variability that is introduced when the cache is disabled. The range of values for temporal accuracy also changes from 32 in Figure-3 to 256 for Figure-4.

The cyclic variation in temporal accuracy with increasing target TSC value can be explained as follows. Two consecutive TSC values that can be read using RDTSC instruction and compared with the target TSC value in the `TSC_loop` is at least 31 to 33 clock cycles apart. This is because of the execution time of the RDTSC instruction and other instructions in the `TSC_loop`. Therefore, unless the target TSC value is exactly a multiple of 31 to 33 clock cycles, the loop will take a number of additional clock cycles (< 33 cyles) before exiting. The periodicity of the cyclic variation in temporal accuracy was found to increase when a more complex RDMSR instruction is used to read

[1]In some experiments we have observed a periodicity of 31 or 33 clock cycles.

```
     movl %ebx,(_TSC_target_hi)
     movl %ecx,(_TSC_target_lo)

TSC_loop:
     rdtsc
     subl %ecx,%eax
     sbbl %ebx,%edx
     cmpl $0,%eax
     jl   TSC_loop

     rdtsc
     movl %edx,(_tsc_final_hi)
     movl %eax,(_tsc_final_lo)
```

Figure 2: TSC loop code

TSC in lieu of RDTSC as shown in figure-5. This further confirms our explanation for the cyclic variation pattern in temporal accuracy with increasing target TSC value.

The alignment of code inside the cache was also found to influence the variations in temporal accuracy significantly [3]. Figure-6 clearly shows the increase in variations when the `TSC_loop` was not aligned to 32 byte boundary compared to aligned code[2] used in the above experiments.

In an effort to determine the best case temporal accuracy values that can be achieved, we added another loop before the `TSC_loop` to introduce an additional delay. The idea was to make the RDTSC instruction execute exactly at a multiple of 31 to 33 clock cycles ahead of the target TSC value by delaying the entry to the `TSC_loop`. We found that the variation in temporal accuracy can be reduced if a delay look-up table (indexed by *target TSC value* % 32) is used for introducing appropriate delay. Figures-7 and 8 clearly show the variations in temporal accuracy without and with the delay loop respectively. We could achieve a temporal accuracy of 2 clock cycles in the best case as shown in Figure-8. with a hand-tuned look-up table and with both the loops aligned to 32 byte boundaries. However, we found that this technique is highly sensitive to even minor modifications in code and alignment of code and data in memory.

5 APIC Timer based experiments

The APIC timer interrupt can be used to schedule an event in the future at a predefined time interval from the present time or at regular intervals in a periodic manner. On our test machine, the highest time base that can be derived from the 200MHz bus clock for the APIC timer is 66MHz, without any pre-scaling. Even though the resolution of the APIC timer is not as good as the TSC counter, an interruptible timer may be used in several ways in a real-time operating system. The objective of this experiment was to measure the variations in the APIC timer interrupt latency due to the non-deterministic nature of the processor state at the time of interrupt. The APIC timer was set up in the periodic mode to generate interrupts at 100 μs and 15 μs intervals in this set of experiments. In the interrupt handler, after the saving of a few registers by `PUSH`ing them to

[2]For code alignment the `.align` assembler directive was used with NOP instructions as “pading bytes”.

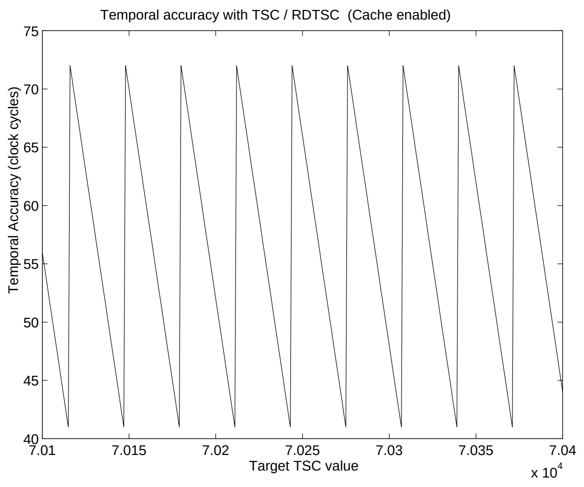

Figure 3: Temporal Accuracy measurements using RDTSC instruction with cache enabled

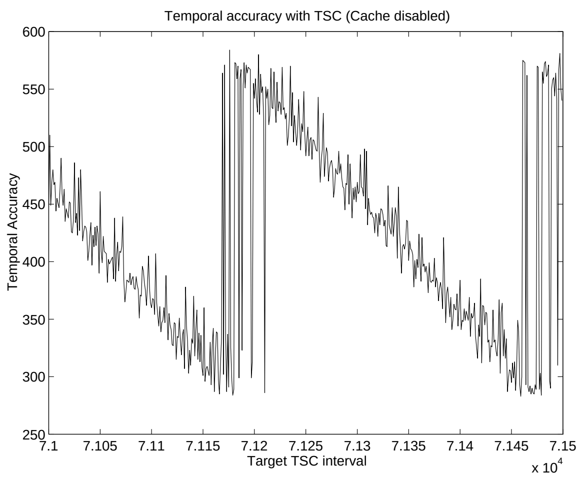

Figure 4: Temporal Accuracy measurements with cache disabled

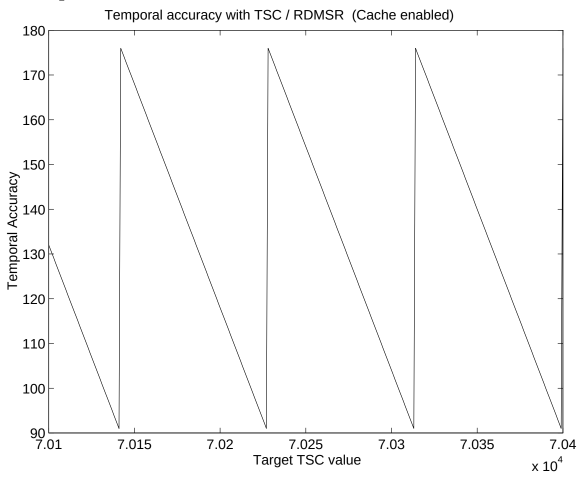

Figure 5: Temporal Accuracy measurements using RDMSR instruction to read TSC with cache enabled

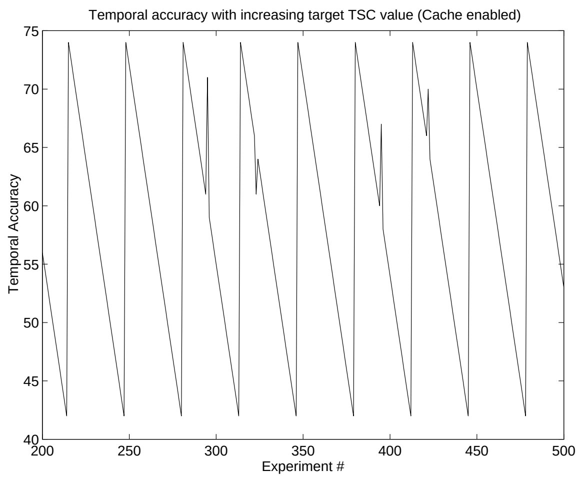

Figure 6: Temporal Accuracy measurements with cache enabled and unaligned code

stack, the first instruction executed is a RDTSC to record the start time of the interrupt handler, as shown below.

```
apic_timer:      # APIC timer interrupt handler
   - save registers (pushl)
   rdtsc
   - save the TSC value
   - restore registers (popl)
   iret
```

The program running in the background consists of a tight-loop. In order to understand the effect of background processing on the APIC timer interrupt latency, we have conducted each experiment with either a NOP or a CPUID instruction in the loop. The CPUID instruction is a *serializing instruction* which will ensure that all modifications to flags, registers, and memory by previous instructions are completed and all buffered writes have drained to memory before the next

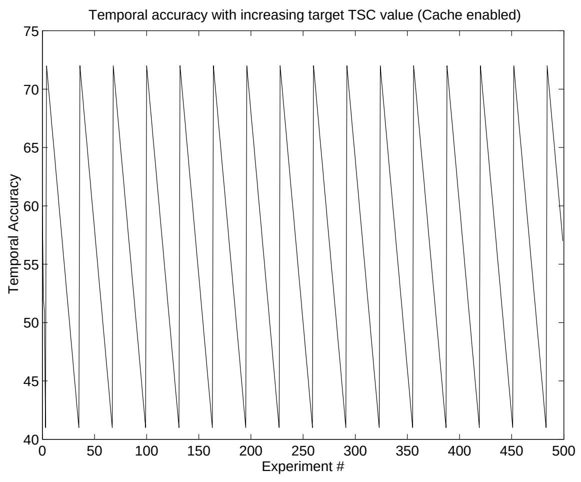

Figure 7: Temporal Accuracy for increasing Target TSC values without delay loop

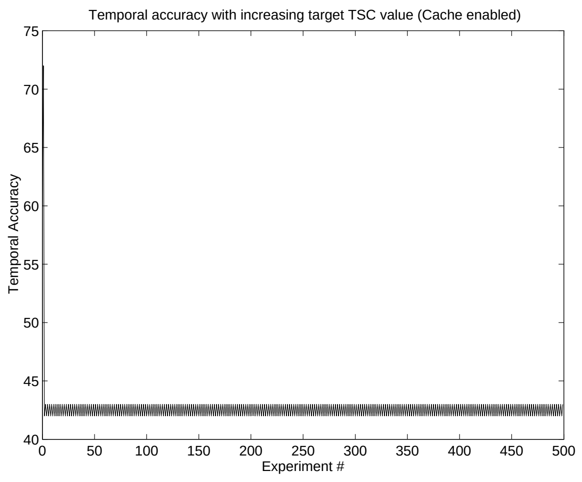

Figure 8: Temporal Accuracy for increasing Target TSC values with lookup table based delay loop

instruction is fetched and executed. The idea of executing a serializing instruction in the loop is to generate the worst case scenario for the APIC timer interrupt handler. On the other hand, the NOP instruction in the loop provides a more or less static environment corresponding to the best case scenario. The result of these experiments are shown in figures-9 and 10. The plot shows the variations in the interrupt latency as an error (measured at the processor clock cycle resolution) from the estimated TSC value for a set of consecutive APIC timer interrupts. The 11 clock cycle variation were observed in temporal accuracy with a periodicity of 512 clock cycles as shown in Figure-10. We were able to confirm that this variation was due to the D-cache miss caused while storing the TSC values in a large array inside the interrupt handler by modifying the interrupt handler to store only the erraneous values.

The experiments were also conducted with the cache disabled. The results of the experiments with cache disabled are shown in figures-11 and 12. A large variation in interrupt latency due to the variability in memory access time was observed in these experiments with the cache disabled. The effect of the instructions executed in the background at the time of interrupt is therefore hard to identify from the experimental results.

6 Discussion

We have used the on-chip Performance Monitoring counters [2] to investigate the reason behind the periodic pattern found in the experimental results. We have used these counters to monitor events such as number of clocks during which DRDY is asserted, number of clocks during which LOCK is asserted, number of mispredicted branches retired, and number of cycles during which there are resource related stalls. The results of using the performance monitoring counters in the set of experiments mentioned above did not give us any insight into the cause of the problem. Moreover, the RDPMC instruction used for reading the performance monitoring counters(PMC) is not a serializing instruction, and hence is not ordered with other instructions [2]. Thus, there is an element of uncertainty on the instant at which the measurements were taken, because it is quite possible for the RDPMC instruction to read the PMCs before or after the event(s) we were trying to

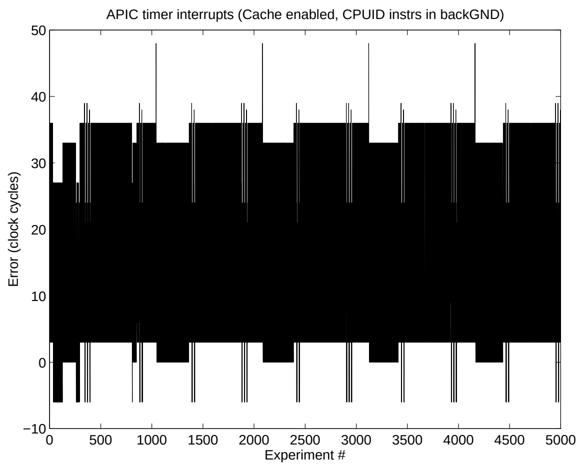

Figure 9: Temporal Accuracy measurements with cache enabled and CPUID instruction in the background loop

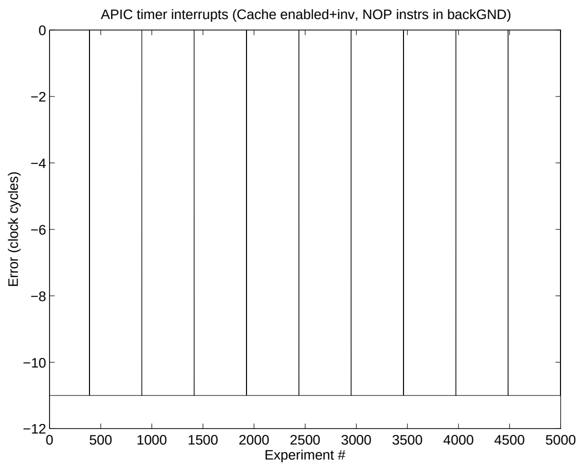

Figure 10: Temporal Accuracy measurements with cache enabled and NOP instruction in the background loop

monitor. The RDTSC instruction also has the same problem. We have also used the performance counters to verify that there were no hardware interrupts during the TSC experiments.

In the Time Stamp Counter based experiments, we have observed that the instructions executed before entering the wait loop can have a tremendous impact on the accuracy with which the loop exits. We were not able to clearly understand this relationship though we have observed some intermittant 100-200 clock cycles long variations in the execution time of small segements of code executed before the `TSC_loop`.

In the APIC timer based experiments, the measured APIC timer interrupt latency is found to vary by a considerable amount, especially when the cache is disabled. The important observation we made is the effect of cache and background process on the interrupt latency. We found that when the cache was enabled, the variations in interrupt latency is significantly reduced to some intermittent spikes. We were able to eliminate these spikes (see figure-10) when no memory access is made from the interrupt handler. However, an attempt to isolate the RDTSC instruction and the memory access instruction by inserting a large number of other instructions inside the interrupt handler in order to eliminate these spikes was not successful. Moreover, if the working set of the background process is static and small, it is possible for the interrupt handler to be always resident in the cache thereby avoiding I-cache misses on interrupt. Similar results were reported by Koopman [4] based on experiments with Intel 80486 cache.

The experiments we have conducted on Pentium Pro processor clearly show that, even in a static environment, it is very hard to achieve a temporal accuracy better than a few tens of clock cycles on a modern superscalar, out-of-order execution processor designed for high average performance. Special conditions have to be set to achieve higher temporal accuracies.

6.1 Maruti run-time system

We have examined how the results of this study can be made use of for improving the temporal accuracy of next release of the Maruti run-time system. Maruti[6] is a hard real-time operating system which manages resources in time base. In Maruti, resources are explicitly reserved prior to execution and the tasks are executed based on a pre-determined schedule called *calendar*. This

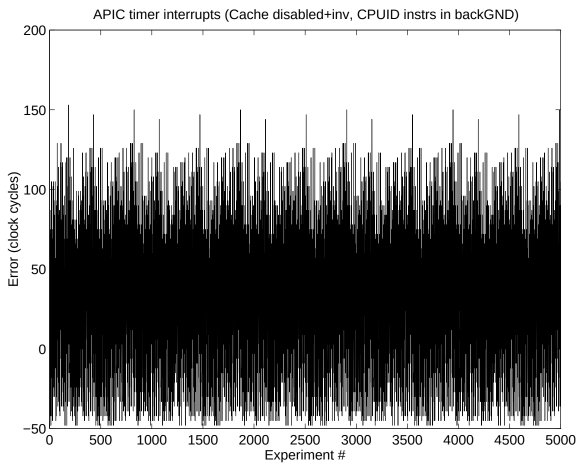

Figure 11: Temporal Accuracy measurements with cache disabled and CPUID instruction in the background loop

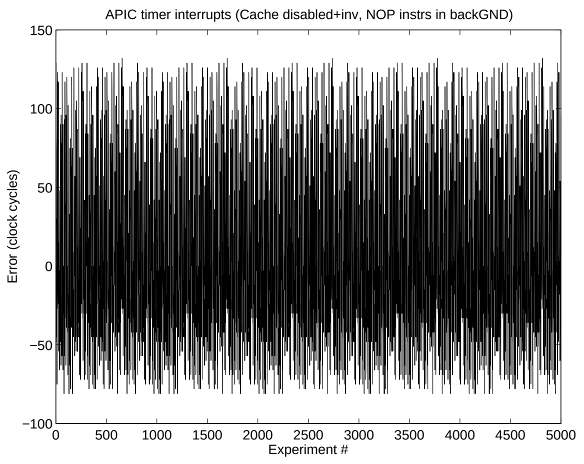

Figure 12: Temporal Accuracy measurements with cache disabled and NOP instruction in the background loop

approach can guarantee the stringent timing requirements of hard real-time applications.

A Maruti application is made up of one or more modules, each of which consists of one or more entries, services, and functions. A module maps at run-time to a Maruti task, which contains multiple execution threads that correspond to the task's entries, services, and functions. Each thread is further divided into a sequence of non pre-emptable scheduling entities called *elemental units*, or EU's. Maruti can deliver high degree of temporal determinacy to the application as long as the underlying timing mechanism is accurate. In order to assure a high degree of temporal accuracy, Maruti has to be able to start the execution of an EU at a predetermined time. In the current implementation of Maruti, a pollable time register is used at run-time to precisely dispatch the threads as specified in the calendar. Clearly, the temporal accuracy of such a system depends on the granularity and the accuracy of the pollable time register and the mechanism to access the register.

Our goal is to construct a mechanism, making use of off-the-shelf modern high performance processors, that can yield a high degree of temporal accuracy that can be guaranteed by Maruti. Based on this study, we have decided to use the TSC for maintaining an accurate system clock of nano seconds resolution. Two new system calls, `readtsc` and `mdelay` are being implemented to read the TSC and to wait for a particular TSC value respectively.

A Maruti thread relinquishes control to the run-time system either voluntarily via an EU break or involuntarily through a time-out interrupt. The new Maruti run-time system will make use of APIC timer for implementing the time-out mechanism. The scheduler will select the next EU for dispatching and compute it's ready time and deadline. The dispatching operation can be implemented using either the `mdelay` function call or the APIC timer interrupt. The APIC timer might be able to provide better temporal accuracy than the `mdelay` if the timer interrupt handler code is in I-cache and all the data structures used by the interrupt handler is in D-cache. We are planning to evaluate the performance of both approaches on actual implementation in order to select the best approach. We expect that in Maruti environment we will achieve a temporal accuracy of less than 100 nanoseconds with a latency of at most a few microseconds.

7 Comments and Conclusion

In this report we have described the experiments we have conducted to measure the temporal accuracy that can be achieved on a Pentium Pro processor and the results of these experiments. Due to the out-of-order and dynamic nature of the execution of instructions, there is an inherent unpredictability in (1) the instruction execution ordering and (2) the delays in the instruction issue, execution and retirement phases of an instruction. The measurement techniques we have used in this study make use of instructions to read TSC and Performance Monitoring counters on such an environment, and these instructions are executed among other instructions by the processor in a similar manner. Hence, our measurement results are not only imprecise but also have a certain amount of uncertainty that can not be eliminated. This fact clearly shows the limitations of instruction level instrumentation on Pentium Pro processor, and in general the limitations on temporally determinate execution of programs on such processor architectures. Despite these limitations, we expect to achieve an application level temoporal accuracy of about 100 nanoseconds when we integrate TSC and APIC based timer interrupts into the Maruti operating system.

Acknowledgment

Authors would like to thank Frank Binns, Architecture group, Intel Corporation, OR for the comments and suggestions and Jan Morales for participation in the initial stages of this study. Pentium, Pentium Pro and Intel 80486 are registered trademarks of Intel Corporation.

References

[1] Intel Pentium Pro Processor Presentation. Web page URL: http://www.intel.com/procs /ppro/info/isscc/index.htm. accessed on Feb 05 16:34:46 EST 1997.

[2] *Pentium Pro Family Developer's Manual*, volume 1-3. Intel Corporation, Mt. Prospect, IL, 1996.

[3] *Optimizations For Intel's 32-Bit Processors.* AP-526 Application note. Intel Corporation, Mt. Prospect, IL, October 1995.

[4] P. Koopman. Perils of the PC Cache. *Embedded Systems Programming*, 6(5):26–34, May 1993.

[5] Robert P. Colwell and Randy L. Steck. A 0.6um BiCMOS Processor with Dynamic Execution. In *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC).* IEEE, February 1995.

[6] M. Saksena, J. da Silva, and Ashok K. Agrawala. Design and Implementation of Maruti-II. In Sang H. Son, editor, *Principles of Real-Time Systems.* Prentice Hall, Englewood Cliffs, N.J., 1995. Also available as University of Maryland CS Tech Report CS-TR-3181.

[7] John A. Stankovic and Krithi Ramamritham. Editorial: What is Predictability in Real-Time Systems? *The Journal of Real-Time Systems*, 2:247–254, 1990.

[8] Clyde E. Taylor and Ronald N. Schroeder. Are RISCs ready for real-time control? *InTech*, 43(12):45–48, December 1996.