Diplomarbeit im Fach Informatik

Optimizing Energy-Consumption by Event-Driven Frequency Scaling

Betriebssystemmechanismen zur Energieeinsparung durch dynamische Anpassung der Prozessortaktfrequenz

Andreas Mull

* 1977–12–14, Nürnberg

Dezember 2001 – Mai 2002

Angefertigt am

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

Prof. Dr. F. Hofmann

Betreuer:

Dr. F. Bellosa, Universität Erlangen, Informatik 4

Abstract

This diploma thesis shows that an operating system aware of detailed information about the processes it serves is able to save energy by means of performance monitoring counters and dynamic frequency scaling. A model for controlling dynamic frequency scaling based on performance monitoring information called Process Cruise Control is presented. For its implementation the Cyclone IQ 80310 development-board was selected as the target platform. Together with an XScale architecture CPU, the Intel 80200 processor performance monitoring counters, as well as frequency scaling, are available.

Contents

Ich versichere, dass ich diese Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass diese 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.

Nürnberg, 2002–05–16

1 Introduction

Increasing acceptance of mobile embedded systems as well as black-outs affecting data centers and server-farms caused by the energy crisis in many parts of the United States show need for broad introduction of power-aware systems. Manufacturers of hardware already provide many power-managed devices. With these devices a tradeoff between saving energy and available performance is introduced. Consequently, power-sensitive systems need to handle energy as a first-class resource. The resource manager has to control power-states of components in such systems.

To predict and control power consumption a multitasking operating system needs to have detailed information about the processes it serves. Resource usage needs to be monitored per process. This accounting has to apply dynamically as resource usage patterns of tasks may change with input data and time.

Current operating systems only have limited knowledge of resource usage. Items like system load, CPU usage, activation of I/O devices and current reported by smart batteries only give few hints for resource usage in a system. Although, many approaches for energy-aware scheduling have already been presented yet. Some of these are dependent on cooperating with underlying applications. This means that tasks would have to become energy-aware for usage with these approaches. In addition, figures like CPU load or general I/O activation only provide a very narrow view of the tasks and their resource usage limiting the prognosis of such approaches.

Information about memory access frequency, execution stalls, activation of specific functional units within the CPU and chip-set like FPU, ALU, branch prediction logic, MMU and cache utilization is desirable for a more profound prognosis of power consumption [Bel01a]. Yet, not all of these aspects are covered by performance monitoring counters implemented in state-of-the-art processor platforms. Therefore, the value of dedicated energy monitoring counters becomes evident. Such units would refine accounting energy consumption through an elaborate view on detailed resource usage of the system.

As energy monitoring counters are not available yet, the approach presented in this work relies on performance monitoring counters to predict the resource usage of tasks. No cooperation or preliminary information from applications is needed for this approach. Power-states controlled in the target system are different main processor clock speeds. The policy presented in this work throttles applications by dynamic frequency scaling to save as much energy as possible while keeping a limit on the decrease of performance. Because of the similarity to a car cruise control we have called our scheduling policy *Process Cruise Control*. Driven by performance monitoring events to scale processor frequency

Event-Driven Frequency Scaling closes the loop to a controlled system.

The opportunity to save energy by scaling frequency grounds on differing energy consumption for accessing main memory at various CPU speeds on the target system. The target system itself and measuring setup is introduced in chapter 2.

Chapter 3 pictures the opportunities to save energy on the specific target system, first. Further, I describe the steps on our way to Process Cruise Control. Aspects are the selection of performance monitoring events and creating a prognosis model relying on performance monitoring counters. An algorithm for generating an energy-saving policy from micro-benchmark sample data is visualized, finally.

Succeedingly, chapter 4 presents our implementation for Process Cruise Control in the Linux kernel. Modules for per-process-performance monitoring, frequency scaling and policy management are explained. Finally, our approach on Process Cruise Control is related to other research.

Chapter 5 discusses actual measurement results on the Process Cruise Control system. Efficiency, dynamic aspects, resulting energy-savings and performance loss of the implementation are debated.

In chapter 6 I conclude and propose further architectural innovations that originate in Process Cruise Control.

A short summary of this work in German is given in chapter 7.

2 Technical Setup

2.1 Target System

Currently several microprocessor architectures support dynamic frequency scaling and performance monitoring. Some offer a wide variety of selectable speeds as for example the Athlon 4 PowerNow! [AMD02b] and the Alchemy Au1100 [AMD02a] from AMD. Others like Intel's Speedstep-M [Int00b] at the time just distinguish slow and fast modes.

As we wanted to explore the possibilities of saving energy in the field of embedded systems we selected the Intel 80200 processor [Int00a]. Its power consumption only rates up to 500 mW and it provides both capabilities for frequency scaling as well as performance monitoring which are crucial tasks for our approach on Process Cruise Control.

The 80200 processor is available with the Intel IQ 80310 evaluation platform [Int01], which is a PCI board. This modular form, in contrast to closed systems like laptops or notebooks, makes power-measurement possible. The evaluation platform mainly consists of the 80310 I/O processor chip-set including the 80200 processor and the 80312 I/O companion chip controlling a 32 MB SD-RAM memory, 8 MB FLASH, a Fast-Ethernet and two RS-232 interfaces.

The IQ 80310 evaluation-board can be mounted on a separate backplane or into a state of the art Personal Computer. We chose the latter method as it eases resetting the target system via network. Likewise, reprogramming contents of the on-board flash memory became possible.

Intel's 80200 processor is based on the XScale micro-architecture [Int00c] supporting the ARM V5TE instruction set. Its 32 KB data and instruction caches are 32-way set associative and have a line size of 32 bytes. The data cache policy is configured to be write-back and write-allocate. Data and instruction MMU both implement a 32 entry full associative TLB.

A programmable clock multiplier (PLL) generates the internal CPU clock which can be adjusted from 200 up to 733 MHz in steps of about 66 MHz. With the development-board speeds are only available from 333 MHz. The lower bound results from a constraint to the memory bus speed which is fixed at 66 MHz: The bus speed has to be significantly less than a third of the CPU clock speed. This would yield a minimum speed of 266 MHz, but as our experiences showed running the system at slower CPU speeds than 333 MHz caused immediate halts. Changing the clock frequency is done by writing a multiplication factor dependent value to a configuration register as shown in table 2.1.

Although the 80200 processor supports variable core voltages depending

Clock frequency:	Multiplication factor:	Register value:
(n/a) 200 MHz	3	1
(n/a) 266 MHz	4	2
333 MHz	5	3
400 MHz	6	4
466 MHz	7	5
533 MHz	8	6
600 MHz	9	7
666 MHz	10	8
733 MHz	11	9

Table 2.1: CPU Frequency Scaling

ID:	Definition:	repeat:
0x0	Instruction cache miss.	
0x1	Instruction cache or TLB miss.	×
0x2	Stall due to data dependency.	×
0x3	Instruction TLB miss.	
0x4	Data TLB miss.	
0x5	Branch instruction executed.	
0x6	Branch mispredicted.	
0x7	Instruction executed.	
0x8	Stall due to full data cache buffers.	×
0x9	Stall due to full data cache buffers.	
0xA	Data cache access.	
0xB	Data cache miss.	
0xC	Data cache write-back.	
0x10	BCU received memory-request.	
0x11	BCU request queue full.	×
0x12	BCU queue drained.	

Table 2.2: Performance Monitoring Events

Mode:	PLL:	Registers:	Wakeup:	
Idle	on	retained	10 cycles	$\approx 20\,\text{ns}$
Drowsy	off	retained	≈ 2000 cycles	$3.1\,\mu\text{s}$
Sleep	off	have to be saved	(code dependent)	$\gg 3.1\,\mu\text{s}$

Table 2.3: Power Saving Modes

on the selected frequency, dynamic voltage scaling could not be used as the development-board lacks features to adjust CPU core voltage.

Furthermore, the processor has facilities for performance monitoring. A time stamp counter (TSC) displays the number of cycles the processor has been active at. Two performance monitoring counters (PMC) may be configured to count processor specific events. The events can be selected from the set shown in table 2.2 which contains the relevant events for this work. Events marked with *repeat* are encountered for every clock cycle they are active at. Other events are counted once each sequence of cycles in which they occur. Some events correspond to common activities and appear twice in the table, differing in the presence of the repeat attribute.

In addition, the 80200 provides several power saving levels displayed in table 2.3. Having the shortest wakeup time idle mode was configured for the system idle thread of our target system.

BlueCat Linux 3.1 from LynuxWorks, which is supplied together with the IQ 80310 evaluation platform, was selected as the target operating system. By using Linux as the target platform, we were able to modify the operating system kernel which is a substantial task for implementing Process Cruise Control. The Linux kernel version used here is 2.2.12.

2.2 Measurement of Power and Energy

The IQ 80310 development-board has no features like test points for tracking its overall or its components' power consumption that is drawn from the PCI bus. Using a PCI raiser card with such test points would be a desirable way to deduce the power consumption of the board. But the tested raiser cards interfered with the PCI initialization procedure of the development system.

Despite these facts we found a way to track the power consumption of the development-board. The 5 V and 3.3 V power lines from the ATX power supply to the host PC main-board were equipped with measurement shunts as shown in figure 2.1. As the development system is the only PCI card connected to the host PC main-board the power lines will only reflect the power consumption of these two units. By keeping the host PC idle, its power consumption is enforced to be nearly constant. This constant level of idle power can easily be subtracted from the actual power values measured while executing the tests.

The development-board only uses 3.3 V power [Int01]. Setting supply voltage to $U_{b,1} = 3.3\,\mathrm{V}$, sample voltage to U_1 and constant idle power to $P_{0,1}$, the actual development-board power P_1 is calculated as follows:

$$P_1 = \frac{U_1}{R_1} U_{b,1} - P_{0,1}$$

However, P_2 would be calculated analogously although of little interest as the development-board does not use 5 V power. Anyway, the 5 V shunt was useful to assure a constant power level of the host PC.

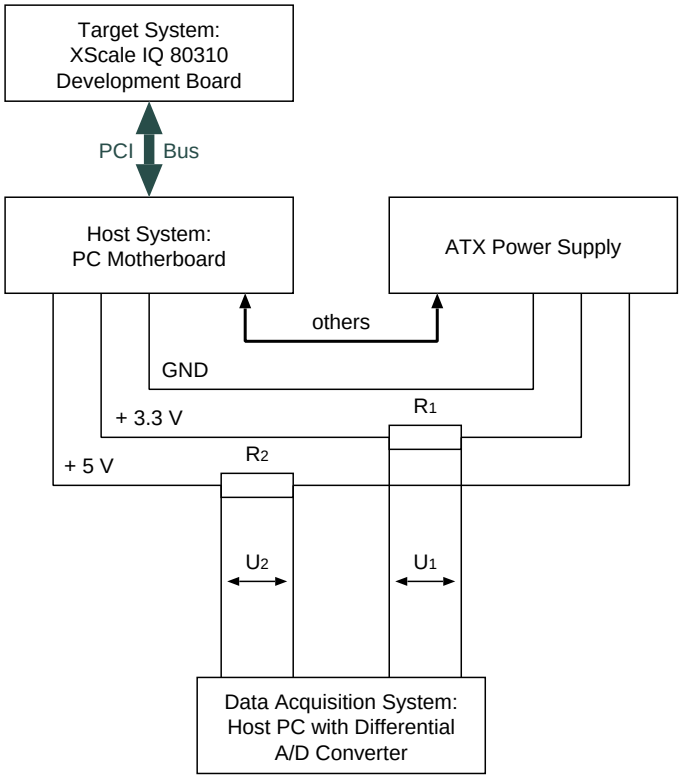

Figure 2.1: Electrical Measurement Setup

2.3 Data Acquisition

The analogous voltages U_1 and U_2 are sampled by two differential A/D converters. A Meilhaus ME 2600 [Mei01] and a special purpose low-cost A/D converter are used to acquire sample data. Both boards are controlled by dedicated Linux device driver modules [Mei02], [Win02]. Special purpose utilities for reading, filtering and synchronization are used to process the measured data.

The synchronizing of start and end of micro-benchmarks is done directly on the read-out data. Ethernet communication is not available for this task as the experimental enabling of data caches corrupted DMA transfers used for the network interface. Serial console activity for communication causes interfering power consumption.

Thus, the synchronization filter is supplied with a complete block of sampled data. As shown in figure 2.2 this block may contain the power progress of several micro-benchmarks, for example seven here. It can start anytime before the first micro-benchmark and end anytime after the last one. As a consequence block synchronization can be done by very simple means, even, if necessary, by hand. With our standard test utility the micro-benchmarks are separated each by short delays of 100 ms. During these delays the target system is idle. This results in gaps in the overall power progress that can be recognized by the synchronization filter.

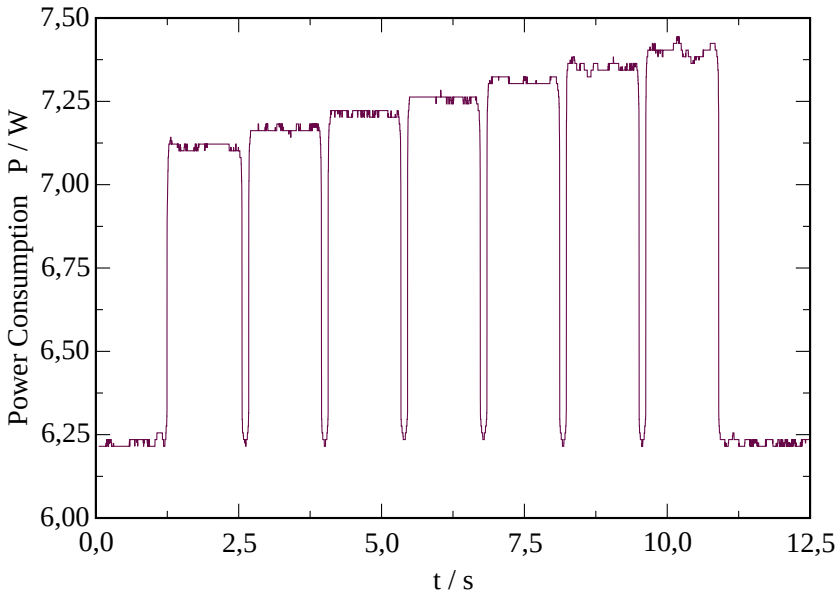

Figure 2.2: Typical Synchronization Filter Input

The software implemented filter acts as a conditional integrator controlled by a trigger with hysteresis: Each slice with a power consumption above a fixed threshold is separately integrated and averaged revealing energy and mean power consumption of the belonging micro-benchmark. Together with the duration of the slice these values represent the filter output. When the power value falls short of a second lower threshold the end of a slice is recognized. At the beginning of the supplied block the average idle power is computed. This idle power is subtracted from any samples before integrating or averaging.

3 A Model for Saving Energy

3.1 Performance and Power Characteristics

To discover possibilities for saving energy we needed to know how much the various components contribute to the total power consumption of the target system.

We created some micro-benchmarks that involve different units of the CPU: An arithmetic test (`add_reg`) exercises the arithmetic unit and registers. `goto_label` makes strong use of branches, `call_sub` uses the stack to pass parameters mixing register operations, cache references and branches. Some cache tests only read (`read_cache`) or read and write (`rw_cache`) the first-level cache of the CPU.

During all tests the processor was constantly busy and operated at 733 MHz. Without involving the main memory and caches power consumption of the 80200 processor is nearly constant at about 600 mW. As can be seen in figure 3.1 the evaluation-board mean power consumption increases to about 800 mW when caches and MMU are utilized by tests `read_cache` and `rw_cache`.

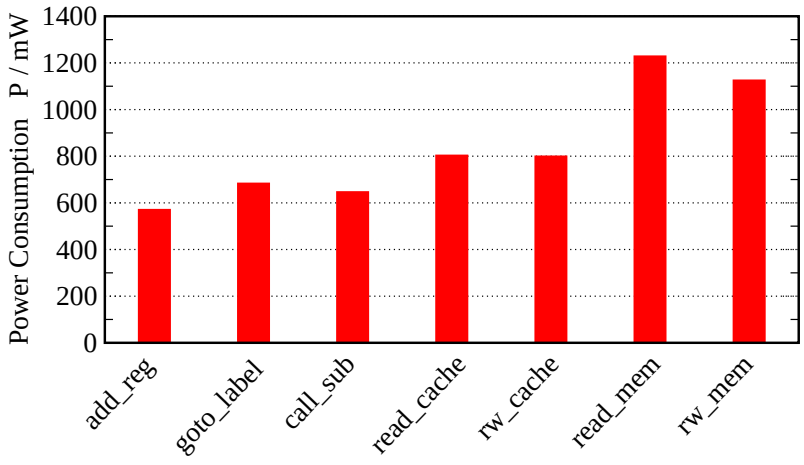

Figure 3.1: Basic Benchmarks Power Breakdown

Further tests also involve the SD-RAM module and memory controller by reading from (`read_mem`) or reading from and writing to (`rw_mem`) the main memory. Most strikingly the mean power consumption now rises up to 1220 mW. We believe that this effect results from activating the memory controller and the memory module and using the memory-processor bus. Lacking test points

on the evaluation-board we cannot prove this thesis.

Running the same benchmarks with different processor speeds reveals similar results. As shown in figure 3.2, power scales proportionally to the CPU clock frequency. Interestingly, this constraint also holds for tests involving the main memory.

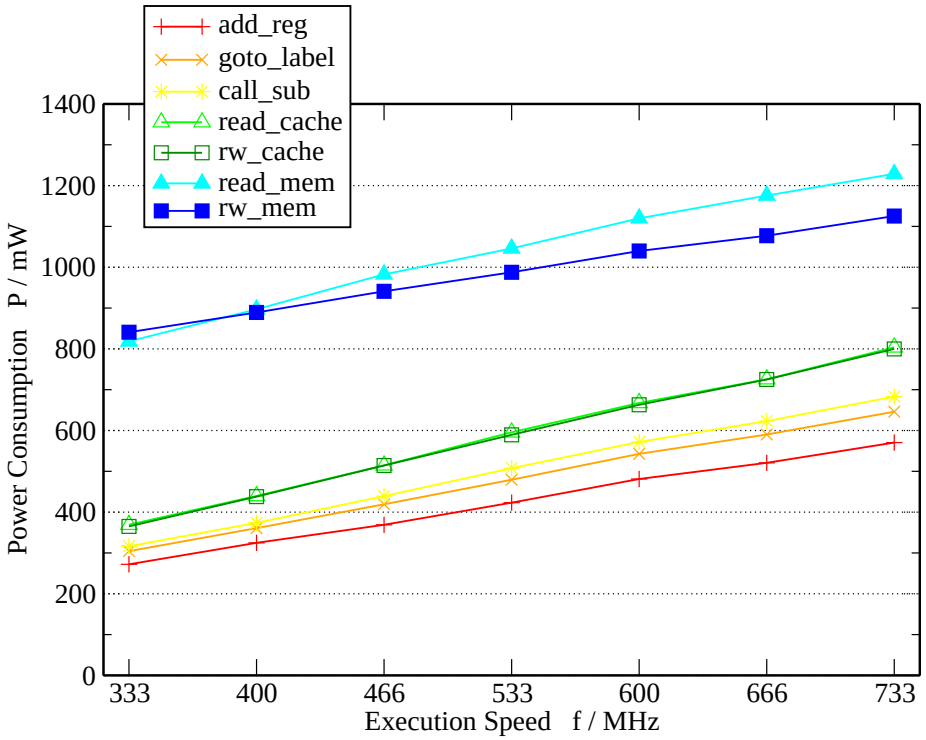

Figure 3.2: Frequency Scaled Power Breakdown

Up to this point, only the mean power consumed while running the benchmarks was considered. To get the amount of energy needed to execute a benchmark, the duration of which has to be heeded as well. Combining mean power consumption and completion time leads to energy consumption. Before focusing on energy consumption one should have a closer look at execution time. When scaling the frequency each test takes another time to complete. Normalized to minimum or maximum execution time's reciprocal will be called *relative application performance*.

Figure 3.3 shows the relative application performances of the benchmarks. These have been normalized to the minimum individual application performance of each test measured at minimum CPU speed.

This figure displays one important fact: while CPU intensive tests scale performance proportionally to clock speed, tests involving the main memory do not significantly increase their application performance when CPU is sped up.

Of course, this fact results from the constant memory access rate. The memory-processor bus is clocked at a fixed frequency of 66 MHz. Memory accesses — for example caused by cache-misses (`read_mem`) — are first passed

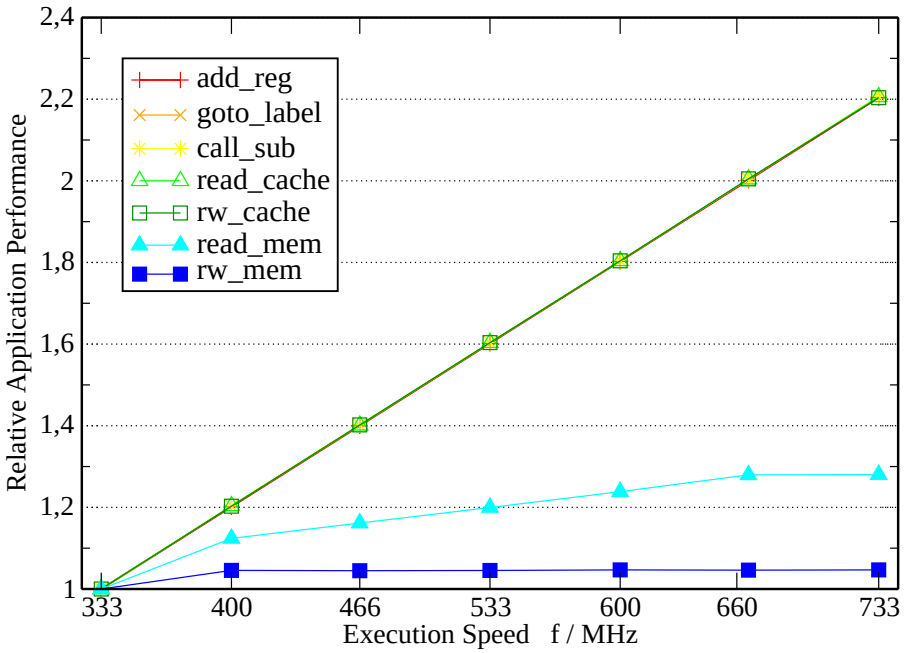

Figure 3.3: Relative Application Performance vs. CPU Speed

to the fill buffer of the 82000 processor. The fill buffer can handle up to four outstanding 32-byte read-requests. When this level is reached by agglomerating memory accesses, the CPU stalls [Int00a].

The other memory intensive test `rw_mem`, which reads and writes memory brings write buffer and cache policies into effect: The write buffer holds up to eight 16-byte write requests. This micro-benchmark stalls for each new allocated cache line as a dirty cache line has to be rewritten to memory before a new one can be stored into cache. These effects explain the worse application performance of the `rw_mem` test.

As power consumption shows, the processor is not powered down during these stalls: While waiting for the memory request to become satisfied the CPU remains in a busy mode.

3.2 Energy Saving Strategy

A comparison of figures 3.2 and 3.3 shows that the application performance for memory intensive applications does not significantly grow parallel to the processor speed while power consumption does. This fact leads to an important basis providing a possibility to save energy:

The amount of additional energy which is consumed by fast running memory intensive applications compared to energy needed by the same applications executed at slower speed can be considered as overhead. Vice versa, energy consumption can be decreased for memory intensive applications without significantly impacting their performance by slowing down the CPU.

The energy statistics of micro-benchmarks in figure 3.4 show the amount of energy saved by selectively throttling the CPU speed. In this figure, energy values for a specific test are relative to its energy consumption at maximum speed, that is 733 MHz.

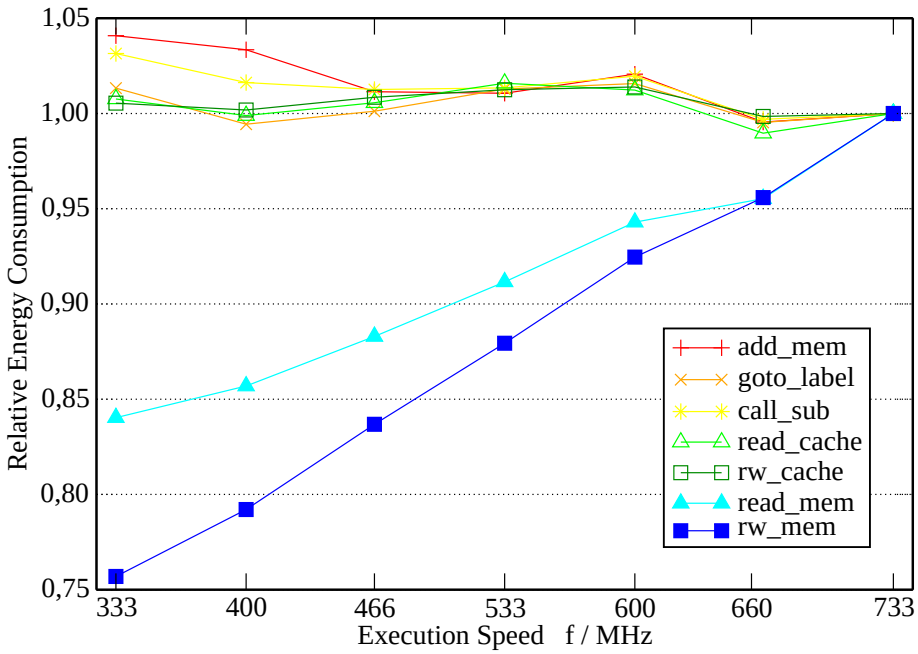

Figure 3.4: Relative Energy vs. CPU Speed

3.2.1 Authentic Tests

Of course, these benchmarks are not representative as they purely exercise specific functionalities of the processor. Consequently, some real-world applications were used as additional tests to prove the memory-related approach for saving energy.

The diagrams from figure 3.5 show the application performance loss and energy saving achieved by throttling CPU clock frequency. Performance loss l and energy saving s of a test are calculated from its execution time t and the energy W needed to run the test. As t_0 and W_0 are t and W at maximum processor speed (733 MHz), the corresponding values of l and s would equal zero in the diagrams and therefore are not shown.

$$l = \frac{t}{t_0} - 1 \qquad s = 1 - \frac{W}{W_0}$$

The top-left diagram shows statistics from the `find/grep` test. Searching the RAM disk for specific data, this test is very memory intensive. Obviously, energy savings outweigh performance loss. The top-right diagram contains the data from the `gzip` test which makes moderate usage of memory while compressing a file using Lempel-Ziv coding (LZ77). In both tests performance loss

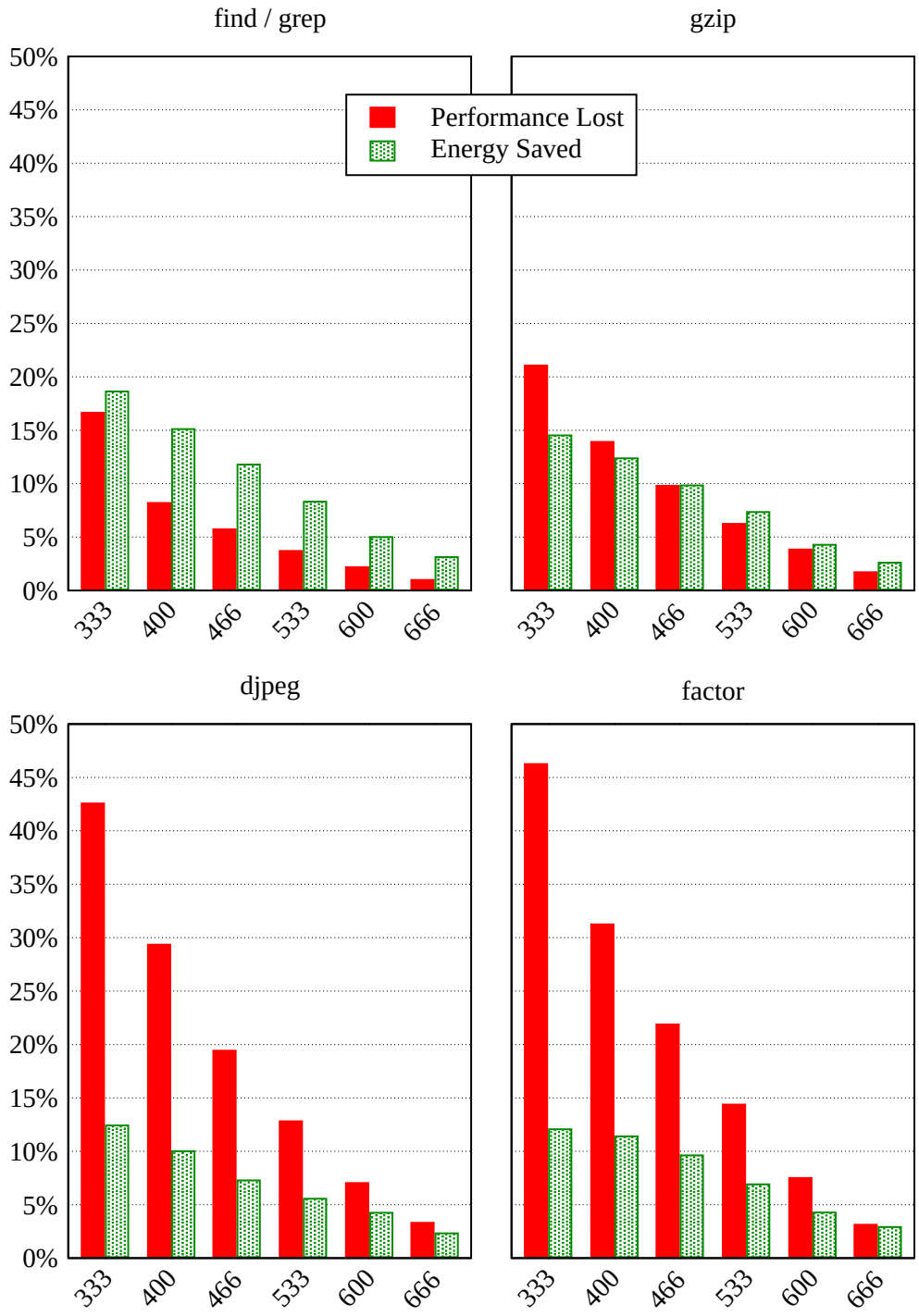

Figure 3.5: Power and Performance Profiles

grows slower than frequency decreases: for example at 333 MHz less than 25 % performance is lost while frequency was decreased more than 50 %. Performance loss and saved energy remain in a similar range: It would be desirable to slow down the processor by a certain degree in order to save energy.

Both lower diagrams show statistics of tests that don't rely much on memory accesses. While `djpeg` decompresses a JPEG encoded picture the other test `factor` factorizes a number. Following consequences arise: On the one hand, energy savings are lower than in both upper diagrams; on the other hand, performance loss has significantly increased in contrast to the upper diagrams. Throttling CPU would result in an undesirably strong decrease in application performance while saving only little energy.

A view back to the micro-benchmarks from section 3.2 shows the following scenario: While CPU intensive benchmarks (`add_mem`, `goto_label`, `call_sub`, `read_cache` and `rw_cache`) scale in performance and energy consumption comparable to `djpeg` and `factor`, memory intensive benchmarks like `rw_mem` hardly lose any performance when throttling CPU speed, that is less than 5 %.

3.2.2 Throttle Control

Throttling applications with high access rates for main memory improves energy efficiency. This happens as the processor is stalled less often while being slowed down.

However, throttling a task also decreases its performance. Consequently, saving energy is always connected to performance loss. From a software approach it is nearly impossible, at least uneconomic, however, to slow down the processor only for memory requests. Determining each single memory access cannot be done without heavy overhead. Thus, granularity for adjusting speed by a kernel extension refers to a process or thread. Of course, when slowing down a complete process or thread, instructions that are not accessing memory become throttled as well as the memory intensive code which surrounds them.

A tradeoff between saved energy and available performance becomes evident. Constant maximum throttling of the CPU would be the best solution if only focusing on saving energy. But after all, performance is an important factor as well. To put it bluntly, saving a bit of energy while halving performance doesn't make sense in most embedded systems.

To keep decreasing performance in a sensible range we orient our model to save as much energy as possible at a fixed maximum performance loss. Restricting performance loss locks one of the two variables of the tradeoff. On this base the model can make a prognosis for a maximum throttle still holding the constraint for performance loss. The central task for the model is to satisfy the performance constraint while increasing energy efficiency as far as possible.

This means that the model is dependent on the parameter of an upper limit for performance loss. Although, by changing the performance loss restriction our model may easily be adapted to other circumstances for which the saving of more or less energy is appropriate.

For this approach we set the limit to an exemplary of 10 %. This value seems to be a reconcilable throttling for most applications. Of course, justifiable

performance impact is dependent on the usage pattern of an embedded system and real world installations could need other policies. The following reflections will rely implicitly on a maximum performance loss of 10 %; we also use the term *policy* for it.

3.3 Frequency Domains

The previous section showed that throttling CPU for memory intensive applications makes sense. By restricting performance loss to a certain percentage a so-called ideal speed for each application can be derived. This ideal speed is the CPU clock frequency at which the application loses just less than for example 10 % of its maximum performance. The ideal speeds for the test applications from section 3.2.1 for a performance loss restriction to 10 % are shown in table 3.1. They may be easily derived from figure 3.5.

TEST:	`find/grep`	`gzip`	`djpeg`	`factor`
IDEAL SPEED:	400 MHz	533 MHz	600 MHz	600 MHz

Table 3.1: Ideal Speeds for the Authentic Tests

Our next point of interest is to find means that may be used to forecast the ideal speed of an application without having initial information about it. As applications should not need to cooperate with the model — that is they do not need to be energy-aware — our basic idea is to figure out relations between rates of processor internal events and possible throttling of a task which, in turn, means saving energy. This approach is designed for a system with two performance events measurable at the same time.

3.3.1 Performance Monitors

If the policy had knowledge of the times it takes to execute an application at different CPU speeds it would be easy to tell the ideal execution speed for the policy. A simple attempt would be collecting ideal speeds for any application that are supposed to run on the target system. With a table of ideal speeds for all these applications a system could automatically adjust the CPU speed for each application. The problems with this simple solution are self-evident: On the one hand you have to exercise any new application for ideal speed before using it; on the other hand the table would never be complete, continously grow, and sometime even become quite bulky.

Notwithstanding these facts, there are applications which change their memory access rate with time and input. To adjust such applications to an ideal speed, the approach needs to get information about the process and its memory access rate during the execution of the process, that is the model has to act dynamically.

Characteristics for resource usage of processes can be obtained by performance monitoring counters. For example, energy-consumption may

be predicted from these characteristics, as already shown in other works [Bel01a], [JM01]. However, this approach tries to directly derivate ideal speed from performance event rates without taking a detour via power-consumption.

Purely obtaining information about the application from performance monitoring counters brings in the two main advantages of this approach: First, no predating knowledge about the application is needed. Second, applications do not need to be energy aware, that is there is no communication between application and scheduler.

Previous work has already explored usage of other kernel level information and application support for latency based cruise control [MGG01]. Likewise, such approaches are less platform dependent while not concentrating on first hand information from performance monitoring counters.

Most modern processors supply performance monitoring units counting specific events that occur in the CPU like cache misses or executed instructions. The 80200 processor offers several events for performance monitoring. It is able to count two of these events simultaneously. A selection of available events is displayed by table 2.2 [Int00a].

3.3.2 Selection of Events

To monitor the rate of a specific event for a single test its occurrence is counted using one of the two available performance monitoring counters. This value is divided by the number of clock cycles that happened while executing monitored by the time stamp counter (TSC). The resulting probability ranges from 0 to 1 and is called *event percentage*.

$$p = \frac{pmc}{tsc}$$

New micro-benchmarks were introduced to figure out those events whose percentages may be, most suitably, related to ideal speeds. The additional micro-benchmarks mix memory accesses with processor internal tasks. Thus, they better cover the spectra for resource usage and ideal speed.

Events basically related to process cruise control are listed in table 2.2. The main task for this approach is to tell how often a process accesses main memory, which is connected to the processor core via a so-called Bus Control Unit (BCU). Likely, percentages of memory-requests that arrived at the BCU and the times its request queue was full will be of primary interest. As external memory is accessed via data caches we expect miss and write-back percentages to reflect such accesses. Moreover, the instruction execution rate will diminish when stalling the processor by intensively accessing main memory. Along, we guess that stalls caused by data dependencies may indicate memory accesses.

Some events better correlate to ideal speed after normalization with execution speed. They do not depend on processor internal timing. Thus, multiplying their percentages with the processor clock speed eliminates the influence of differing TSC values as they reflect internal timing.

We call this figure *event factor* — as we will do with other normalized event rates as well.

$$p^* = \frac{pmc}{tsc} \cdot f_{exec}$$

Multiple events yield differing percentage and factor ranges. Percentage for executed instructions nearly covers the full available range from 0 to 1 while, for example, memory request percentage only ranges up to 0.2. For considering multiple event rates in the same diagram an equalization of the different ranges becomes desirable. In consequence, event percentages and factors in figure 3.6 are normalized so that the maximum mean rate for each event equals to one. The resulting value is called *event factor*, again. This equalization shouldn't be confused with predating normalization with execution speed.

Figure 3.6 shows how pure and normalized relative event percentages correlate with ideal speed. The coloured bars display the mean percentages of the above discussed events. Error bars denote the dispersion of a specific event percentage to its minimum and maximum. The horizontal position of the bar group gives the ideal speed of the micro-benchmarks whose event percentages were accumulated to that group.

The effect of execution speed normalization shows up if you compare both diagrams of figure 3.6. Normalization causes less variance for most event percentages. Although, normalization has nearly no effect: Value ranges of percentages for the same event still overlap for different ideal speeds. Precision of processor internal event percentages — like that for data dependencies — would even suffer from normalization. Thus, we will postpone execution speed normalization for now.

Memory request percentage has least dispersion for the whole spectrum of ideal frequencies. Along, mean values scale very well with ideal speed. Consequently, memory request percentage will be the main candidate for ideal speed prognosis. Other events have undesirably high variances, BCU queue full percentage being the worst example.

Apart from percentage dispersion executed instructions could be a good means for prognosis, as well. Being the only percentage directly proportional to ideal speed executed instructions percentage has the widest value spectrum for different ideal speeds. Executed instructions show a crease at 600 MHz both for normalized and pure event factors. Although not resulting from measurement error — as repeated test runs showed — this effect could currently not be explained. However, this error is not too significant and the event percentage is used for the model furthermore.

3.3.3 Multiple Dimensions

Just taking a single event into consideration for ideal speed prognosis will not lead to satisfactory results. The main problem remains in the percentage dispersion of the single events: For example, percentage range for a certain ideal speed mostly overlaps with that for another speed. The correct ideal speed can not be distinguished just upon knowledge of a single event percentage.

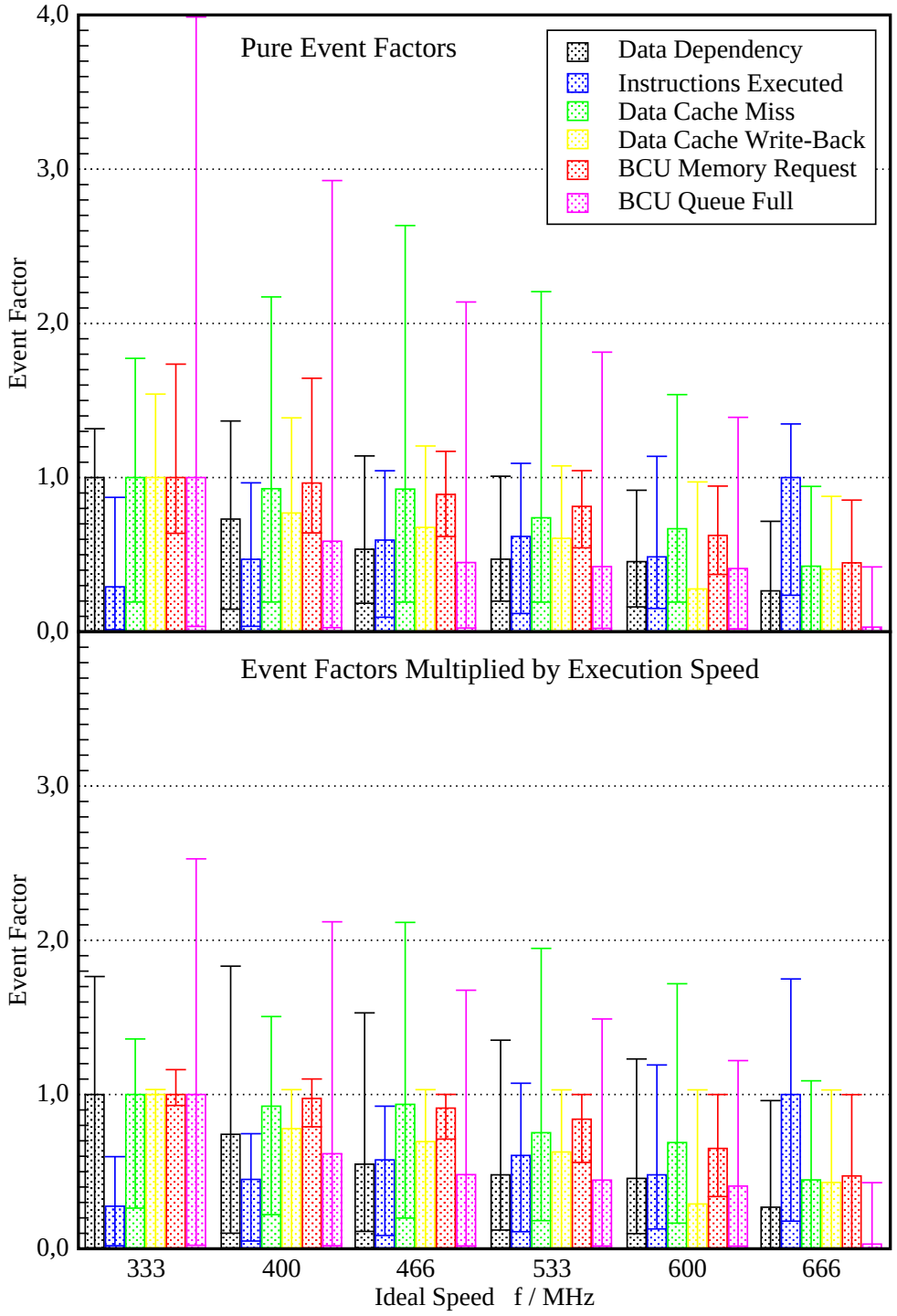

Figure 3.6: Singular Event Factors vs. Ideal Speeds

To cope with this problem a two-dimensional approach is necessary for building a model. Instead of watching a single event percentage two will be used for ideal speed prognosis. Lying at hand, both above mentioned top candidates will be selected for prognosis: executed instruction and memory request percentage.

Combining these percentages yields diagram 3.7. It shows one point for each possible execution speed of each micro-benchmark. Thus, there are seven points in the diagram for one micro-benchmark. All points display the ideal execution speed of the belonging test by their shape and colour. Coordinates of the points are determined by the two above mentioned event percentages and — for the y-coordinate — execution speed of the micro-benchmark:

$$x = \frac{pmc_0}{tsc} \qquad y = \frac{pmc_1}{tsc} \cdot f_{exec}$$

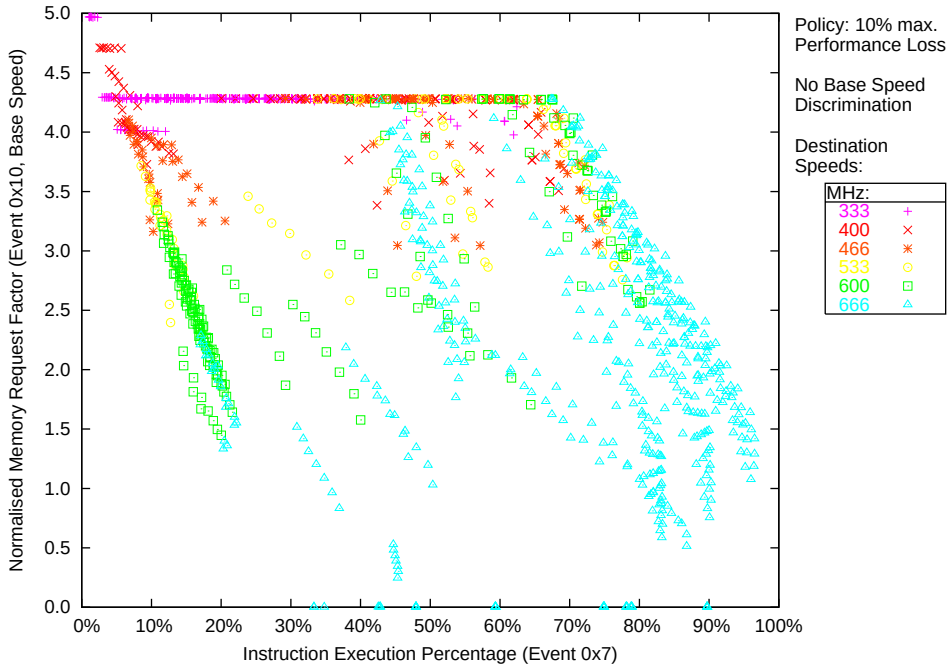

Figure 3.7: Ideal Speed Diagram for all Speeds

Different memory request percentage ranges were joined by execution speed normalization producing a memory request factor. However, this diagram still does not distinguish ideal speeds exactly enough. Sets of differently coloured points intermix especially in the upper right quarter of the diagram. Only few areas may be clearly zoned for identical ideal speed.

Browsing through data shows that points of different ideal speed can be better distinguished when looking at the samples for a single execution speed at one time only. This means either introducing a third dimension or splitting up the diagram into seven diagrams of their own, one for each execution speed. Then, of course, execution speed normalization may be neglected as points of different execution speed are separated to different diagrams or plains anyway.

As this approach is presented on a two-dimensional medium and as occurring execution speeds are discontinuous the diagram is splitted into seven diagrams of their own. However, the following considerations may be exercised on a three-dimensional cubicle as well.

Figure 3.8 shows just the allocation of the sample points for an execution speed of 400 MHz exemplarily. Of course, this is only one of the seven diagrams sampled from executing micro-benchmarks. With constellations like in figure 3.8 we are able to separate points with different ideal speeds easily. Geometrical partitions of identical ideal speed may be built as being explained in the next section. Again, these diagrams all relate to the event percentages for executed instructions and memory requests.

When receiving a new — uncoloured — point by observing an unknown application its ideal speed can be forecasted. First, one of the seven diagrams is selected by looking at the actual execution speed of the application. Then, the point is associated to one of the ideal speed partitions giving the prognosis.

To take a short digression I want to have a look at other event percentages than executed instructions and memory requests: Different combinations for such diagrams resulted in graphs with worse parting for ideal speed zones like shown in figure 3.9. Event percentages displayed here are data cache accesses versus misses. Again, this diagram is just one of the seven diagrams for that two events. Separation for execution speed was done just like in figure 3.8. However, coherent zones of equal ideal speeds are hard to find in this diagram. Points with completely different ideal speeds nearly combine. Obviously, such diagrams are insufficient for a usable model.

3.3.4 Creation of the Model

Having decided which events to use for the model each micro-benchmark is attributed with the following data:

- its ideal execution speed
- seven pairs of performance event percentages, one pair for each execution speed

To generate the model which will consist of seven model graphs the data is split up for geometrical inspection. Percentage pairs with equal execution speed form points in one of the seven diagrams both percentages of one pair giving the coordinate for one point:

$$x = \frac{pmc_0}{tsc} \qquad y = \frac{pmc_1}{tsc}$$

Points are attributed according to their ideal execution speed $f_{point} \in \{333, 400, 466, 533, 600, 666, 733\}$. The following method for balancing border lines is applied to each diagram separately.

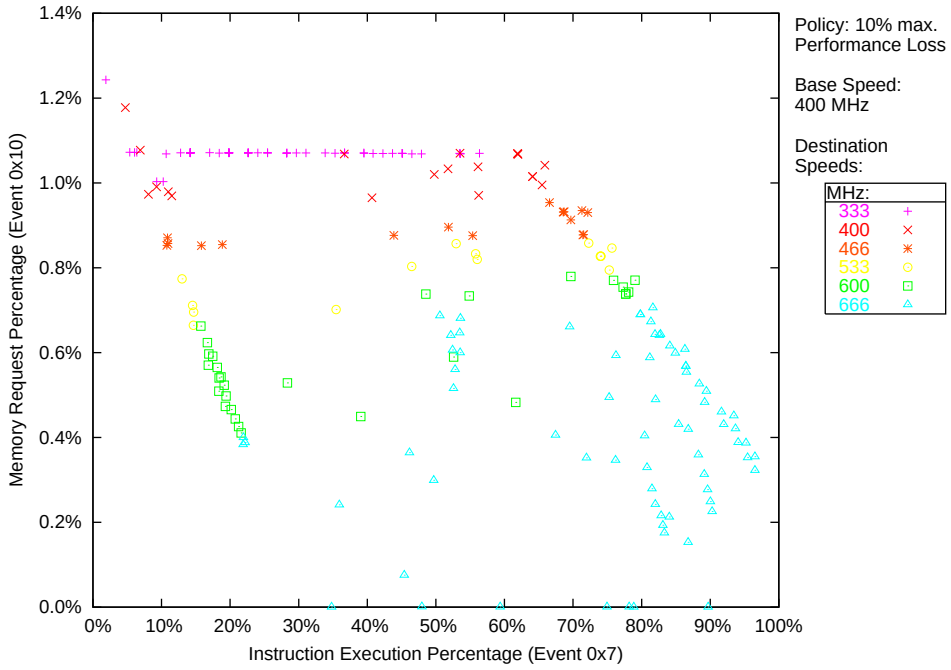

Figure 3.8: Useful Ideal Speed Diagram with Micro-Benchmarks at 400 MHz

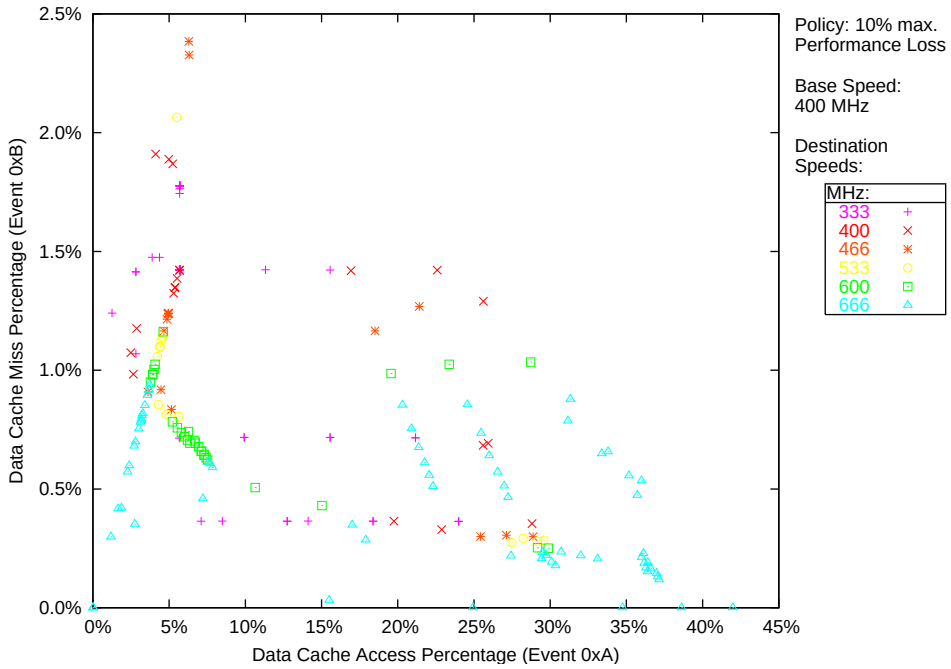

Figure 3.9: Unusable Ideal Speed Diagram with Micro-Benchmarks at 400 MHz

Space in the model graph is partitioned into seven zones with six border lines. Each partition consists of the space between two lines or one line and the graph boundaries. Each line is attributed with a border speed $f_{line} \in \{400, 466, 533, 600, 666, 733\}$. The points which set below this line are to be of that ideal speed. More precisely, the partition above a border line is to contain all sample points with $f_{point} = \text{pred}(f_{line})$, whereas the partition below the line is to hold all sample points with $f_{point} = f_{line}$.

For each possible border line points are classified into four misplacement sets as shown in table 3.2. Consequently, these sets change with varying lines being considered. Points setting not into any of these sets are defined to be placed correctly. The geometrical aspect of this classification is shown using exemplary points in figure 3.10. An optimization method is used to balance the border lines in between the points. This method has to provide constraints for the sets and the position of the line.

SET:	POSITION:	CONSTRAINT:	MISPLACEMENT:
A	above line	$f_{point} > f_{line}$	total
B	below line	$f_{point} < \text{pred}(f_{line})$	
C	above line	$f_{point} = f_{line}$	uncritical
D	below line	$f_{point} = \text{pred}(f_{line})$	

Table 3.2: Classification of Points into Sets

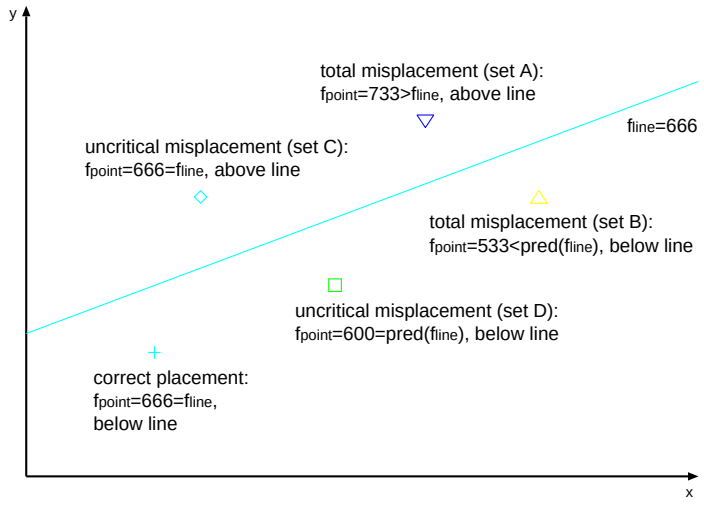

Figure 3.10: Geometrical Aspect of Classification Sets

The following weighting method was found appropriate for finding good border lines. It consists of four rules which have decreasing priority. A rule of higher priority has to be satisfied before selecting lines according to subsequent rules begins. That means, a found minimum for $|\mathbf{A} \cup \mathbf{B}|$ has to be kept while searching for lines that make $|\mathbf{C}| < |\mathbf{D}|$, for example. The rules are applied to each of the six border lines separately.

1. Minimize $|\mathbf{A} \cup \mathbf{B}|$.
2. Minimize $|\mathbf{C} \cup \mathbf{D}|$.
3. Keep $|\mathbf{C}| < |\mathbf{D}|$ if possible.
4. Maximize the distance to the nearest point placed correctly.

To exclude applications seldom accessing main memory for computation from throttling the border line with $f_{line} = 733$ was fixed to a memory access rate of 0.1 %. This value was found experimentally.

The algorithm places six border lines into one model diagram. It is applied to each of the seven model graphs. Figures 3.11 and 3.12 show a selection of two out of the seven graphs from the model used for this work. They contain points of the micro-benchmarks as well as the resulting border lines.

The current implementation for finding border lines uses a brute-force algorithm to find the best matches out of a set of eight million preassumed lines per diagram. By using a weighting function the best matching line is reported. Of course, this method is not the optimal solution as it takes some time to generate a model. For example a geometrical approach would be more elegant. But as the model is generated only once for a target system the usage of a slow brute-force algorithm is acceptable while easing the implementation.

To simplify weighting the four rules were put together into a one-dimensional function where factor n is a number bigger than the count of sample points in the diagram. Making use of the fact that the distance d to the nearest correctly placed point is always less than or equal to $\sqrt{2}$ four dimensions from the rule-set may be simply collapsed into different value ranges:

$$n \quad > \quad |\mathbf{C} \cup \mathbf{D}| \quad > \quad 0.5 \quad > \quad \frac{\sqrt{2}}{10}$$

The best match is found by maximizing the weighting function:

$$w = -n \cdot |\mathbf{A} \cup \mathbf{B}| - |\mathbf{C} \cup \mathbf{D}| + k + \frac{d}{10}$$

$$\text{where} \quad k = \begin{cases} 0.5 & : \quad |\mathbf{C}| < |\mathbf{D}| \\ 0 & : \quad \text{else} \end{cases}$$

Border lines produced by the model generator are stored as axis intercepts and gradients. Execution speed (identifying a graph) and predicted ideal speed (also called destination speed) build indices to a table with all 7×6 border line entries. One group of six table entries refers to exactly one model diagram. The table may be loaded by the corresponding kernel module that implements the model policy. Table 3.3 represents the stored form of the model data.

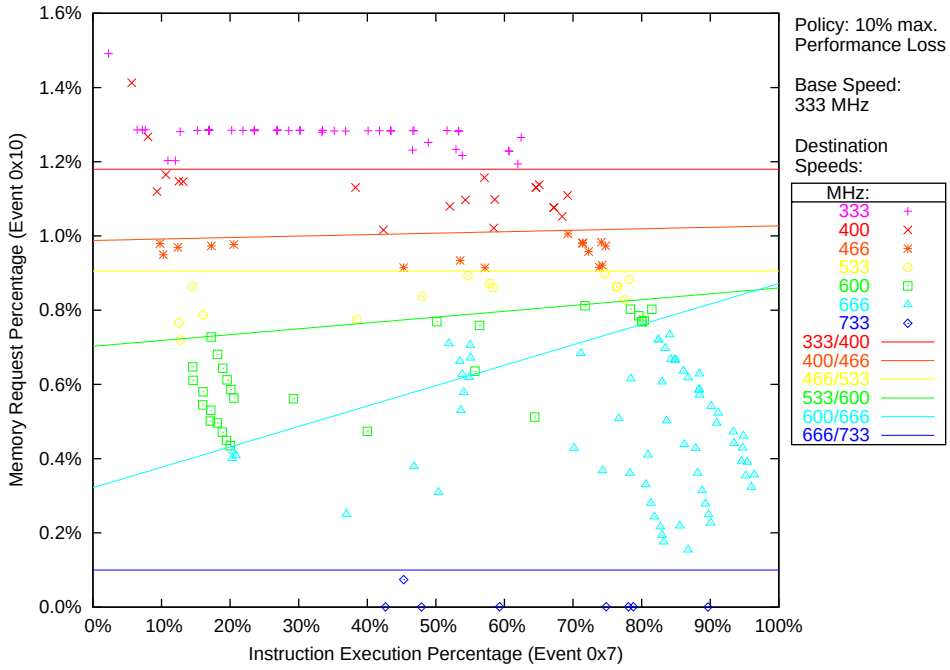

Figure 3.11: Model Diagram with Micro-Benchmarks at 333 MHz

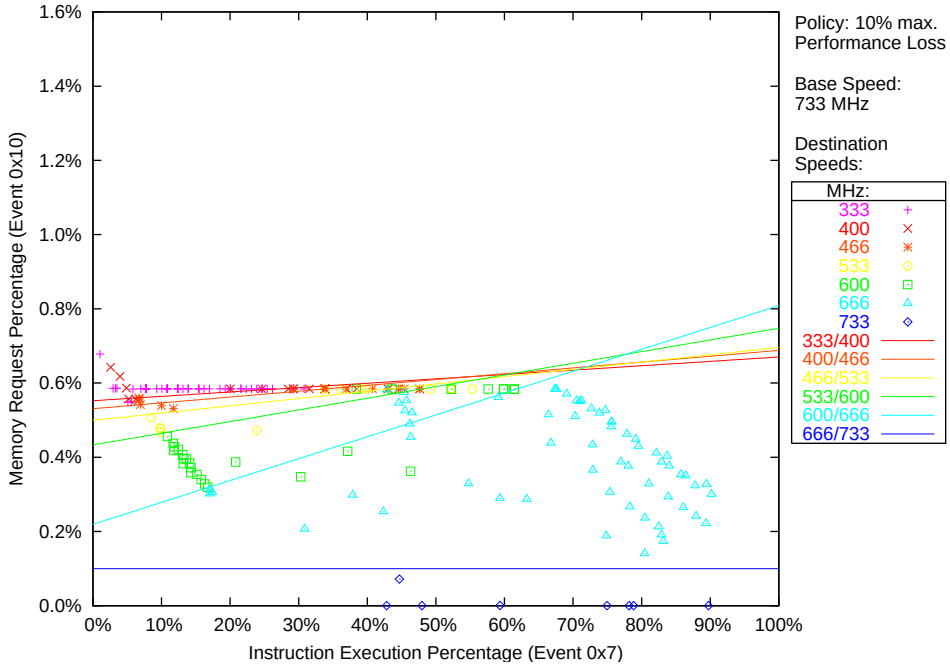

Figure 3.12: Model Diagram with Micro-Benchmarks at 733 MHz

Execution Speed:	Destination Speed:	Gradient:	Axis Intercept:
333	400	+0.000000	+0.011799
333	466	+0.000393	+0.009879
333	533	+0.000000	+0.009055
333	600	+0.001571	+0.007028
333	666	+0.005498	+0.003219
333	733	+0.000000	+0.001000
400	400	+0.001571	+0.009785
400	466	+0.001178	+0.008901
400	533	+0.000393	+0.008404
400	600	+0.002749	+0.006217
400	666	+0.005498	+0.002859
400	733	+0.000000	+0.001000
466	400	+0.001571	+0.008445
466	466	+0.002356	+0.007668
466	533	+0.001571	+0.007232
466	600	+0.003142	+0.005661
466	666	+0.005498	+0.002723
466	733	+0.000000	+0.001000
533	400	+0.001571	+0.007360
533	466	+0.001571	+0.007187
533	533	+0.002356	+0.006422
533	600	+0.003142	+0.005457
533	666	+0.005891	+0.002487
533	733	+0.000000	+0.001000
600	400	+0.000393	+0.007012
600	466	+0.001571	+0.006488
600	533	+0.002356	+0.005740
600	600	+0.003534	+0.004853
600	666	+0.005891	+0.002379
600	733	+0.000000	+0.001000
666	400	+0.000393	+0.006298
666	466	+0.001963	+0.005691
666	533	+0.002749	+0.005250
666	600	+0.003142	+0.004611
666	666	+0.005891	+0.002276
666	733	+0.000000	+0.001000
733	400	+0.001178	+0.005524
733	466	+0.001571	+0.005309
733	533	+0.001963	+0.004995
733	600	+0.003142	+0.004334
733	666	+0.005891	+0.002196
733	733	+0.000000	+0.001000

Table 3.3: Model Data in Tabular Form

3.3.5 Application of the Model

For analyzing the following steps by means of figures 3.11 and 3.12 these diagrams have to be considered without points, as if they contained only the border lines.

Querying the model for an ideal speed prognosis works as follows: The two performance event percentages yield the coordinates of a point for a certain graph. The graph is selected according to the execution speed of the application when the percentages were measured.

The diagram will be queried for the ideal speed prognosis, next. Therefore, a line suitable for the point has to be found. Beginning with the line which is attributed with the fastest speed, searching is done towards the slowest one. The first line above the point is considered as a match. The speed attribute of that line is the ideal speed prognosis of the model. If the point is not below any line ideal speed prognosis defaults to 333 MHz.

Figure 3.13 shows a hypothetical model diagram and how application points would be colourized according to it. Each point in the diagram displays the dedicated ideal speed for its surrounding area. In contrast to figures 3.11 and 3.12, figure 3.13 contains points which are coloured according to the model prognosis already. The border lines in this hypothetical diagram intersect strongly to demonstrate the special mode for querying the model.

I am going to exercise some steps of querying the hypothetical diagram from figure 3.13. Exemplarily, the diagram will be coloured completely, catching any pitfall of possible application points. Numbers and ranges denote areas in the graph. At first, the blue line is considered colouring both areas (1–2) below it blue. The intersection of the cyan line does not matter here. The cyan line is focused next. It fills any area below itself and above its blue predecessor line to be cyan (3–7). Again, intersections from subsequent lines do not play any role yet. Continuing with the green line areas 8 through 10 become green. Areas below predecessor lines (1–5) are not considered any more. Exactly the same method is applied to the remaining lines. When the red line is finished, the last area yet uncoloured (11) is filled magenta according to the default speed.

The data flow pictured in figure 3.14 shows the relation of the Process Cruise Control model to a car cruise control or any other controlled system. Sampled event percentages along with corresponding execution speed run into the model policy. An ideal speed prognosis is made and applied to the application. Again, the new CPU speed is fed to the model along with newly measured event percentages closing the loop of control.

3.3.6 Divergent Ideal Speed

While most applications are scheduled for a definitive ideal speed the pure geometrical approach for partitioning the model diagrams introduces a problem: Some applications (and even micro-benchmarks) spawn points that are not destined to one constant ideal speed by the model. That means the sample point for an application which lies just above a certain border line in one diagram sets just below the corresponding border line in another diagram. Such applications

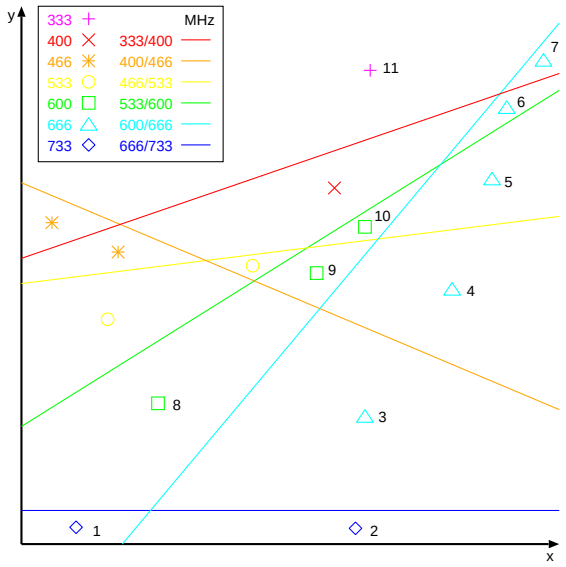

Figure 3.13: Geometrical Aspect of Model Diagram and Application Points

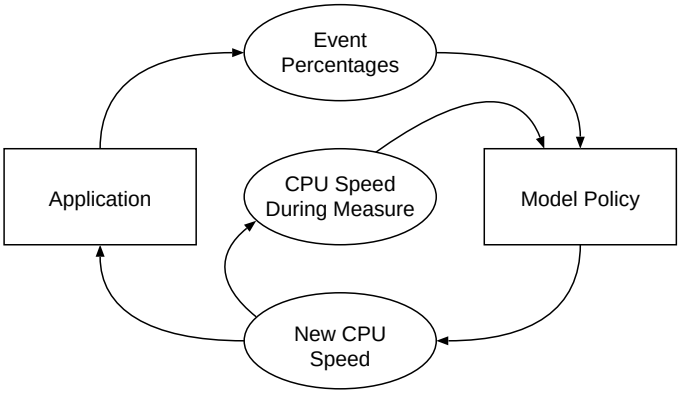

Figure 3.14: Process Cruise Control Loop

become scheduled with two different ideal speeds, rotatively. We say they have *divergent ideal speed*. This is a lack of the model, that is an uncertainty in the neighbourhood of border lines. Consequently, divergent ideal speed always covers only two speeds between which scheduling oscillates.

The effect may already be shown from sample points of micro-benchmarks and the generated model. A cyclic test examines each micro-benchmark whether it becomes scheduled to a definitive ideal speed or cycles between different speeds. The cyclic test relies just on the model data and the sample points. Therefore, no further measurement is needed to indicate the existence of applications with divergent ideal speed.

For example, out of 212 micro-benchmarks used for this approach only four become associated to divergent ideal speeds. Table 3.4 displays the contradictory destination speeds for ideal speed prognosis on these four micro-benchmarks. Participant speeds of each cycle are typeset bold.

		Execution Speed:						
Destination Speed:		333	400	466	533	600	666	733
Micro-Benchmark:	`fimul21`	533	533	**533**	**466**	466	533	466
	`memmix1-9`	**400**	**333**	400	333	400	333	400
	`mul3`	**400**	**333**	333	333	333	333	333
	`mul7`	600	600	600	**600**	**533**	533	466

Table 3.4: Micro-Benchmarks Scheduled to Conflicting Destination Speeds

For example, when considering benchmark `mul7` starting at an initial speed of 733 MHz it first becomes scheduled to 466 MHz. From there it is directed to 600 MHz entering the divergence loop. From now on, the benchmark will be scheduled rotatively with 533 and 600 MHz.

In systems with considerable delay for switching CPU speed overhead may become a problem. Frequent adjustments of CPU speed for divergent ideal speed could slow down execution. The amount of overhead is dependent on how much time passes by between two subsequent prognoses. To minimize such overhead a scheduling threshold may be implemented enlarging intervals between subsequent CPU speed adjustments.

3.4 Related Work

This project grounds on considerations for indirectly accounting energy in runtime presented by Joseph and Martonosi [JM01] and by Bellosa [Bel01a]. Methods to correlate energy consumption with performance counters were adopted to the Process Cruise Control model. Concepts for dynamic frequency and voltage scaling and corresponding strategies for speed regulation were already presented by Morrey et al. [MGG01] and Pouwelse et al. [PLS00]. Policies presented in these papers regard application-level and scheduler information to control frequency scaling. Performance monitoring counters are not considered to identify applications for throttling.

Further work details discussion of energy-efficiency in context of dynamic frequency and voltage scaling [MLH+01]. It is shown that points of lower performance could well be less energy efficient than points with higher performance. While originating from the view of a complete system, these considerations share direction with results of some work focusing on memory dependent possibilities for saving energy. Likewise, this project was initially inspired by the work of Vollrath et al. [VHS98] concentrating on SD-RAM power consumption for varying memory access patterns. Nevertheless, we could not show significance on the discussed memory related effects in our system.

As implementation and validation will outline more elaborate concepts for data acquisition and collection like resource containers [BDM99] may improve accurateness of the Process Cruise Control system.

4 Implementation

The implementation of the Process Cruise Control model has to fulfill three central tasks:

1. Measuring event percentages of associated tasks.
2. Considering an ideal speed for these applications.
3. Controlling the speed of processes under cruise control.

To obtain utilization characteristics for different CPU units the micro-benchmarks have to be monitored using the performance counters contained in the 80200 CPU. Accessing and accumulating such performance monitoring counters has already been implemented by several packages designed for PC and similar architectures. We chose the Perfctr-Package [Pet01] which is also used by PAPI [BDG+00], [Muc01], a common interface for programming performance counters. An extended version of the Perfctr-Package represents the first of four modules in the implementation which is shown in figure 4.1.

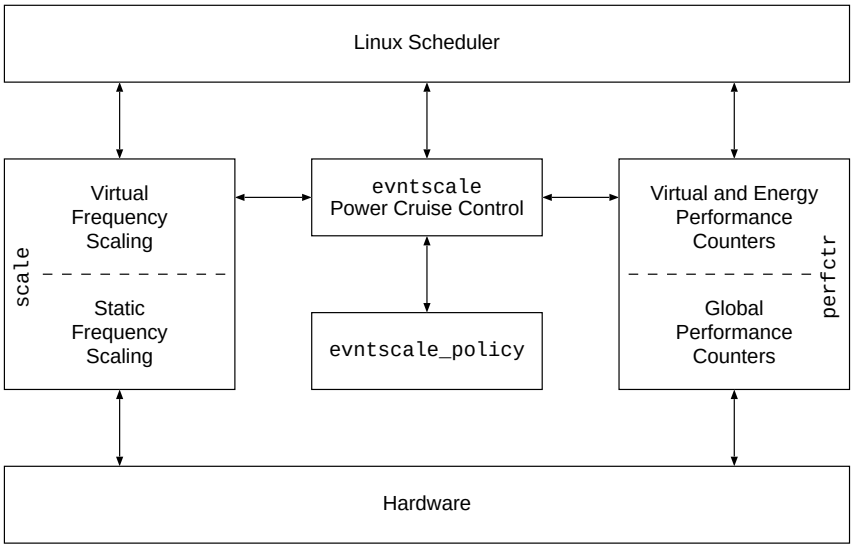

Figure 4.1: Kernel Module Structure

Controlling the overall data flow and considering an ideal speed is split into two further modules. The central module `evntscale` manages the cooperation of the modules. The policy module (`evntscale_policy`) predicts ideal speeds and administrates the model data table.

Separating the policy into a module of its own brings in better replace-ability for cruise control strategy: New models could easily be implemented and used with the existing three modules replacing just the current policy module.

A fourth module is responsible for frequency scaling. Although its initial implementation had been taken from the LART project [BM02], [PLS00] the module was majorly rewritten to support virtual frequency scaling.

4.1 Frequency Scaling

For executing micro-benchmarks and other tests at different speeds a basic frequency scaling interface is necessary. This static interface allows to change the overall clock speed of the system affecting all processes. Per-process-frequency scaling (also known as *virtual frequency scaling*) is based on top of static frequency scaling. Both static and virtual frequency scaling are implemented with the module `scale`.

For the virtual part each process becomes attributed with its individual speed. However, the scheduler had to be extended for a callback-stub. This stub enables the module to adjust the CPU speed just after changing processes. The attribute may also indicate that the process has no individual speed at all; it is scheduled with the global speed, then. All processes have void individual speed by default. Thus, they are scaled with the global frequency when the module comes up. As the individual speed attribute of a process is stored with its thread struct, child processes inherit the setting of their parent.

Actual change of the core clock is realized by storing the new speed value to the CCLKCFG configuration register of the 80200 processor. This register may also be read to get the current speed, but after starting up the module keeps its own copy of that value for easier access. The value stored to and read from the CPU register is not the frequency in MHz, but rather a 4 bit speed value. The relation between speed values and frequencies is shown in table 2.1.

Along with changing the core clock the loop for short delays in the kernel needs to be adjusted. As calibration of the BogoMIPS value `loops_per_sec` takes some time it is done for each available CPU speed during module initialization. The resulting values are stored for later use, allowing fast adaption of the delay loop.

Interfaces

The module `scale` has two control interfaces: One for the global speed and another for the individual process speeds. A process may write or read the overall clock speed as a register value as a decimal on a single line to or from `/proc/scale`.

The interface for per process frequency scaling is implemented similarly. When being read, it gives a list of processes currently under control of dynamic frequency scaling. The list maps process ids (PID) to the individual speed setting.

Writing is line oriented and associates a certain process with an *individual speed*. Using zero for *individual-speed* releases the process from control through

virtual frequency scaling, making it disappear from the list mentioned above. The format of the control file `/proc/vscale` is as follows:

<pid> <individual-speed>

For use with the cooperating modules two interfaces are provided. Both set the virtual speed of a process, whereas the latter one does not apply the actual core clock setting directly. Instead, frequency is scaled first when processes are switched.

void **set_thread_speed** (struct thread_struct *__thread__, u32 **speed**)
void **set_thread_speed_lazy** (struct thread_struct *__thread__, u32 **speed**)

4.2 Energy Performance Counters

The Perfctr-Package [Pet01] supports global and virtual performance counters. Virtual counters are used to obtain per process information about resource usage while global counters proved helpful to obtain utilization characteristics for different CPU units when executing micro-benchmarks. Of course, some low level routines which access the actual hardware registers had to be adapted for the 80200 processor. Additionally, in conjunction with Process Cruise Control some extensions to pure virtual performance counters were necessary.

Basic virtual performance monitoring behaves like shown in figure 4.2 (a): Forking a new process inherits the current virtual performance counter configuration to the child process. When the child exits, its counter values are accumulated to its parent if the child did not modify its inherited performance counter configuration. Thus, counters of the parent become debited for actions of the child.

For Process Cruise Control each process is considered for its own. Accumulating counter values of children would interfere with resource usage observation for the parent process.

For example, consider a memory intensive parent process that forks a child process which makes massive usage of arithmetics. Both processes are under Process Cruise Control. The parent process — as accessing memory very often — is scheduled to a slow speed, according to the policy of the model. The child process will be scheduled to a fast speed — it uses the CPU only.

What would happen if the counter values of the child were accumulated to its parent? As counter values reflect the resource usage pattern for Process Cruise Control it looks as if the parent process were CPU intensive. Thus, the model would react by scaling the parent process to a faster frequency.

To circumvent such wrong decisions the accumulation of child counter values was disabled for energy virtual performance counters. These counters represent an extension to normal virtual counters; their behavior is shown in figure 4.2 (b). Virtual counters are transformed to energy performance counters by a special function call in kernel space:

int **vperfctr_energy_control** (struct thread_struct *__thread__,
struct perfctr_control **control**)

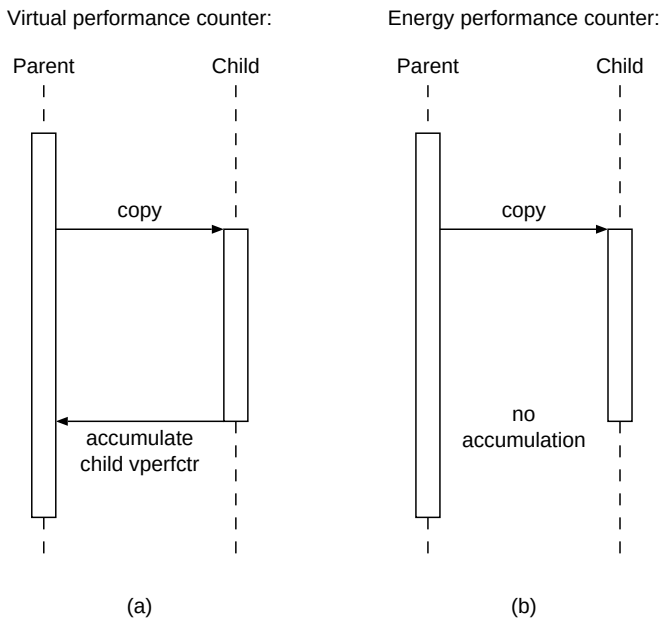

Figure 4.2: Virtual vs. Energy Performance Counters

Activating a virtual performance counter through this call sets the energy mode for it. This energy mode will be canceled when the counter is disabled through the same call. These semantics still make using virtual performance counters for processes which are not under Process Cruise Control possible. Beyond that, performance counters are protected against user space accesses when brought into the energy mode by the special function call.

4.3 Process Cruise Control

The Process Cruise Control kernel module (`evntscale.o`) is the central instance which controls the cooperation of the other modules and the kernel:

- It hooks itself into the context switch and timer queue of the kernel to get called whenever necessary.
- It makes use of energy performance counters to obtain information about the processes put under its control.
- It delegates the decision for a new per-process CPU speed to the policy module explained in section 4.4.
- Finally, it sets a new per-process CPU speed using virtual frequency scaling.

The implementation consists of two parts: A small callback stub is bound into the kernel whereas the major part of the system is contained in the kernel

module. The callback stub becomes activated just before processes are switched by **switch_to**. If the module has registered its control function with the stub, the function is called and is passed the old thread as a parameter.

Initialization

When loading the module, it hooks the Process Cruise Control function (**__evntscale_adjust_thread**) into the callback stub which resides in the kernel. To control processes which exhaust their time slice between context switches, the module registers its control function with the timer interrupt as well. Thus, the control function becomes called periodically and can rescale the execution speed of the process.

Control Interface

The module creates an entry in the proc-file-system (`/proc/evntscale`) which is used to control the set of tasks which are put under Process Cruise Control. The list maps process ids (PID) to a value which can be evaluated by the policy: A non-zero value means Process Cruise Control is active for the identified process. Writing a value of zero into the proc-file deactivates control of event-driven frequency scaling for a certain process and purges its entry from the list. Children of tasks under Process Cruise Control inherit the energy-saving setting of their parents.

The format of the file is line oriented; a line consists of a PID to value mapping and is built as follows:

<pid> <value>

Control Function

The control function (**__evntscale_adjust_thread**) is activated before processes are switched or when the timer has ticked. It manages the event-driven frequency scaling.

At first, the virtual energy performance counters are accumulated. The values of the counters are, together with the thread struct, passed to the Process Cruise Control policy. The policy is queried whether the speed for the current process has to be altered.

If the speed has to be altered, the virtual frequency scaling mechanism is used to do the job: During a timer interrupt the speed has to be adjusted instantaneously using **set_thread_speed**. When called by the scheduler the new speed is stored in the thread struct only by **set_thread_speed_lazy**. The CPU is not adjusted to the new speed as processes will switch immediately, anyway. When reactivating the current process the new speed setting will take effect as it is interpreted by the virtual frequency scaling.

4.4 Event-Driven Frequency Scaling Policy

The decision policy for Process Cruise Control implementation is divided into a separate module easing exchange of policies for future models.

Currently, only one policy exists to implement the model discussed in this thesis (`evntscale_policy.o`). The policy maintains a dynamic lookup table with the frequency scaling directives. Each directive is represented by a border line from the model stored as gradient and axis intercept. The table consists of 7×6 border line entries. This corresponds to seven model diagrams, one for each base speed.

In addition, seven threshold values are stored, again one for each base speed. Being heuristic figures threshold values do not appear in the model diagrams. They tell how old a performance counter sample has to be to consider it meaningful. A threshold value expresses the elapse in clock cycles. Scheduling threshold values is considered for three reasons:

1. Minimizing overhead caused by divergent ideal speeds.
2. Preventing points from the bottom left corner of a diagram from influencing cruise control decisions.
3. Obviating overhead caused by logging debug messages.

Fortunately, our implementation does not suffer from significant overhead when switching CPU speeds. As already explained in section 3.3.6 the first point becomes negligible, therefore. The actual switching overhead will be discussed in section 5.1.

During debugging, the threshold value should range from about 5 ms to 10 ms. Smaller values would result in non-negligible overhead from logging each policy decision. For normal use the threshold values are preset to 500 µs.

Policy Interface

The policy module creates its own proc-file-system entry `/proc/evntscale_policy` through which policy and thresholds can be written to and read from the module. Reading the file shows the whole table including the thresholds. Writing to the file is line oriented, there are two different sorts of lines:

<source-speed> <destination-speed> <gradient> <axis-intercept>

fills one entry in the lookup table. The entry is determined by *source-speed* and *destination-speed*. The entry itself being a border line is formed by its *gradient* (m) and *axis-intercept* (t).

<source-speed> <threshold>

sets the *threshold* value for a certain *source-speed*. The value is given in clock cycles.

Query of the Policy

The policy is queried by the Process Cruise Control using the function `evntscale_policy_calculate_speed` and gets the thread struct of the process to be considered. The query returns a new speed for that process or zero if the speed does not change. The following variables are used to decide on a new speed:

- Time stamp counter (TSC)
- Both performance counters (PMC), which currently are number of instructions executed and memory requests (event ids 0x7 and 0x10 in table 2.2)
- Current speed of the process

The procedure that controls the cruise of processes does not rely on an aging strategy or anything similar. Old samples are neither stored nor used for prognosis. Instead, sample values are accumulated until they are considered old enough. Age and event count values are first reset when using them for prognosis. Before having convenient age the values are not touched, no prognosis is made and per process speed will not be changed, consequently.

The actual query of the policy works as follows: At first, the age of the sample is checked. If TSC is less than the threshold corresponding to the current speed no decision is made. Zero is returned keeping the current speed. Otherwise the sample is converted to a point for a diagram by calculating its coordinates:

$$x = \frac{pmc_0}{tsc} \qquad y = \frac{pmc_1}{tsc}$$

Afterwards, the sample values of the virtual performance counters are reset. One of the seven table sets is selected according to the current (source) speed. Beginning with the fastest destination speed it is checked if the point sets below a corresponding border line ($y < m \cdot x + t$). The first matching line is picked and its destination speed is returned. As there are only six border lines but seven speeds the slowest speed being the default speed is selected if the point does not set below any line. This method corresponds to the proposed application of the model from section 3.3.5.

While coordinates, gradient and axis intercept are expressed here and in the policy interface file using real numbers, the actual internal arithmetics are simulated using integer numbers. From outside they look like fixed point arithmetics with six decimals. This is because floating point arithmetic is banned from kernel space.

Control File Format Summary

proc-File:	Format:	Function:
`scale`	<*speed*>	Global Speed
`vscale`	<*pid*> <*speed*>	Per-Process Speed
`evntscale`	<*pid*> <*flag*>	Enable/Disable Cruise Control for a Process
`evntscale_policy`	<*src*> <*dst*> <*m*> <*t*>	Policy Rule *src*: Source Speed *dst*: Destination Speed *m*: Line Gradient *t*: Axis Intercept
	<*speed*> <*threshold*>	TSC Minimum Age for *speed*

Table 4.1: Control File Format Summary

5 Validation

5.1 Overhead

Before going on to energy and performance statistics, focus will be set on overhead caused by using Process Cruise Control. Reading performance counters, querying the model and adjusting CPU speed all take some time to complete. As these steps are taken whenever switching tasks which are under Process Cruise Control it has to be considered if the overhead introduced by these actions is still reasonable.

Table 5.1 shows durations measured from the Linux scheduler and the extensions that were made to it for event-driven frequency scaling. Elapses, respectively overheads, in this table are displayed both in CPU cycles and equivalent time. As processor frequency may be altered from 333 MHz up to 733 MHz constant amounts of cycles correspond to ranges in time. Vice versa, constant time values correspond to ranges in cycles.

Task, Sub-Task:	Overhead/Elapse	
	Cycles:	Time:
Scheduler overall	$< 140\,000$	$< 420\,\mu s$
Task switch	$> 26\,000$	$> 35\,\mu s$
Event-scaling	$6\,000 - 18\,000$	$8 - 54\,\mu s$
Read PMCs	184	$250 - 550\,ns$
Adjust speed	$1\,000 - 2\,300$	$3.1\,\mu s$

Table 5.1: Overhead Factors

Scheduler overall values outline the size of the procedure that was extended for Process Cruise Control. As we positioned our expansion to the Linux kernel before and after the task switch, which is only one part of the whole scheduling routine, its timings are given here for comparison. Whenever tasks are switched service routines for event-driven frequency scaling are called.

Event-scaling routines consist of several sub-tasks, namely querying the model which involves some amount of calculations, reading performance counters and adjusting processor frequency. Both latter tasks are denoted separately, policy calculations take the rest of the time used for event-scaling.

Reading performance monitoring counters needs a fixed count of cycles while adjusting speed takes a fixed amount of time for recalibrating the PLL which is used to multiply the input clock of the processor. Fortunately, both mentioned hardware dependent actions don't generate too much overhead. Consequently,

calculation for the policy of Process Cruise Control surpasses hardware factors.

Comparing elapses for task switch and event-scaling it is obvious that both values range in the same dimension. We can say that the task switch is elongated less than 70 % of its pristine duration. When comparing the elongation caused by Process Cruise Control to the overall scheduler times it shows that overhead introduced by the extensions rates below 15 %.

5.2 Energy Savings

In this section I am going to discuss actual energy savings resulting from Process Cruise Control along with corresponding performance loss caused by slowing down the target system.

First, the authentic tests from section 3.2.1 are revisited. These are considered as homogenous applications as their behavior on accessing main memory should be mostly constant over the whole execution time. Later, an inhomogeneous test application will be analyzed. Its access rates for main memory vary strongly between single phases of the execution.

To prove our approach on Process Cruise Control test applications are analyzed on execution time, energy and mean power consumption. All three values are measured both with and without Process Cruise Control. In the latter case applications run with a constant speed of 733 MHz.

Each value from application of Process Cruise Control was normalized to the corresponding value without event-driven frequency control. Savings, respectively leakages, are displayed as percentages in figures 5.1 and 5.2.

5.2.1 Homogenous Applications

To prove our approach for homogenous main memory access rates the initial test applications `find/grep`, `gzip`, `djpeg` and `factor` from section 3.2.1 are revisited. One additional test application, namely `rjpeg`, is introduced anew for discussion about weaknesses of the model.

As expected, performance loss ranges below or about 10 % for all four original test applications. With mean power reduction always ranging above corresponding performance loss, energy is saved for each test.

Along, about 10 % of energy was saved for memory intensive applications as `find/grep` and `gzip`. Less energy was saved for tests more concentrating on CPU like `djpeg` and `factor`. These savings solely result from throttling CPU speed.

Test `rjpeg` shows how the model fails in limiting performance loss if the application strongly uses the Linux pipe mechanism. Accounting main memory accesses caused by pipe operations of the connected applications does not work cleanly. More memory transactions than accounted are actually performed. In consequence, Process Cruise Control throttles exaggeratedly.

While both tests `find/grep` and `rjpeg` make use of pipes, why is the first one not affected by missing pipe accounting? To find out their command lines have to be regarded...

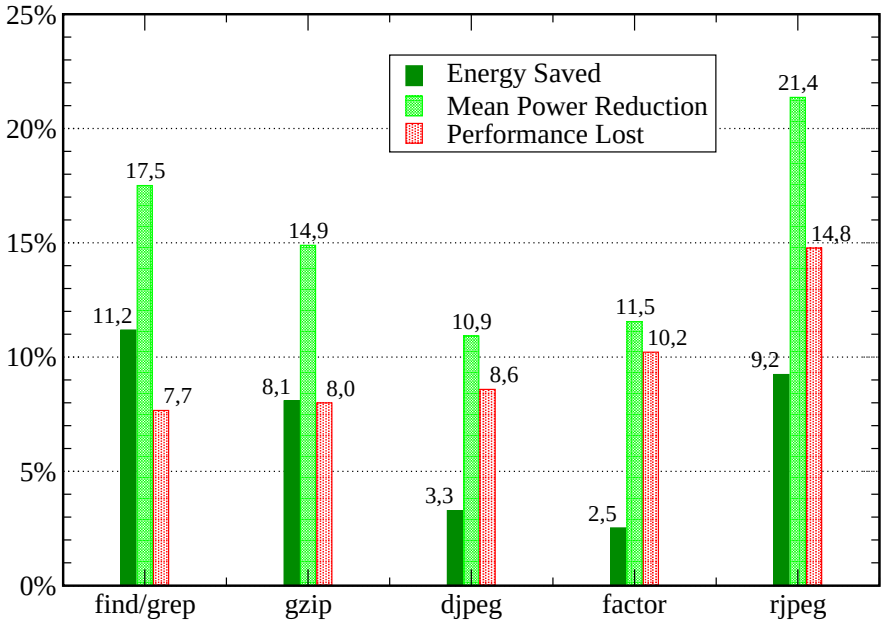

Figure 5.1: Savings/Leakages of Homogenous Applications

`find/grep`: `find` <*dir-list*> `-type f` | `grep` <*string*>
`rjpeg`: `djpeg` <*file*> | `cjpeg`

Computation of the first test is concentrated to `grep` in our test constellation. Only a relatively small list of files is passed through the pipe of `find/grep`. Copy actions of the kernel don't carry weight here. In contrast, `rjpeg` balances execution to decompression and recompression. Considerable amount of data — the uncompressed image — has to be passed through the pipe causing more copy actions in the kernel.

A more elaborate accounting of kernel activities is desirable. Mechanisms like resource containers and performance counters with distinction between kernel and user space could assist further approaches.

5.2.2 The Inhomogeneous Application

To prove the real usability of our approach an inhomogeneous test application was designed. Limitations of the target system denied using existing benchmarks etc., which have distinguishable phases of execution with varying main memory access rates. As a matter of fact we were not able to allocate enough space in RAM-disk for storing these tests and appendant runtime environments.

The inhomogeneous test used here involves several subtasks which are described in table 5.2. The split up savings and leakages of the individual tasks along with the totals are shown in figure 5.2.

As expected, very memory intensive tasks like `free db` and `memcpy` gain energy-efficiency from Process Cruise Control without losing measurable performance. Average memory access rates from `fill string` and `file rw` result

Sub-test:	Description:
`loop`	Simple `for`-Loop
`fact1`	Multiplication Loop
`primes`	Eratosthenes' Sieve
`fact2`	Conditional Multiplication Loop
`file rw`	Write Data to a File and Verify
`fact3`	Recursive Multiplication Loop
`read db`	Read Word Database into Memory
`sort db`	Quick Sort
`search db`	Full Text Search in Database
`fill string`	Dump Word Database Using `strcat`
`free db`	Release Memory
`memcpy`	Swap Blocks of Memory

Table 5.2: Sub-Tasks of the Inhomogeneous Application

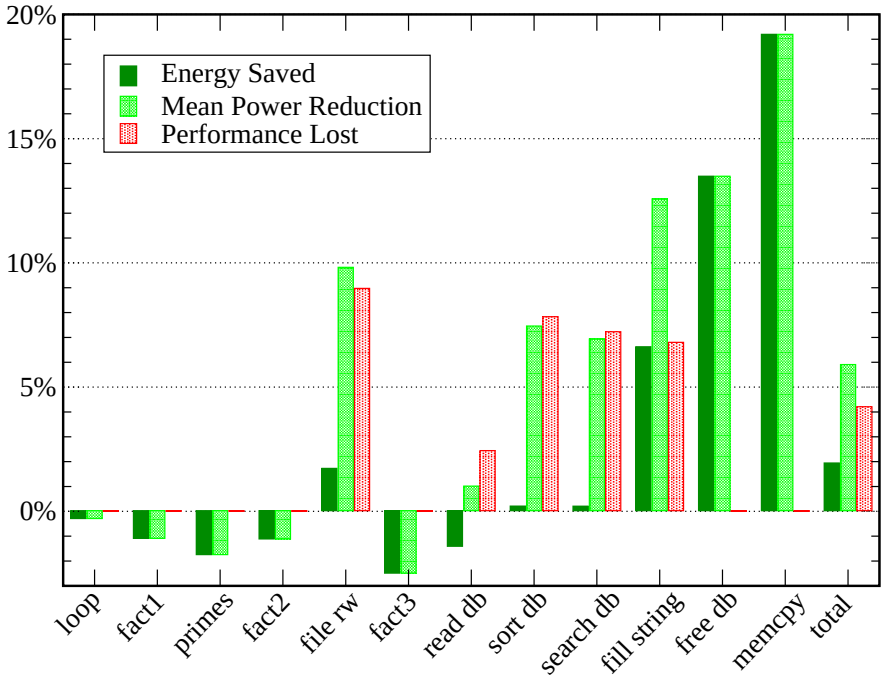

Figure 5.2: Savings/Leakages of the Inhomogeneous Application

in energy saving with considerable but limited impact on performance.

Tests with low memory access rates as `read db`, `sort db` and `search db` become slowed down considerably. However, constraints for maximum performance loss are satisfied. Depending strongly on CPU speed, the performance of these sub-tasks suffers most, compared to other tests. At least, little energy is saved while performance decreases considerably.

Sub-tasks with very low memory access rate fall below the heuristic border line attributed with 733 MHz (see section 3.3.4). Accordingly, `loop`, `fact1`–`fact3` and `primes` are not slowed down making performance loss disappear. Energy leakages of these tests result from logging overhead. The overhead was caused by additional memory accesses of `klogd`. Logging with `klogd` had to be introduced to monitor schedule progress detailed below.

Rightmost bars in diagram 5.2 display overall values of the inhomogeneous test. As it concentrates on alternating resource usage between sub-tests and has many sub-tasks, which are not very memory intensive, overall savings rate relatively low.

Figure 5.3 displays trends during execution of the inhomogeneous test: Red graphs represent values measured at a constant execution speed of 733 MHz while green graphs show results from activated Process Cruise Control. The upper diagram denotes power consumption while schedule progress indicating CPU speed for the current point of execution and durations of the subtasks are displayed in the lower part.

5.3 Effective and Predicted Ideal Speed

The model generation algorithm has to find border lines that best match test points gathered from micro-benchmarks. This step is correspondent to a quantization of the ideal speed distribution in the prediction diagram. Consequently not every micro-benchmark point falls into the zone with matching ideal speed.

Furthermore, points collected from tests or applications which, of course, were not used for the generation of the model may have another effective ideal speed than the model predicts. The effective ideal speed is the speed at which the test ran most energy-efficiently. The predicted ideal speed is that speed the test would be scheduled under Process Cruise Control. Figure 5.4 displays the position of the test points from the homogenous applications at an execution speed of 333 MHz. The measured ideal speed of a test is shown by its colour and shape. The predicted ideal speed of the same point is given by its position according to the query method explained in section 3.3.5.

While tests `find/grep`, `djpeg`, `factor` and `gzip` have equal predicted and effective ideal speeds the latter test is near a model border already. The test `rjpeg` is scheduled with incorrect ideal speed. Its predicted ideal speed is higher than the effective one. Although, prediction error in scheduled speed resulting from different predicted and effective ideal speed is ranging, relatively low, at about 10 %.

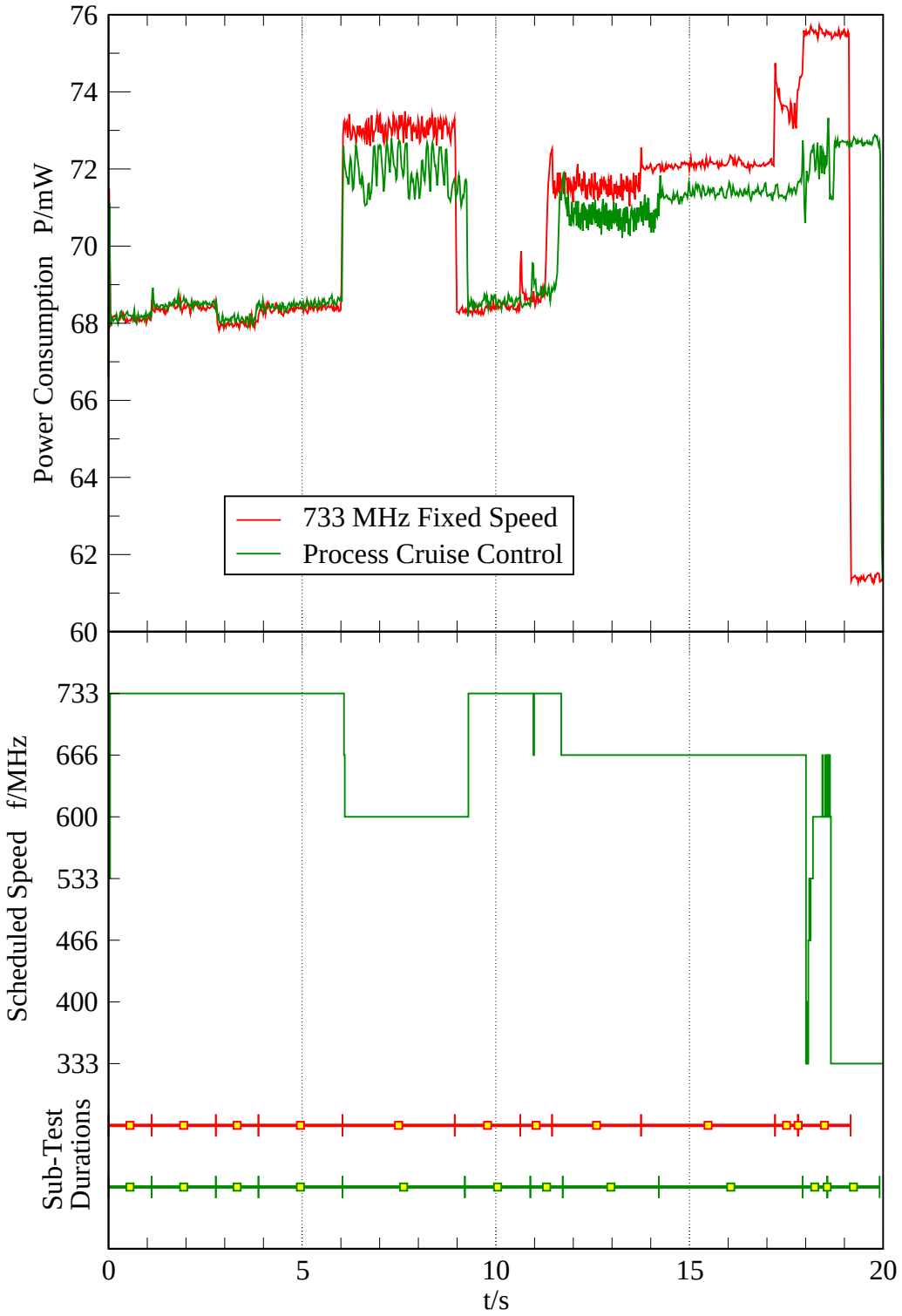

Figure 5.3: Power and Scheduling Trend of the Inhomogeneous Application

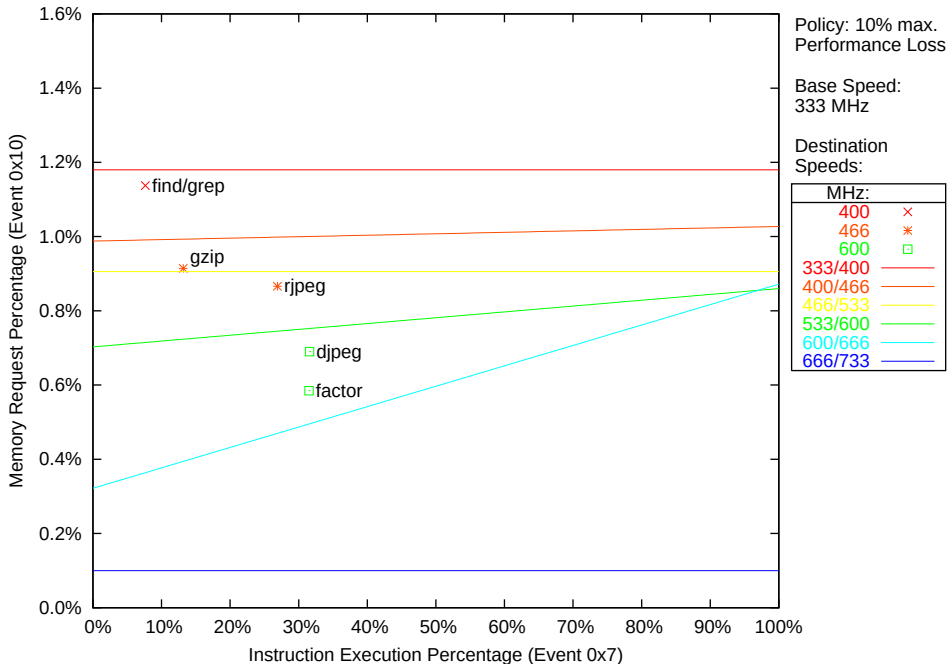

Figure 5.4: Ideal Speeds of Homogenous Applications

5.4 Shortcomings of the Implementation

Of course, the current implementation of Event-Driven Frequency Scaling is an experimental platform. While creating, debugging and using this implementation some weak details showed up.

Code Locality

The current implementation of Process Cruise Control consists of four separate kernel modules. As each module becomes allocated into its own memory page TLB misses are generated for every call between the four modules. Resulting overhead could be decreased by merging participant code into one single kernel module.

Further, module code is called back from the scheduler referring in additional overhead. Instead, this code should be integrated into the scheduler itself thus moving it into the kernel. The activation of this code may be controlled by semaphores when loading and unloading the module. Time consuming callbacks can be avoided then.

Policy Query

Querying the policy currently needs time-expensive calculations to consider an ideal speed for the given query values using a table with gradients and axis intercepts presented as decimal values. Workarounds for missing floating point operations in the kernel not only take much time for calculation but are tricky

to implement as we have observed from practical reimplementation in student assignments [FS02]. In addition, the current model is only capable of linear borders between partitions of equal destination speeds. As is thinkable, other platforms could yield model diagrams with borders that could be approximated with polynomial functions only.

Section 5.3 deals with the difference between effective and predicted ideal speed. This leak in the model results from design: On the one hand, the border generation quantizes the initially fine-spread ideal speed distribution; on the other hand, knowledge of ideal speed zones for prognosis is limited due to a limited number of micro-benchmarks influencing model creation.

Weak Accounting

As mentioned in section 5.2.1 current Process Cruise Control has problems with the pipe mechanism in the Linux kernel. This lack appears due to missing policies for kernel handlers and interrupts. For example, a user process that caused or participated in causing some kernel transactions or background activities: The energy for executing these operations is not accounted correctly to originator processes, that is it is not charged for the involved processes. The current implementation lacks mechanisms to accumulate information on activities related only indirectly to inducing processes. Such accounting would be achieved with multiple policies for different code sections, for example.

Despite that, means for data acquisition are weak. Performance monitoring counters were initially created as a mechanism for accounting events corresponding to system efficiency, data throughput and performance profiling. Abusing these counters for energy or ideal speed prognosis has limited success as a consequence of the very beginning of this approach. Currently, no other means than performance monitoring units are available for collecting resource usage information. Gathering such information directly from similar counters dedicated to energy accounting is not possible as such units have not been implemented yet. Additionally, performance monitoring counters used in this system do not provide any distinction between kernel and user space. Using separate accounting policies for both processor modes would be inefficient as underlying software could not be supported or accelerated by hardware.

Like other processors, the Intel 80200 just contains one time stamp and two performance monitoring counters. Along with that, the current implementation of the model makes no use of counter multiplexing for gaining statistics about more than two events. For considering only singular effects of energy consumption, simultaneous sampling of two events may be sufficient. To obtain information on the energy state of a complete system considering only two events would not serve the purpose.

6 Conclusions

6.1 Aims of this Work

The approach introduced in this thesis achieves the saving of energy by Event-Driven Frequency Scaling. The method for saving energy by these means was called *Process Cruise Control*.

Before finding any methods a target system had to be selected upon requirements for this approach: The system had to provide performance monitoring capabilities as well as functions to scale processor core clock. Consequently, Intel's 80200 processor was used as target system. While measuring power consumption and performance loss of the system for several test applications and benchmarks at different processor speeds scenarios were found when throttling CPU by frequency scaling saves energy while only slightly impacting performance. These usage patterns were identified with accumulated memory accesses.

In the next step, values collected from the performance monitoring unit of the processor were investigated for indicating accumulated memory accesses. Thus, by relating event rates from performance counters to possible energy savings while limiting maximum performance impact a model for speed prognosis was found: This policy predicts an ideal speed for an application from the event rates that are created. The ideal speed is selected with respect to a limited performance decrease of the controlled application.

The model found appropriate for this approach was related to other work that directs to the same intention: systems that are energy-aware, controlling their power state on their own, thus adapting automatically to the current need and urgency of computation. As well, concepts were illuminated that would be helpful for solving weaknesses of the approach which are discussed later.

Subsequently, the Linux kernel was modified and extended to support frequency scaling, performance monitoring and — basing upon these — Process Cruise Control. The actual implementation showed the conceptual feasibility of Event-Driven Frequency Scaling: Energy can be saved by watching resource usage and throttling applications which would waste energy otherwise. A control loop is built to regulate execution speed of tasks to save energy. Resulting performance impact is limited by the policy of Process Cruise Control.

This approach works without support from the application. Programs to be used in a system with Process Cruise Control do not need to be energy-aware. Thus, any application may be used as is; neither recompilation nor modification are required.

The policy does not need any preliminary information of the applications

that are supposed to run under its control. The only data to be gathered in advance are attributes of the target platform. These characteristics are used to generate model data which in turn are the working set for the Process Cruise Control policy. Although sampling platform data with micro-benchmarks and automatically generating the model are time consuming jobs, they have to be done only once for each platform.

Finally, the implementation was examined for energy efficiency. It was shown that the current approach satisfies its design goals, mostly: while limiting performance impact a considerable amount of energy was saved. Furthermore, shortcomings of the current model and implementation were discussed. Consequences to solve those problems will be displayed next.

6.2 Future Work and Improvements

Matrix Model

Introducing a matrix oriented model eliminates all of the problems related to overhead and inflexibility while querying the policy. Ideal speeds would then be predicted by just looking up a cell of the matrix. The lookup position directly corresponds to event percentages. As unsigned integer values are used to indicate the matrix arithmetics for calculating event percentages may be done with unsigned integer operations as well. Implementation would be eased by this concept. Further, borders of ranges with equal ideal speeds may run in any form over the whole graph. It is only resolution, which limits the accuracy of the prognosis diagram.

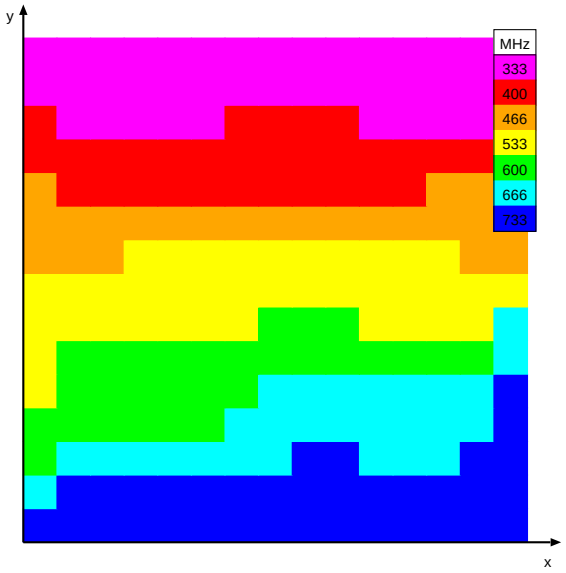

Figure 6.1: Hypothetical Matrix Diagram

Figure 6.1 illustrates what such a matrix model could look like. This ideal speed prognosis diagram is purely hypothetical and just meant to demonstrate the independence from linear borders.

In a first attempt a matrix model could be obtained from the line oriented model. More straightforward, the matrix model would be generated directly from the micro-benchmark results. When only having test points distributed unevenly, a model generation algorithm for averaging points with different ideal speeds in same or neighbouring matrix cells is still needed.

Even more sophisticatedly, test points of micro-benchmarks may be positioned directly with matrix cells. Such approaches base on accurate knowledge of the target system and require a generator for micro-benchmarks with a specified event rate [JBM01]. As the coordinates of test points result directly from measured event rate equidistant test points could be placed in the model diagrams, then. In this case, a model generation algorithm would become superfluous.

Refined Accounting

Resource containers [BDM99] are a fundamental concept that would help joining statistics accumulated from this approach with other strategies for energy aware housing with resources which are provided by the kernel. As well, interrupts and other background kernel activities may be accounted directly to processes via resource containers.

Along with elaborating accounting means of data acquisition may be improved. More detailed statistics about resource usage would yield a better basis for spotting energy sinks and help to assign energy consumption to single processes. Such improvements include larger sets of counters for simultaneously sampling events. Along with that, events available for energy monitoring counters should have close reference to different energy sinks in the system. Further, still existing performance monitoring could be used to concentrate on limiting performance impact of the system. Consequently, more complex and accurate models for energy consumption and ideal speed prognosis would be possible.

To observe such actions more precisely multiple sets of energy monitoring counters would be helpful. Based on these, kernel and user space actions could be monitored separately. Likewise, other sets of counters could be used for handlers in the kernel and its interrupt routines. Distinguishing counters on level of task privileges — for example kernel and user mode — would not blow up time critical interrupt handling routines or the system call mechanism while supporting differentiated accounting of energy. However, before introducing new hardware these improvements can be evaluated in advance by software support emulating multiple sets of counters. Moreover, software approaches could multiplex performance monitoring counters widening the sight of the model while not yet having more performance counters in current processors.

7 Kurzzusammenfassung

Die vorliegende Diplomarbeit schildert ein Verfahren zur Energieeinsparung durch eine dynamische Anpassung der Prozessortaktfrequenz. Indizien für eine mögliche Drosselung von Prozessen werden dabei mittels Ereigniszählern des Prozessors gesammelt. Die Arbeit baut auf bereits vorgestellte Methoden [Bel01a], [JM01] zur Energievorhersage durch Ereigniszähler auf. Das System hat kein Vorwissen über die Applikationen und kommt ohne deren Mithilfe aus, wie sie zum Teil in anderen Implementierungen [MGG01] nötig ist. Dafür ist das vorliegende Modell abhängig von der eingesetzten Plattform. Betrachtungen aus dieser Arbeit können jedoch auf andere Plattformen angewandt werden. Von der Konzeption bis zur Implementierung wird wie folgt vorgegangen:

Zunächst erfolgt die Beschreibung des technischen Aufbaus der Experimentierumgebung. Zum Einsatz kommt die IQ 80310-Entwicklungsumgebung mit dem 80200-Prozessor von Intel, da dieser sowohl Ereigniszähler als auch dynamische Geschwindigkeitsanpassung unterstützt. Weiterhin lässt sich relativ einfach die Leistungsaufnahme des Systems überwachen. Aufgrund seiner Veränderbarkeit und Flexibilität wird Linux als Betriebssystem eingesetzt.

Vor dem Entwurf einer Vorschrift zum Energiesparen muss nach Effekten im System gesucht werden, die Möglichkeiten zur Reduktion der Leistungsaufnahme bei vergleichbarer Anwendungsleistung bieten. Situationen mit gehäuften Speicherzugriffen bieten dabei eine solche Möglichkeit, die für das Modell Verwendung findet: Ohnehin ausgebremst durch den Speicherbus kann der Prozessor in seiner Arbeitsgeschwindigkeit während solchen Zugriffshäufungen gedrosselt werden, ohne dass nennenswerte Verzögerungen eintreten. Die Leistungsaufnahme des Systems wird dabei gesenkt.

Bei realen Prozessen können Speicherzugriffe nicht vollständig isoliert werden, eine Drosselung geht immer mit einer Reduktion der Anwendungsleistung einher. Um dennoch Vorhersagen für eine Drosselung machen zu können, wird der maximale Verlust an Anwendungsleistung beschränkt. Zusammenhänge der Ereigniszählerwerte mit Speicherzugriffsverhalten und möglicher Drosselbarkeit von eigens dafür entwickelten Tests werden als Nächstes gesammelt. Wie sich zeigt, können im vorhandenen System die Häufigkeiten zweier zählbarer Ereignisse — ausgeführte Instruktionen und Bus-Anfragen — zu einer vertretbaren Vorhersage für eine *ideale Geschwindigkeit* des Prozessors verwendet werden. Bei der idealen Geschwindigkeit wird bei den Applikationen möglichst viel Energie gespart und dennoch das Limit für den Abfall der Anwendungsleistung eingehalten. Ein entsprechendes Vorhersagemodell wird aufgestellt; es basiert auf diesen Häufigkeiten und auf der linearen Auftrennung des Messwertraumes in verschiedene Zonen idealer Geschwindigkeit. Das Verfahren wird in Bezug

zu anderen Arbeiten gesetzt und verglichen.

Eine Implementierung des Vorhersagemodells und der unterliegenden Regel- und Messmethoden zeigt die Praktikabilität des erläuterten *Process Cruise Control*-Verfahrens. Messungen am fertigen System dienen einer Diskussion von tatsächlicher Energieersparnis und Reduktion der Anwendungsleistung. Der Anteil von zusätzlichem Aufwand für die Messung, Bewertung und Steuerung wird betrachtet. Weiterhin werden auch Schwächen des Modells und der Implementierung aufgezeigt: Die Aufnahme der Ereignishäufigkeiten erfolgt nicht lückenlos, die Bewertung der Messwerte ist zu aufwändig und die nutzbaren Ereignisse lassen zu wenig direkten Aufschluss über tatsächliche energierelevante Aktivitäten.

Entsprechende Verbesserungsvorschläge werden nach einer Zusammenfassung der Arbeit vorgestellt und umfassen zunächst eine Vereinfachung und Verallgemeinerung des Modells, die Verwendung weiterführender Konzepte für die Akquisition und eine Verwaltung der Messwerte, wie zum Beispiel Resource Container [BDM99]. Ein Vorschlag von zentraler Bedeutung ist die Einführung von speziellen energiebezogenen Ereigniszählern, um genaueres Wissen über Energiever(sch)wendung im System zu bekommen. Damit schließt diese Arbeit.

List of Tables

List of Figures

Bibliography

[AMD02a] AMD: The Alchemy Au1100 from AMD, Internet Edge Processor Data Book, Apr 2002
`http://www.alchemysemi.com/`

[AMD02b] AMD: Athlon Processor Model 4 Data Sheet, Apr 2002
`http://www.amd.com/`

[BDG+00] S. Browne, J. Dongarra, N. Garner, K. London, P. Mucci: A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings of the Conference on Supercomputing SC2000*. Nov 2000

[BDM99] Gaurav Banga, Peter Druschel, Jeffrey C. Mogul: Resource Containers: A New Facility for Resource Management in Server Systems. In *Operating Systems Design and Implementation*, 45–58. 1999

[Bel01a] F. Bellosa: The Case for Event-Driven Energy Accounting. Technical Report TR-I4-01-07, University of Erlangen, Department of Computer Science, Jun 2001

[Bel01b] F. Bellosa: Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management. Technical Report TR-I4-01-11, University of Erlangen, Department of Computer Science, Nov 2001

[BM02] J.D. Bakker, J.A.K. Mouw: Linux Embedded Radio Terminal design page, Nov 2002
`http://www.ict.its.tudelft.nl/~erik/open-source/LART/`

[FS02] Matthias Färber, Holger Scherl: AKBP/ES II: Team 4, Cruise Control, Feb 2002
`http://www4.informatik.uni-erlangen.de/Lehre/WS01/V_AKBPES2/Skript/team4_cruise.pdf`

[Int00a] Intel: *Intel 80200 Processor based on Intel XScale Microarchitecture Developer's Manual*, Nov 2000

[Int00b] Intel: *Intel SpeedStep Technology*, Jan 2000

[Int00c] Intel: *Intel XScale Microarchitecture Technical Summary*, Jul 2000

[Int01] Intel: *Intel IQ80310 Evaluation Platform Board Manual*, Jul 2001

[JBM01] Russ Joseph, David Brooks, M. Martonosi: Live, Runtime Power Measurements as a Foundation for Evaluating Power/Performance Tradeoffs. In *Workshop on Complexity Effectice Design WCED, held in conjunction with ISCA-28*. Jun 2001

[JM01] Russ Joseph, M. Martonosi: Run-time Power Estimation in High-Performance Microprocessors. In *The International Symposium on Low Power Electronics and Design ISLPED'01*. Aug 2001

[Mei01] Meilhaus Electronic GmbH: *Handbuch ME-2000/2600 PCI/cPCI, Revsion 1.8D*, Sep 2001

[Mei02] Meilhaus Electronic GmbH: ME-2000/2600 Linux Driver, Apr 2002 `http://www.meilhaus.de/download/me-boards/linux/me2600.tar.gz`

[MGG01] Charles B. Morrey III, Marco Gruteser, Dirk Grunwald: Simple Mechanisms For Dynamic Voltage Scaling In General Purpose Operating Systems. Technical report, Department of Computer Science, University of Colorado, Boulder, Nov 2001

[MLH+01] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, Raj Rajkumar: Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling. Technical report, Real-Time and Multimedia Systems Lab, Department of Electrical and Computer Engineering, Carnegie Mellon University; Austin Research Laboratory, IBM, Nov 2001

[Muc01] Philip Mucci: The Performance API PAPI. White Paper of the University of Tennessee, Mar 2001 `http://icl.cs.utk.edu/projects/papi/`

[Pet01] Mikael Pettersson: Linux x86 Performance-Monitoring Counters Driver, Nov 2001 `http://www.docs.uu.se/~mikpe/linux/perfctr/`

[PLS00] J. Pouwelse, K. Langendoen, H. Sips: Dynamic Voltage Scaling on a Low-Power Microprocessor. In *Proceedings of the International Symposium on Mobile Multimedia Systems & Applications MMSA 2000*. Nov 2000

[VHS98] Jörg Vollrath, Markus Hübl, Ernst Stahl: Power Analysis of DRAMs. In *Proceedings of the Seventh Asian Test Symposium*. 1998

[Win02] Christian Winter: *Design and Implementation of a Power Measurement System*. Pre-master thesis, University of Erlangen, Department of Computer Science, 2002