CPU Shielding:
Investigating Real-Time Guarantees via Resource Partitioning

Progress Report 1

John Scott Tillman
jstillma@ncsu.edu

CSC714 Real-Time Computer Systems
North Carolina State University
Instructor: Dr. Frank Mueller

Project URL:
http://www4.ncsu.edu/~jstillma/csc714/

OVERVIEW

This project seeks to investigate the feasibility and limitations of using *CPU shielding* to allow hard real-time operation in commercial, off-the-shelf (COTS) systems. This is being accomplished by theoretically bounding and experimentally verifying worst case interrupt response times (CPU contention), worst case bus reaction times (bus contention), and worst case slowdown associated with additional cache misses (cache contention). The goal is to verify the models/predictions and evaluate the predictability that can be achieved using this co-hosting method.

TIMELINE

Completed	March 16th	Project proposal submitted
Completed	March 22nd	Replicated system setup from [1]. Evaluate literature predictions of interference from known sources (PCI, Interrupt and Cache induced)
85%	March 29th	Designed experiments to test latency from known sources
Completed	April 2nd	Initial Project Status Report
0%	April 5th	Gather and evaluate initial test results. Identify and categorize unknown latency factors.
0%	April 19th	Demonstrate mixed (real-time and non-real-time) mode operation at predicted highest frequency.
0%	April 21st	Final Project Status Report

TEST SYSTEM DETAILS

The system being tested contains an AMD Athlon x2 +3800 processor. This dual core processor contains a 512K L2 cache *per core* [6]. This limits cache contention effects under CPU shielding, but should still exhibit FSB contention. This property places this contention in the same category as all other external device DMA and can be considered using the techniques from [4].

The PCI bus in the test system contains a variety of devices typical of a commodity hardware configuration such as:

Device	Bus ID	Latency Timer
ATI Technologies Inc RS480 Host Bridge (rev 10)	00:00.0	0x00
ATI Technologies Inc RS480 PCI-X Root Port	00:02.0	0x01

ATI Technologies Inc RS480 PCI Bridge	00:05.0	0x01
ATI Technologies Inc 4379 Serial ATA Controller	00:12.0	0x00
ATI Technologies Inc IXP SB400 USB Host Controller	00:13.0	0x80
ATI Technologies Inc Std. Dual Channel PCI IDE Controller	00:14.0	0x00
ATI Technologies Inc IXP SB400 PCI-PCI Bridge	00:14.4	0x81

This is not an exhaustive list. The *Latency Timer* in the PCI configuration space is “a mechanism to constrain a master’s tenure on the bus” [7]. Large latency timer values imply that the given device can take control of the PCI bus for longer times during master (usually DMA) transfers. Since the latency timer is writable it should be possible, at a loss of efficiency, to minimize the effects of PCI bus contention. Since this value is readable (as part of the standard PCI configuration space) it should provide the means to determine an upper bound to the worst case delay.

WORST CASE PCI CONTENTION

According the the PCI 2.1 specification [7] the maximum latency of the bus is:

latency_max(clocks) = 32 + 8 * (n – 1)   (n is the # of data transfers)

The latency timer defines the number of bus clocks beyond which no further data transfers can begin. The latency timer is only considered when there is bus contention.

The other aspect of PCI bus latency has to do with bus access arbitration. Given the requirement of a “fair” arbitration algorithm it is very likely that a bus with N masters (including the CPU) could potentially wait latency_max * (N-1) bus cycles before being granted read or write access to the PCI bus.

There are 17 devices visible on the test platform’s PCI bus. I have not yet established which might perform master transfers, but it is reasonable to assume all may have the capability. Calculating the exact worst case latency yields:

```
00:00.0, 00 -> 32          latency_max(1) =    32
00:02.0, 01 -> 32          latency_max(2) =    40
00:05.0, 01 -> 32          latency_max(3) =    48
00:12.0, 00 -> 32          latency_max(4) =    56
00:13.0, 80 -> 80          latency_max(5) =    64
00:13.1, 00 -> 32          latency_max(6) =    72
00:13.2, 00 -> 32          latency_max(7) =    80
00:14.0, 80 -> 80          latency_max(8) =    88
00:14.1, 00 -> 32
00:14.3, 80 -> 80
00:14.4, 81 -> 88
00:14.5, 80 -> 80
```

```
00:18.0, 80 -> 80
00:18.1, 80 -> 80
00:18.2, 80 -> 80
00:18.3, 80 -> 80
01:00.0, 00 -> 32
02:00.0, 00 -> 32
03:0b.0, 80 -> 80
03:0b.1, 00 -> 32
03:0b.2, 00 -> 32
```

This gives a total latency count of 1160 bus cycles. At 33 MHz the worst case bus access latency is 35.2 microseconds. Lowering all latency maximum numbers to 0 yields 544 cycles (16.5 microseconds). The probability of every device on the PCI bus requesting its maximum allotment simultaneously is extremely small. This may prevent latency measurements within even an order of magnitude of the worst case.

PCI contention will be measured using the CPU's TSC register. The unshielded CPU will be asked to run a series of stress tests. The shielded CPU will perform timed reads from a single register on a PCI peripheral (the parallel port hardware).

CACHE CONTENTION

Our CPU's non-shared cache should prevent cache access interference between the two cores. To verify this, a simple test will be performed. The non shielded CPU will perform sequential reads to bytes of memory to guarantee maximized cache misses. The shielded CPU will wait (a random time interval) and read from a memory location (theoretically) guaranteed to be cached. The CPU's TSC register will verify the memory access did not miss.

MEMORY (FSB) CONTENTION

Almost all strategies for bounding cache miss effects on WCET rely on a structural knowledge of the code and data access patterns inherent in the code being evaluated. Given the nature and scale of this project it isn't feasible to attempt a general solution to this problem. Also there is nothing specifically unique to CPU shielding in these calculations. Given that our platform has separate L1 and L2 caches per core, and that the purpose is to dedicate one core to real-time processing, we will bring all real time data and instructions into cache and simply keep it there. Were this not the case the WCET calculation would still be calculable by using any of the variety of methods in the current literature (see [8] for a survey of methods).

TEST SYSTEM SETUP

The system setup from [1] is a somewhat standard Linux kernel. While that report was focused on the 2.4 line of kernel development, almost all of the system setup is the same. A variety of CPU affinity control methods exist, but for our purposes the system call *set_cpus_allowed* is ideal.

One limitation of this method lies in the fact that an interrupt must occur before its interrupt service routine (in Linux) can be migrated across CPUs. Solving this requires the kernel be aware of the shielding requirements from the initial bootstrap time. This is the correct solution for longer term less controlled environments, but for our environment this should be unneeded. This assumption must be verified by comparing interrupt dispatches for the shielded CPU before and after our tests. This comparison will be performed by our test suite.

The *Realfeel* timing utility (used by [1]) uses the standard Real-Time Clock interface and the processor's TSC (Time Stamp Counter) to calculate cycle accurate response times. The existing implementation of *Realfeel* exhibits unexpected behavior when used on the test platform. The reason for this has not yet been established.

RedHat provides a number of stress testing utilities that have been designed specifically to exert pressure on various systems within our shielded platform:

TTCP	Loopback test, high memory
FIFOS_MMAP	Alternate IPC using FIFOs and MMAP
P3_FPU	Floating point matrix operations
FS	A mix of major file system operations
CRASHME	Jump to execute in random memory locations
NFS_COMPILE	Kernel rebuild over a loopback mounted NFS device

These six stress tests will form the basis of the workload placed onto our unshielded CPU. The described measurements will be performed under each load condition to verify the predictability of the Shielded CPU.

WEBSITE

http://www4.ncsu.edu/~jstillma/csc714/

REFERENCE WORKS

[1] S. Brosky. *Shielded CPUs: real-time performance in standard Linux*. Linux Journal, May 2004, pg 121

[2] M. Caccamo. *Toward the Predictable Integration of Real-Time COTS Based Systems*.

[3] R. Pellizzoni, B. Bui, M. Caccamo, L. Sha. *Coscheduling of CPU and I/O Transactions in COTS-based Embedded Systems*. Real Time Systems Symposium, 2008

[4] T. Huang, J. Lui, J. Chung. *Allowing cycle-stealing direct memory access I/O concurrent with hard-real-time programs*. International Conference on Parallel and Distributed Systems, Tokyo, 1996.

[5] S. Schönberg. *Impact of PCI-bus load on applications in a PC architecture*. Proceedings of the 24th IEEE international Real-Time Systems Symposium, Cancun, Mexico, December 2003

[6] *AMD Athlon 64 Processor Product Data Sheet, Revision 3.18*, Semptember 2006

[7] *PCI Local Bus Specification, Revision 2.1*, June 1, 1995

[8] R. Wilhelm, J. Engblohm, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, *The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools*, ACM Transactions on Embedded Computing Systems, Vol. 7, No. 3, Apr 2008, pages 1-53.