Charakterisierung der Leistungsaufnahme von mobilen Geräten im laufenden Betrieb

Studienarbeit im Fach Informatik

vorgelegt von
Florian E.J. Fruth
geboren am 26. Januar 1979 in Schillingsfürst

Institut für Informatik,
Lehrstuhl für Verteilte Systeme und Betriebssysteme,
Friedrich Alexander Universität Erlangen-Nürnberg

Betreuer:	Dipl.-Inf. Andreas Weißel
	Prof. Dr. Wolfgang Schröder-Preikschat

Beginn der Arbeit:	01. November 2004
Abgabedatum:	11. April 2005

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.

Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 01. Mai 2005

Run-time Energy Characterization of the Intel PXA

Student Thesis

by

Florian E.J. Fruth

born January 26th, 1979, in Schillingsfürst

Department of Computer Science,
Distributed Systems and Operating Systems,
University of Erlangen-Nuremberg

Advisors:	Dipl.-Inf. Andreas Weißel
	Prof. Dr. Wolfgang Schröder-Preikschat

Begin:	November 1st, 2004
Submission:	April 11th, 2005

Abstract

An important issue of mobile devices is energy consumption. There are different approaches on how to optimize the available energy. One method is to reduce the CPU frequency to save power. Other methods include scheduling strategies and if possible the complete shutdown of hardware components.

For power management, energy accounting, battery lifetime estimation and similar approaches it is necessary to measure how much energy was consumed before and after the changes. The measurement can be done with external hardware. In cases when there is no developer board available it can be difficult to measure the consumed energy, e.g. if you want to do tests on a mobile phone or organizer with embedded processors.

Therefore this thesis is an approach to provide energy consumption estimation without the need for external measurement hardware. Exemplary the Intel PXA 255 architecture will be used. It has two performance counters which make it possible to estimate the energy. The first part of this work is to get an energy consumption per event depending on the current processor and RAM frequency. The Intel PXA 255 processor supports different CPU and memory speeds. As a consequence there are different internal CPU voltage settings. So it is necessary to map different events to energy consumptions depending on the current CPU and memory frequencies.

There are only two performance counters and a variety of different applications with different usage of memory, processor and I/O. This thesis will show that two performance counters are not sufficient to estimate all types of programs in an accurate way but it is possible to train the energy weights for representing specific applications in an accurate way. It also evaluates the approach to switch the performance counters during runtime. This multiplexing technique gives a virtual view of more counters. The acquired results lead to the conclusion that the performance counter multiplexing is not sufficient to represent all types of applications at once but for a larger subset of applications compared to only two counters. For example it was possible to get an average estimation error below 2% with three tested real world applications.

Contents

Chapter 1

Introduction

1.1 Overview

For mobile devices the amount of energy usage is important. There are different approaches to maximize the device uptime, battery lifetime or the performance. For these methods of resolution it is necessary to measure the energy consumed by the mobile device.

Naturally it is done by external measurement devices. This means extra costs for the measurement equipment and this is not always possible, e.g., if there is no developer board available or the form factor of the device under test does not allow measurement hardware to be attached. Additionally if the results are needed for run-time purposes it is necessary to get them back inside the device and an interface between the operating system and the measurement hardware has to be established. This thesis shows a way to avoid the need for extra measurement hardware in specific situations. It is an approach to use the build-in performance counters of the Intel PXA 255 processor for energy estimation. These performance counters provide a way to measure two out of 14 different events at the same time.

The first part of this thesis shows how to assign different amounts of energy to the triggered events. We will show that with the drawback of only two performance counters included in the PXA processor it is difficult to measure different types of applications and give an exact energy estimation. To unmake this drawback it is possible to use the two performance counters to count different events by switching the triggered events during run-time. This makes it possible to measure more than two events with the drawback that there is overhead for switching the counters.

The second part of this thesis is the evaluation of the measurements. It examines if it is better to trigger two static events or switch them on run-time. It also shows the error estimation of the different methods and compares the results of the test programs with benchmarks and real world applications. This gives a clue that if you trigger the right events for the right application the energy consumption estimation can be more accurate. But if you trigger the wrong events (e.g. memory access for an application which uses only registers and many ALU operations) the estimation can be improper. This thesis will also investigate if it is possible to train the energy weights that they represent specific applications in an accurate way.

1.2 Related Work

There are two theses which are closely related to this one. The first is *Accounting and control of power consumption in energy-aware operating systems*[17] from Martin Waitz. It introduces an accounting model based on resource containers. Resource containers represent an accounting model of used resources. It is a model which makes it possible to always charge the party that is responsible for resource usage. Waitz uses the performance counters of the Intel Pentium 4 architecture for measuring the consumed energy. This was realized with a Linux kernel patch. The advantage of the resource containers patch is that it provides an easy way of monitoring the performance counters on a per container basis. So it is possible to determine which application used how much energy. This thesis is based on the Intel PXA 255 processor on an Intel XScale developer board. Ka-Ro electronics[6] patched the Linux kernel 2.6 to make it work on the Intel XScale board. So the patch from Martin Waitz was ported to the patched Ka-Ro kernel. This provides the possibility to use the resource containers for energy consumption on a per process basis together with the PXA 255 processor.

The second related thesis is *Event-driven temperature control in operating systems*[8] from Simon Kellner. He used the performance counters of the Intel Pentium 4 processor to estimate the energy consumption. Based on this power usage estimation he evaluates the temperature of the processor. The difference from his and my work is that he had more than two performance counters. Additionally the Intel PXA processor is able to work with different clock speeds. This is another aspect of estimating the power consumption because a different clock speed also means a different voltage supply for the processor. And so the event counters do not always represent the same amount of energy. Instead a triggered counter presents an amount of energy based on the current clock speed while keeping in mind that the clock speed may be changed at run-time.

Furthermore there is the work of Russ Joseph and M. Martonosi[5] who used the performance counters for an energy estimation on an Alpha 21264 Simulator and an Intel Pentium Pro. They show that performance counters are meant to measure performance and not energy. To get accurate results it would be better to have counters which were meant to measure energy, e.g. they address the lack of register and memory usage counters. In the Alpha 21264 simulation they used a heuristic model for mapping the existing performance counters to represent energy counters. They also used a multiplexing technique to measure more than the two existing performance counters. Their results show that directly measurable events give errors around 5% while counters which use their heuristics can lead to errors above 20%. This especially happens with floating point operations and an nondeterministic use of load and store operations. Their tests on a real Pentium II architecture give errors up to 15%.

Bellosa[2] shows that for an accurate online energy consumption estimation it is necessary to use extra hardware which represents the current usage of processor and memory. He investigates if the performance counters are suitable enough for this task. He uses an Intel Pentium III processor for power usage estimation with a modified version of PAPI[9]. The Pentium III processor also supports only two performance counters which gives errors within 15% of the real usage. To analyse if more performance counters would reduce the error the test programs were rerun several times while measuring different counter types. Five counters were enough to reduce the error ratio below 5%. The problem were especially counters which represent the usage of different computer components. For example if a load instruction occurs it is possible that the data is stored in the first level cache. This means that one event only represents a first level cache read and register store. On the other hand if the data is not cached but stored in the RAM this needs much more energy. Bellosa also analyzed the memory energy consumption. Different types of memory show different energy consumption but he showed that it is possible to use only one counter to estimate the energy usage within a 3% error. His conclusion is that if there were more counters an accurate measurement would be possible. And he suggests that not only the processor but all hardware components should implement performance counters. This would make it possible to get a very accurate energy model.

There are several projects for reading and accounting the performance counters. The first example is PAPI[9]. It provides an interface for multiple architectures including x86, IA-64 and Alpha. It consists of a kernel and an userspace part. The kernel part is responsible for reading and accounting the events depending on the used architecture. The user space part provides an API which is independent from the hardware architecture. That makes it easy to port userspace programs to different architectures which are supported by PAPI. It also includes the possibility of multiplexing performance counters in hardware and if this is not possible also in software.

Other performance counter tools include perfmon[15] and different versions derived from it. perfmon also consists of a kernel and an userspace part but only supports SPARC I/II and Intel Pentium I/II with Solaris. There is another version for the Itanium I/II[3] using Linux with kernel 2.4 and 2.6. perfmon does not support performance counter multiplexing but there is an extension called vEC[7]. vEC is based on perfmon and supports multiplexing which was used for memory system energy estimates.

Chapter 2

Test Case

2.1 Basic Setup

The basic setup can be split into the external and internal parts. The external part is represented by figure 2.1. It shows the Intel XScale Developer Board with an Intel PXA 255 processor[4] which was used for the tests in this thesis. To measure the energy consumption the measurement device NI SC-2345[11] was connected to the 3.3V voltage wire. For data acquisition the SC-2345 was connected to a PCI card of the PC. The PC runs National Instruments Labview[10] for data acquisition. Figure 2.2 shows a screenshot of Labview. It shows which components were used and how they were put together. The main component is the *DAQ assistant* which is used to setup the measurement details. It contains what should be measured, scaling factors, time intervals and other variables. The output of the DAQ assistant is connected to the *signal graph* and the *write lvm-files* module. The write lvm-files module writes the measured data to a text file. These files include a header and a tabular separated list of time and measured value pairs. The signal graph output is shown in figure 2.3 where the x-axis represents the time in seconds and the y axis the used power in watt. Labview provides an easy graphical user interface for a graphical investigation of the mea-

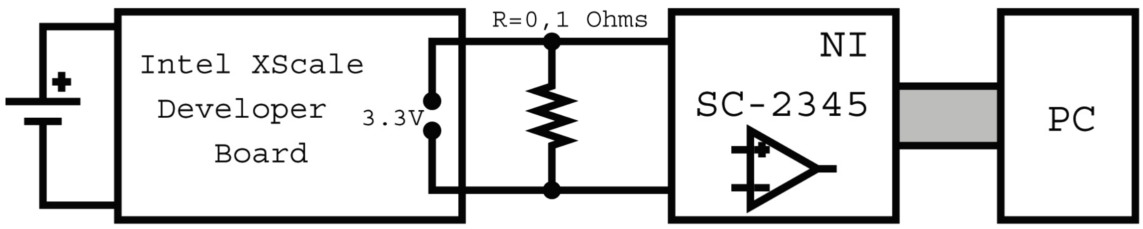

Figure 2.1: The test setup

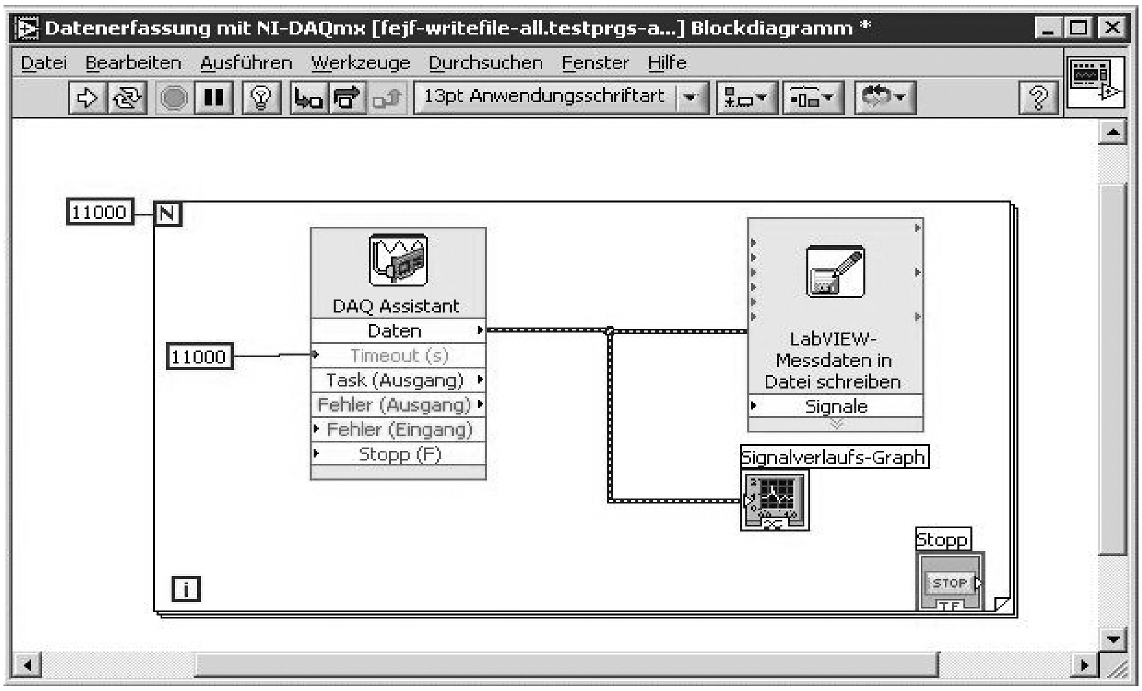
Datenerfassung mit NI-DAQmx [fejf-writefile-all.testprgs-a...] Blockdiagramm *
Datei Bearbeiten Ausführen Werkzeuge Durchsuchen Fenster Hilfe
13pt Anwendungsschriftart
11000
N
DAQ Assistant
Daten
Timeout (s)
Task (Ausgang)
Fehler (Ausgang)
Fehler (Eingang)
Stopp (F)
11000
LabVIEW-Messdaten in Datei schreiben
Signale
Signalverlaufs-Graph
Stopp
STOP
i

Figure 2.2: Screenshot of Labview

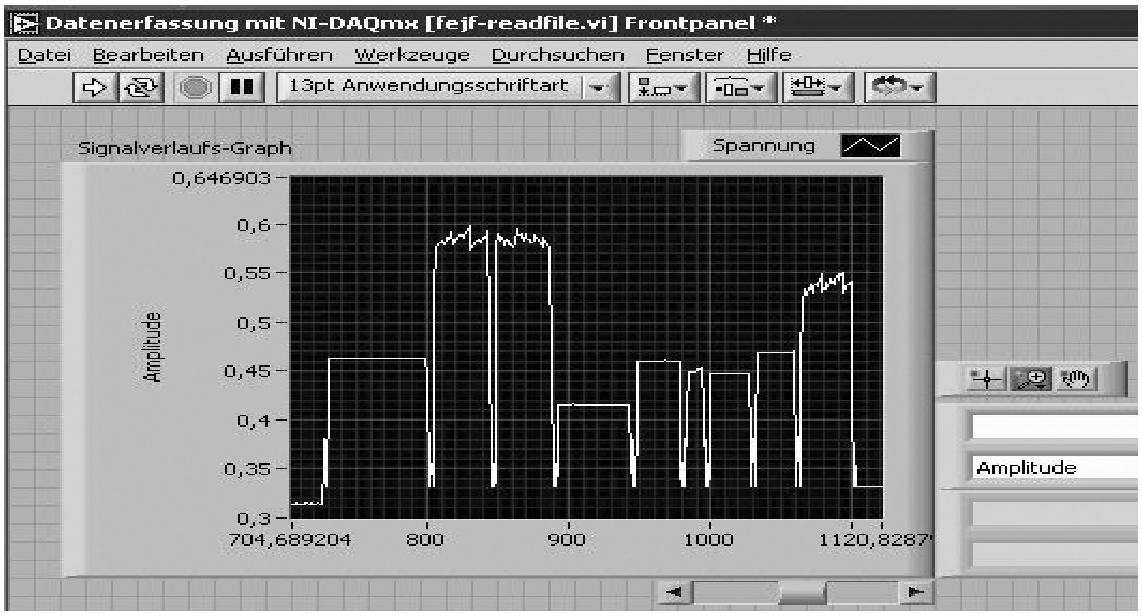
Datenerfassung mit NI-DAQmx [fejf-readfile.vi] Frontpanel *
Datei Bearbeiten Ausführen Werkzeuge Durchsuchen Fenster Hilfe
13pt Anwendungsschriftart
Signalverlaufs-Graph
Spannung
Amplitude
0,646903
0,6
0,55
0,5
0,45
0,4
0,35
0,3
704,689204
800
900
1000
1120,8287
Amplitude

Figure 2.3: Screenshot of Labview

sured data. For a better evaluation it is possible to write the data in lvm files. The first screenshot additionally shows two other aspects. The First one is setting the *timeout for the DAQ assistant*. The first measurement approach always failed after a few seconds because the default timeout is set to a very low value. The last module is the box around the others which represents a *for loop* currently set to a value of 11000 which represents the approx. three hours of measurement for the test applications if the *DAQ assistant* is set to measure one second intervals.

The consumed energy is

$$P_{CPU} = (U_{SUPPLY} - U_R) * \frac{U_R}{R}$$

The NI SC-2345 supports different measurement modules for different voltage ranges. In this setup the SCC AI-06 module[11] was used which supports input voltages from $\pm 10mV$. The output range is $\pm 10V$ which represents a gain of 100. This leads to the formula

$$P_{CPU} = (U_{SUPPLY} - \frac{U_{MEASURED}}{100}) \cdot \frac{U_{MEASURED}}{100 * R}$$

With a Resistor of $0,1\Omega$ the final approximation is

$$\Rightarrow P_{CPU} \approx 0.33 \cdot U_{MEASURED}$$

This value can be directly set in the NI Labview software which makes it possible to directly measure the power consumption.

The NI SC-2345 measures the voltage on the 3.3V power supply. This power supply is mainly used by the CPU and the ram. It would be also possible to measure the 12V power supply but as the performance counters only represent power consumption of the processor this would be more inaccurate.

The internal part can be summarized to a Linux 2.6.3 kernel modified by Ka-Ro electronics[6] to run on the Intel Developer Board. The kernel had to be extended to make it possible to setup and read the performance counters.

2.2 Problems with the Test Setup

There were three main problems with the measurement encountered during this thesis. The first one is that the whole test program measurement took about three hours. With a data rate of 50,000 samples per second this makes 540 million data records. The initial Labview setup stored all these records in the memory which didn't work because there was not enough memory to store the 11GB of data. So the setup was modified that it stores the data after each second while continuing the measurement. This try also failed. It leaded to gaps in the measurement. After some testing this problem was partly solved by terminating all other processes beside the Labview program on the Windows computer. As long as nobody does anything on the PC while the measurement is running it works. But something as simple as the screen saver could cause a gap again. This misbehavior could only be solved by starting the measurement and wait three hours to check if it was a successful measurement without gaps. A Perl script was written to check for gaps. It runs directly on the Windows computer with cygwin[12]. The cygwin port of GCC made it also possible to compile the program which sums up the energy on a per second basis for Windows.

The second problem were some random hangups. Sometimes after a measurement was finished it was not possible to start a new one. It seamed as the process which used the PCI measurement card still locked it. Trying to find the corresponding process in the Windows process table to shut it down gave no results. So the only solution was a Windows reboot. At least this hangups did not happen during measurements.

Another Problem is the automation of the measurement. The measurement was always started by hand for a predefined time interval. The start could be done by a remote desktop session. Together with a serial line connection to the developer board the complete measurement process could be handled via a remote connection. As the XScale board runs Linux it would have been also possible to login via ssh but that would have falsified the measurements more then the serial line connection. It would be much easier if it would have been possible to trigger it from inside the developer board. The XScale developer board has outputs and the SC-2345 supports digital inputs which should make this possible, but as some tries failed to accomplish this behavior it was easier to start and stop the measurement by hand. Also in mind that the focus of this thesis is not the setup of an automated measurement with Labview for the Intel XScale developer board but on the energy consumption estimation with performance counters. Another aspect is that the start and stop trigger signals for each test program could lead to other problems. For example when starting a measurement Labview sometimes took multiple seconds before actually recording data. This behavior was easily bypassed by the manual start because it is possible to see when Labview starts the data acquisition. After that start the test programs can be run without losing any data.

Chapter 3

Energy Estimation with Fixed Performance Counters

The Intel PXA 255 supports only two performance counters. This chapter shows how the estimation is done and how accurate it can be compared to the real consumption.

3.1 Implementation

The basis for the performance counter measurement was a Linux-2.6.3 kernel which was patched by Ka-Ro electronics[6] to make it run on the Intel XScale board with the PXA processor. This kernel was extended by the resource containers patch from Marin Waitz[17]. The main part was to replace the performance counter setup and read calls to fit the ARM architecture of the PXA processor. The port of the resource containers patch instead of a new implementation was made because the containers make it easy to account the performance counters and corresponding energy separately for each program. The resource containers also provide the possibility to group different programs to the same container or group one process to several containers. The structure is hierarchical which means the consumed energy is not only added to its corresponding container but also to its parents. The root of this hierarchy is the root container which stores the used energy of the whole system. Additionally a clock counter was added to the two performance counters to represent the base energy usage. The accounting of the resource containers is implemented by modifying the kernel scheduler and timer interrupt. The modified timer interrupt is responsible for the accounting and the scheduler was modified to recognize task switches.

3.2 Determination of the energy weights

To estimate the energy weights for the performance counters it is necessary to measure the real energy consumption and count how often the counters were triggered. These two values can be used to calculate the weights which are needed for an energy estimation.

3.2.1 Test programs

For an accurate estimation there were test programs written which trigger specific counters again and again. It is obvious that it's not possible to trigger only one event at a time. For example a branch instruction always additionally triggers an instruction executed, program counter change event and perhaps also some others. The task was to trigger mainly one specific event. It is also clear that some event counters such as instruction or data cache misses could not be triggered as often as others.
The test programs were

- level 1 cache reads
- level 1 cache writes
- level 1 cache reads together with writes
- memory reads
- memory writes
- memory reads together with writes
- ALU utilization with add statements
- program counter changes with branch instructions
- a mix of many counters realized with a factorization program

3.2.2 Analysis

The first view on the Labview data records showed further problems with the measurement. As figure 3.1 shows the average power usage at 100MHz is about 320mW. It also shows the first problem: there are unpredictable spikes from time to time which get up to 550mW. The second and even bigger problem can be seen in figure 3.2. There are steady energy fluctuations below the normal value in an interval of $\approx 0.0171s$. Deeper investigation also showed a little peek above the normal level following the lower peeks by $\approx 0.0079s$. The first idea was that this is caused by the power supply which converts the 230V AC to 12V DC. But switching the power converter to another model did not change this behavior. Another sign that the power supply might not be the cause is that the AC has a frequency of 50Hz. This means that the interval would be $0.02s$. Another problem may be the conversion from 12V to 3.3V.

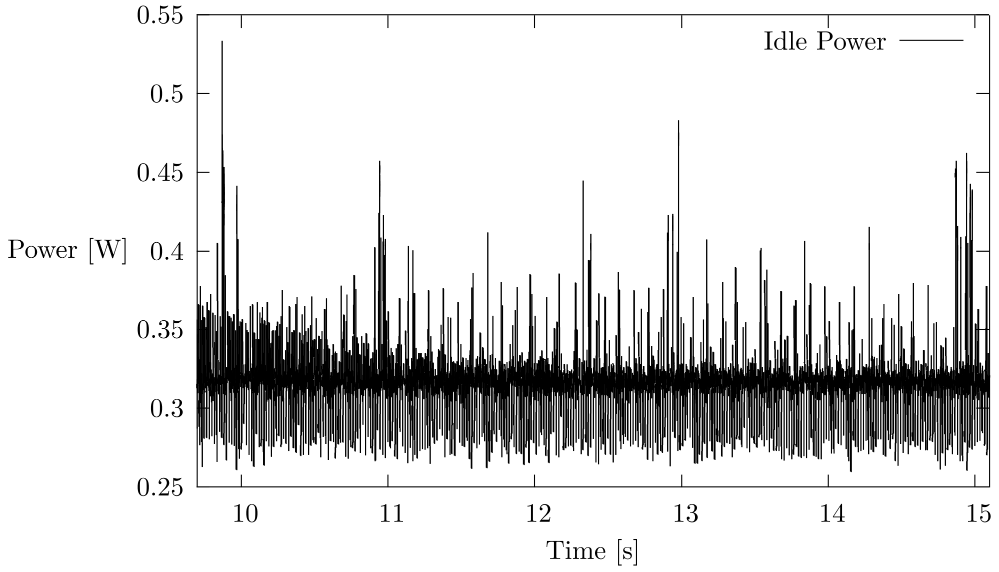

Figure 3.1: Energy Spikes

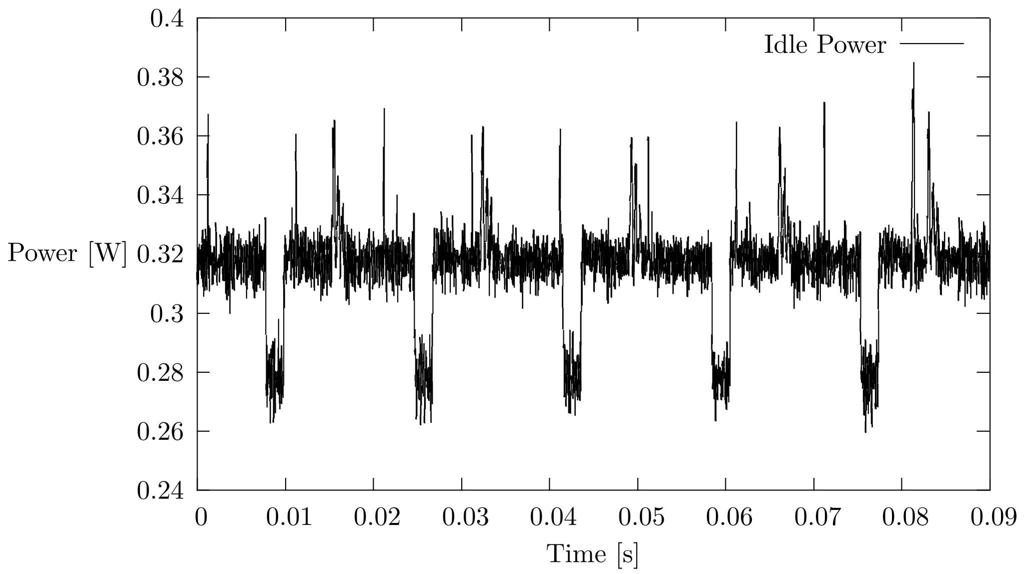

Figure 3.2: Short Energy Fluctuations

3.2.3 Toolchain

The measured values of the energy usage and the number of events triggered during the three hours of one test run had to be prepared for the energy weight estimation. As the Labview data for this time period was around 11GB of data it was impossible to split it to pieces by hand. In addition to this the test programs should be run multiple times to give more accurate results leading to even more data.

Three Perl scripts edited the data to get the energy weights. The first one splits the 11GB of measurement data into pieces corresponding to the several test programs. The problem shown in figure 3.2 made this more complicated then it had to be. The easy approach that an energy around the idle level means idle and everything above it represents a test program was useless because the energy fluctuations which happened while a test was running went below the idle energy level. The same with the energy peeks in idle mode which were above the work energy level. The solution for this problem was a hysteresis like energy level detection.

The second Perl script is a wrapper for a C program which accounts the energy of each produced log piece. Finally a last Perl script used this data in conjunction with the measured event counters.

As most of the tools were written in Perl and only the energy calculation was written in C it was possible to run these tools on Windows and Linux. First the tools were run on the measurement computer on Windows. But as the calculation also took about two hours it was better to copy the 11GB of data to a computer with a better performance running Linux. During the calculation process on the Linux computer it was possible to run the next test on the Windows computer.

The sophisticated part was to use the gathered data to calculate the actual weights. That topic is discussed in the next section of this thesis.

3.2.4 Energy Weight Calculation

The Energy weight calculation can be represented by solving the equation

$$counter_1 * weight_1 + counter_2 * weight_2 + \cdots + counter_n * weight_n = energy$$

where n is the number of available performance counter types. In the case of the Intel PXA 255 processor ($n = 14$). This equation has to be solved for each program. The counter values can be determined by running each programs with each counter. As there are two counters this means that every program has to be run seven times with different counter pairs to get all counter values. Now the problem is that at run time it is not possible to measure all 14 counters but only two and the clock counter. This simplifies the equation to

$$perfcounter_1 * weight_1 + perfcounter_2 * weight_2 + clockcounter * weight_3 = energy$$

Determining the clock counter is easy. The clock counter should represent the base energy that is always used even if the processor does nothing. It is specified by measuring how much energy is used if the processor is idle. Then divide the measured energy by the clock counter.

$$weight_3 = \frac{energy_{idle}}{clockcounter}$$

The calculation of *weight*$_1$ and *weight*$_2$ can be easily solved for each program itself. The difficult part is to find weights which match for every program with the least possible deviation. This extends the problem to find a solution for

$$\begin{bmatrix} counter_{1,1} & counter_{2,1} & clockcounter_1 \\ \vdots & \vdots & \vdots \\ counter_{1,j} & counter_{2,j} & clockcounter_j \end{bmatrix} * \begin{pmatrix} weight_1 \\ weight_2 \\ clockweight \end{pmatrix} = \begin{pmatrix} energy_1 \\ \vdots \\ energy_j \end{pmatrix}$$

where j is the number of programs. It is obvious that this equation could not be solved but it can be demanded to find values which minimize the total error. The following table lists all possible performance counter types for which these equations had to be solved.

Counter	Description
0x00	instruction cache miss
0x01	instruction cache can not deliver
0x02	data dependency stall
0x03	instruction tlb miss
0x04	data tlb miss
0x05	branch instruction executed
0x06	branch mispredicted
0x07	instruction executed
0x08	data cache full stall (every cycle)
0x09	data cache full stall (only first occurrence)
0x0A	data cache accesses
0x0B	data cache misses
0x0C	data cache write-back
0x0D	software changed the PC

The first approach to get a solution was to use the same programs that Simon Kellner[8] did. Several approaches to get plausible values failed. lrslib[1] and dgels (part of netlib[14]) always calculated some values below zero. There was no way to tell these programs to only search for positive values. From the mathematical point of view this behavior is intentional because it is obvious that there is a bigger change of minimizing the overall deviation if you use positive an negative values. The third program which is also part of netlib was dqed. dqed also calculated negative weights but luckily it has the option to set upper and lower boundaries for the results. Unluckily if setting the lower boundaries to a non-negative value represented by zero did not give the expected results. When setting a lower boundary to zero it changed every value which was below zero before to exactly zero.

As the equation to solve contains only two unknown variables, an iteration over an appropriate range should be feasible with only little overhead. From early appraisals it was known that the weights would be between about 10^{-12} and 10^{-6}. That means 10^{12} passes for each counter pair. After 24 hours without the first result it was obvious that this approach had to be improved. After multiple rounds of improvements the algorithm was able to calculate the needed weights in about 60 seconds. These improvements include

- only calculate the weights for energy pairs which were determined to give reasonable results
- check if the initial counter weights are big enough that the target energy can be reached with weights between the predefined range
- check if the initial counter weights are small enough that the target energy can be reached with weights between the predefined range
- instead of searching through the complete range in steps of 10^{-12} first use steps of size 10^{-10} and if the value is near a minimum error refine the step up to 10^{-12}
- if the minimum for $weight_2$ was passed continue with the next value for $weight_1$ instead of testing all possible values for $weight_2$
- instead of calculating the equation for every program with each step the counter values and energies are summed up first and only one addition plus one multiplication has to be done per step

3.2.5 Clock Counter Problems

Kellner ran into the problem that the processor still consumes power even if it is idle. This is the same with the PXA255 processor. His approach to give the clock counter an energy weight which represents the idle consumption was easily added to the kernel. Just to identify an important difference between the Pentium 4 processor he used and the PXA 255 processor used for this thesis: The PXA processor is targeted for mobile battery supply. So the main task is to save power. To achieve this objective the processor tries to do as much idle cycles as possible. Now the problem is that the built-in clock counter does not count the idle cycles as clock cycle. This means that this approach did not change anything on the idle consumption estimation error. Observing the system time getting changed even if there were many idle cycles it was apparent that there must be a way to get an accurate time counter. After some approaches failed when using the time in seconds, the kernel jiffies and the clock from the developer board the use of the kernel nano seconds counter was successful.

3.2.6 Determined weights

The only reasonable counter pairs for an accurate estimation were determined to be:

Counter	Description
0x02	data dependency stall
0x05	branch instruction executed
0x07	instruction executed
0x0A	data cache accesses

The other performance counters were not triggered often enough which means it needs an unreasonable high weight to get the total energy. For example one data write back event presents a unpredictable amount of data cache reads and writes. This makes it difficult to foretell how much data cache writes are represented by one data cache write back event. It is better to compensate the write back energy usage by a slightly too big data cache access weight.

The calculated values for the optimum weights for the set of test programs is shown in the following table. It shows the calculated values in [nJ] for the different frequencies and combinations.

	100MHz	200MHz	300MHz	400MHz
0x02	1759	1572	2390	2086
0x05	4120	3310	1400	5310
0x02	1970	1711	1930	1082
0x07	430	350	420	1250
0x02	1221	1666	1876	1157
0x0A	2120	1000	1200	3220
0x05	350	7500	8500	11800
0x07	1320	300	330	340
0x05	1361	3312	4843	8041
0x0A	3200	2230	2140	2200
0x07	165	810	515	212
0x0A	3200	1200	2430	4430
Clock	314569	332394	364850	387110

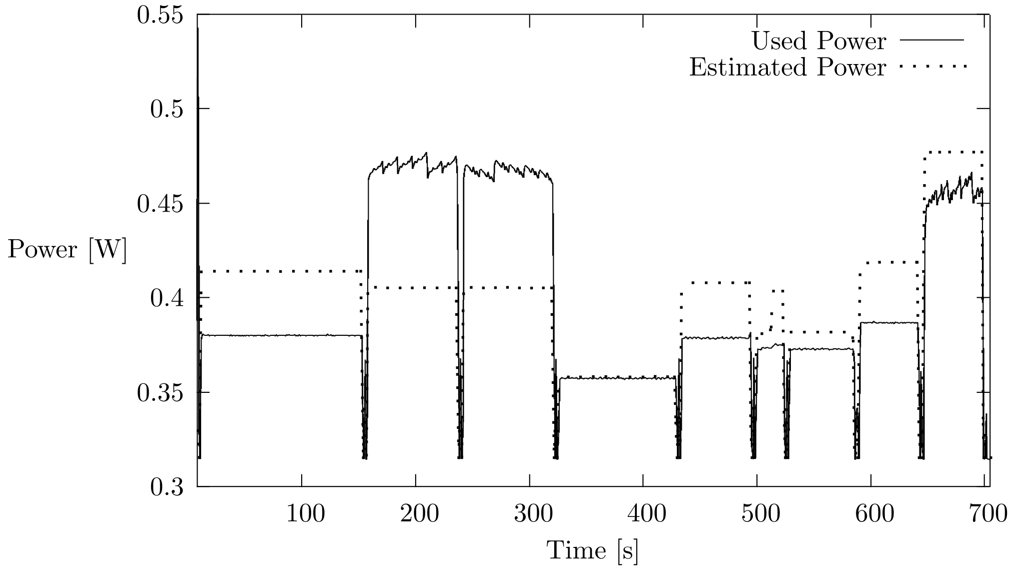

Figure 3.3: Test Programs with Event Counters 0x02 and 0x07

It shows that for different counter pairs the same event got different weights. This is due to the lack of more counters. As there are only two these two must also represent the other events. For example if a branch instruction is executed this also means ”instruction executed” and ”program counter changed” events. In case of using the counter pair 0x05 (branches) and 0x07 (instructions) the branch weight does not need to contain the energy of the instruction itself. In the other cases it does.

3.3 Evaluation

The energy weights have been determined. It is now necessary to examine how exact they are. As a first check the test programs which were used to calculate the weights are run again. The Labview measurement part is the same. The difference is that the kernel does not only count the events triggered by the programs but also calculates the estimated energy usage. A program reads the estimated energy every second. It calculated the difference to the value measured before and so it was possible to have a log which showed the estimated energy for each second. The data from the Labview log was also added up to show the energy consumption on a per second basis. Now these two logs could be compared to show how accurate the energy usage estimation is.

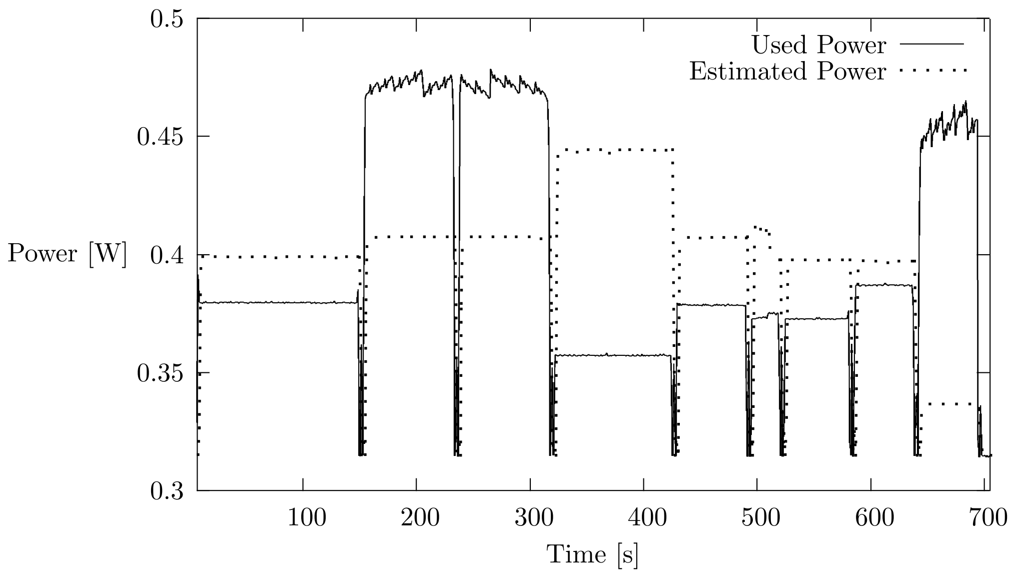

Figure 3.4: Test Programs with Event Counters 0x05 and 0x07

3.3.1 Test Programs

The figures 3.3 and 3.4 show the results with the counter pairs 0x07 (instructions executed) and 0x02 (data dependency stall) respectively 0x05 (branch instructions executed). Each high-level area is one test program. Figure 3.3 points out that the weights are nearly perfect for the accumulator program (fourth plateau). Though it also shows that the estimation may be below or above the real consumption for different program types because the used counter types are not optimal for all of these applications.

Together with figure 3.4 the last program (memory writes) shows significant differences on accuracy. This is obvious because memory writes tend to cause more data dependency stalls and less branch instructions. So the 0x02 and 0x07 counter pair represents programs with many data accesses in a better way.

Examining the 0x07 and 0x0A counters results gives another interesting aspect. The add program there shows a too low estimated power. So there are 3 different states of the estimation depending on the used counters. 0x02 gives the exact energy, 0x05 a energy which is too high and 0x0A one that is too low. The same behavior can be noticed with the memory writes. They are too low for the 0x07 and 0x05 pair and twice to high with the other combinations. This is an indication that the performance counter multiplexing which is described in the second part of this thesis might be an improvement for this type of applications. Despite of this the values for the level 1 cache reads were always a bit too high and for writes too low.

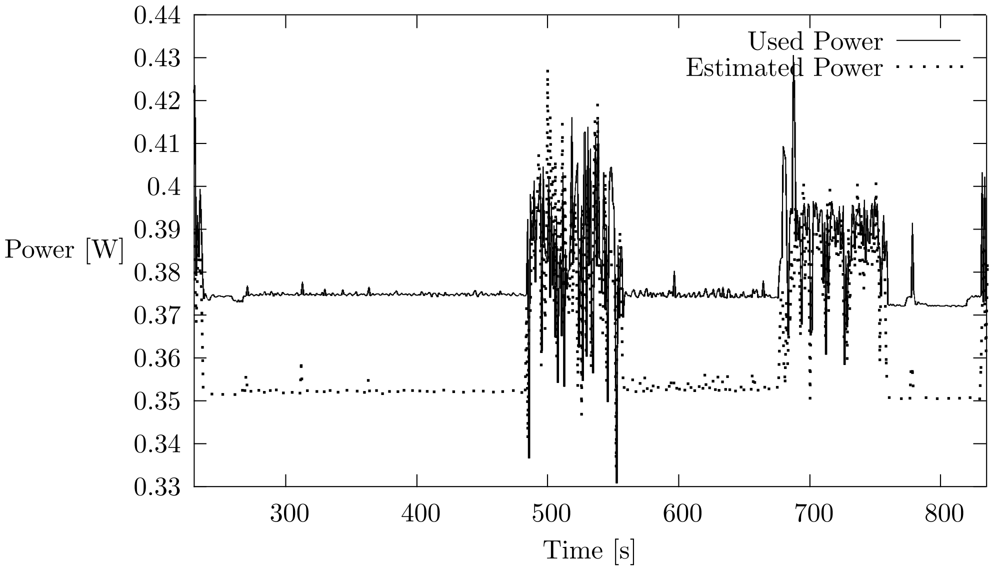

Figure 3.5: Event Counters 0x02 and 0x07

3.3.2 Benchmarks

The next step of evaluation was to run the benchmarks mibench[13] and unixbench[16]. Mibench is a free, commercially representative embedded benchmark suite. It consists of six categories of benchmark types. These categories include up to eight benchmarks. So this benchmark includes different micro benchmarks which use different components of the processor and other external hardware like I/O operations. The complete list of benchmarks included in mibench and unixbench are shown in the following tables.

mibench					
Industrial	Consumer	Office	Network	Security	Telecomm
basicmath	jpeg	ghostscript	dijkstra	blowfish enc.	CRC32
bitcount	lame	ispell	patricia	blowfish dec.	FFT
qsort	mad	rsynth	CRC32	pgp sign	IFFT
susan (edges)	tiff2bw	sphinx	sha	pgp verify	ADPCM enc.
susan (corners)	tiff2rgba	stringsearch	blowfish	rijndael enc.	ADPCM dec.
susan (smoothing)	tiffdither			rijndael dec.	GSM enc.
	tiffmedian			sha	GSM dec.
	typeset				

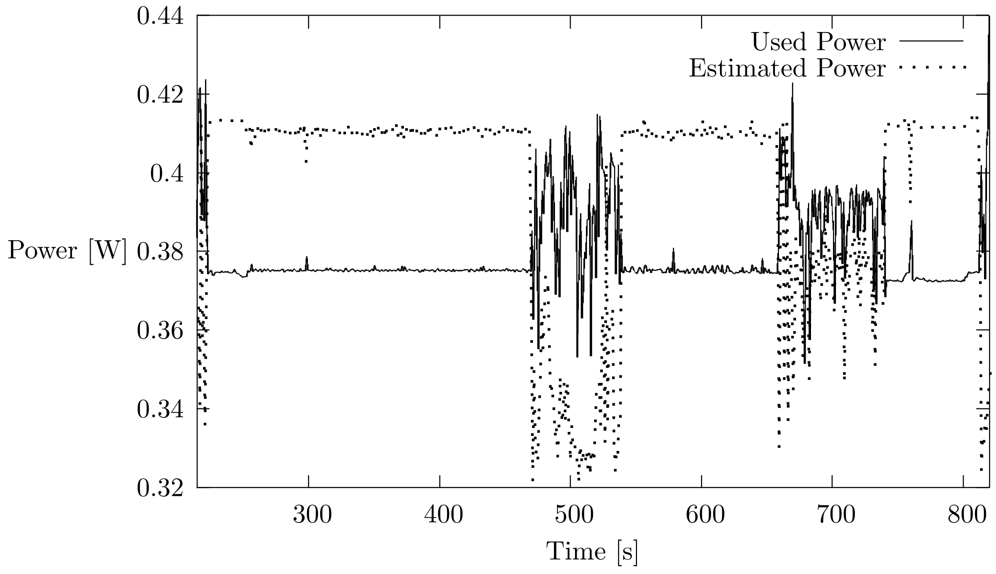

Figure 3.6: Event Counters 0x07 and 0x0A

unixbench			
arithmetic	system	misc	dhry
double-prec. whetstone	system call overhead	compile and link	dhrystone 2 w/o regs
arithmetic overhead	pipe throughput	calculations with dc	dhrystone 2 with regs
register arithmetic	pipe context switch	recursion	
short arithmetic	process creation		
int arithmetic	execl call		
long arithmetic	filesystem throughput		
float arithmetic			
double arithmetic			

Three test programs of the mibench suite failed to complete their execution by giving an *Illegal instruction* error. This includes ghostscript, tiff2bw and tiff2rgba. The result of the mibench run is shown in figures 3.5 and 3.6. There are three flat areas and two areas with many changes. The three flat area represent the run of lame, rsynth and FFT. The areas with the changes correspond to the programs between these three programs. These figures substantiate that the multiplexing might get correct results for this benchmark while the results for unixbench were not so promising. The cause may be that mibench mainly uses the processor and unixbench also performs many I/O operations for benchmarking.

3.3.3 Real World Programs

For further evaluation some real world programs have been tested. This includes:

- Gnu C Compiler v2.95.4 to compile the test programs used before
- GZip v1.3.2 to compress and decompress a 10MB random data file
- pdfTeX (Web2C 7.3.7) 3.14159-1.00a-pretest-20011114-ojmw to convert a small latex file to pdf

The real world programs results were bad news. Nearly every combination of performance counters gave values which were too low. Half of the tested combinations showed estimated values which differ from the real usage by over 25%. This showed that the test programs and benchmarks do not represent the real energy consumption of normal programs. The main problem is that the real world programs do not mainly use the CPU but also other components of the board. This includes memory accesses which in cooperation with the data caches give an nondeterministic behavior.

A possible solution for this problem might be to exchange the initial basic test programs by real world applications and rerun the measurement and counter calculations again. But nevertheless it is assumed that the test programs and benchmarks than will be wrong. It is also feasible that the event counter multiplexing will change this situation. Another aspect is that it is possible to train the energy weights with a subset of real world programs. For example if the weights are calculated on base of GCC test runs the errors should decrease when using these weights for measuring other program compilations with GCC.

3.3.4 Table of Estimation Errors and Conclusions

All tested programs with all used counter pairs at four frequencies give the following errors in %. A positive value means that the estimated energy was too high, a negative represents a too low value. The **real world average** row is the average of the three real world programs GCC, GZip and pdfTeX which is calculated by the following formula.

$$error(realworldavg) = \frac{error(GCC) + error(GZip) + error(pdfTeX)}{3}$$

100MHz	0x02-0x05	0x02-0x07	0x02-0x0A	0x05-0x07	0x05-0x0A	0x07-0x0A
L1 read	7.3%	7.9%	12.5%	4.8%	12.6%	13.9%
L1 rw	-12.1%	-14.5%	-8.3%	-13.3%	-5.2%	-4.9%
L1 write	-11.8%	-13.0%	-9.0%	-12.9%	-5.6%	-4.5%
add	-10.5%	0.1%	-10.3%	23.1%	-10.0%	-7.1%
branch	14.5%	6.9%	10.9%	7.5%	15.0%	14.2%
factor	1.3%	3.4%	3.4%	7.7%	3.5%	4.6%
mem read	6.8%	1.9%	3.1%	6.5%	6.3%	5.4%
mem rw	15.4%	7.2%	7.2%	2.5%	6.4%	5.2%
mem write	0.0%	3.8%	-6.0%	-24.4%	-23.3%	-21.7%
mibench	-4.5%	-4.5%	0.7%	-1.6%	4.8%	4.1%
unixbench	-2.6%	-2.9%	-3.1%	-4.8%	-4.4%	-4.3%
gcc	-7.0%	-3.8%	-9.0%	-19.7%	-15.0%	-15.7%
gzip	-6.4%	-5.1%	-8.7%	-20.0%	-17.1%	-16.8%
pdftex	-2.9%	-0.8%	-2.5%	-13.6%	-8.0%	-9.1%
real world avg	**-5.4%**	**-3.2%**	**-6.7%**	**-17.8%**	**-13.4%**	**-13.9%**

200MHz	0x02-0x05	0x02-0x07	0x02-0x0A	0x05-0x07	0x05-0x0A	0x07-0x0A
L1 read	7.0%	7.7%	11.7%	3.4%	13.3%	12.2%
L1 rw	-14.6%	-17.0%	-12.0%	-8.1%	-3.9%	-6.2%
L1 write	-14.6%	-17.1%	-12.6%	-8.8%	-4.7%	-7.7%
add	-17.1%	-3.3%	-16.7%	-4.9%	-15.8%	16.6%
branch	15.0%	5.2%	8.1%	32.2%	22.4%	14.8%
factor	2.0%	0.7%	2.9%	1.9%	4.8%	12.4%
mem read	5.7%	-1.3%	0.1%	20.9%	11.3%	8.1%
mem rw	18.6%	7.8%	11.4%	24.5%	10.0%	4.3%
mem write	10.4%	15.2%	13.6%	-30.7%	-29.8%	-28.6%
mibench	-6.9%	-7.0%	-4.0%	-5.2%	1.4%	-2.0%
unixbench	-2.2%	-2.2%	-6.8%	-4.8%	-4.7%	-4.8%
gcc	-5.2%	-1.1%	-1.3%	-21.5%	-15.5%	-22.6%
gzip	-4.7%	-2.8%	-2.6%	-26.7%	-24.3%	-24.1%
pdftex	0.2%	1.2%	3.0%	-18.7%	-12.7%	-12.0%
real world avg	**-3.2%**	**-0.9%**	**-0.3%**	**-22.3%**	**-17.5%**	**-19.6%**

300MHz	0x02-0x05	0x02-0x07	0x02-0x0A	0x05-0x07	0x05-0x0A	0x07-0x0A
L1 read	10.1%	11.2%	16.0%	5.1%	15.0%	22.7%
L1 rw	-19.6%	-19.1%	-14.0%	-9.5%	-5.1%	-0.7%
L1 write	-20.3%	-18.8%	-15.8%	-8.6%	-8.3%	-2.8%
add	-23.9%	-4.3%	-22.6%	-8.3%	-20.3%	0.6%
branch	9.2%	8.9%	10.2%	44.8%	29.3%	26.0%
factor	-5.3%	5.3%	0.8%	11.5%	1.5%	11.3%
mem read	-0.5%	-1.7%	-0.4%	25.5%	15.4%	8.1%
mem rw	19.6%	11.8%	15.0%	34.3%	13.4%	10.9%
mem write	27.9%	22.5%	15.2%	-35.9%	-35.7%	-32.3%
mibench	-10.6%	-10.0%	-5.7%	-8.0%	-0.9%	0.0%
unixbench	-1.4%	-2.1%	-2.1%	-5.3%	-4.8%	-4.9%
gcc	5.5%	-0.1%	0.0%	-25.2%	-17.5%	-26.6%
gzip	5.7%	-1.4%	-0.9%	-32.3%	-30.1%	-27.8%
pdftex	9.4%	2.3%	0.4%	-22.8%	-17.3%	-6.0%
real world avg	**6.9%**	**0.3%**	**-0.2%**	**-26.8%**	**-21.6%**	**-20.1%**

400MHz	0x02-0x05	0x02-0x07	0x02-0x0A	0x05-0x07	0x05-0x0A	0x07-0x0A
L1 read	2.3%	4.8%	18.6%	1.6%	11.5%	33.2%
L1 rw	-19.3%	-18.0%	-6.0%	-4.4%	-2.2%	13.5%
L1 write	-20.7%	-18.1%	-9.7%	-3.1%	-7.3%	9.8%
add	-31.9%	32.5%	-29.5%	-17.2%	-25.7%	-27.0%
branch	14.8%	6.5%	12.3%	57.1%	34.4%	35.7%
factor	-3.2%	5.3%	-0.8%	4.3%	14.2%	8.2%
mem read	4.5%	1.2%	0.6%	34.6%	21.6%	15.9%
mem rw	23.1%	3.8%	6.8%	44.8%	13.9%	16.2%
mem write	32.3%	10.1%	1.3%	47.7%	46.5%	8.6%
mibench	-11.8%	-11.6%	-1.7%	-12.1%	-5.7%	1.5%
unixbench	-2.2%	-27.1%	-26.2%	-28.1%	-28.5%	-28.9%
gcc	-1.0%	-6.4%	-6.6%	-26.3%	-7.9%	-30.8%
gzip	0.9%	-17.7%	-12.7%	-38.4%	-36.1%	-32.4%
pdftex	5.3%	-9.7%	-8.9%	-27.7%	-13.9%	-9.6%
real world avg	**1.7%**	**-11.3%**	**-9.4%**	**-30.8%**	**-19.3%**	**-24.3%**

The following conclusions can be made when investigating the table of errors.

- L1 cache reads get always too high values because the read and write accesses can not be measured on their own. So one data cache access represents the probability of a data cache read mixtured with a data cache write.
- L1 cache writes get too low values for the same reason. Additionally the L1 cache read and write program gets a too low value because it is more probable that data is read than written. So the program with an equal amount of reads and writes has a value estimated too low.
- The add program gets too low values with the data counters because it does unusual heavy CPU usage and low memory usage.
- The branch program gets a too high value. Paradoxically this especially happens when measuring the branch and instruction counter. This can be certified by the fact that these too counters need to represent all other counters in normal operation. But with the branch test program there are few other counters triggered which leads to a too high estimation.
- Factor gets a good average error for all counter pairs because it is a mixture of many of them
- The memory programs seam to be very unpredictable. These can be due to the available caches.
- Mibench and unixbench work very well because they are also a mixture of programs. As seen in the figures 3.5 and 3.6 the different programs even out the too low and too high estimations.
- The real world programs are nearly all estimated with a too low energy. This is because they heavily use the memory and there are no appropriate event counters.
- In general GZip gets higher error rates than GCC and pdfTeX. The cause could be that GZip needs to move more data in the memory and the other programs tend to mainly use the CPU.
- Column three in the 200MHz table shows that it is possible to get the real world estimation errors of all three programs within a 3% range. But it also shows that this involves higher errors for other applications.
- The average of the real world programs gives the lowest errors with the performance counter 0x02 (data dependency stall) and one of the others depending on the frequency.
- There is always one counter pair combination which gives an average error below 5% but there is no pair which is always the best.

- **Another issue seen in the table is that higher frequencies corresponds to higher errors. This may be due to the fact that the test programs need less time to finish and so the impreciseness can increase. Another possible cause is that a higher CPU frequency corresponds to a higher RAM frequency with a higher cache miss penalty. And as the RAM accesses can not be measured directly this impreciseness corresponds to a higher energy estimation error.**
- **Memory accesses correspond to higher error rates because they can not be measured directly.**
- **There is always one counter pair for each program with an error below 7%. This means it is possible to train two counters for specific applications but not for all possible program types at once.**

Another interesting issue is if it is possible to train the counters not only for a specific application but also a subset of applications. To investigate this additional tests were made. These tests were based on program compilations. Three test programs were chosen for the energy weight calculation: compilation of GZip, TCSH and WGet which included the execution of make, GCC, LibTool and others. The results showed that the counter pair 0x05 (branches) and 0x0B (data cache misses) had the smallest deviation. These weights were used to estimate the energy usage of the test programs and additionally the compilation of MC. To test if these counters are also feasible for other applications a GZip test run was also included. The estimation errors for 100MHz:

	make gzip	make tcsh	make wget	make mc	gzip file1	gzip file2
0x05-0x0B	5.9%	6.2%	6.6%	3.44%	-4.6%	-6.4%

These errors show that it is possible to train the energy weights for a subset of applications. The estimation errors are all within a +10% to -10% range around the measured energy. Additionally the GZip estimation is also in this range. The problem is that no counter pair was found which gives errors below 10% for all types of programs. Every type of program has a specific counter pair which is best to get an accurate estimation. The next chapter will show if it is possible to use the software multiplexing of the performance counters to extent the variety of applications that can be measured without calculating new energy weights.

Chapter 4

Energy Estimation with Alternating Performance Counters

This chapter shows a possible way to improve the energy consumption estimation. Bypassing the drawback of only two available performance counters is done by switching them in fixed time intervals. That makes it possible to not only count two events but more. The current patch allows a maximum of six events to be monitored. It is possible to change the maximum by modifying two single statements in the kernel.

With the evaluation I will investigate if this multiplexing technique gives a more accurate estimation.

4.1 Implementation

The implementation was done by changing the resource container account procedure in the kernel. Normally the kernel updates the resource containers every tick. A tick is generated by a timer interrupt. This timer interrupt is hardware dependent. For example for a IA-32 architecture the value is 1000 times per second. For the used ARM architecture it is 100 times per second. After the change the update procedure does not only store the counter usage since the last tick but also sets new counter pairs to be measured until the next tick occurs.

As stated above a limitation of six maximum counters was chosen. The reason is that more counters mean more overhead. Furthermore using more counter pairs also means that it takes more time till the first two counters are used again. It would be possible to use all 14 available counters but there are some counters which are not very feasible for the energy estimation used in this thesis. Using all 14 counters and solving the energy weight equation for it gives the possibility to implement another approach. This approach would be to calculate the weights and use seven different counter pairs which leads to seven time slices. After seven time slices all 14 counters were counted at least once. The problem with this approach is that it would take at least seven kernel ticks before an accurate estimation can be done and depending on the scheduler it is possible that one program is always represented by one counter pair. To prevent this behavior it would be necessary to store the last used counters for each process. This would mean additional overhead and a higher latency of the scheduler. That is the reason why this approach was not used.

4.2 Estimation

As there are still only two event counters present in hardware and the patched kernel only gives a virtual view of up to six counters, the estimation of the performance counter weights is the same as in the previous chapter. There are always two pairs of performance counters and the clock counter which represent the current use of energy. This makes it possible to read the energy estimation at any time and it is not necessary to wait till a time slice with all combinations of counter pairs passed.

There are multiple combinations of performance counters possible. For an deeper investigation the following three combinations of counters were chosen:

- 0x02 - 0x07 (data stall - instructions) alternating with 0x05 - 0x07 (branches - instructions)

Combination 0x02 - 0x07 and 0x05 - 0x07				
	100MHz	200MHz	300MHz	400MHz
0x02	1970	1711	1930	1082
0x07	430	350	420	1250
0x05	350	7500	8500	11800
0x07	1320	300	330	340
Clock	314569	332394	364850	387110

- 0x02 - 0x05 (data stall - branches) alternating with 0x07 - 0x0A (instructions - data cache misses)

Combination 0x02 - 0x05 and 0x07 - 0x0A				
	100MHz	200MHz	300MHz	400MHz
0x02	1736	1593	2390	2086
0x05	4300	3230	1400	5310
0x07	167	811	1296	1708
0x0A	3200	1200	340	400
Clock	314569	332394	364850	387110

- 0x02 - 0x07 (data stall - instructions) alternating with 0x05 - 0x07 (branches - instructions) and 0x0A - 0x07 (data cache misses - instructions)

0x02 - 0x07 and 0x05 - 0x07 and 0x0A - 0x07				
	100MHz	200MHz	300MHz	400MHz
0x02	1970	1711	1930	1082
0x07	430	350	420	1250
0x05	350	7500	8500	11800
0x07	1320	300	330	340
0x0A	3200	1200	340	400
0x07	167	811	1296	1708
Clock	314569	332394	364850	387110

4.3 Evaluation

To get a better impression on how much the accuracy has been improved respectively declined it was necessary to compare the errors. As there are many values which had to be compared it was easier to write a Perl script which does the work automatically. The script uses the table of errors from the non-multiplex error table, calculates an average and compares them with the multiplex error table.

The average error calculation is done in two ways. The first way is to take the pairs from the multiplex table and only use these pairs for the average error calculation. These values are shown in the *two/three pairs* columns of the error improvement table.

For example using the counter pairs 0x02-0x05 alternating with 0x07-0x0A leads to the average calculation formula:

$$averageerror = \frac{|error(0x02 - 0x05)| + |error(0x07 - 0x0A)|}{2}$$

This comparison is done to get an impression if the multiplexing technique improves the estimation by using the same weights for the same programs. This should be especially work very well if the two counter pairs had errors with opposite algebraic signs. This means if one counter pair produced a too high estimation and the other a too low estimation the multiplexing of these two should be nearly zero. Or when speaking of errors: a negative error and a positive error should add up to zero with the multiplexing. For example the add program at 200MHz had an error of -17.1% with the 0x02-0x05 counter pair and an error of +16.6% with the 0x07-0x0A counters. The multiplexing of these two counter pairs gives an error of -0.4% which is nearly the sum of the both values.

As second approach the six errors from the first estimation are used all together (*All Pairs* column). This gives a better overview of the total error.

$$averageerror = \frac{|error(0x02-0x05)|+|error(0x02-0x07)|+\ldots+|error(0x07-0x0A)|}{6}$$

It will show if it is better to stick with two counters or if it is wise to use the multiplexing scheme. As both ways provide possibilities of further improvement this should show the better one.

To give an overview of the results the table on the next two pages shows the errors with the multiplexed counters followed by the table of improvements and declines.

100MHz	0x02-0x05 X 0x07-0x0A	0x02-0x07 X 0x05-0x07	0x02-0x07 X 0x05-0x07 X 0x07-0x0A
L1 read	10.1%	6.6%	8.6%
L1 rw	-9.5%	-12.7%	-11.8%
L1 write	-8.4%	-13.1%	-11.3%
add	-8.3%	11.9%	5.9%
branch	14.1%	6.8%	9.1%
factor	3.0%	4.5%	4.6%
mem read	5.9%	4.0%	4.3%
mem rw	10.1%	4.9%	4.7%
mem write	-11.0%	-10.5%	-15.1%
mibench	0.2%	-2.9%	-0.4%
unixbench	-3.6%	-3.7%	-3.7%
gcc	-10.1%	-11.4%	-12.4%
gzip	-12.2%	-12.4%	-14.1%
pdftex	-4.6%	-5.5%	-7.3%
real world avg	**-9.0%**	**-9.8%**	**-11.3%**

200MHz	0x02-0x05 X 0x07-0x0A	0x02-0x07 X 0x05-0x07	0x02-0x07 X 0x05-0x07 X 0x07-0x0A
L1 read	8.7%	5.8%	6.6%
L1 rw	-11.3%	-13.5%	-13.8%
L1 write	-11.4%	-12.7%	-14.6%
add	-0.4%	-3.9%	2.5%
branch	13.8%	17.9%	14.7%
factor	2.9%	2.1%	-1.9%
mem read	6.1%	8.8%	4.4%
mem rw	10.5%	15.9%	9.8%
mem write	-8.4%	-7.9%	-16.4%
mibench	-4.3%	-5.9%	-4.4%
unixbench	-3.7%	-3.7%	-3.8%
gcc	-9.8%	-7.8%	-12.6%
gzip	-14.7%	-14.8%	-19.2%
pdftex	-8.6%	-8.2%	-9.6%
real world avg	**-11.0%**	**-10.3%**	**-13.8%**

300MHz	0x02-0x05 X 0x07-0x0A	0x02-0x07 X 0x05-0x07	0x02-0x07 X 0x05-0x07 X 0x07-0x0A
L1 read	10.3%	6.9%	5.4%
L1 rw	-16.4%	-16.0%	-15.2%
L1 write	-16.0%	-15.4%	-16.6%
add	8.2%	-6.7%	10.0%
branch	12.5%	23.0%	23.1%
factor	5.5%	-1.1%	10.8%
mem read	3.4%	9.6%	12.3%
mem rw	11.0%	20.6%	16.8%
mem write	-0.1%	-9.0%	-19.9%
mibench	-8.6%	-9.1%	-8.1%
unixbench	-3.4%	-4.1%	-4.0%
gcc	-5.4%	0.5%	-14.7%
gzip	-12.8%	-16.4%	-21.8%
pdftex	-7.6%	-9.4%	-10.1%
real world avg	**-8.6%**	**-8.4%**	**-15.5%**

400MHz	0x02-0x05 X 0x07-0x0A	0x02-0x07 X 0x05-0x07	0x02-0x07 X 0x05-0x07 X 0x07-0x0A
L1 read	5.4%	1.7%	4.0%
L1 rw	-14.6%	-15.0%	-13.1%
L1 write	-14.3%	-14.1%	-13.0%
add	10.9%	6.4%	25.8%
branch	17.1%	25.3%	29.5%
factor	4.5%	-6.0%	24.2%
mem read	7.6%	13.1%	13.2%
mem rw	10.2%	19.1%	15.9%
mem write	-16.4%	-13.2%	-15.3%
mibench	-11.2%	-12.7%	-11.4%
unixbench	-17.9%	-17.8%	-18.1%
gcc	-2.5%	0.1%	-11.9%
gzip	-18.4%	-26.5%	-31.3%
pdftex	-12.4%	-8.7%	-12.3%
real world avg	**-11.1%**	**-11.7%**	**-18.5%**

The previous table leads to the following table of improvements/declines.

100MHz	0x02-0x05 X 0x07-0x0A		0x02-0x07 X 0x05-0x07		0x02-0x07 X 0x05-0x07 X 0x07-0x0A	
	2 Pairs only	All Pairs	2 Pairs only	All Pairs	3 Pairs only	All Pairs
L1 read	0.5%	-0.3%	-0.2%	3.2%	0.3%	1.2%
L1 rw	-1.0%	0.2%	1.2%	-3.0%	-0.9%	-2.1%
L1 write	-0.2%	1.1%	-0.2%	-3.6%	-1.2%	-1.8%
add	0.5%	1.9%	-0.3%	-1.7%	4.2%	4.3%
branch	0.2%	-2.6%	0.4%	4.7%	0.4%	2.4%
factor	-0.1%	1.0%	1.0%	-0.5%	0.6%	-0.6%
mem read	0.2%	-0.9%	0.2%	1.0%	0.3%	0.7%
mem rw	0.2%	-2.8%	-0.1%	2.4%	0.3%	2.6%
mem write	-0.2%	2.2%	3.6%	2.7%	1.5%	-1.9%
mibench	4.1%	3.2%	0.1%	0.5%	3.0%	3.0%
unixbench	-0.1%	0.1%	0.1%	-0.0%	0.3%	-0.0%
gcc	1.2%	1.6%	0.3%	0.3%	0.7%	-0.7%
gzip	-0.6%	0.2%	0.2%	-0.0%	-0.1%	-1.7%
pdftex	1.4%	1.5%	1.7%	0.6%	0.5%	-1.2%
real world avg	**0.7%**	**1.1%**	**0.7%**	**0.3%**	**0.4%**	**-1.2%**

200MHz	0x02-0x05 X 0x07-0x0A		0x02-0x07 X 0x05-0x07		0x02-0x07 X 0x05-0x07 X 0x07-0x0A	
	2 Pairs only	All Pairs	2 Pairs only	All Pairs	3 Pairs only	All Pairs
L1 read	0.9%	0.5%	-0.2%	3.4%	1.2%	2.6%
L1 rw	-0.9%	-1.0%	-0.9%	-3.2%	-3.4%	-3.5%
L1 write	-0.2%	-0.5%	0.3%	-1.8%	-3.4%	-3.7%
add	16.5%	12.0%	0.2%	8.5%	5.8%	9.9%
branch	1.1%	2.5%	0.8%	-1.6%	2.7%	1.6%
factor	4.3%	1.2%	-0.8%	2.0%	3.1%	2.2%
mem read	0.8%	1.8%	2.3%	-0.9%	5.7%	3.5%
mem rw	1.0%	2.3%	0.2%	-3.1%	2.4%	3.0%
mem write	11.1%	13.0%	15.0%	13.5%	8.4%	5.0%
mibench	0.2%	0.1%	0.2%	-1.5%	0.3%	0.0%
unixbench	-0.2%	0.5%	-0.2%	0.5%	0.1%	0.5%
gcc	4.1%	1.4%	3.5%	3.4%	2.5%	-1.4%
gzip	-0.3%	-0.5%	-0.1%	-0.6%	-1.3%	-5.0%
pdftex	-2.5%	-0.6%	1.8%	-0.2%	1.0%	-1.6%
real world avg	**0.4%**	**0.1%**	**1.7%**	**0.9%**	**0.7%**	**-2.7%**

300MHz	0x02-0x05 X 0x07-0x0A		0x02-0x07 X 0x05-0x07		0x02-0x07 X 0x05-0x07 X 0x07-0x0A	
	2 Pairs only	All Pairs	2 Pairs only	All Pairs	3 Pairs only	All Pairs
L1 read	6.1%	3.0%	1.2%	6.4%	7.6%	7.9%
L1 rw	-6.2%	-5.1%	-1.7%	-4.7%	-5.4%	-3.9%
L1 write	-4.4%	-3.6%	-1.7%	-3.0%	-6.5%	-4.2%
add	4.1%	5.1%	-0.4%	6.6%	-5.6%	3.3%
branch	5.1%	8.9%	3.8%	-1.6%	3.5%	-1.7%
factor	2.8%	0.5%	7.3%	4.8%	-1.4%	-4.9%
mem read	0.9%	5.2%	4.0%	-1.0%	-0.5%	-3.7%
mem rw	4.2%	6.5%	2.4%	-3.1%	2.2%	0.7%
mem write	30.0%	28.1%	20.2%	19.2%	10.3%	8.4%
mibench	-3.3%	-2.7%	-0.1%	-3.2%	-2.1%	-2.2%
unixbench	-0.2%	0.0%	-0.4%	-0.7%	0.1%	-0.6%
gcc	10.7%	7.1%	12.2%	12.0%	2.6%	-2.2%
gzip	3.9%	3.6%	0.4%	-0.0%	-1.3%	-5.4%
pdftex	0.1%	2.1%	3.2%	0.3%	0.3%	-0.4%
real world avg	**4.9%**	**4.3%**	**5.3%**	**4.1%**	**0.5%**	**-2.7%**

400MHz	0x02-0x05 X 0x07-0x0A		0x02-0x07 X 0x05-0x07		0x02-0x07 X 0x05-0x07 X 0x07-0x0A	
	2 Pairs only	All Pairs	2 Pairs only	All Pairs	3 Pairs only	All Pairs
L1 read	12.3%	6.6%	1.5%	10.3%	9.2%	8.0%
L1 rw	1.8%	-4.0%	-3.8%	-4.4%	-1.1%	-2.5%
L1 write	0.9%	-2.8%	-3.5%	-2.6%	-2.7%	-1.5%
add	18.5%	16.4%	18.5%	20.9%	-0.2%	1.5%
branch	8.1%	9.7%	6.5%	1.5%	3.6%	-2.7%
factor	1.2%	1.5%	-1.2%	0.0%	-18.3%	-18.2%
mem read	2.6%	5.5%	4.8%	-0.0%	4.0%	-0.1%
mem rw	9.4%	7.9%	5.2%	-1.0%	5.7%	2.2%
mem write	4.1%	8.0%	15.7%	11.2%	6.8%	9.1%
mibench	-4.5%	-3.8%	-0.8%	-5.3%	-3.0%	-4.0%
unixbench	-2.3%	5.6%	9.8%	5.7%	9.9%	5.4%
gcc	13.4%	10.7%	16.2%	13.1%	9.3%	1.3%
gzip	-1.8%	4.6%	1.5%	-3.5%	-1.8%	-8.3%
pdftex	-5.0%	0.1%	10.0%	3.8%	3.4%	0.2%
real world avg	**2.2%**	**5.1%**	**9.2%**	**4.5%**	**3.6%**	**-2.3%**

The results of the performance counter multiplexing gave error values which were not as good as expected. In some cases it is better, in some cases it is worse. The best improvements can be notified with the add and memory write test programs. One main improvement is that the estimation in total is better. For example the worst case of two counter pairs was the branch test program measured with the counters 0x05 and 0x07. This gave an estimation error of 57.1%.with multiplexing this error was decremented to a 29.5% estimation error.

Looking at the error table shows one important issue. All programs making use of the memory writes tend to get a higher error ratio. This may be because the 3.3V power supply is also used by the memory. Now the problem is that there is no counter available to measure memory accesses but only counters for instruction and data caches. At least there is one counter for data write back (counter 0x0C). This counter was not considered for investigation because it did not trigger very many events and so it is very unfavorable for the energy usage estimation. Nevertheless the previous results showed that for a better estimation it is necessary to count memory accesses. Even if the data write back counter only counts memory access in one direction the other direction is already covered by the data dependency stall counter (0x02). As the memory read operations were covered in the above tests there were no multiplexing tests covering the memory writes.

The next step was to investigate this behavior. All programs were run again with the counter combination 0x02-0x07 (data stall - instructions) alternating with 0x05-0x07 (branches - instructions) and 0x07-0x0C (data cache write back - instructions). The results amplified the initial assumption. The data write back counter is not very convenient for estimation. For write accesses the estimated usage is far too low for all frequencies. When using memory read and write operations together it gives too high values.

Further conclusions:

- Higher frequencies also correspond to higher error ratios. Especially at 400MHz the errors get above 30% while the biggest error for 100Mhz is only 15.1%. That is the same behaviour as seen without multiplexing.
- The error table shows that the multiplexing technique can not compensate the drawback of only two counters because there are still errors +15% and -15% percent. But the errors are not as big as without multiplexing. This means with multiplexing it is possible to train the energy weights for a larger variety of program types but still not for every one.
- Many programs show improvements with the multiplexing but there are still some with a worse estimation.
- There is no counter pair combination that shows the best improvements for all tested applications.

- The average of the real world programs shows improvements for nearly all counter combinations. The only exception is the triple 0x02-0x07, 0x05-0x07 and 0x07-0x0A. If the multiplexed error estimation is compared to the three single errors there is also an improvement but when compared to all counter pair estimation errors there is a decline.

As chapter 3 showed the combinations which include the *data dependency stall* (0x02) performance counter showed the best estimation for the real world programs. As the combinations investigated in this chapter did not include only these combinations it is necessary to evaluate if these pairs give a smaller error. Exemplary the triple combination 0x02-0x05, 0x02-0x07 and 0x02-0x0A have been tested. The results of the real world programs are shown in the following error table.

0x02-0x05 X 0x02-0x07 X 0x02-0x0A			
100MHz	Errors	Improvements/Declines	
		3 Pairs only	All Pairs
gcc	-4.3%	2.3%	7.4%
gzip	-7.2%	-0.5%	5.2%
pdftex	-4.0%	-1.9%	2.1%
real world avg	**-5.2%**	**0.0%**	**4.9%**
200MHz	Errors	Improvements/Declines	
		3 Pairs only	All Pairs
gcc	0.5%	2.0%	10.7%
gzip	0.2%	3.2%	14.0%
pdftex	1.0%	0.5%	7.0%
real world avg	**0.6%**	**1.9%**	**10.6%**
300MHz	Errors	Improvements/Declines	
		3 Pairs only	All Pairs
gcc	1.1%	0.8%	11.4%
gzip	0.9%	1.8%	15.5%
pdftex	3.4%	0.6%	6.3%
real world avg	**1.8%**	**1.1%**	**11.1%**
400MHz	Errors	Improvements/Declines	
		3 Pairs only	All Pairs
gcc	-0.1%	4.6%	13.1%
gzip	0.4%	10.0%	22.6%
pdftex	2.1%	5.9%	10.4%
real world avg	**0.8%**	**6.8%**	**15.3%**

The results show that if the right counter pairs are used together with the correct weights it is possible to get very accurate estimations. The real world average error is only for the 100MHz measurement slightly above 5% and for the other frequencies always below 2%. The problem is that this counter pair combination works well for the real world programs used in this thesis but not for all available application types. The memory read and write accesses for the real world programs are also balanced and not single sided like the memory write only or memory read only programs. This emphasizes that it is possible to train the counters for specific applications but not for all available types. For example programs which heavily use the processor, or only do memory read accesses but no writes or vice versa get inaccurate results with these counter settings. These results correspond to the conclusion of Bellosa[2] that the accuracy is limited by the number of counters. The multiplexing extends the set of applications that can be measured at once but two event counters are still not enough to get more accurate results for all possible applications. There are always programs which can produce an estimation error above 15%. This is the same for the PXA processor used in this thesis and for the Pentium Pro processor used by Martonosi[5].

All these results lead to this conclusion:

The performance counter multiplexing technique is good for compensating the lack of only two counters for measuring all types of applications. But there are still errors above 15% for lower and 30% for higher frequencies which leads to the conclusion that the performance counters have to be trained for specific applications to get accurate results. At least the multiplexing technique makes it possible to measure a larger variety of applications without recalculating the energy weights. Even higher accuracy for a wider range of applications can be achieved if there would be more than two performance counters and different counter types available.

Chapter 5

Future Work

There are a few ways to improve the estimation. The first way is to use other hardware with more performance counters. For example the successor of the processor used for this thesis is the Intel PXA 270. There are two versions of this processor, one of them supporting four event counters. Using these four counters together with the multiplexing technique shown in chapter four gives a good chance of decreasing the errors. Though the event types stay the same which again may make an estimation difficult if many memory accesses occur.

Another possible way is to change the calculation of the energy weights. This thesis always uses the weights of two counters which represented the whole set of counters for a fixed time period. It is also possible the calculate weights for the complete set of counters together. Then set the kernel to measure each counter pair for one time period and multiply the results by seven. After seven time periods every counter was counted once. By multiplying the scores this should give exact results. The problem of this approach may be that it takes very long until all counters get to the same level. If there are process switches during this time they have to be considered by the resources containers.

There is also the possible improvement to patch the kernel together with the compiler. The compiler knows if it creates code which will make much CPU load or likely more memory accesses. This gives the possibility to create code statements which tell the resource containers that a program is of a specific type. For example there could be statements for monitoring ten times more the data events then the instructions if an application mainly uses the memory. And vice versa for programs which use mainly the processor. This allows optimizing the measurement for specific applications to give exact results.

This approach could be also realized in a more automated way. It does not require programming statements or a patched GCC. To achieve this goal it should be feasible to measure which counters are used by a program if it is started. For example always use one second after a program start for counter type determination. After that period only use the little subset of counters which were used the most by the program for the remaining energy calculation.

Chapter 6

Conclusion

This thesis shows that it is possible to use the performance counters of the Intel PXA 255 processor for energy estimation. It also shows that it is not enough for a very accurate estimation because some specific applications may differ up to 30% of the real usage. The main problem is that the event counters are good enough to characterize the processor itself but not the hardware that is used by the CPU. The cause is that the performance counters are meant to measure performance and not energy. There are only two counters but the technique of switching them in fixed interval gives the possibility for accurate measuring of the processor. The problem is that the processor depends on other resources. The best example is the main memory. The memory also consumes much energy (when there is some data to read from or write to it) and there are no event counter types that could directly measure the RAM usage. A compensation of this lack was tried by investigation of the data stall and data cache write back counters. As normal programs mostly do not only use the CPU with it's registers it is difficult to do a good estimation. The data caches which cause unpredictable memory access are also difficult to deal with.

To get better results the approach to optimize the performance counters on real world applications was investigated. This gives a bigger error for programs which mainly use the CPU and less for real world applications. For a good estimation that would make external measurement devices useless it would be necessary to have

- more performance counters
- more event types that could be monitored by the counters

The potential of the work provided in this thesis is that it is possible to trim the performance counters for a specific application or a subset of applications. This means it is possible to measure an application once with external hardware and from then on the performance counters can estimate the energy consumption in a representative way if the energy weights are set to the correct values. The multiplexing technique provides a way to measure a larger variety of applications without the need for recalculating the energy weights again. They also minimize the error if the performance counters are used to measure all types of applications but there are still some applications possible which produce estimation errors up to 15% for 100Mhz and up to 30% for higher frequencies. The further investigation of the real world programs showed that it is possible to get an average estimation error below 2% if the performance counters and energy weights are optimized for these applications. The multiplexing extends the subset of applications that can be measured at once but for measuring all programs there need to be more performance counters present in hardware.

Bibliography

[1] AVIS, D. lrslib. http://cgm.cs.mcgill.ca/~avis/C/lrs.html.

[2] BELLOSA, F. The case for event-driven energy accounting. Tech. Rep. TR-I4-01-07, University of Erlangen, Department of Computer Science, June 2001.

[3] ERANIAN, S. The perfmon2 interface specification. Tech. Rep. HPL-2004-200R1, HP Labs, February 2005.

[4] INTEL. *Intel XScale® Microarchitecture Technical Summary*, July 2000.

[5] JOSEPH, R., AND MARTONOSI, M. Run-time power estimation in high-performance microprocessors. In *The International Symposium on Low Power Electronics and Design ISLPED'01* (August 2001).

[6] KA-RO ELECTRONICS GMBH. Ka-ro electronics triton starter kit. http://www.karo-electronics.de/.

[7] KADAYIF, I., CHINODA, T., KANDEMIR, M., VIJAYKIRSNAN, N., IRWIN, M. J., AND SIVASUBRAMANIAM, A. vEC: virtual energy counters. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering PASTE'01* (June 2001).

[8] KELLNER, S. Event-driven temperature-control in operating systems. Department of Computer Science, student thesis SA-I4-2003-02, April 2003.

[9] MUCCI, P. The performance API PAPI. White Paper of the University of Tennessee, March 2001.

[10] NATIONAL INSTRUMENTS. National Instruments Labview. http://www.ni.com/labview/.

[11] NATIONAL INSTRUMENTS. *SCC Series User Manual*, September 2000.

[12] VARIOUS AUTHORS. Cygwin. http://www.cygwin.com/.

[13] VARIOUS AUTHORS. mibench. `http://www.eecs.umich.edu/mibench/`.

[14] VARIOUS AUTHORS. netlib. `http://www.netlib.org/`.

[15] VARIOUS AUTHORS. Perfmon - performance monitoring tool. `http://www.cse.msu.edu/~enbody/perfmon.html`.

[16] VARIOUS AUTHORS. unixbench. `http://www.tux.org/pub/tux/benchmarks/System/unixbench/`.

[17] WAITZ, M. Accounting and control of power consumption in energy-aware operating systems. Master's thesis, Department of Computer Science 4, January 2003. DA-I4-2003-02.

Charakterisierung der Leistungsaufnahme von mobilen Geräten im laufenden Betrieb

Diese Studienarbeit zeigt, dass es möglich ist, mit Hilfe der Performance Counter des Intel PXA 255 Prozessors, die verbrauchte Energie abzuschätzen. Es wird jedoch auch deutlich, dass die Genauigkeit von mehreren Faktoren abhängt. Sowohl die Tatsache, dass der Prozessor nur zwei Performance Counter besitzt, als auch die Einschränkung nur bestimmte Ereignisse zählen zu können, wirkt sich negativ auf die Genauigkeit aus. Wichtige Ereignistypen wie z.B. das Zählen von Arbeitsspeicherzugriffen fehlen komplett.

Durch das Umschalten der Performance Counter in regelmäßigen Abständen (Multiplexing) ist es möglich diese Unvollkommenheit etwas auszugleichen. Hierdurch werden die Ergebnisse genauer. Jedoch sind trotzdem noch Abweichungen möglich die bei 100MHz bis zu 15% betragen, bei höheren Frequenzen sogar bis zu 30%. Dies führt zu dem Schluß, dass es zwar nicht möglich ist die verbrauchte Energie für sämtliche Programmtypen mit einer bestimmten Kombination aus Performance Counter und Energiegewicht vorauszusagen, es jedoch durchaus machbar ist die Counter auf ein Programm oder einige Programmtypen zu optimieren. Somit ist es möglich eine Testanwendung einmal mit externer Hardware zu messen und ab diesem Zeitpunkt kann für dieses Programm auf die externe Meßhardware verzichtet werden. Als Beweis wurden die Energie Gewichte mit Hilfe von drei verschiedenen Programmübersetzungen (hauptsächlich bestehend aus GCC, Make, LibTool) ausgerichtet. Anschließend wurde ein weiteres Programm übersetzt und zusätzlich ein Testlauf mit GZip durchgeführt. Diese Programme zeigten einen maximalen Fehler von knapp über 5%. Desweiteren zeigte sich, dass mit Hilfe des Multiplexings der durchschnittliche Fehler von Testläufen mit GCC, GZip und pdfTeX für die Frequenz 100MHz mit einem Fehler von 5,2% abschätzen lies. Bei höheren Frequenzen lag der Fehler sogar jeweils nur bei unter 2%. Dies zeigt, dass die richtige Wahl der Performance Counter Typen und Gewichte für die Abschätzung wichtig sind und diese auf bestimmte Programmtypen abgestimmt werden können. Das Multiplexing erhöht die Anzahl der Programmtypen die ohne Änderung der Performance Counter mit einer geringen Fehlerrate abgeschätzt werden können, ein gänzlicher Verzicht ist jedoch nicht möglich. Hierfür wären mehr Event Counter und andere Event Counter Typen erforderlich.

Es sind verschiedene Erweiterungen dieser Arbeit denkbar. Zum einen, dass zum Bestimmen der Gewichte ein anderer Ansatz erfolgt. Unter anderem wäre es möglich Compiler-Anweisungen einzuführen die dem Kernel mitteilen welche Counter Typen für das aktuelle Programm am besten geeignet sind. Außerdem wäre es möglich, dass der Kernel zuerst alle Counter für ein spezielles Programm beobachtet und ausgehend von dieser Beobachtung sich für die besten Zähler entscheidet.