Studying Sleep Modes on the PowerPC 405LP processor

Project Report

Prepared by

Tricia Glidewell (taglidew), Deepa Srinivasan (dsriniv)
Web page: http://www4.ncsu.edu/~dsriniv/csc714/index.html

Fall 2005

CSC 714: Real-time Systems

North Carolina State University

1 Problem description

Energy consumption has become a vital design constraint in embedded systems. The demand for efficient energy management is critical in portable and embedded devices where the available battery service life is critical. It is also important in non-embedded systems so as to conserve energy consumed to bring down costs and also for accurate infrastructure support (for e.g., in a datacenter, the power consumed by several servers can impact the cooling needed for the room). Hence, it is an important field of study to enable efficient power management of devices, both embedded and otherwise. One of the main ideas in power management is to detect when a system is idle and scale down the power consumed in that state. When the system is at peak (or comparable), it is scaled back up so that the performance impact is kept minimum. Dynamic Voltage Scaling (DVS) is widely supported in current processors so as to maximize battery life/minimize power consumption in various systems. Further, certain components of a system can also be set to different sleep or standby modes wherein they are completely inactive and hence consume minimum or no power, thus reducing leakage power or power consumed in an idle state of the system. With recent advances in manufacturing technology, leakage or static power dominates the power consumption in a system, rather than the power consumed when a device or system component is active. Hence, it is important to study this aspect of power management in detail.

In our project, we used the PowerPC 405LP processor board – the 405LP is based on the PowerPC 405 processor core and additionally offers power efficiency through Dynamic Voltage Scaling (DVS). It also supports standby or (two different) sleep modes, as described above. The main goal of our project was to study these sleep modes on the 405LP processor board. In particular, we experimentally measured the overhead of using/enabling the sleep modes – i.e. when a component in the system goes into a sleep mode, what is the time delay, from when it is needed to be active again to when it actually becomes active. This time delay will be important for real-time applications since it will need to be taken into account while determined task schedules (according to deadlines) and system idle time. As a stretch goal for our project, we integrated the measured overhead for sleep modes with a real-time scheduling algorithm (EDF).

2 Overall accomplishments

As part of our project, we completed the following major milestones. Note that all parts of this project were worked on and completed by Deepa Srinivasan and Tricia Glidewell together.

1. Acquainted ourselves with the 405LP development board and development environment. This involved connecting to the board and running sample programs. We also went through the relevant parts of the Linux kernel (including the Dynamic Power Managament or DPM module) to understand the power management modifications done for the 405LP.

2. We read relevant current literature (see References section), to better understand the 405LP processor system as well as power management techniques.

3. We investigated various timer mechanisms available on the 405LP to determine which one is most suited for use to measure sleep overhead in fine granularity (in the scale of microseconds). Through our research and experiments, we determined that the

Programmable Interrupt Timer (PIT) was the best suited. However the PIT did not seem to be available during a sleep mode, and hence could not be used for microsecond-level wakeups. Still, we were able to use the PIT right before and after a sleep and wakeup to determine overhead (as presented in the Results section).

4. We modified an existing application program to exercise the various sleep modes on the 405LP to test our modification and take overhead measurements.

5. As part of our stretch goal, since we accomplished the above milestones, we incorporated the measured overheads into a real-time scheduling algorithm (EDF) simulator (the source code and outputs from the program are attached with our report).

3 Results

This section presents the overhead values that we measured experimentally after incorporating our kernel modifications. These measurements were obtained using the PIT that decremented at a rate of 1.048576 MHz. For each of the modes we studied, we obtained 10 different readings, that are listed in Tables 1, 2 and 3. The modes we studied are:
Clock-suspend: In this mode, the output of the PLL is stopped. No units are powered down, and so no state is lost. SDRAM must held in self refresh to avoid losing SDRAM contents. On wakeup, the PLL continues and processing resumes exactly where it left off. Hence we expect no overhead on wakeup, as is evidenced in Table 2.

Suspend: In this mode, the SDRAM is not powered off, so state can be saved there as long as SDRAM is put into self-refresh mode prior to the sleep.

Further detailed descriptions of the power modes can be found in the 405LP-sleep.txt file that is part of the 405LP Linux kernel source tree.

Table 1: Measured overhead values for "suspend" mode of 405LP

#	Sleep Overhead		Wakeup Overhead	
	Cycles	**µs**	**Cycles**	**µs**
1	218	228.59	80	83.89
2	216	226.49	79	82.84
3	227	238.03	77	80.74
4	219	229.64	79	82.84
5	215	225.44	77	80.74
6	215	225.44	77	80.74
7	216	226.49	79	82.84
8	214	224.40	77	80.74
9	215	225.44	79	82.84
10	215	225.44	76	79.69

Table 2: Measured overhead values for "clock-suspend" mode of 405LP

#	Sleep Overhead		Wakeup Overhead	
	Cycles	µs	Cycles	µs
1	161	168.82	0	0.00
2	160	167.77	0	0.00
3	161	168.82	0	0.00
4	161	168.82	0	0.00
5	161	168.82	0	0.00
6	160	167.77	0	0.00
7	161	168.82	0	0.00
8	161	168.82	0	0.00
9	160	167.77	0	0.00
10	161	168.82	0	0.00

Table 3: Measured overhead values for "suspend" mode of 405LP (invoked via the U63 button on the processor board)

	Sleep Overhead		Wakeup Overhead	
	Cycles	µs	Cycles	µs
1	229	240.12	78	81.79
2	228	239.08	78	81.79
3	229	240.12	77	80.74
4	226	236.98	79	82.84
5	228	239.08	78	81.79
6	227	238.03	78	81.79
7	216	226.49	81	84.93
8	227	238.03	76	79.69
9	227	238.03	76	79.69
10	221	231.74	79	82.84

4 Detailed Notes

In this section, we present complete details of the work we did for this project, both to demonstrate our methodology and research, as well as serve as reference for future extensions and/or related work.

4.1 Experiment 1 Notes

We modified the do_suspend function in ibm405lp_pm.c to use the RTC & PIT. We then compiled the kernel with our changes and flashed the board. The following tests in this section were driven by placing the board into suspend mode (mode = clock-suspend) and then waking it up via the proc interface.

On the first run, we were not able to wake the board up after a suspend. Therefore, we changed our code to only read registers without writing to any. We also added print statements to output

the value of the PIT before suspend and after wakeup. In these experiments, it looked like the PIT was incrementing and not acting as expected. Therefore, we changed our code to just read registers except we did set rtc0_cr0 so that the periodic interrupt rate was 500ms (1111 for last 4 bits). Additionally, we forced the print statements to output all 32 bits from the PIT.

When executed, the PIT decremented by 1440 which is not what we expected. Since it was 46 seconds, we had expected the PIT to only decrement by around 92 (we expected it to decrement once every 500 milliseconds).

Analysis: It seems that setting the periodic interrupt for the RTC does nothing - in suspend mode it does not set the RTC to drive the PIT.

Next we changed our code to write 0xFFFFFFFF to the 32 bit PIT and ran it again. The results are presented in Table 4. Before writing the PIT with the value 0xFFFFFFFF, what we were reading did not make sense. So, our conclusion is that maybe the PIT was not enabled. Additionally, since regardless of how long the board is suspended the decrement is the same, it looks like the PIT is not being decremented while it is actually suspended. We think this is because the clock that is driving the PIT is not available during suspend.

Table 4: Measured PIT decrements (from right before actual suspend and right after wakeup) without changing the clock driving the PIT

Run	Seconds suspended	Cycles PIT decremented
1	30	19872
2	6	19871
3	48	19870
4	120	19870

Our next step was to go back and see if we can get the RTC to drive the PIT and still be able to wakeup. When we initially ran our code that set the RTC to drive the PIT, it was not able to wake-up. Therefore, we changed the order of configuring the registers so that we setup the RTC as we would like it and then changed the clock (in the CPC0_CR1 register) that is driving the PIT. We also ensured the PIT was read at these times: right before suspend, right after wakeup, and after finished waking up (although in clock-suspend the last 2 are really executed one after the other). With this change, we were able to wake-up the board after a suspend. The results are presented in Table 5.

Table 5: Measured PIT decrements (from right before actual suspend and right after wakeup) with RTC driving the PIT

Run	Seconds suspended	Cycles PIT decremented
1	30	3452
2	36	3489

Note: We noted that a call is being made that delays the do_suspend method for 5 clock cycles. We added another print statement to print PIT right after delay call (call that delays for 5 clock cycles).

We then searched through the code to determine if the PIT is being manipulated in any way during a suspend. We found in 'ibm405lp_pmasm.S', where the PIT is being restored if auto-reload was not enabled. To see if this restore was being executed, we entered a print command to print out the TCR (we wanted to check bit 9 to see if it was set to auto-reload or not). The value of the TCR = 0x04400000, therefore it should not be restored during a wakeup.

Analysis:
It seems that the PIT is not available during suspend but we need to double check our register settings to ensure we are doing everything correctly and additionally to further check into the kernel code to ensure they are not disabling the PIT.

Other options may be:
Even if the PIT is not decremented during the sleep, it is decremented during the wakeup so we can place the code that measures the PIT to read immediately after wakeup and again after finished wakeup(after restoring states). This option may cause the measured overhead to lose some accuracy, but it will be more accurate than the 1-second RTC reading. But for this to work, we need to find out the frequency at which the PIT is decremented.

As a note, in these experiments, we are not able to get back the console (we were using minicom) after the board wakes up. Instead, we the board had to be rebooted.

4.2 Experiment 2 Notes

Based on previous experiments from the week before, we were able to change the clock that drove the PIT timer to the RTC. Starting from there, the first thing we want to do is to ensure that the PIT reading is as expected based on the frequency we set. In order to do this, we changed RTC0_CR0 so that we use a 4.2MHz clock frequency and set periodic interrupt to all zeroes to ensure this setting does not skew what we expect. We then changed the do_suspend to not actually suspend by commenting out the block of code that does this. We then inserted in its place code to sleep for a specified amount of time and then we will see if the PIT decrements by the expected amount during the sleep time.
The method we used is udelay(int microseconds). The results are presented in Table 6.

Table 6: Measured PIT decrements (with printk included in overhead) with RTC driving the PIT

Run	Microseconds suspended	Cycles PIT decremented
1	1	3274
2	10	3284
3	500	526731

At this point, we noticed that our printk was between the first read of PIT and the second read. This means that our results included the overhead to execute this printk command. We moved the printk to after the second read so that the read of PITS was back to back. We then executed again. The results are presented in Table 7.

Table 7: Measured PIT decrements (with printk removed from overhead) with RTC driving the PIT

Run	Microseconds suspended	Cycles PIT decremented
1	0	0
2	1	1
3	10	11
4	20	21
5	500	524
6	1000	1047
7	5000	5234

Notes:

1. This seems to be following 1.048576 (~1.5) MHz setting and not 4.2MHz.
2. There does seem to be some error coming from somewhere. We think this is actually an error in the udelay timer.

We then commented out all the code that modified the RTC0_CR0 register since it doesn't seem to be changing the frequency. We will use the 1.05 MHz frequency so we do not need to set this anyway. We ran another test just to make sure that it is still running at 1.05 MHz and the removal of this code didn't change anything. It ran as expected.

Just to see if mdelay reduces error, we will try mdelay(int milliseconds):
5 milliseconds = 5234 cycles decremented in PIT
This didn’t seem to affect the results which is not surprising.

4.3 Experiment 3 Notes

Now that we have set the RTC to drive the PIT and determined the frequency of the RTC (1.05MHz), we added back the suspend code and removed all of the sleep calls. We then placed the first PIT read as the first call after wakeup initiated and the second PIT read after wakeup activity completed. The results are presented in Table 8.

Table 8: Measured PIT decrements (wake-up overhead) for various suspend modes

Run	Mode	Cycles PIT decremented (wake-up overhead)
1	Clock-suspend	0
2	Suspend (via U63 button)	77
3	Suspend (via U63 button)	79
4	Suspend (via proc interface)	77

4.4 Experiment 4 Notes

Before continuing further, we moved the initialization of the PIT back to right before we suspend. Although the PIT is not available during the suspend, we think that this will provide better accuracy. Using suspend mode, we ran a few tests varying the amount of time we were suspended. In all cases, the PIT was about the same but was not a value that was expected. It appears that the PIT is overwritten during the suspend, therefore we removed this alteration.

We then changed our code to measure both the suspend overhead and the wakeup overhead. This was done by initializing the PIT to 0xFFFFFFFF upon entering the do_suspend method and then reading it right before the actual suspend.
After the wakeup, we immediately set the PIT to 0xFFFFFFFF and then read it again after the wakeup has finished. We also added code to calculate the suspend overhead and the wake-up overhead and output them in the number of cycles – this is just subtracting the read PIT value from the 0xFFFFFFFF initialized value. These results are detailed in the 'Results' section.

Additionally, we tried 'standby' mode via the proc interface, but it would never wake-up. To ensure that our code did not affect how this is not waking-up, we reverted to the original kernel without our code and tried to set the pm_alarm to wakeup after suspending in standby mode. This still did not work. Tried with the U63 button as well but still did not work.

4.5 Experiment 5 Notes

We integrated a sleep algorithm with a simulated real-time EDF scheduler. We modified code from hw3 and used the following algorithm. When an idle task is detected, we calculated how much idle time we would have. If more than the overhead associated with a suspend, we will enter the suspend mode, else if more than the overhead associated with a clock-suspend, we will enter the clock-suspend mode, else we would not enter any sleep mode. This algorithm only applies to periodic tasks since with aperiodic tasks, we would not be able to determine when the next task will get scheduled. The source code and outputs from the program are attached with our report.

5 Future work

Future work would include modifying an existing real-time scheduler to use a sleep mode and port our simulated algorithm to execute on the 405LP platform. However, the current hardware has a limitation in terms of the granularity that can be specified for wakeups. Another direction for future work would be to extend DVS schedulers that use synchronous voltage switching. Since no instructions are executed during this time, it may be beneficial to sleep while waiting for the switch to complete. The logic to implement this would include estimating the time to switch frequencies and then setting the wake-up alarm based on this estimation. Then after initiating the voltage change, the system can enter sleep mode. Upon wake-up, the frequency has already been switched and the system can continue processing tasks. In order for this to be possible, we must be able to set the alarm in increments less than a second, preferably microseconds.

6 References

1. "Feedback EDF Scheduling Exploiting Hardware-Assisted Asynchronous Dynamic Voltage Scaling" by Y. Zhu and F. Mueller in ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05), Jun 2005, pages 203-212.
2. H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. IEEE Trans. Comput., 53(5):584–600, 2004.
3. B. Brock and K. Rajamani. Dynamic power management for embedded systems. In IEEE International SOC Conference, Sept. 2003.
4. A. Dudani, F. Mueller, and Y. Zhu. Energy-conserving feedback edf scheduling for embedded systems with realtime constraints. In ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02), pages 213-222, June 2002.
5. IBM and MontaVisa Software. Dynamic power management for embedded systems. white paper.
6. K. Nowka, G. Carpenter, and B. Brock. The design and application of the powerpc 405lp energy-ef_cient system on chip. *IBM Journal of Research and Development*, 47(5/6), September/November 2003.
7. Y. Zhu and F. Mueller. Feedback edf scheduling exploiting dynamic voltage scaling. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 84.93, May 2004.
8. P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Symposium on Operating Systems Principles*, 2001. K. Govil, E. Chan, and H.Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *1st Int'l Conference on Mobile Computing and Networking*, Nov 1995.
9. R. Minerick, V. W. Freeh, and P. M. Kogge. Dynamic power management using feedback. In *Proceedings of Workshop on Compilers and Operating Systems for Low Power*, 2002.
10. PowerPC 405LP user manual.