CPU Shielding:
Investigating Real-Time Guarantees via Resource Partitioning

Progress Report 2

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 is investigating the feasibility and limitations of using *CPU shielding* to allow hard real-time operation in commercial, off-the-shelf (COTS) systems. 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.

TEST SYSTEM SETUP

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 test system uses a standard Linux kernel (2.6.25). A variety of CPU affinity control methods exist for processes, but for our purposes the system call *sched_setaffinity* is ideal (see also cpu sets). A utility was written which moves all existing processes away from the second CPU. The *shield* utility does not continuously monitor these affinity masks, but it will restore them when it is exited. There is no documented way to monitor for new process creation in linux, however, new processes will inherit the affinity mask of their parent process (they can, themselves call sched_setaffinity to override it though). Most processes remain unaware of their affinity mask, but there are a few utilities in Linux which spawn helper tasks for each processor. Standard processes can be started directly on a given processor (or set of processors) using the *taskset* command line utility (see man pages for details).

Interrupt vectors can be assigned an affinity mask as well. These masks are available through their respective entries in the ‘proc’ filesystem (/proc/irq/#). 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. Reliably 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 is available in our shielding helpers.

Another point of concern regarding interrupts are the various per-processor IRQs. These interrupts cannot be moved to another processor. The only solution for this is either disabling interrupts (which we do in some of our tests) or rewriting the kernel to allow the disabling of these vectors (local timer,

rescheduling, TLB helper, etc). The presence of these timers makes a realistic test case difficult to create. Most of our tests are performed with all possible tasks and interrupts migrated to the unshielded CPU (0) and interrupts disabled on the shielded CPU. This scenario is stable and workable under our very restricted conditions. Because of this the majority of our test utilities must be run with root privileges, since the interrupt disable instructions are privileged.

RESULTS

Cache Contention

Our CPU's non-shared cache should prevent cache access interference between the two cores. To attempt to verify this, a simple test was be performed. The non shielded CPU performs sequential reads to bytes of memory guaranteed to maximize cache misses. The shielded CPU continuously reads from a memory location (theoretically) guaranteed to be cached. The CPU's TSC register was used to verify the memory access did not miss.

<ticks vs. hit count>
5 298434689
6 1644
7 959
8 298433425
9 746
10 1758
13 14
14 24
15 55
16 41
17 9
<graph histogram of timings for shielded core when unshielded is exercised>

<ticks vs. hit count>
5 188649016
6 1070
7 516
8 188647306
9 350
10 1182
13 23
14 52
15 115
16 111
17 11
<graph histogram of timings for shielded core when unshielded is not exercised>

Running this test yields some unexpected results. The preponderance of cycle times fall at 5 and 8 cycles, and represent L1 and L2 memory access times (timings are not adjusted for the additional rdtsc instruction timing). The times at 6, 7, 9 and 10 cycles may represent timing anomalies in the instruction execution (and/or possibly the TSC counter).

Times up to 10 cycles show a very consistent distribution, whereas the distribution at higher cycle times are impacted by shielding and processor loading. One interesting point is that exercising the unshielded CPU actually *decreased* the incidence of cache misses on the shielded core. This result is unexpected and has not yet been explained.

In any case, exercising the cache of the unshielded core does not measurably decrease the performance of the shielded core. This is the expected result, though the unexplained variances in the timings observed leave room to question this result.

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 bring all real time data and instructions into cache and simply lock 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).

PCI Bus Contention

The PCI bus in the test system contains a variety of devices typical of a commodity hardware configuration. The PCI bus is required to use a 'fair' method for multiplexing the bus amongst the various installed devices. This means that the worst case latency must be the sum of each device's longest bus access time.

The *Latency Timer* in the PCI configuration space is the "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 provides 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 possible 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 and it is reasonable to assume all may have the ability to act as bus masters. Calculating the exact worst case latency yields:

```
00:00.0, 00 -> 32
00:02.0, 01 -> 32
00:05.0, 01 -> 32
00:12.0, 00 -> 32
00:13.0, 80 -> 80
00:13.1, 00 -> 32
00:13.2, 00 -> 32
00:14.0, 80 -> 80
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
```

```
latency_max(1) =    32
latency_max(2) =    40
latency_max(3) =    48
latency_max(4) =    56
latency_max(5) =    64
latency_max(6) =    72
latency_max(7) =    80
latency_max(8) =    88
```

This gives a total latency count of 1160 bus cycles. At 33 MHz the worst case bus access latency is 35.2 microseconds. This does not account for additional latencies which may be generated due to bus arbitration. 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.

PCI contention was measured using the CPU's TSC register. The unshielded CPU was asked to run a series of stress tests while the shielded CPU performed timed writes to a single register on a PCI peripheral (the parallel port hardware).

<usec vs % times recorded>
1.8, 1.094842
1.9, 72.150420
2.0, 11.362581
2.1, 0.563029
2.2, 0.008082
2.3, 0.011968
2.4, 0.045099
2.5, 0.005072
2.6, 0.049861
2.7, 0.375097
2.8, 0.490162
2.9, 0.021546
3.0, 1.147367
3.1, 3.186867
3.2, 1.914145
3.3, 2.161552
3.4, 3.293120
3.5, 1.581903
3.6, 0.417095
3.7, 0.019558
3.8, 0.002737
3.9, 0.002901
4.0, 0.002408
4.1, 0.002864
4.2, 0.002773
4.3, 0.002372
4.4, 0.002426
4.5, 0.001897
4.6, 0.001788
4.7, 0.001770
4.8, 0.002153
4.9, 0.001806
5.0, 0.001478
5.1, 0.001460
5.2, 0.001569
5.3, 0.001423
5.4, 0.001532
5.5, 0.001569
5.6, 0.001861
5.7, 0.002080
5.8, 0.001532

5.9, 0.001569
6.0, 0.001660
6.1, 0.001733
6.2, 0.001788
6.3, 0.001861
6.4, 0.001916
6.5, 0.002408
6.6, 0.001806
6.7, 0.002317
6.8, 0.001934
6.9, 0.001970
7.0, 0.001715
7.1, 0.002135
7.2, 0.002226
7.3, 0.002007
7.4, 0.003029
7.5, 0.001916
7.6, 0.001514
7.7, 0.001715
7.8, 0.001569
7.9, 0.001843
8.0, 0.001751
8.1, 0.001569
8.2, 0.002153
8.3, 0.001076
8.4, 0.001168
8.5, 0.001368
8.6, 0.001204
8.7, 0.001113
8.8, 0.000876
8.9, 0.000876
9.0, 0.001295
9.1, 0.001423
9.2, 0.000839
9.3, 0.000474
9.4, 0.000128
9.5, 0.000055
9.6, 0.000073
9.7, 0.000109
9.8, 0.000018
10.0, 0.000018
48.9, 0.000018
<graph of measurements of PCI with extra bus contention>

<usec vs. % times recorded>

1.8, 1.248036
1.9, 82.799835
2.0, 12.789639
2.1, 0.619157
2.2, 0.000737
2.3, 0.009682
2.4, 0.040753
2.5, 0.001904
2.6, 0.052072
2.7, 0.310100
2.8, 0.391054
2.9, 0.007348
3.0, 0.023416
3.1, 0.062736
3.2, 0.058070
3.3, 0.556666
3.4, 0.843514
3.5, 0.174987
3.6, 0.009825
3.7, 0.000164
3.8, 0.000020
9.4, 0.000041
9.5, 0.000020
9.6, 0.000082
9.7, 0.000082
9.8, 0.000061
<graph of measurements without extra contention>

The measurements taken while the system was unstressed are reasonable normal cases. The PCI bus is typically under utilized, but the occasional DMA (hard drive, video card, ethernet) will hold the bus a longer period of time. Note that 3.8us is about 125 bus cycles. When 'idle' some DMA processes still occur. these 125 cycles could easily be attributed to external devices. The distinct jump from 3.8us to 9.5us is harder to explain since no one device should be able to lock the bus for this additional time period (5.7us). Given this and the consistent lack of any measurements between 3.8 and 9.4us we must consider how this additional latency is being introduced.

It is important to realize that the dual core processor in use has only a single PCI bus presence. In looking at the given numbers this provides a reasonable explanation for the sudden jump. Imagine a scenario where both cores requested the PCI bus simultaneously. Hardware internal to the chipset will multiplex the requests, potentially granting to the unshielded processor first. The fair nature of the PCI bus arbitrator will then:

1) Complete any previously pending PCI bus requests

2) Schedule the unshielded CPU's transaction
3) Complete any newly pending PCI requests
4) Schedule the shielded CPU's transaction

This doubling of the worst case latency was not accounted for in our earlier calculation. Verifying this as the cause of this discontinuity requires a PCI bus protocol analyzer.

If we assume this is the cause of the discontinuity We must extend our worst case calculation from above from 1160 bus cycles, to 2320 plus the CPU's worst case bus cycle latency. I can find no documentation of the CPU's latency value, but it should not exceed that used by the PCI bridge, 80. This would push our worst case bus timing to 2400 cycles or 72.7us.

This extended worst case timing helps explain (very) occasional timings from the stressed executions which measure upwards of 50us, well beyond our single core worst case calculation. Still it is surprising that numbers even 2/3 of the worst case would be experienced.

CHALLENGES

RDTSC Instruction

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* exhibited unexpected behavior when used on the test platform. The cause of this appears to be a bug in the GCC compiler which improperly compiles the rdtsc instruction:

```
__asm__ __volatile__ ("rdtsc" : "=A" (TSC))
```

This should result in the 64 bit value edx:eax to be stored into TSC. However when compiled only the low 32 bits are stored. A simple work around, which manually reconstructs the whole TCS value was used. Once inlined the additional code was optimized away.

```
inline u64 rdtsc() {
   u32 TSC_low, TSC_high;
   u64 TSC;
   __asm__ __volatile__ ("rdtsc" : "=a" (TSC_low), "=d" (TSC_high));
   TSC = TSC_high;
   TSC <<= 32;
   TSC += TSC_low;
   return TSC;
}
```

IRQ Affinity

The IRQ affinity method described above does not currently work in the x86_64 platform in Linux. This is because no system has been put in place to allow requested IRQ affinities to be cached until the interrupt is pending. Our test system automatically directs (almost) all interrupts to CPU0. It obviously cannot redirect the CPU specific interrupts. Since the interrupt mask is not preventing execution on our shielded processor we must rely on the natural affinity of the IRQs for CPU0. The *shield* utility will display the interrupt counts for each system IRQ after the shield is established and again before it is taken down.

This problem also prevents specific interrupts from being moved to the shielded CPU set. The real-time clock interrupt is implemented as a low latency real-time notification system for user-space applications. However, since the inability to move its service exclusively to the shielded CPU set limits its usefulness for our purposes.

CONCLUSIONS

CPU shielding with the standard Linux kernel is possible, but providing hard response time guarantees is not. There are three reasons for this. First the inability to restrict processes' access to individual CPU's makes it possible to create a shielded CPU, but nearly impossible to keep it shielded. This problem is solvable with some relatively minor kernel additions. The second problem is the fact that not all platforms properly implement the IRQ affinity capability. This appears to be either a bug or an oversight, and could certainly be overcome with kernel modifications. The final, and most troublesome, shortcoming lies in Linux's handling of per CPU interrupts. There is no existing facility to disable these interrupts, and adding such a capability looks to be quite complex, touching on some of Linux's core subsystems (memory management, scheduling).

Nothing in this evaluation has indicated that CPU shielding is not a viable method for co-hosting real-time and normal processes. With the rising prevalence increasing parallelism, and falling costs of multi-core computing, CPU shielding should be further studied as a viable co-hosting method.

FUTURE DIRECTIONS

One of the most concerning aspects of using CPU shielding with the standard Linux kernel is the lack of control Linux grants over the shielded processors. The scheduler, memory management, and interrupt handling are all within control of software that is running on both the shielded and unshielded CPU sets. The CPU hot-plug API in Linux may present the ability to better isolate the CPU from interference.

At a maximum worst case latency of 72.7us for this relatively simple system the PCI bus is proving to be one of the highest contributors to worst case response

times. A study of the tradeoff between efficiency and response time with respect to the PCI latency timer is certainly needed for PCI based real-time systems. Also studying the possibility of an unfair bus arbitrator to improve response times might be worthwhile. Reevaluating these numbers for faster variants of the PCI bus may yield better results. However those faster busses tend to have higher bandwidth peripherals attached, so this may balance out. Creating a tool which will evaluate the PCI devices and provide a worst case access time may be a worthwhile side project, since it can be automated.

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.