ANALYSIS OF A DYNAMIC FREQUENCY SCALING ALGORTIHM ON XSCALE.

Group 9

Ishdeep Singh Sawhney (issawhne@unity.ncsu.edu)
Shobhit Kanaujia (sokanauj@unity.ncsu.edu)
Prasanth Ganesan (pganesa@unity.ncsu.edu)

TABLE OF CONTENTS

Solved Issues: _____3

Results: _____5

Simulator Design: _____9

Open Issues: _____ 10

References: _____ 11

Solved Issues:

Overall, we have completed the implementation of simulator for both, the static-EDF and the cycle-conserving EDF. Here we talk about the various issues that were resolved.

1. Mapping hardware addresses. (Shobhit, Prasanth)

In order to do frequency scaling, we need to setup registers in the processor. Thus we need a way to write to these registers from the application-level. We explored 3 possible solutions:

Ioctl-approach:

KernerlIoControl () is an API provided by WinCE, which can be used to access hardware locations. Here we actually depend on the WinCE API to provide us an ioctl () call (i.e. an appropriate ioctl-code) for accessing clock/power management registers. There is not great detail in documentation; the ioctl-codes that we could find did not seem to be ones for accessing the power/clock management registers. Hence this approach was abandoned.

Mmap- approach:

Most of the processor registers exist at well known addresses. We can obtain a virtual address for those well-known addresses by using WinCE APIs. The APIs available for this approach are:

VirtualAlloc (), VirtualCopy (), MapIoSpace (), TransBusAddrToVirtual ()

Inline assembly:

CCLKCFG coprocessor register is not mapped to a well-known hardware address, so in order to access that register we'd most-probably have to do inline assembly.

We are using the mmap () - approach to set the CCCR register, and linking a function written in assembly to set the CCCLKCFG register. This problem was solved by using a combination of Mmap approach and inline assembly.

2. Windows API (Prasanth, Ishdeep)

Windows CE provides a large API. This API is overwhelming to start with and we had to identify the necessary API functions. We required functions to manipulate thread priority, thread quantum and for maintaining time. The following API functions are the important few that we have used:

Thread API :CreateThread(),ResumeThread(),SetThreadPriority(),GetThreadPriority(), CeSetThreadQuantum().

Time related API: GetTickCount()

Suplementary API : List, Text and File manipulation API.

3. Assembly programming under eVC++ for ARM (Shobhit)

The ARM Programmers guide [4], talks about interleaving C and Assembly code. The most relevant part of it is the APCS (Arm Procedure Call Standard) which specifies the register-usage for the ARM procedures and also the syntax for procedures implemented in assembly. We have implemented two assembly routines, to read/write from CCLKCFG (the co-processor register).

4. Scheduling (Shobhit, Prasanth, Ishdeep)

There were various choices available to do scheduling. We could have taken any of the following approaches:

Scheduler Thread:

We can map each job on a separate thread and then use a separate thread for the scheduler. The scheduler runs at a high priority and since Windows CE does not preempt a higher priority thread for a lower priority thread (unless there is resource contention), we can run the scheduler as a high priority thread and then schedule the lower priority threads for the required time quantum. There are probably some threads for the kernel that are being scheduled which give non-deterministic behavior to this scheme. This is further discussed in the open issues.

Scheduler as wrapper:

In this scenario the scheduler would be executed as a wrapper around the tasks. This simple scheme does not protect against malicious user tasks and does not provide preemption. The additional control can be provided by timers but this approach was not used because of other simpler solutions.

Tasks as functions:

Jobs can be scheduled as function calls. This gives good estimates of time but has problems similar to the previous scheme. The main advantage is that this is a very lightweight method.

Simulation:

This method provides good control over timing and scheduling of jobs. According to the calculated alpha (utilization), the processor frequency is varied. The scheduler is executed at the highest priority (THREAD_PRIORITY_TIME_CRITICAL) and has complete control on timing. The time that the scheduler runs is adjusted into the next job's execution time and hence effectively we have scheduler running in zero time. This scheme implements the salient features of all the above schemes without the added complexity. However, this forms a simulation environment and real jobs cannot be scheduled with this approach.

5. Minimizing interference (Shobhit, Ishdeep)

There are a few kernel threads that are scheduled routinely to do housekeeping jobs. These are unpredictable and there is lack of information due to the proprietory OS. This caused our scheduling assumptions to fail in our scheduler implementation. We have reduced such interferences by removing the extra battery pack, the PCMCIA card running 802.11 and also disabling the Bluetooth system. Further we run our scheduler at the highest priority to reduce preemption.(The Windows CE documentation warns against running the threads at the highest priority for long durations.)

6. Scheduler Time included in job (Prasanth, Shobhit)

There were a couple of choices available to maintain the time.We could have maintained time ourselves but this has overhead of incrementing time. We can use the kernel timebut this also does not yield a clean solution. Windows CE provides a function called GetTickCount() which returns the time in milliseconds since system startup, this was a convenient function to use for keeping time. However since the scheduler runs between jobs but is not accounted for in the task utilization calculations, this can possibly

cause deadline misses which have to be prevented under all circumstances since this is a hard realtime system. As described above, we include the scheduler execution as part of the next job's execution time and hence have the scheduler running for free.

7. Voltage scaling algorithm (Shobhit, Prasanth)

We read the various papers on DVS algorithms and decided to work on the algorithms presented in the paper by Padmanabhan Pillai. The paper presents 3 algorithms. We started with the basic static scheduling algorithm. This was easy to implement and weeded out or exposed the platform issues. We implemented the Cycle-conserving algorithm next and were to implement the look-ahead DVS before running out of time.

8. Preemption not discussed but implemented (Ishdeep,Shobhit, Prasanth)

Pillai's paper does not discuss preemption of tasks but we have taken care of preemption of job's because of higher priority job being released during the execution of a job. If preemption will happen, we schedule the lower priority job till the preemption point in time rather than scheduling it to run to completion.

Results:

We present the simulation results for three task-sets for both the static EDF and the Cycle-Conserving EDF Algorithm.
The task sets used are:

U = 0.5

Task	P	D	E
A	4	4	1
B	20	20	3
C	30	30	3

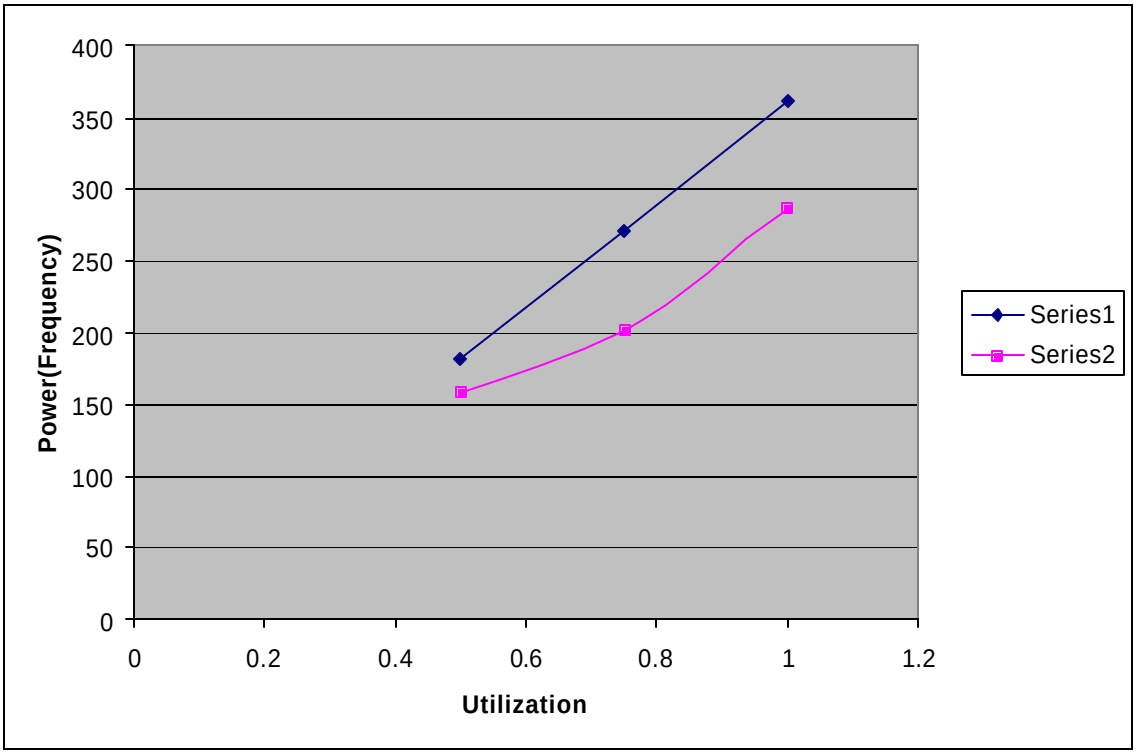
U = 0.75

Task	P	D	E
A	8	8	3
B	14	14	1
C	10	10	3

U = 1.0

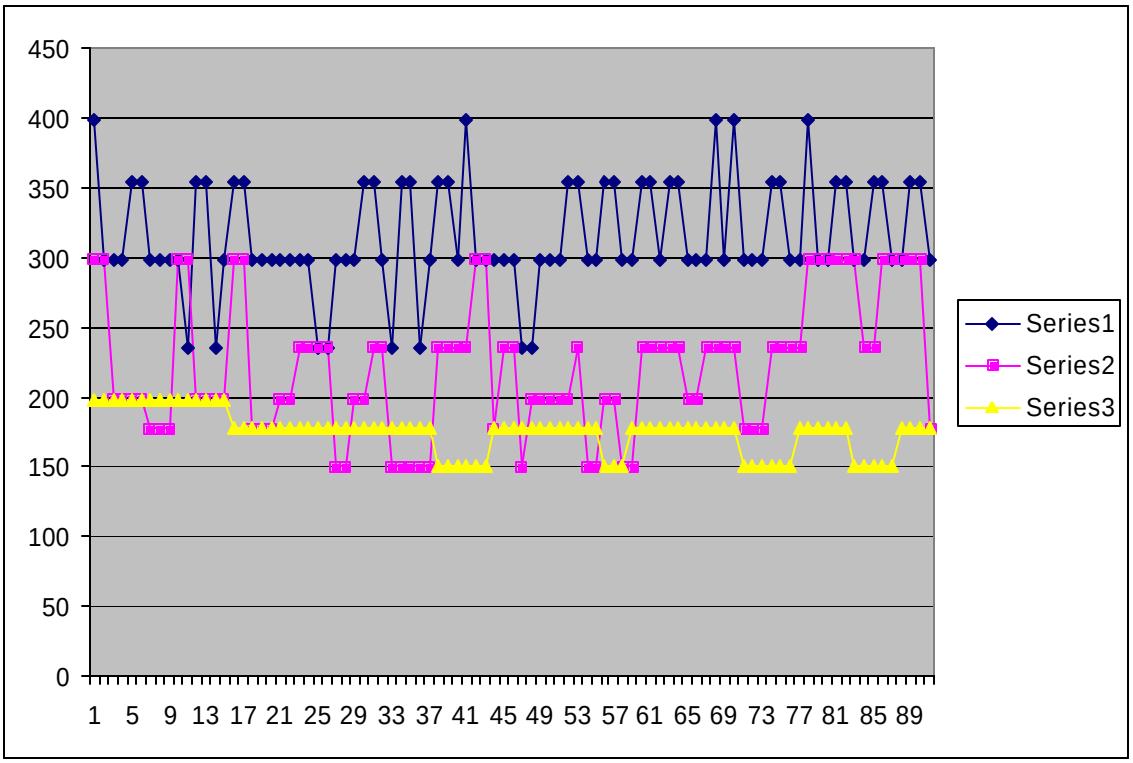
Task	P	D	E
A	4	4	2
B	8	8	2
C	12	12	3

Average Frequency v/s Utilization.



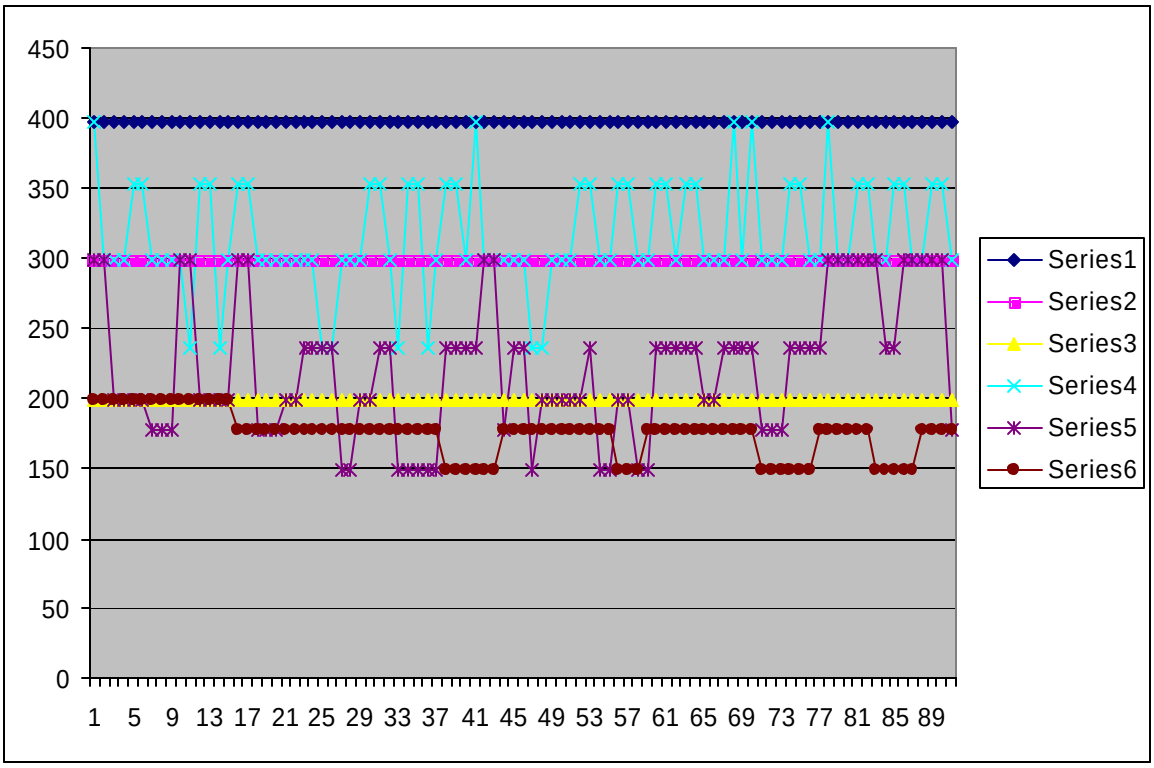
Series 1 Static
Series 2 Cycle-Conserving

Processor Frequency v/s Tick (Cycle Conserving).



Series 1 CC, U =1
Series 2 CC, U = 0.75
Series 3 CC, U = 0.50

Processor Frequency v/s Tick (Cycle Conserving and Static)



series 1 Static, U =1
series 2 Static, U = 0.75
series 3 Static, U = 0.50
series 4 CC, U =1
series 5 CC, U = 0.75
series 6 CC, U = 0.50

Simulator Design:

Implementation (Prasanth, Shobhit)
The salient features of the implementation are below:

- The design is similar for both the static scheduling and the cycle conserving case. Both follow the EDF scheme while the computation of the operating frequency occurs only once in the static scheduling case while it occurs at each job release and completion stage in the cycle conserving scheduling case.
- The scheduler runs at every tick. As per the simulation, the scheduler is part of each job as well as the idle task. To handle the overhead of the simulator at the lowest frequency we have set our tick to 10 msec. This ensures the scheduler completes execution before the tick is over. The remaining time available in the tick period is used to do the work of the task (which in our case is an idle loop). Thus our scheduler takes zero-time, since the scheduler time is accounted towards task-execution time.
- The scheduler operates in the following steps at every tick:
    - Check if there are any new jobs that have been released and add them to the job queue in an EDF manner.
    - Check if any job has missed it's deadline and remove the job from the list.
    - Release the first job in the job queue or execute a previously uncompleted job.
    - On job release compute the utilization (alpha) based on WCET for the current task and actual execution for the other tasks in the task set.
    - Scale the frequency as per the computed alpha.
    - Run the job for a factor of time of the WCET, but as a multiple of the tick count.
    - Depending on the scaled frequency update the running time of the job to the newer period. (expand of the job time to map the lowering in frequency)
    - In case of preemption, the frequency at which the pre-empted job operates is the frequency set by pre-empting job on it's completion.
    - On job completion update the actual execution time (used by other jobs to compute alpha) and scale frequency again.

Open Issues:

Kernel Scheduling of thread (Ishdeep)

The Windows CE documentation describes priority levels available for threads but the documentation falls short on description of kernel threads. This lack of information causes inability in correctly predicting the order of execution of threads. One of the problems encountered was that a thread gets preempted if it makes a kernel call, irrespective of its priority. It was difficult to use SuspendThread() and ResumeThread() to get better predictability because a call to SuspendThread() can fail repeatedly and also ResumeThread() should be called the exact number of times as SuspendThread() to resume execution of a thread. To get complete predictability; complete control over the processor is required otherwise deadline misses may occur. Also there is a huge time penalty due to thread creation for each new job. These difficulties can be overcome by making the system utilization around 60% and also running jobs for maximum 60-70% of their worst case execution time.

Screen flickering

According to Intel Developer's Manual, the following steps should be followed before a power change sequence:

- Disable the LCD Controller or configure it to avoid the effects of an interruption in the LCD clocks and data from the application processor.
- Configure peripheral units to handle a lack of DMA service for up to 500 µs. If a peripheral unit can not function for 500 µs without DMA service, it must be disabled.
- Disable peripheral units that can not accommodate a 500 µs interrupt latency. The interrupts generated during the Frequency Change Sequence are serviced when the sequence exits.

These operations are not being performed at present and must be performed.

Look Ahead RT-DVS

Future work can involve implementing the third algorithm suggested by Pillai or any other DVS algorithm.

References:

[1]. Electrical, Mechanical, and Thermal Specification Datasheet for PXA250. ftp://download.intel.com/design/pca/applicationsprocessors/manuals/278524-001.pdf
[2].Padmanabhan Pillai, Kang G. Shin, “Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems (2001)”, 18th ACM Symposium on Operating Systems Principles.
[3] www.msdn.microsoft.com
[4]. http://www.mculand.com/sub2/arm7tdmi/pg174000.pdf