Implementing a Frequency Scaling Algorithm for iPAQ H3975
(Final Report)

CSC 714
Prof. Frank Mueller

Prakash Ramrakhyani
Mark Wah
Benjamin Welch

Table of Contents

1) Introduction

In the world of computing today there is a great need for low power general-purpose processors. This is evident by the number of battery-operated devices in use. Support for energy awareness comes in two flavors: (1) hardware-driven or (2) software-driven. In our research we plan to focus on a software-driven approach. More specifically, we plan to study the benefits of a dynamic voltage-scaling (DVS) algorithm. A brief overview of the project is presented next.

Currently, there is much research focusing on this topic as listed on the References section. Most of them present algorithms and test results on homegrown systems or open systems environments. This project aims at implementing some of the solutions on a commercial embedded device in a simulated environment.

We propose to implement the chosen DVS algorithm via a simulated task scheduler on Windows CE 3.0.

2) Detailed Progress Report and Contribution

a) Simulation vs Device Driver approach

Description: After researching the methodology of implementing the DVS algorithm we have found some obstacles with the device driver approach. In the end, we chose a simulation approach instead.

Problem(s): Writing a device driver requires more hardware and software support. Platform Builder 3.0 is the software needed to write the device driver for Windows CE 3.0. Other than that, there is very little support for the relatively new processor (Intel XScale PXA250). There is very little access to OEM (Compaq) information and Windows CE 3.0 source code.

Solution: After discussion with Dr. Mueller, all members agreed that writing a simulator would let us focus more on the algorithm, experimentation and results. There was also a great learning curve cost associated with the development environment.

Contribution: All members

***Notes**:* Due to time constraints, the simulator approach is more feasible. We agree that we would learn more using this approach.

b) Power Management API

Description: Initially, we attempt to find as many existing Power Management API as possible to aid us in our goal to implement the DVS algorithm. After some research, we were only able to find a battery status API and mode set API. GetSystemPowerStatusEx() was used a little initially to get some power measurement but was not used in the end due to long run time problem.

Problem(s): There is no existing API to change frequency as expected. The granularity of battery status is very coarse and may not be helpful enough.

Solution: Use the existing API for informational purpose and seek additional method to change frequency, like using assembly language.

Contribution: All members.

Notes: In the end we did not use any of these APIs found, since they did not meet our requirement. We also faced a problem where the frequency change causes some problems in the stability of the system and opted against their use since we cannot extract the information.

c) Scheduler and timing resolution

Description: Since we are creating a simulation, we need to control the timing of the scheduler and all the tasks in the system. In order to implement the scheduler, we must be able to allow tasks to run for a precise amount of time. We have found that the GetThreadTimes() function is extremely accurate and can be used for the timing of tasks. Another issue is putting the scheduler to sleep. This can be accomplished via the Sleep() function.

The following code was to be used as the busy loop for the tasks.

```
do{
        //useless calculations
          x = x * x;
          y = (z << 3) * (23 / (w * 1.1));
          x = x / 1.1;

          GetThreadTimes(GetCurrentThread(),     // specifies the thread of interest
                    &lpCreationTime,             // when the thread was created
                    &lpExitTime,                 // when the thread was destroyed
                    &lpKernelTime,               // time the thread has spent in kernel model
                    &lpUserTime                  // time the thread has spent in user mode
                   );

}while( lpUserTime.dwLowDateTime < time*TIMESTEP);
```

where *time* is the relative time in the simulation and *TIMESTEP* is a value of time with units of 100ns.

Another issue is: knowing when a task has completed. The following function can be used for this:

```
//check to see if the thread has finished executing
GetExitCodeThread(hThread[index],          // handle to the thread
                  &lpExitCode              // address to receive termination status
                  );
```

By testing the lpExitCode variable, we can determine if the thread specified by hThread[index] has finished.

```
if(lpExitCode != STILL_ACTIVE)
…recreate the thread….
```

Problem(s): Some basic timing functions like Sleep() were not working as expected. Some information revealed that the OEM (Compaq) might have changed some of its specification. Attempts to use POSIX threads are not successful either since Windows CE 3.0 is not fully POSIX compliant.

Solution: A lot of testing and experiment was done and in the end, we found the function Sleep() worked properly. It turns out that the method used to verify the tick function did not wrok properly under the Pocket PC 2002 SDK. We were initially testing the amount of time used by the Sleep() function by using the GetSystemTime() function. This function appears to not be working properly. Instead, we used GetTickCount() which gives good timing granularity. However, we were unable to use any of this functionality since we changed the way we obtained the data. We used a simulated timing environment. This functionality will be useful for future work or other projects that may want to follow up our findings.

Contribution: Benjamin Welch

Notes: Most of the work done here is research in various websites, discussion groups and a lot of testing. We regret that we don't have enough time to implement a useful 'real time' environment.

d) Assembly Language approach

Description: We resorted to use assembly language since there are no proper power management APIs. Reference to several Intel manuals on the XScale PXA 250 processor showed that ARM assembly language can be used to write to the CCLKCFG register on co-processor 14 which deals with the FCS (Frequency Change Sequence), and the turbo and run modes.

Problem(s): Research to several websites and discussion groups showed that assembly language support in eVC++ is poor. The conventional inline assembly language is discouraged. There is also a compatibility problem since we are not sure if the ARM assembler will work with the new XScale processor

Solution: Assembly language can be written in a separate .asm file and be added to the workspace in eVC++. Instruction to build the .asm file needs to be explicitly set also. The notes below show how to use assembly language in eVC++.

Contribution: Mark Wah

Notes: Steps for assembly language use:

1) Write the required ARM assembly language into a .asm file and link it into the Project Workspace, the code in bold below is writing 2 to the CCLKCFG Register which will set the FCS (Frequency Change Sequence) bit.

```
AREA |.data|, DATA
|?ControlFreq@@3PCKC| DCD 0x41300000 ; ControlFreq

EXPORT |?enable_frequency_change@@YAXXZ| ; enable_frequency_change

AREA |.pdata|, PDATA
|$T24415| DCD |?enable_frequency_change@@YAXXZ|
DCD 0x40000501

AREA |.text|, CODE

|?enable_frequency_change@@YAXXZ| PROC ; enable_frequency_change

; Line 7
sub sp, sp, #4
|$M24413|
; Line 9
mov r3, #2
str r3, [sp]

mov r3, #2
mcr p14, 0, r3, c6, c0, 0

; Line 10
add sp, sp, #4
mov pc, lr
|$M24414|

ENDP ; |?enable_frequency_change@@YAXXZ|, enable_frequency_change
```

2) Set the build instructions for this file and the output, for example:

```
        build command: armasm file.asm ARMDbg\file.obj

        output: ARMDbg\file.obj
```

e) Frequency Change on Intel Xscale PXA250

Description: The frequency change sequence is a 2-step process. Besides writing to the CCLKCFG register, the CCCR (Core Clock Configuration Register), which is a memory-mapped register, needs to be written to. The value in the CCCR determines the frequency that the processor will run at after setting the FCS bit in the CCLKCFG register.

Problem(s): eVC++ support to write to the memory-mapped register is not documented. Research shows that VirtualAlloc() and VirtualCopy() can be used for this purpose. However, VirtualCopy() is not an exported function in eVC++ although it is defined in coredll.lib. Platofrm Builder 3.0 has this function exported.

Solution: To enable the use of VirtualCopy(), we defined it in a .h file as follows:

```
extern "C" BOOL VirtualCopy(LPVOID dest, LPVOID src, DWORD size, DWORD flags);
```

This is needed since none of the .h file in the SDK or eVC++ has the above function defined although it is defined in the library file coredll.lib. Details to write to the CCCR are shown below on the Notes section.

Contribution: Prakash Ramrakhyani

Notes: Steps to write to the CCCR

1) Define VirtualCopy() in a .h file since it is not an exported function in eVC++ but exported in Platform Builder. Coredll.lib contains the implementation and both eVC++ and Platform Builder have access to it.

```
extern "C" BOOL VirtualCopy(LPVOID dest, LPVOID src, DWORD size, DWORD flags);
```

2) Use VirutalAlloc() to allocate the memory

```
 LPVOID VirtualCCCR = VirtualAlloc(0, sizeof(DWORD), MEM_RESERVE,
PAGE_NOACCESS);
```

3) Use VirtualCopy() to map the memory location we want to write to:

```
if(!VirtualCopy((LPVOID)VirtualCCCR, (LPVOID)CCCR, sizeof(DWORD), PAGE_READWRITE
|PAGE_NOCACHE | PAGE_PHYSICAL))
      {
       VirtualFree(VirtualCCCR,0,MEM_RELEASE);

       VirtualCCCR = NULL;
       }
```

4) Using the different frequency level we have computed in the table below, write the value to it, for example:

```
*(int *)VirtualCCCR = 289; (for 100 Mhz, using the default memory frequency)
```

The values for these registers for the four frequency settings are detailed in the following table:

	100 MHz	200 MHz	300 MHz	400 MHz
CCCR	289	321	449	577
CCLKCFG	2	2	3	3

Table 1: Control Register Values for Various Frequencies

The table below summarizes the typical voltage settings for the four frequency settings. (Source: Electrical, Mechanical, and Thermal Specification Datasheet for PXA250 [7])

Note: There are more than four frequency settings for the PXA250. However, we have chosen to utilize only four of these settings. The register values given above specify a single frequency for memory i.e. for our experiments we do not scale the memory frequency

Frequency(MHz)	Voltage (V)
100	0.85
200	1.00
300	1.10
400	1.30

Table 2: Voltage/Frequency Relationship

f) Implementing the Algorithm

Description: The algorithm is based on the feedback EDF scheduling described in [1] and [3]. It addresses

a. Using idle slots that may appear in the Worst Case schedule,
b. Slack generated due to early completion
c. Passing slack safely to a preempting task without causing any missed deadlines

For slack distribution our algorithm uses a greedy approach as described [1] i.e. allocating all available slack to the next task being scheduled, so that it may run at the lowest possible frequency.

Problem(s): The approach for the utilizing idle slots, as hinted in the paper, seemed a little too tedious. The paper hints at using "future knowledge" of a basic EDF schedule for the task set to locate idle slots, such that only those idle slots in the base-EDF schedule, that occur before the deadline of the task to be scheduled, are utilized for frequency scaling. This approach seems to be non-trivial for task sets that have large hyper-periods, requiring maintenance of large structures.

Solution:
Our algorithm has a slightly different approach for utilizing idle slots. It is based on the "Maximum Constant Speed" technique as described in [2]. "Maximum constant speed is the lowest possible clock speed that guarantees a feasible schedule for the task set at hand."[2]

$$f_{max} = U_{WC} \cdot f_{peak}$$

f_{max} : Maximum Constant Speed (frequency).
U_{WC} : Worst Case Utilization at peak frequency(f_{peak}).
f_{peak} : Peak operating frequency of the processor

The idea is to statically choose a frequency at which at which the worst-case utilization is 1, so that there are no idle slots in the worst-case schedule. For slack estimation due to early completion WCET for each task in the task set is now scaled up to account for the lower maximum frequency. (WCET' = WCET/ U_{WC})

Contribution: All members

Notes: The algorithm has been implemented on a time-driven scheduling simulator that is run as a Real Time task on Windows CE. We first built a basic EDF scheduling simulator, and then enabled it for frequency scaling. The task set is input from a text file (very much like the homework schedulers). Within the scheduler, there is the notion of relative time and actual time. We have decided to reuse the basic EDF scheduler from homework 2 and add the notion of actual time. Actual time is accounted for via a Windows API. Windows CE provides a mechanism for accessing the amount of time that a thread has been active (as discussed earlier). Using this data we are able to interrupt executing tasks at regularly scheduled intervals. Our code and data from the experiments we performed have been posted on our website.

g) Experiment and Results

Since we do not have a standard "benchmark" to test our scheduler ran our scheduler for 3 distinct task sets. One of these is described in [1] and the other two are arbitrary test cases that we built. The task sets and the duration of simulation for each task set are described below.

a) 3-task Task set WC Utilization = 0.75 Hyper-period: 40
Duration of Simulation: 5 Hyper-periods

Task	Period	Deadline	Phase	WCET
A	8	8	0	2
B	5	5	0	2
C	20	20	0	2

Table 3: Task Set 1

b) 10-task Task set WC Utilization = 0.718 Hyper-period: 120
Duration of Simulation: 1 Hyper-period

Task	Period	Deadline	Phase	WCET
A	8	8	0	1
B	10	10	0	1
C	12	12	0	1
D	15	15	0	1
E	20	20	0	1

F	24	24	0	1
G	30	30	0	2
H	40	40	0	3
I	60	60	0	4
J	120	120	0	5

Table 4: Task Set 2

c) 3-task Task set described in [1] WC Utilization = 0.718 Hyper-period: 280
Duration of Simulation: 1 Hyper-period

Task	Period	Deadline	Phase	WCET
A	8	8	0	3
B	10	10	0	3
C	14	14	0	1

Table 5: Task Set 3

Problem(s): We are unable to run the algorithm for long durations due to instability of the system after initiating the FCS (Frequency Change Sequence) repeatedly. The Intel manual reveals that during the Frequency Change Sequence, for Hardware and Watchdog Resets, the resets takes precedence over the FCS. This may fouled our experiments for runs greater than 3.5 minutes. Moreover, if the GPIO Reset is asserted, the contents of the SDRAM will be lost. This is critical since we are gathering data and there is a potential loss when this happens. Due to our inability to run experiments for a long duration we have not been able to prove the energy benefits DVS-EDF by measuring battery life.

Solution: Instead of showing benefits of EDF-DVS using battery life consumption we have shown a comparison of the schedule produced by EDF vs. that produced by our implementation of the DVS-EDF algorithm. We have also shown, for one test case a comparison of analytical estimates of energy consumption of the two algorithms.

Contribution: All members

3) Results

Analytical Estimate of energy consumption for when task–set 1 is executed for 1 Hyperperiod.

$$P\ a\ f^3 \qquad E\ a\ f^2$$
$$P = x \cdot f^3 \qquad E = y \cdot f^2$$

To compare energies for the two algorithms, we assume x = y = 1. Consider power at peak frequency (400 MHz) = 64 units and energy consumption at peak frequency for 1 scheduling time period is 16 units. Further consider no energy is consumed in idle slots. By the equations above, power and energy consumption at other frequencies is described in the table below

Frequency	**Power**	**Energy**
100 MHz	1	1
200 MHz	8	4
300 MHz	27	9

Using the above assumptions we estimated the average power over the first hyper period for task set 1 for DVS-EDF algorithm to be 21.875 units while that for the EDF schedule to be 40 units This implies approximately 45% power savings over the base case. Energy consumption over the first hyper-period for this task set was estimated to be 293 units with the DVS-EDF schedule and 400 units with EDF schedule. This indicates that the DVS-EDF algorithm, for the given scenario had 27% energy savings over that base-case.

The figures below show a comparison of excerpts from the schedule produced by basic EDF scheduler and that produced by our DVS-EDF scheduler for the 3 scenarios described above. We shall discuss some observations for the second (b) scenario described above

Fig 3 shows an excerpt from the basic EDF schedule between scheduling instants 35-50
Fig 4 shows the DVS-EDF schedule for the same time duration

Y – axis indicates the fraction of the peak frequency at which a task is run
X axis indicates progressing time.

In the DVS-EDF schedule G1 is pushed to execute at a later time, due to its later deadline. Earlier tasks like D2 take advantage of this and run at a lower frequency (75 % of peak frequency). This indicates utilization of idle slots.
C3 utilizes slack passed from previous tasks due to early completion and runs at 25% of peak frequency.
A5 and A6 are initiated at same scheduling instants in both the schedules but DVS-EDF runs them at 75% of peak frequency. This is an indication that our DVS-EDF schedule utilizes future idle slots to run jobs at a lower frequency.

a) 3 task set with WC Utilization = 0.75

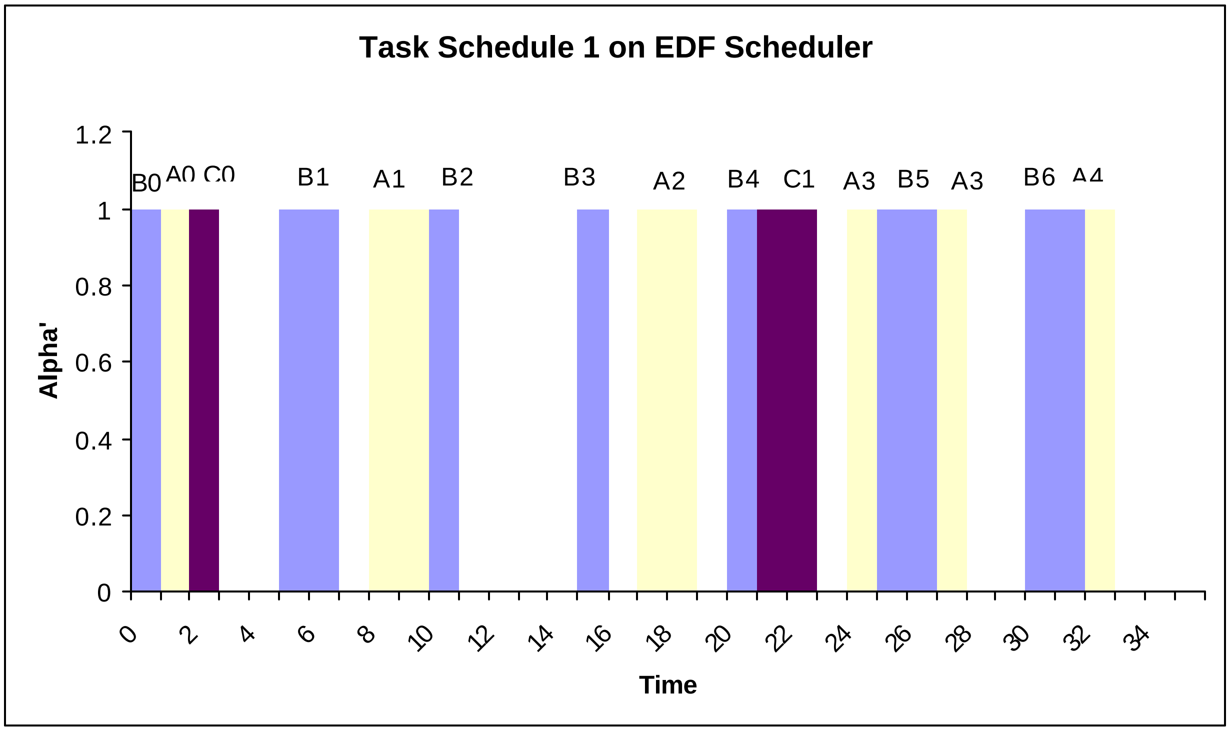

Figure 1: Partial EDF Schedule for Task Set 1

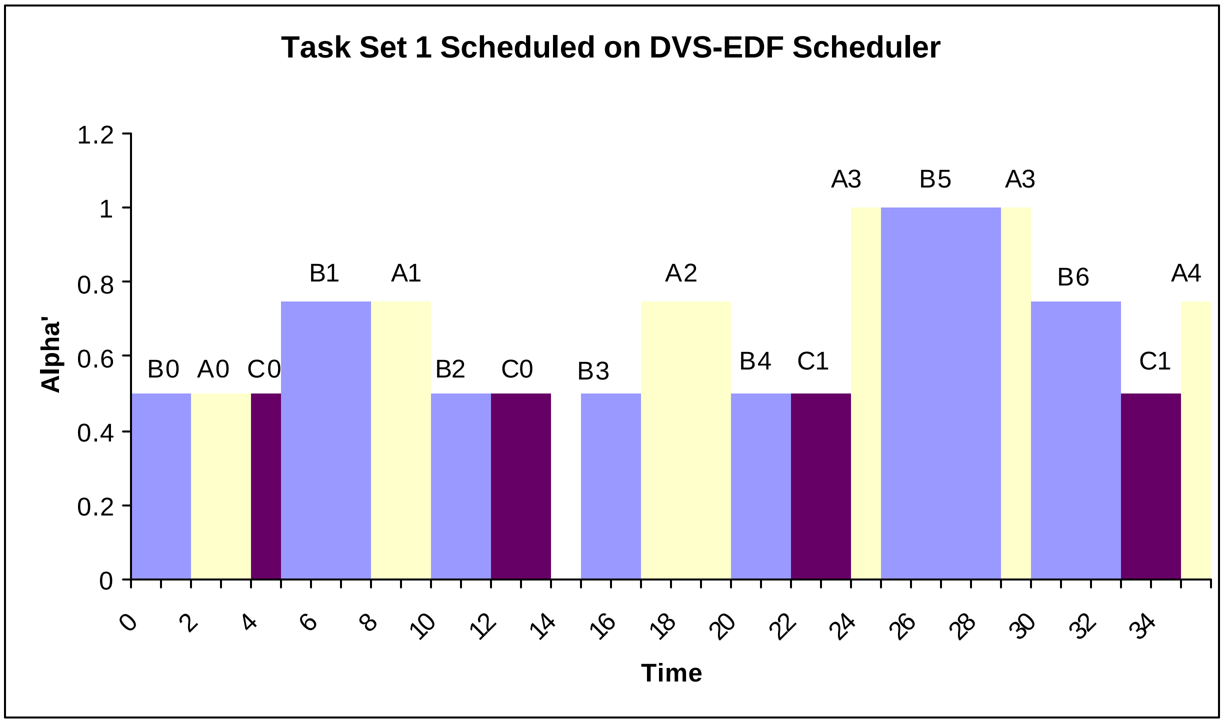

Figure 2: DVS-EDF Schedule (partial) for Task Set 1

b) 10 task set with WCET utilization = 0.718

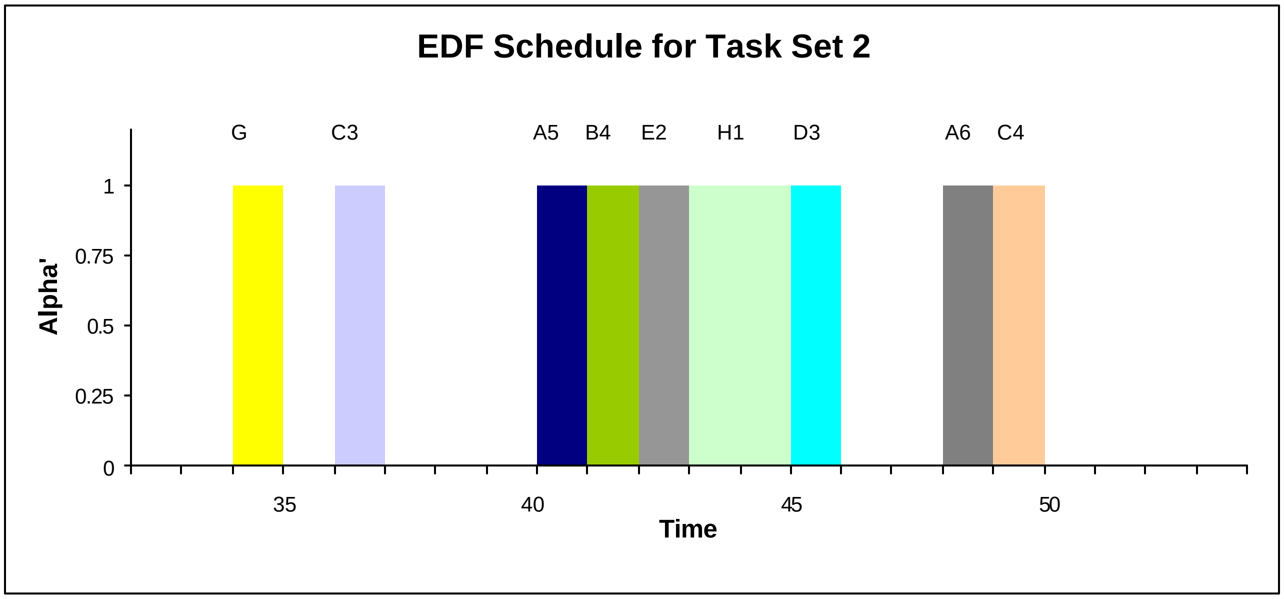

Figure 3: Partial EDF Schedule for Task Set 2

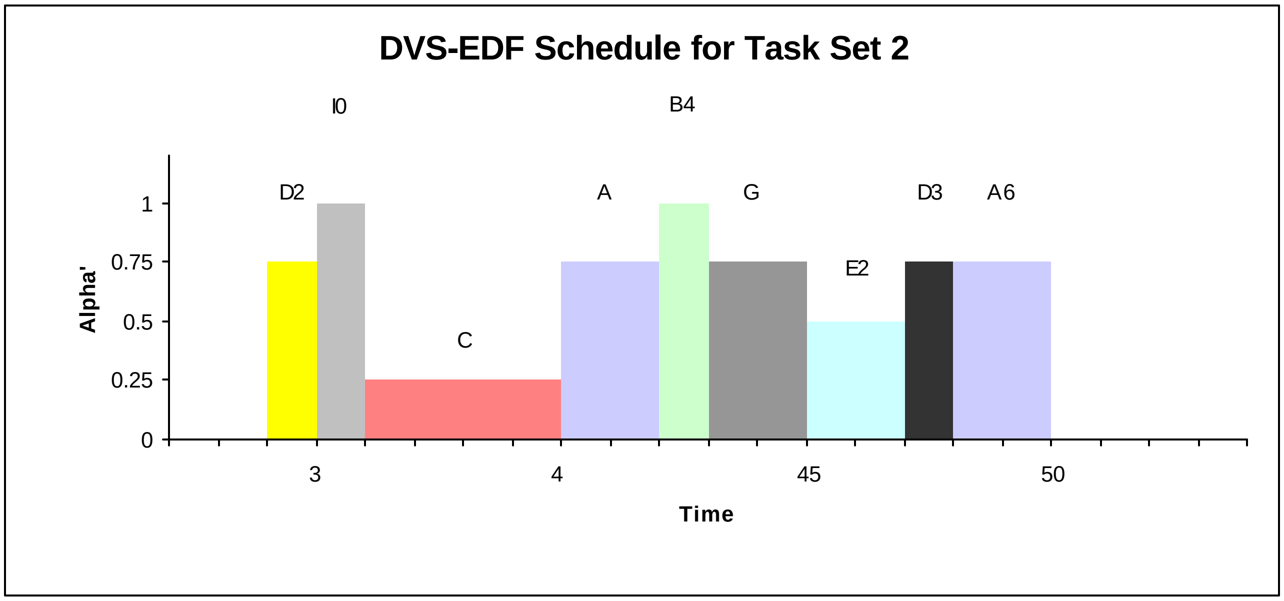

Figure 4: DVS-EDF Schedule (partial) for Task Set

c) Task Set 3. Utilization = 0.718

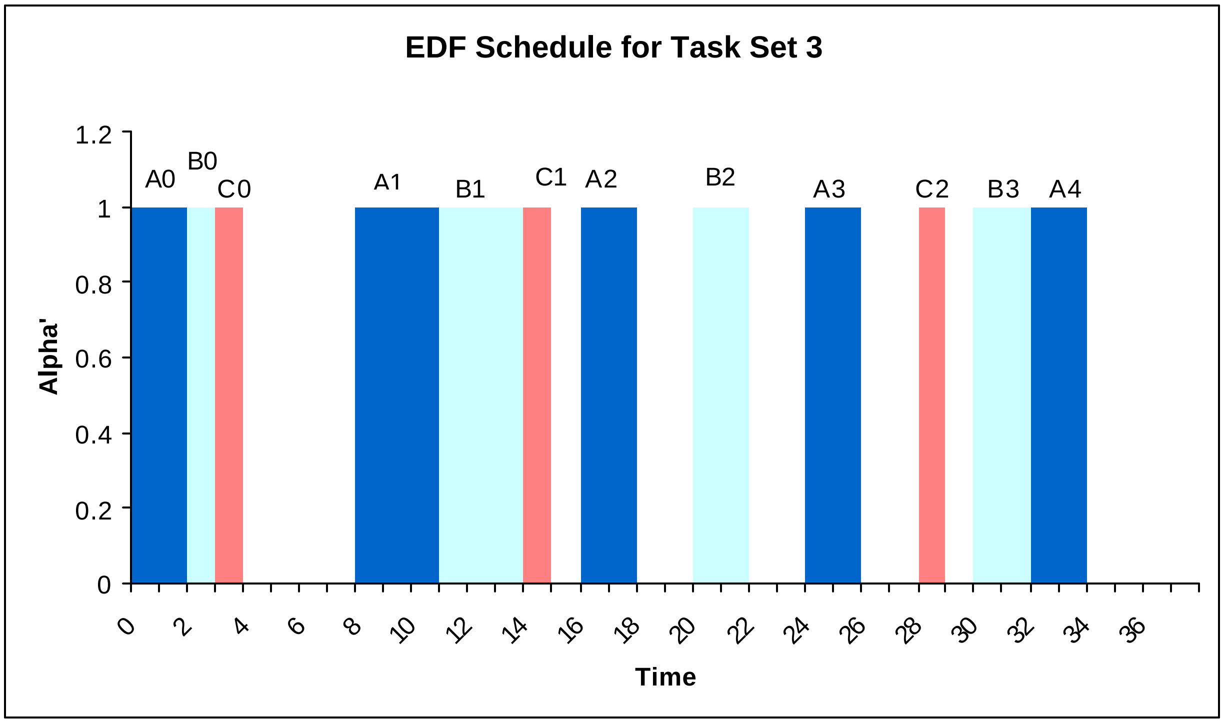

Figure 5: Partial EDF Schedule of Task Set 3

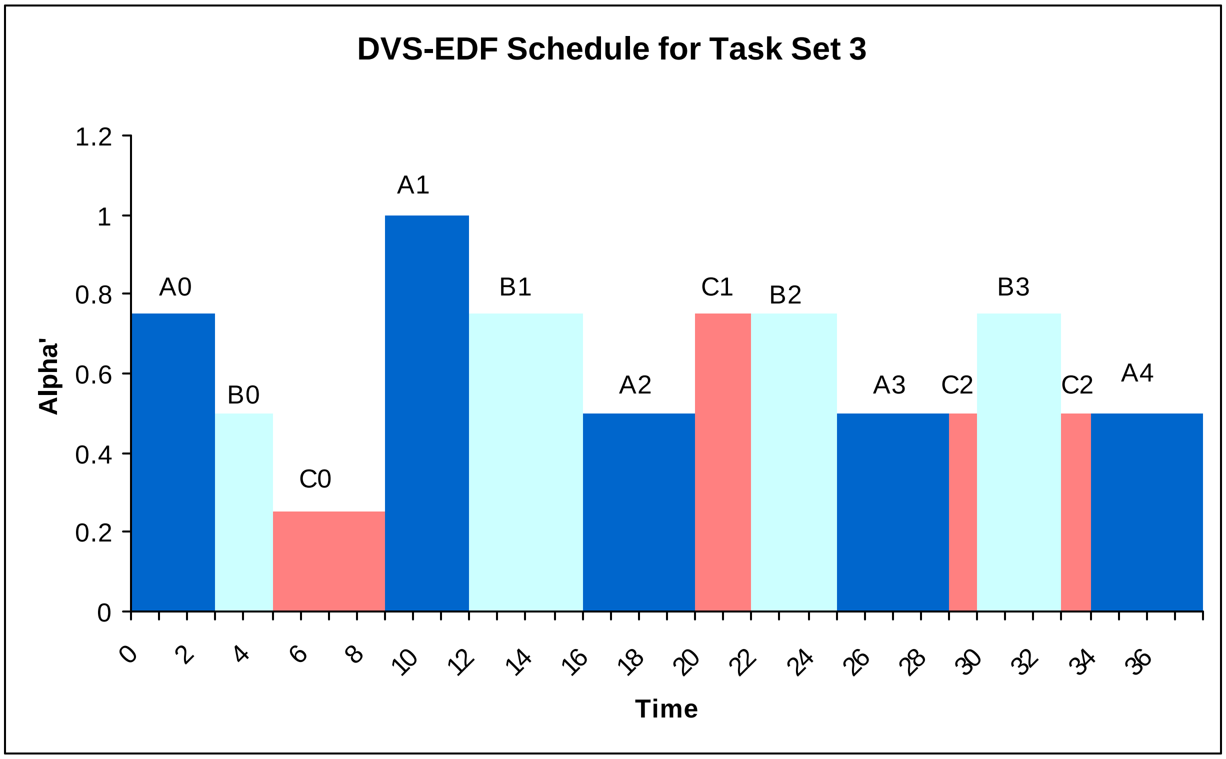

Figure 6: DVS-EDF Schedule (partial) for Task Set 3

4) Conclusion

The DVS-EDF algorithm shows potential energy savings while meeting all deadlines. Due to several major setbacks, we were unable to produce a stable environment on the Windows CE 3.0 platform. However, we have successfully implemented a DVS algorithm capable of functioning in a simulated environment. Although we were unable to meet all the goals set forth by our proposal, we have delivered a base for future work to build upon.

5) Future Work

There is more work to be done to get accurate real world results. This project provides a good base for future work, which includes but not limited to:

a) Stabilization of iPAQ following frequency scaling
b) Running simulation on Windows CE 3.0 platform
c) Implementing a device driver
d) Implementing DVS algorithm in the scheduler of Windows CE 3.0, since the source code is available for download.

Our project demonstrates that running the DVS-EDF algorithm on a commercial embedded device is possible with more effort, and also has significant power and energy savings.

6) Project homepage

http://www4.ncsu.edu/~mwah/index.htm

7) References

[1] A. Dudani, F. Mueller, Y. Zhu "Energy-Conserving Feedback EDF Scheduling for Embedded Systems with Real-Time Constraints."

[2] C. J. Hughes et. al. "Variability in the Execution of Multimedia Applications and Implications for Architecture," *Proc. of 28th International Symposium on Computer Architecture*, June 2001.

[3] F. Mueller, Y. Zhu "Preemption Handling and Scalability of Feedback DVS-EDF."

[4] T. Pering, T.Burd, and R. Broderson. "Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor System."

[5] Intel® XScale™ Core Developer's Manual *[http://www.intel.com/design/intelxscale/27347301.pdf]*

[6] Intel® PXA250 and PXA210 Applications Processors: Operating System Developer's Guide.
[http://www.intel.com/design/pca/applicationsprocessors/manuals/278535-001.pdf]

[7] Intel Electrical, Mechanical, and Thermal Specification Datasheet for PXA250
[ftp://download.intel.com/design/pca/applicationsprocessors/manuals/278524-001.pdf]

[8] Microsoft®: Windows CE
[http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wceintro/htm/cestart.asp]

[9] Microsoft®: Introduction to Development for Pocket PC
[http://msdn.microsoft.com/library/default.asp?url=/library/e-us/dnppc2k/html/ppc_ntro.asp]

[10] Microsoft®: Mobile Device Developers
[http://www.microsoft.com/mobile/developer/default.asp]

[11] (Almost) No POSIX OS Is An Island
[http://www.embedded.com/story/OEG20020111S0071]

[12] Pocket PC Developer Network
[http://www.pocketpcdn.com/]

[13] Open Source POSIX Threads for Win32
[http://sources.redhat.com/pthreads-win32/]