Karlsruher Institut für Technologie

Multi–Core Energy Accounting

Studienarbeit
von

Johannes Weiß

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch

Bearbeitungszeit: 15. Juni 2011 – 15. Oktober 2011

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der gültigen Fassung beachtet habe.

Karlsruhe, den 14. Oktober 2011

Johannes Weiß

Multi–Core Energy Accounting

Johannes Weiß <mcea@tux4u.de>

Typeset using LaTeX on October 14, 2011, 13:32.
Repository Hashes: tree: 07158b3af604fd0ac753e9d0e8107e45f79eea32, last commit: 0daf6414adb914209ebb31a5be3cd4c3fea42d14

Contents

Chapter 1

Introduction

Energy is a crucial resource especially for mobile devices. Since the available energy is either limited—on mobile devices—or can become expensive—on devices connected to the regular power grid—as little as possible should be used. Even though most modern operating systems try to maximize the CPU usage by lowering the frequency [22] to eventually maximize the energy efficiency, this reaction is not always appropriate. A lower CPU frequency may even decrease energy efficiency [22, 25].

The basis to intelligently minimize the energy consumption is per–task energy accounting because it reveals where exactly the energy is consumed. For every running process the operating system should be aware of the present contribution to the machine's total power consumption. Furthermore, the overall energy consumed by a process should be known after its termination for accounting purposes.

Thus, for being able to develop good power management strategies, a good live energy estimation is crucial. The approach in this study thesis is to use the CPU's *performance monitoring counters*. Since the turn of the millennium [1] there have been many papers [2, 3, 14, 15, 25] doing energy estimation using performance counters with impressive results.

Besides these points, various other applications of energy estimation are possible: Temperature control [15] and thermal management [18] are just one auxiliary field. Accounted energy may also serve as a customer cost model for computing centers [2], because it much more accurately reflects the real costs than simple computing time based models. Another possible application in this field are migration decisions of either virtual machines between real machines in clusters or processes between processing units [19]. Power–aware

scheduling can may help to improve the computing time efficiency.

1.1 Charges and Restrictions

Because of the limited amount of time and the start from scratch, some energy–relevant features of processor and operating system have been disabled. Since all of these individually raise some kind of events, this is not seen as a major drawback of this work. Future work may certainly be able to flawlessly integrate them into the known model. In particular, the following features lasted disabled:

- ▷ Dynamic Frequency and Voltage Scaling [10]
- ▷ Hyper–threading [9]
- ▷ ACPI Processor States other than C0 [5]
- ▷ Intel® Turbo Boost Technology 2.0 [11]

In addition to the advanced processor and operating system features mentioned above, all auxiliary processing units were not taken into account. The problem with the auxiliary processing units, such as the floating point unit (FPU), MMX [27], SSE [29] and AVX [16], is that they are mostly uncovered by the processor performance events [7]. They somehow act as a black box not revealing the work they do internally. Hence it is almost impossible to count their energy using a performance event model.

1.2 Acknowledgments

- ▷ Prof. Dr. Frank Bellosa, my advisor
- ▷ Rainer Dosch who designed and soldered the circuits
- ▷ Simon Kellner for the introduction into the subject and valuable hints
- ▷ James McCuller who is responsible for the university workstations I used

1.3 Preliminaries

To be able to distinguish ordinary text from special entities, different fonts and decorations have been typeset. File system paths (e. g. `/bin/ls`), CPU performance events (e. g. `CPU_CLK_UNHALTED`) and measuring channels (e. g. `TRIGGER`) appear in a `typewriter` font. Proper nouns of products, programs and libraries (e. g. R's leaps package) are typeset using a sans–serif font. The names of programs and libraries specifically developed for this work (e. g. DATADUMP) appear in SMALL CAPS. Finally, typescripts of terminal sessions are decorated as in the following example:

```
$ echo 'Hello␣World!'
Hello World!
```

The *International System of Units (SI)* is used wherever appropriate. Additionally, the following new definitions are introduced: S meaning *samples*, MiB and KiB (1 MiB $\hat{=}$ 1024 KiB $\hat{=}$ 1 048 576 B = 1048576 byte).

Chapter 2

Technical Prerequisites

In the following chapter the hardware examined in this work will be discussed in detail.

2.1 Products

The examinded computer consisted of:

- ▷ CPU: Intel® Core™ i7–2600K Processor (*Sandy Bridge* microarchitecture)
- ▷ Motherboard: ASUS P8P67–M PRO (using a dedicated video controller)

2.2 Sandy Bridge Characteristics

The most evident characteristics of the Intel® Core™ i7–2600K Processor are the four cores (on one chip) and the uniform distribution of the caches (see figure 2.1). All caches except for the last–level cache (L3) are present on each core [4]. The CPU's *performance monitoring unit* (PMU) appears—because of its central importance—in a separate chapter (2.2.1). A short overview of the key features follows [8]:

- ▷ Number of cores: 4
- ▷ CPU clock speed: 3.4 GHz

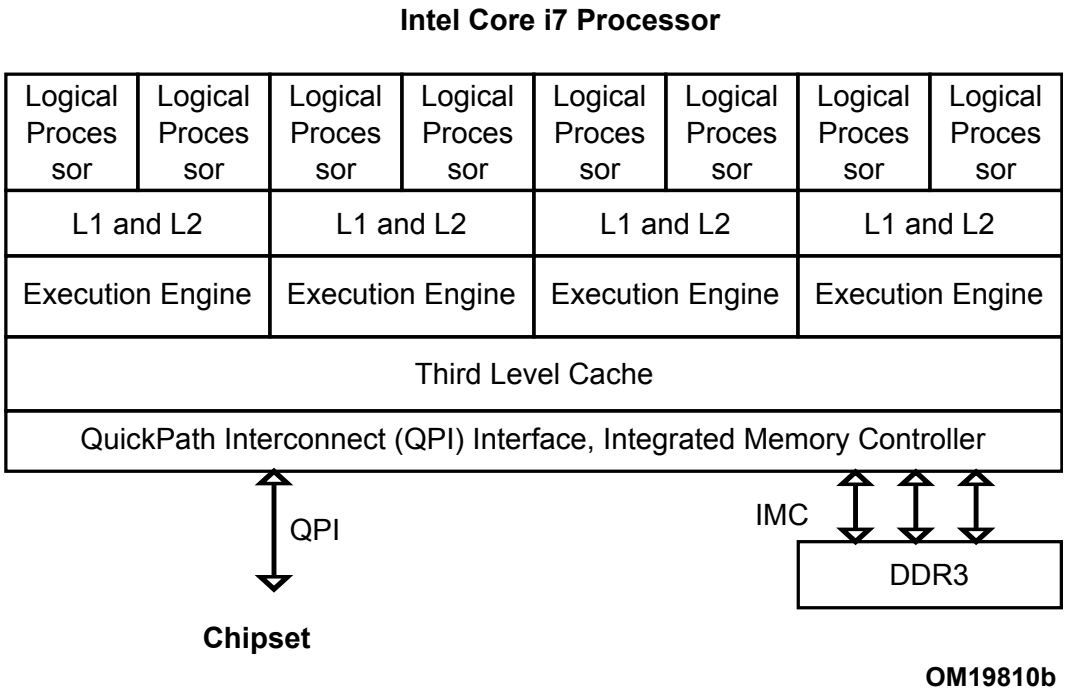

Figure 2.1: Intel® Core™ i7–2600K Processor cache organization (taken from [12])

- ▷ L1 cache of 64 KiB per core [12]
- ▷ L2 cache of 256 KiB per core [12]
- ▷ shared L3 cache of 8 MiB [12]

2.2.1 Performance Monitoring Unit

The CPU's *Performance Monitoring Unit* (PMU) is present since the Intel® Pentium processor. It is able to monitor several of the CPU's performance parameters while the system is running. Originally meant for tuning system and application performance [13], it is mostly used by compiler developers.

In prior work [1, 3, 15, 22, 25] some of these performance events have proved to be somehow related to the power consumption of the CPU. The selection of the events and the degree of their correlation to energy highly varies among CPUs, or at least CPU microarchitectures. Therefore, selection and fitting have been done again for the Intel® Sandy Bridge microarchitecture and in particular the Intel® Core™ i7–2600K Processor.

By examining the processor user's manual [7] 184 events have been found available and usable. Because the CPU is only able to count eight (four in Hyper–threading [9] mode) user–programmable performance events simultaneously [12] the most useful events have to be selected. In addition to the eight (four) events, the CPU provides three counters for fixed events, that will be counted in any case: `CPU_CLK_UNHALTED.REF_TSC`, `CPU_CLK_UNHALTED.THREAD` and `INST_RETIRED.ANY`.

The selection, configuration and usage of these performance event counters is done via special *model specific registers* (MSRs) [13]. In this work a more high–level approach via the perf_event-API (unofficial documentation at [24]) of newer Linux Kernels and libpfm4 (see chapter 4.2.1) has been used.

2.2.2 Architectural Differences between Sandy Bridge and its Predecessors

The Intel® *Sandy Bridge* microarchitecture is a further development of the Core and Nehalem architectures [4]. Among other simplifications of the branch prediction unit, the special loop predictor has been discontinued presumably to reduce the overall pipeline length and to minimize the misprediction penal-

ties [4]. For speed improvements a micro–operation (μop) cache and macro–operation fusion have been introduced [4].

Chapter 3

Design

3.1 Big Picture of the Setup

As figure 3.1 illustrates, an additional workstation—the *Examining Workstation (EW)*—has been used while evolving this thesis. The *System under Test (SuT)* counts the CPU's performance events itself and the EW records the energy consumption using a measuring device in the meantime. The resulting data sets have thereafter been used to build up the energy model.

3.2 Measuring Setup in Detail

To fit the energy model afterwards, the current flows of the CPU and the motherboard's 12 V supply have to be measured. Voltage drops across the sensing resistor are measured using four–terminal sensing [26] to deduce the current flow. The motherboard's 12 V current flow has been measured in this thesis because it was unclear if the CPU is entirely fed by its own 12 V power supply.

Because the SuT and the EW (see figure 3.1) have to agree about the examination (time) interval a trigger wire is used. It is realized as a simple analog signal using the parallel port's *high* and *low* voltages.

This can be summed up to measure three potential differences. Since the measuring device (chapter 3.2.1) provides up to eight differential, analog input channels this seems easy at first. Unfortunately, two caveats apply: On the one hand according to the user's manual [21] the best measuring accuracy can

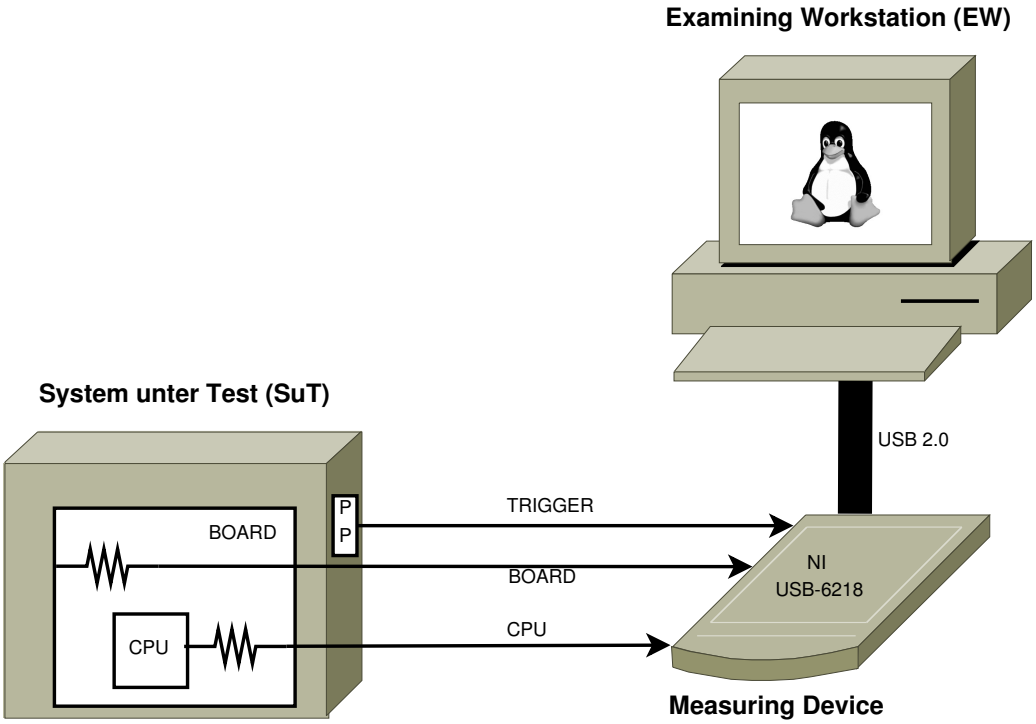

Figure 3.1: Measuring setup overview

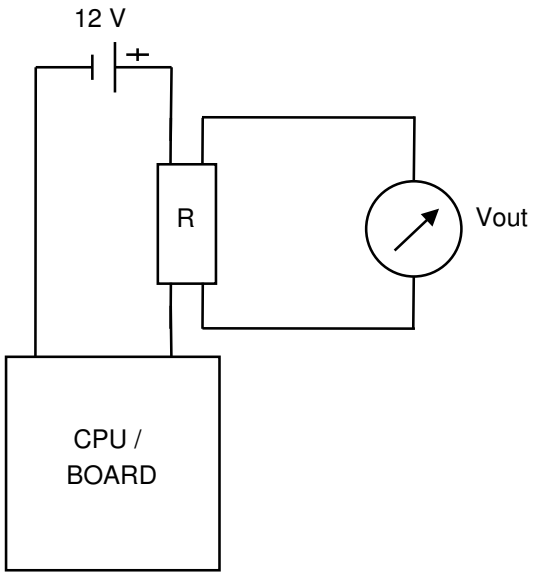

Figure 3.2: Measuring circuit for CPU (R=10 mΩ) and BOARD (R=5 mΩ)

be achieved in range of −200 mV and 200 mV. On the other hand the overall potential differences may not exceed ±10.4 V [20].

Choosing adequate sensing resistors for the `CPU` (R = 10 mΩ) and `BOARD` (R = 5 mΩ) channels (see figure 3.2) worked out well. The parallel port trigger wire has been a problem at first, though. The parallel port has a potential difference range of more than 200 mV and our test machine's port had a very different potential level than the `CPU` and `BOARD` channels, exceeding the allowed range of ±10.4 V. The potential equalizer illustrated in figure 3.3 solves both problems.

Finally, three differential channels `CPU`, `BOARD` (12 V supply only!) and `TRIGGER` in the range of ±200 mV and alike potential levels can be connected to the measuring device. The performance events get counted on the SuT itself which controls the trigger wire, too: The trigger is set to *On* directly before executing the program to examine and is set to *Off* promptly after its termination. To safely register all CPU energy consumption peaks, a high sampling rate of 50 kS/s is chosen. An exemplary plot of an examination can be found in figure 3.4.

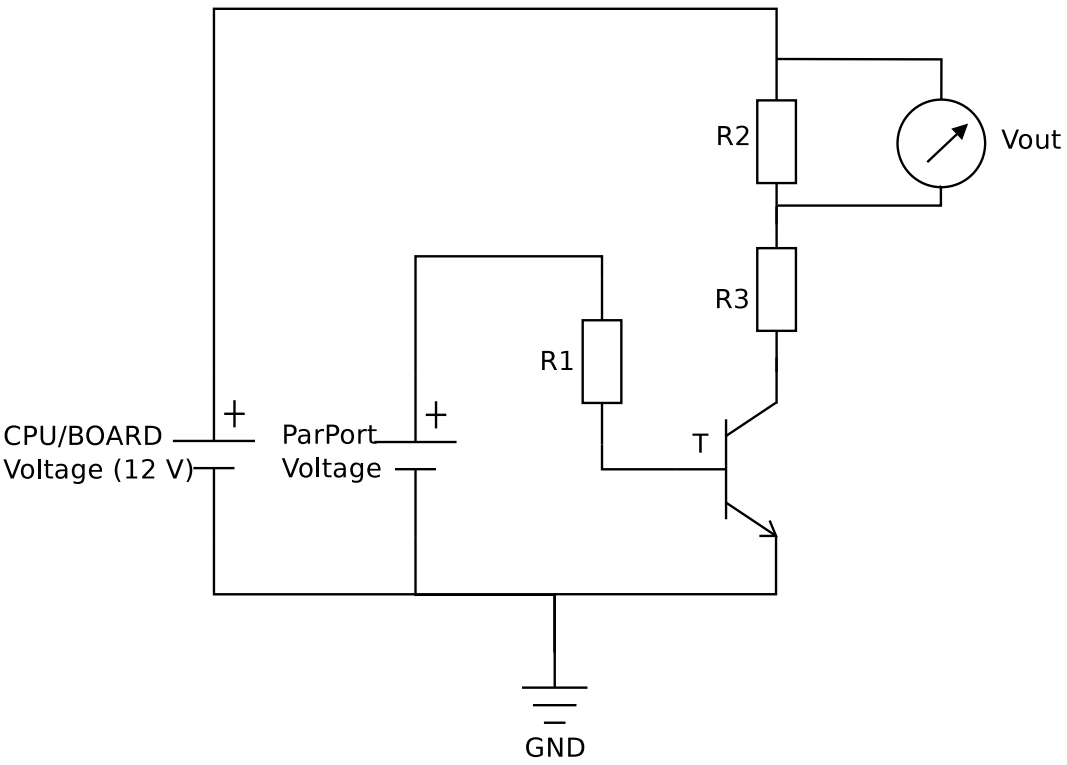

Figure 3.3: Potential equalizer

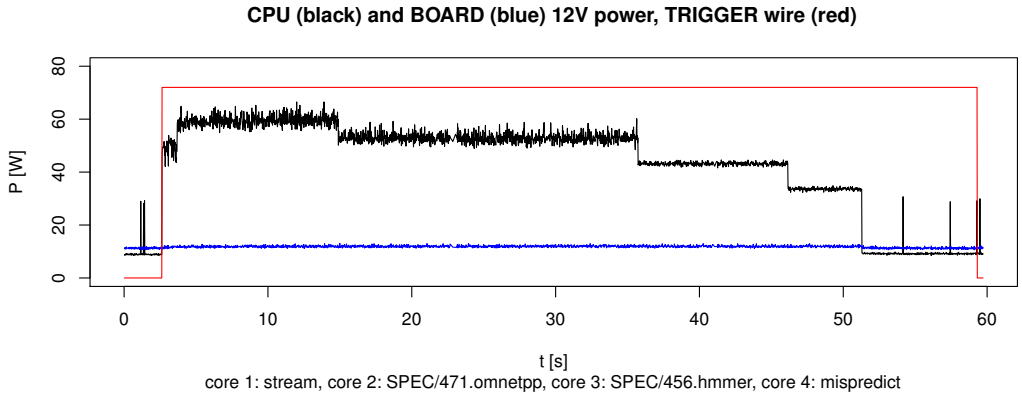

Figure 3.4: Sample examination

Figure 3.5: NI USB–6218 (picture from `http://www.pressebox.de/pressemeldungen/national-instruments-germany-gmbh/boxid/75241`)

3.2.1 Measuring Device

For measuring the voltage drops a NI USB–6218 from National Instruments (shown in figure 3.5) was chosen because its support for high sampling rates of up to 250000 samples per second (250 kS/s) and its high accuracy (accuracy $< 2.69\,\mathrm{mV}$) [20].

3.3 Calculation of the Electrical Work

From elementary physics (resistance R, voltage U, electric current I and instantaneous electric power P):

$$U = R * I \iff I = \frac{U}{R} \tag{3.1}$$

$$P = U * I \tag{3.2}$$

So, the current flow is calculable from the voltage drops across the measuring resistor (U_R). Accordingly, the instantaneous power can be calculated as:

$$P = U * I \tag{3.3}$$

$$= U * \frac{U_R}{R} \tag{3.4}$$

$$P = \frac{12V * U_R}{R} \tag{3.5}$$

Hence, integrating will result in the electrical work

$$W = \int P(t)dt \tag{3.6}$$

3.4 The Energy Model

The following chapters will define the term *energy model* along with the formal methods suggested to build such a model.

3.4.1 Properties

In this work, an energy model is considered as a linear function. It describes a system with n_c cores that is able to monitor n_e performance events per core simultaneously. Additionally to the per–core event counters, n_g global event counters make the model up. In the formal description (see chapter 4.3 for the practical implementation) we assume four functions providing the actual values:

- $c_g(i, t_0, t_e) \in \mathbb{N}^{\{1,\dots,n_e\}\times\mathbb{R}_{\geq 0}\times\mathbb{R}_{\geq 0}}$, the global event i's count in the time interval (t_0, t_e)

- $c_e(j,k,t_0,t_e) \in \mathbb{N}^{\{1,\dots,n_c\}\times\{1,\dots,n_g\}\times\mathbb{R}_{\geq 0}\times\mathbb{R}_{\geq 0}}$, the performance event k's count on core j in the interval (t_0,t_e)
- $w_g(i) \in \mathbb{R}^{\{1,\dots,n_g\}}$, the global event i's energy weight in J
- $w_e(j,k) \in \mathbb{R}^{\{1,\dots,n_c\}\times\{1,\dots,n_g\}}$, the weight of performance event k in J on core j

Though, an energy model equates to a linear function. The function's value is the electrical work performed between two instants of time t_0 and t_e:

$$W(t_0,t_e) = \sum_{i=1}^{n_g} c_g(i,t_0,t_e)w_g(i) + \sum_{j=1}^{n_c}\sum_{k=1}^{n_e} c_e(j,k,t_0,t_e)w_e(j,k) \tag{3.7}$$

The functions c_e and c_g contain the system's live data whereas the energy weight functions w_e and w_c can be calculated a priori as done in this work. Obviously, the selection of the events and their respective weights highly depend on the type of microprocessor. To calculate the electrical power the system consumes, typically an arbitrary frequency f (period length $T = \frac{1}{f}$) is chosen and the variables t_0 and t_e are set accordingly ($t_0 =?$, $t_e = t_0 + \frac{1}{f} = t_0 + T$). The instantaneous power is then calculated as:

$$P(t) = \frac{W(t-T,t)}{T} \tag{3.8}$$

$$= \frac{\sum_{i=1}^{n_g} c_g(i,t-T,t)w_g(i) + \sum_{j=1}^{n_c}\sum_{k=1}^{n_e} c_e(j,k,t-T,t)w_e(j,k)}{T} \tag{3.9}$$

3.4.2 Finding the Energy Weights

Having seen what exactly constitutes an energy model (chapter 3.4.1), it is crucial to find a small and significant set of performance events and appropriate energy weights. This chapter will focus on how to find the weights. A separate chapter (3.4.3) describes the downsizing of the event set. To find reasonable energy weights test program (also called benchmark) execution observations have to be recorded and evaluated. Each test program run record contains the following data:

- ▷ b, the point in time the run began
- ▷ e, the point in time the run ended
- ▷ The n_g values of the functions $c_g(1 \cdots n_g, b, e)$
- ▷ The $n_c * n_e$ values of the functions $c_e(1 \cdots n_c, 1 \cdots n_e, b, e)$
- ▷ The electrical work j the system performed between b and e

Since the fitting of the energy weights needs a large data set, a matrix representation is appropriate. So, n_o observations in the non–overlapping time intervals $(b_{1 \cdots n_o}, e_{1 \cdots n_o})$ lead to:

- ▷ $b_{1 \cdots n_o}$, the points in time the runs began
- ▷ $e_{1 \cdots n_o}$, the points in time the runs ended
- ▷ The matrix Cg containing the global counter values (see equation 3.10)
- ▷ The matrix Ce, containing the performance event counter values (see equation 3.12)
- ▷ The vector $j =< j_1, \cdots, j_{n_o} >$, containing the electrical work performed

$$Cg \in \mathbb{N}^{n_o \times n_g} \tag{3.10}$$

$$Cg = \begin{pmatrix} c_g(1, b_1, e_1) & \cdots & c_g(n_g, b_1, e_1) \\ \vdots & \ddots & \vdots \\ c_g(1, b_{n_o}, e_{n_o}) & \cdots & c_g(n_g, b_{n_o}, e_{n_o}) \end{pmatrix} \tag{3.11}$$

$$Ce \in \mathbb{N}^{n_o \times n_c n_e} \tag{3.12}$$

$$Ce = \begin{pmatrix} c_e(1, 1, b_1, e_1) & \cdots^* & c_e(n_c, n_e, b_1, e_1) \\ \vdots & \ddots^* & \vdots \\ c_e(1, 1, b_{n_o}, e_{n_o}) & \cdots^* & c_e(n_c, n_e, b_{n_o}, e_{n_o}) \end{pmatrix} \tag{3.13}$$

*) an arbitrary incrementation scheme—consistent with vector w—may be chosen

Using a linear regression (of equation 3.14), suitable energy weight vectors w_c and w_g can be found. The goal is to minimize the error term, i. e. $\sum \epsilon^2$ as small as possible.

$$j = Cw + \epsilon \tag{3.14}$$

$$\begin{aligned}
j &= \begin{pmatrix} j_1 \\ \vdots \\ j_{n_o} \end{pmatrix} \\
C &= \big(Cg|Ce\big) \in \mathbb{N}^{n_o \times n_g + n_c n_e} \\
&= \begin{pmatrix} Cg_{1,1} & \cdots & Cg_{1,n_g} & Ce1,1 & \cdots & Ce1,n_c n_e \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ Cg_{n_o,1} & \cdots & Cg_{n_o,n_g} & Cen_o,1 & \cdots & Cen_o,n_c n_e \end{pmatrix} \\
w &= \begin{pmatrix} w_g(1) \\ \vdots \\ w_g(n_g) \\ w_e(1,1) \\ \vdots \\ w_e(n_c,n_e) \end{pmatrix}
\end{aligned}$$

3.4.3 Minimizing the Set of Performance Events

It is not practical to take into account all events the microprocessor is aware of. Today's processors offer much more events than they can count simultaneously [12]. The function and matrix names of the previous chapter (3.4.2) also apply here. The single exception is that this chapter only focuses on the CPU's performance events. The global events are not taken into account here because they are pseudo–events not obtained by the limited PMU and therefore have no maximal quantity. The approach to find a subset of n_e events (of e_{max} available) used in this work can be summed up to the following two steps which will be discussed in detail later:

1. Generation of the matrix of performance event counters Ce and the corresponding vector of electrical works j
2. Obtaining Ce_{final} containing the n_e most correlating columns of Ce that will form the energy model's performance events

Thereafter, the performance events the model finally uses are found. The next step is to find the final energy weights as in the previous chapter (3.4.2).

Step 1: Generating Ce and j

(a) Choose p test programs which use the CPU differently. The test programs have to be independent from external events. We consider subsequent runs of a test program as equal.

(b) Divide the e_{max} available events in g disjoint, non–empty sets $E_{1..g}$ each of size up to n_e.

(c) For each set $E_{1..g}$, run all the p test programs and record the electrical work performed and the event counters of the set's events.

(d) The electrical work of each of the runs of a test program should be roughly equal. If they differ a lot, there is either a dependency on external events in the benchmark or the machine is otherwisely stressed. In the former case the test program selection should be improved, in the latter the affected test runs have to be repeated.

(e) Folding all the results leads to a vector $j \in \mathbb{R}^p$ containing the electrical work a run of each of the test programs performed. Additionally, a matrix Ce (as in chapter 3.4.2) accrues, containing each event counter's value for a run of each of the test programs.

Step 2: Deriving Ce_{final} from Ce

(a) Eliminate duplicate columns in Ce

(b) Eliminate columns that contain only zeros in Ce

(c) Eliminate linear dependent columns in Ce

(d) Generate all column combinations of size n_e (without repetition) of the remaining columns in Ce and deduce the energy weights (as in chapter 3.4.2)

(e) Ce_{final} is the combination with the smallest error term ($\sum \epsilon^2$)

Chapter 4

Implementation

4.1 Data Formats

This chapter will give a rough outline of the formats used to store data between the various phases of the analysis. The lingua franca for most complex on–disc formats are the schrieblesque Google™ Protocol Buffers. Though, when interfacing with external software such as R, other formats had to be chosen. The choice in favor of Protocol Buffers has been made because of *think XML, but smaller, faster, and simpler* (Protocol Buffer's slogan).

4.1.1 Shot IDs

A shot ID (shot identifier) is a string value uniquely identifying a measuring experiment. More precisely it is a character string containing only a defined set of ASCII characters. The following regular expression should match all valid shot IDs: `[a-zA-Z0-9@_-]{1,64}`.

Such shot IDs are always used to associate power and performance event measurements.

4.1.2 Electrical Power Data Point Files

The main matter of the *electrical power data point files* (called *data point files* for simplicity) is to log the CPU's power consumption by time. As described in chapter 3.2, voltage drops are measured and electrical power and work are calculated later on. This is why data point files do literally contain voltage drop

values by time. But since the sole reason they exist is to serve as calculation input later on, they are named after their future purpose.

For practical reasons the representation is designed rather flexible: It supports arbitrary sampling rates, an arbitrary number of (named) channels and high–resolution timestamps (up to 1 ns).

While researching for this thesis, three channels have been recorded with a sampling rate of 50 kS/s: `CPU`—the CPU's voltage drops—, `BOARD`—the motherboard's voltage drops (unused)— and `TRIGGER`—the trigger wire used to mark the periods of time the other machine counts performance events.

The data point files are also one notable exception of the *Protocol Buffers for everything* principle in this work. Protocol Buffers were not designed to handle large messages but to serve as individual messages *within* a large data set [6]. So (as you can see in appendix B) small Protocol Buffer messages are streamed one after another. The size of one of these chunks is not specified. Usually one makes up a chunk when he receives one from a measuring device. The rule of thumb is: Make them as large as practical, but not too large since we only record one time stamp per chunk. The data points inside each chunk are considered as equally distributed.

4.1.3 Counter Files

The *counter files* are used to save the performance event counter values after the completion of one measuring experiment. Associating the performance event counter values and the electrical work of a measuring experiment is an essential part toward calculating an energy model (see chapter 4.3). The match of corresponding data point files (chapter 4.1.2) and counter files is done using the shot ID (chapter 4.1.1).

The counter files' on–disc representation is—except for some magic bytes—a Protocol Buffers–only format. For the technical definition see appendix B.

4.1.4 Work Files

The content of the work files is just the electrical work of a certain measuring experiment in Joule. The file format is the ASCII representation of the floating point value followed by optional garbage (separated by an ASCII space or newline). In contrast to the other file formats, the work file's file names have to match the following format: `work_<SHOT-ID>_.work` (e. g.

`work_SPEC-gcc@2011-08-31_16-07-46.work`).

The following listing shows a valid work file:

```
$ hexdump -C work_SPEC-gcc@2011-08-31_16-07-46.work
0000  37 34 2e 30 31 35 32 35  36 0a   |74.015256.|
000a
```

4.2 Software Tools

Since the building of a reasonable energy model is not an easy task, numerous tools have been used. Most software was developed specifically for the purpose of this study thesis. All of it is open–sourced and available on GitHub (`https://github.com/weissi/studienarbeit`).

This chapter will give an overview of the software used, both standard software and tools developed specifically.

4.2.1 Standard Software

- Google™ Protocol Buffers — for saving and loading of all kinds of data
- protocol-buffers — for parsing Google™ Protocol Buffers files and generating code in Haskell
- protobuf-c — for parsing Protocol Buffer files and generating code in C
- R — for statistical computations and data plots
- leaps — a R library for regression subset selection (to minimize the set of performance event counters)
- lm — a R library to fit linear models
- libpfm4 — for reading the performance event counters from user space
- NI–DAQmx Base — for interfacing NI USB–6218
- Glasgow Haskell Compiler
- Linux Kernel
- numerous GNU tools

4.2.2 Special Developments

LIBDATAPOINTS

LIBDATAPOINTS is responsible for loading and saving the measured data points from and to the data point files (see chapter 4.1.2). Its API is straight forward and the library is able to handle arbitrarily large files. The API can be found in `libdatapoints/datapoints.h`.

DATADUMP

DATADUMP is the tool used to dump the measuring data to data point files (see chapter 4.1.2). It currently records the three channels `CPU`, `BOARD` and `TRIGGER` with a sampling rate of 50 kS/s using NI–DAQmx Base (see chapter 3.2.1 and 4.2.1) to a file. Again, the usage is quite simple and straight forward:

```
datadump out.dpts my-shot-id
```

The example saves the measuring experiment `my-shot-id`'s data to the file `out.dpts`. The record runs until a `SIGINT` signal is caught. Optionally a third parameter representing the desired maximal running time in seconds is supported.

FASTCALCWORK

FASTCALCWORK can calculate the electrical work from data point files quickly. As explained in chapter 3.3 it calculates and integrates the instantaneous power to electrical work. But since discrete data is obtained by sampling (see chapter 3.2), calculating the electrical work is easy and fast:

$$P(t) = \frac{12V * U_R(t)}{R} \tag{4.1}$$

$$W = \int_{t_0}^{t_{max}} P(t)dt = \sum_{i=1}^{max} P(t_i) * (t_i - t_{i-1}) \tag{4.2}$$

The following example illustrates the usage of FASTCALCWORK:

```
fastcalcwork captured-17:15:00.dpts CPU 0.01 TRIGGER
```

Using the command line above, FASTCALCWORK will calculate the electrical work with a measuring resistor of 0.01 Ω. The measured data points will be taken from column `CPU`, the analog trigger's value (see chapter 3.2) from `TRIGGER`.

DATAEXPORT

DATAEXPORT exports data point files (see chapter 4.1.2) to R's `read.table` format [23]. This format is less efficient and contains only parts of the information but enables the user to exploit R's excellent analysis and plotting capabilities.

DUMPCOUNTERS

DUMPCOUNTERS is a tool for retrieving the performance event counter values on the target machine. Giving it a set of performance events and a command to execute, it will record the events while the task is running. It always records the whole system's performance events for all CPUs. Along the way it also controls the trigger wire. The trigger is crucial to match the time intervals for counting performance events and measuring electrical power.

A working example executing `/bin/ls` and counting `CPU_CLK_UNHALTED` and `INST_RETIRED` along the way follows:

```
dumpcounters -e CPU_CLK_UNHALTED,INST_RETIRED \
             -o ls_clock-cycles_inst-retired.ctrs \
             -r /bin/ls
```

After the execution, the file `ls_clock-cycles_inst-retired.ctrs` will contain the performance event counter values. The file format is described in chapter 4.1.3.

CTRBENCHMARK suite

CTRBENCHMARK suite is a front end and a very small library making it easy to write and use microbenchmarks. It is primarily meant to stress individual performance event counters, hence its name. Working samples can be found in the directory `ctrbenchmark/benchlets/`.

BuildSLE

BuildSLE is a Haskell program which is able to *build* the *s*ystem of *l*inear *e*quations. It takes several counter files (chapter 4.1.3) and the corresponding work files (4.1.4) as input and outputs a system of linear equations.

Given that R was always used to post–process the results of BuildSLE it is obvious that, again, R's `read.table` [23] format is outputted. Building a system of linear equations and solving it in R works as the following example shows:

```
$ buildsle *.work *.ctrs > /tmp/sle.rtab
BuildSLE, Copyright (C)2011, Johannes Weiss
[...]
Processed work files: 500
[...]

$ R
R version 2.12.1 (2010-12-16)
[...]
> sle <- read.table('/tmp/sle.rtab', header=TRUE)
> m <- lm(WORK~., data=sle)
```

High–level Scripts

So far, there are several simple tools for simple tasks. To compose the ensemble many of them have to be invoked correctly with each other. Since this is not a trivial task, some high–level scripts have been developed, most notably `scripts/measure-n-counters-m-benchmarks.sh` and `scripts/do_measuring.sh`. The former script can control records of many benchmarks with a fixed (chapter 4.3.3), rotational (chapter 4.3.2) or incremental (chapter 5.3) set of performance event counters; the latter automates the following steps as shown in the example below:

1. Check if no other measuring experiment is currently running.
2. Check that NI USB–6218 is connected.
3. Check that the login to the remote machine (which has the hostname `i30pc59` in the example) works password–less and the remote user is allowed to use sudo.

4. Locally and remotely build the needed software tools.

5. Locally start DATADUMP, remotely launch `/bin/ls` using DUMPCOUNTERS. The counted performance events are `CPU_CLK_UNHALTED` and `INST_RETIRED`.

6. Wait until `/bin/ls` has terminated and then stop DATADUMP.

7. Calculate the electrical work using FASTCALCWORK.

8. Save the data dump file, the counter file and the work file under the directory `/tmp/demo-shot`. The shot–id is `demo-shot@2001-09-09_-15-41-32`.

```
$ do_measuring.sh -f \
                  -s 2011-09-09_15-41-32 \
                  -p demo-shot \
                  -o /tmp/demo-shot \
                  i30pc59 \
                  'CPU_CLK_UNHALTED,INST_RETIRED' \
                  /bin/ls
No other 'datadump' running: OK
Checking if NI device 3923:7272 is plugged: OK
Testing password-free SSH: OK
Testing password-free sudo: OK
Building: OK
Remote building: OK
INFO: logfile is '/tmp/measuring_log_[...].log'
Waiting for sloooow NI call (e.g. 29s) [...].OK[...]
Writing remote script: OK
Running remote benchmark: OK (time=1)
INFO: remote log is '/tmp/remote-[...].log'
Telling datadump to stop (SIGINT): OK
Waiting for dump process (23472) to finish ....OK
Calculating work to 'work_[...].work' OK
Doing transformation

GREAT SUCCESS, EVERYTHING WENT FINE :-)
```

4.3 Toward the Energy Model

In this chapter a description of the steps toward the final energy model(s) is given.

4.3.1 The Set of Benchmarks

The first step toward the energy model is to choose a good set of benchmarks. The benchmarks should stress different parts of the CPU and should collectively stress all parts and units of the CPU. As chapter 1.1 states some units have been deliberately excluded in this work. The benchmarks should therefore avoid the excluded units because their results would have bad consequences on the final model.

The benchmark choice fell to the CINT2006 (Integer Component of SPEC CPU2006) suite, the memory bandwidth benchmark STREAM, a benchmark provoking many branch mispredictions and an ALU stress benchmark.

For the multi–core tests, different instances of these benchmarks have been pinned to each core and were therefore running simultaneously.

4.3.2 Finding a Useful Subset of Events

Because of the symmetric architecture of the CPU (see chapter 2.2) the final subset of events in the model has been gathered using only one enabled core of the CPU. The other cores have been disabled using the BIOS.

To model the baseline energy of a core (the energy consumed when no performance event is triggered), one pseudo–event per core (modeling the time it is enabled) has been added. These events fall in the group of the *global events* (see chapter 3.4.1). A core can only be enabled or disabled when the machine gets (re)booted. Therefore each of these pseudo–events is either 0 (core *disabled*) or is equal to the entire running time of a benchmark (core *enabled*).

The general methodology has already been described in chapter 3.4.3. First the performance events were randomly distributed in disjoint groups of eight events which is the maximal number of performance events the CPU can count simultaneously (see chapter 2.2). Thereafter, for each group of events all benchmarks have been executed consecutively. Then, all the runs of each benchmark were considered as one run recording all available events. Since the electrical

work consumed and the running time were recorded for each run of its own, they were averaged per benchmark. This was reasonable because the maximal relative standard deviation was always below 4%.

The result was considered as one huge system of linear equations and the best subset of events was chosen using R's leaps package. The events `CPU_CLK_UNHALTED` (= `CPU_CLK_UNHALTED.THREAD` $\propto$ `CPU_CLK_UNHALTED.REF_TSC`) and `INST_RETIRED` (= `INST_RETIRED.ANY`) are counted using fixed counters (see chapter 2.2.1) and were therefore forced into the model. Except the two fixed counters, R's leaps package has been configured to find the best model for eight counters.

Appendix A describes the events forming the energy model presented in this paper.

4.3.3 Final Energy Model

After the event selection process, suitable energy weights for the (pseudo–) events had to be found. To find these, the benchmark runs were repeated, once again on a single core but recording all of the final events in one shot. Thereafter all reasonable permutations for two and three cores and 500 random permutations for all four cores have been generated. All in all over 3000 different benchmark runs have been recorded. The way the benchmark runs were recorded is illustrated in figure 4.1. It was necessary to always reboot and reconfigure the machine because the number of active cores cannot be changed at run time. The taskset [17] utility has been used to pin the benchmark processes to their target core.

To train the model only about 50 well chosen benchmark runs were used, iteratively adding adequate sample runs when weaknesses of the model have been discovered. To evaluate the model (chapter 5) the whole set was used.

Presumably due to the symmetric architecture of the CPU, the events' energy weights proved to be similar on all cores. Hence, the model's general form could be simplified. Consequently the function w_c is not anymore parametrized by core. The following equation shows the new form, identifiers as in chapter 3.4.1.

$$W(t_0, t_e) = \sum_{i=1}^{n_g} c_g(i, t_0, t_e) w_g(i) + \sum_{k=1}^{n_e} w_e(k) \sum_{j=1}^{n_c} c_e(j, k, t_0, t_e) \tag{4.3}$$

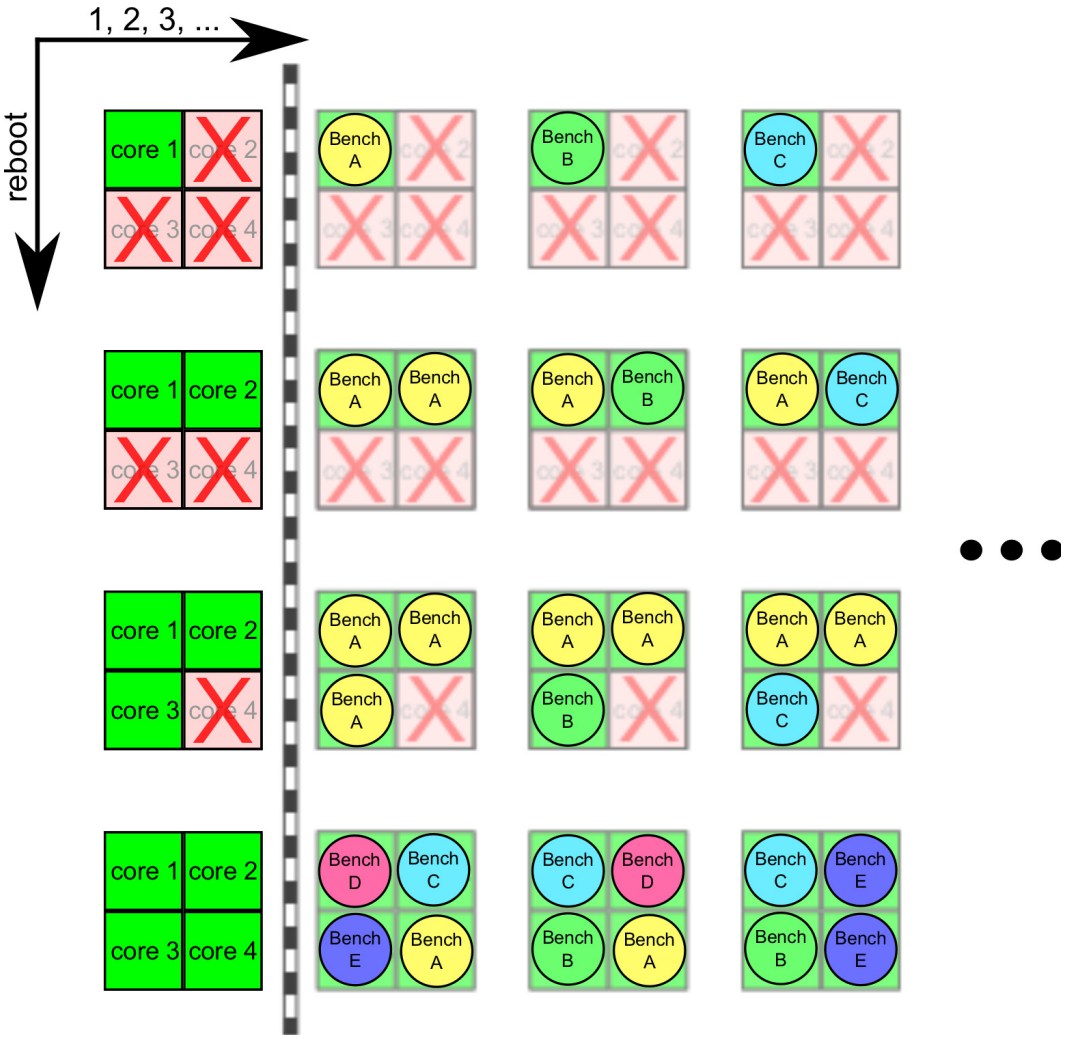

Figure 4.1: Data acquisition by running benchmark permutations

The following table shows the energy weights which form the functions w_c and w_g. c_g and c_e are the system's live data, available from the PMU (via libpfm4). When running live on a system, the parameters t_0 and t_e result from the sampling process: t_0 is the point in time of the last sample, t_e is "now". The functions' values are then obtained by the performance event counter's difference between the last and the current sample. Obviously, c_g and c_e are partial functions in reality: They are only defined if t_0 matches the time of the last sample and t_e the time of the current sample. The formula above and the weights below represent the energy model presented in this thesis. That is enough to estimate the electrical power the CPU consumes in selectable sampling intervals.

(pseudo–)Event	Energy Weight (w_c and w_g)	Maximal Frequency compared to `CPU_CLK_UNHALTED`
time core 1 is enabled [ns]	7.122 nJ	∞
time core 2 is enabled [ns]	1.486 nJ	∞
time core 3 is enabled [ns]	1.591 nJ	∞
time core 4 is enabled [ns]	2.350 nJ	∞
`INST_RETIRED`	−0.224 nJ	245.01%
`CPU_CLK_UNHALTED`	4.546 nJ	100.00%
`LD_BLOCKS:DATA_UNKNOWN`	−6.281 nJ	83.28%
`LD_BLOCKS:ALL_BLOCK`	5.504 nJ	83.28%
`UOPS_DISPATCHED:STALL_CYCLES`	−2.508 nJ	72.44%
`ILD_STALL:IQ_FULL`	−1.425 nJ	18.17%
`DSB2MITE_SWITCHES`	−5.972 nJ	5.31%
`DSB_FILL:ALL_CANCEL`	64.249 nJ	2.60%
`L2_RQSTS:PF_HIT`	−22.837 nJ	1.23%
`BR_INST_RETIRED:FAR_BRANCH`	−18 031.295 nJ	0.05%

Chapter 5

Evaluation

This chapter evaluates usefulness, accuracy and overhead the energy model imposes. The model's strengths and weaknesses are pointed out and it is compared to a simple computing time based model.

5.1 Error of Estimation

Most of the 3000 benchmark runs mentioned in chapter 4.3.1 and 4.3.3 are estimated with an error of 5% or better, as *model* in figure 5.1 shows. The bars named *cpu_time_based* are a comparison to a simple computing time based model, discussed in a chapter of its own (5.2). The overall worst sample was estimated with an error of about 40% (see *model* in figure 5.3). This sample is from a special benchmark run where all four cores were enabled but mostly idle (real average power usage 8.9 W, 12.5 W estimated). This erratic behavior is explainable by the design of the model: The goal has been to develop a model which will not produce completely faulty results but focuses on higher power consumptions. The power consumptions observed were between 6 W and 60 W and the focus has been set on the range above 15 W. This decision was motivated by the fact that an operating system will typically spend little time in ACPI Processor State C0 [5] when little energy is consumed. But as chapter 1.1 states, the energy model worked out in this thesis is valid iff the CPU is in state C0 all along. Nevertheless, a similar energy model may be evolved with particular support for low–power CPU usage and the other ACPI Processor States in mind.

Regarding most of the benchmarks—which consume more than 15 W on

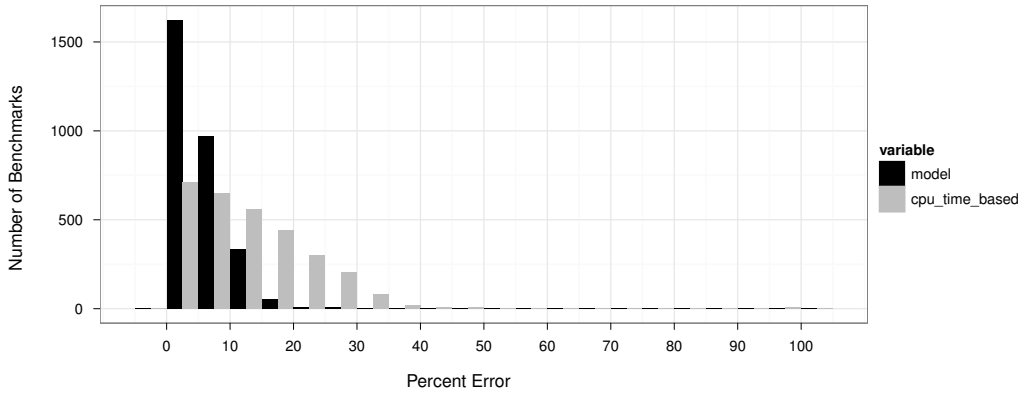

Figure 5.1: Histogram of percent error, sophisticated versus simple model

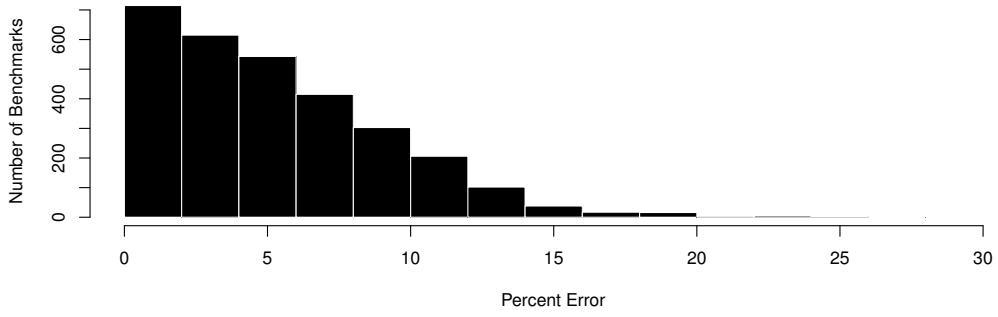

Figure 5.2: Histogram of percent error for 15 W^+–benchmarks

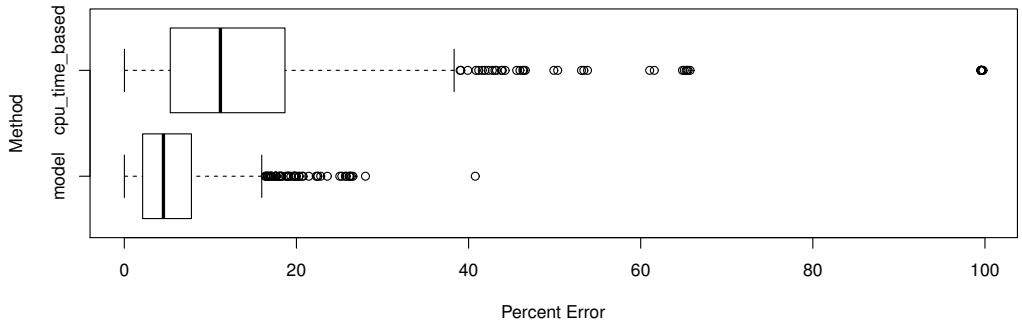

Figure 5.3: Percent error, sophisticated versus simple model

average—the model's estimation is clearly better. Figure 5.2 illustrates a percent error classification of the benchmarks consuming more than 15 W on average. The 15 W^+–benchmarks show an average estimation error of only 5.3%.

5.2 Comparison to a Simple Time Based Model

As it is still today's standard to use computing time based models, this chapter will compare the estimations to the more sophisticated energy model presented here (chapter 4.3.3).

The comparison of the overall average energy estimation error shows the benefit of a reasonable energy model: 5.4% versus 13.1% for the simple computing time based model. Figure 5.3 illustrates the results, where the final energy model is listed as *model* in the figures. The computing time based model was built by using only the event `CPU_CLK_UNHALTED` and is referred to in the figures as *cpu_time_model*. It has been fitted using a linear regression on the same training data as the regular model, resulting in the formula

$$W = 5.885J * 10^{-9} * \texttt{CPU\_CLK\_UNHALTED}. \tag{5.1}$$

5.3 Overhead of this Implementation

To evaluate the overhead of this implementation three metrics have been applied: Running time, average power usage and unhalted CPU clock cycles. To compare the results the *473.astar* benchmark of the CINT2006 (Integer Component of SPEC CPU2006) benchmark suite has been used. Besides a warm–up run, every configuration has been executed ten times. The performance events have been added consecutively in the order they are listed in appendix A. Before measuring the costs of adding a counter, the system has been benchmarked without even setting up libpfm4 (see chapter 4.2.1). In the plots these configurations are listed as *0* performance events measured. Because the unhalted CPU clock cycles are measured using the PMU (chapter 2.2.1) and libpfm4, no benchmark could be recorded without even setting up libpfm4.

As one can see in figures 5.4 and 5.5, the number of counted performance events or not using the PMU (see chapter 2.2.1) does not seem to correlate to the CPU clock cycles or the time a process needs to complete. The Pearson

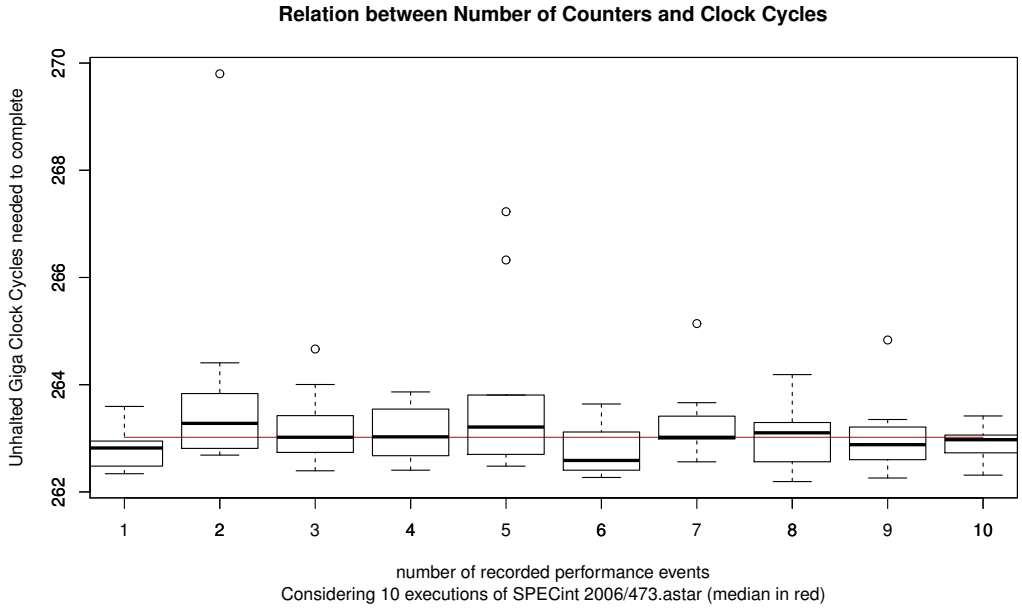

Figure 5.4: Counter Costs Cycles

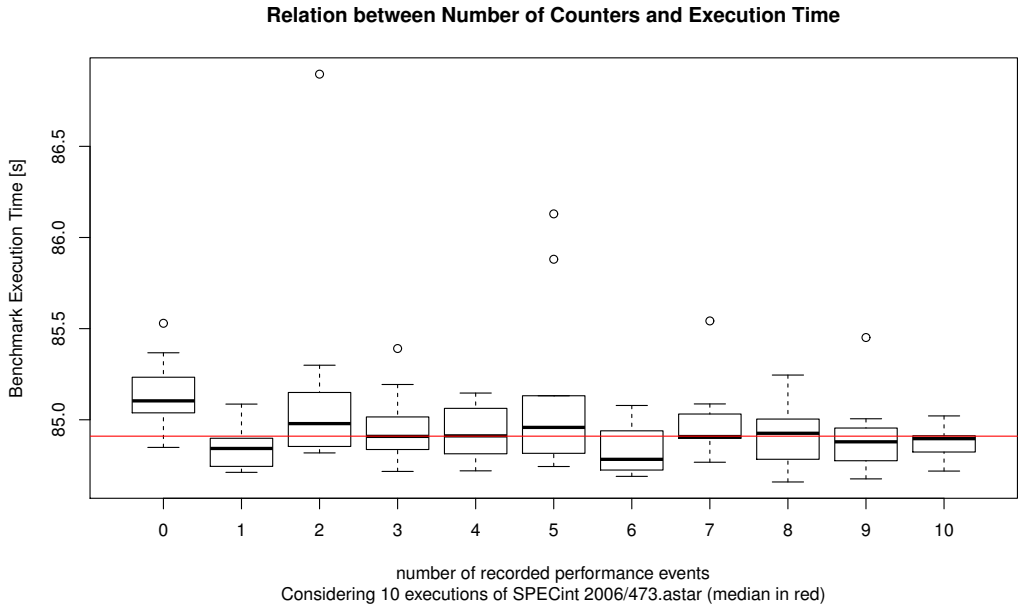

Figure 5.5: Counter Costs Time

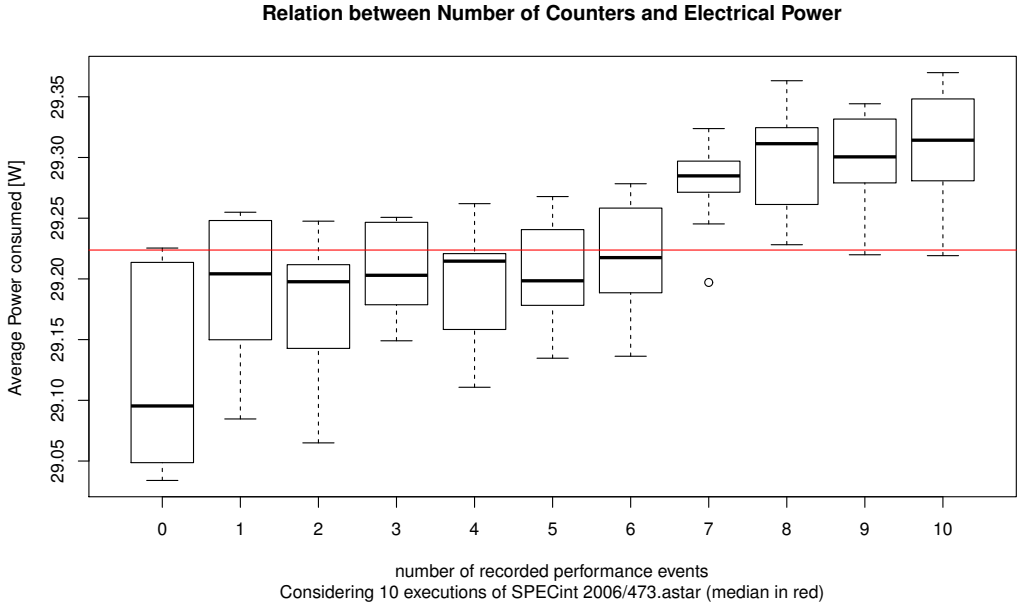

Figure 5.6: Counter Costs Power

product–moment correlation coefficients (PCCs) [28] are -0.18 with respect to the running time and -0.12 to the clock cycles. Considering the average electrical power there's some correlation (PCC: 0.72, figure 5.6), but the absolute value ($< 0.3\,\mathrm{W}$) is negligible.

So, this implementation does not harm the system's performance and its contribution to the overall energy consumption is negligible.

Chapter 6

Conclusion

As of our knowledge the energy model presented in this thesis is the first performance event counter based for the Intel® Sandy Bridge microarchitecture. More than that, it is the first for CPUs with more than two cores. The evaluation results prove the general concept is also applicable to today's and tomorrow's multi–core CPUs.

The software developed along with this thesis provides a convenient and freely available way to build energy models in the future.

6.1 Problems and Outlook

Even though the resulting energy model already proved its efficiency and necessity, there is room for further improvements. On the one hand for the model itself, on the other hand for the process of building energy models for new target architectures. First, the restrictions mentioned in chapter 1.1 should be eliminated. To improve the practical usefulness real multi–threading programs should be better kept in mind. The challenge with threads is that shared memory regions get accessed on the same time. This will probably attract interest on other or additional performance events which are not well covered here. The latter also shows the need to develop a convenient event selection process. Supplementary, the upcoming many–core architectures with $\gg 4$ cores might become challenging.

Bibliography

[1] F. Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 37–42. ACM, 2000.

[2] R. Bertran, Y. Becerra, D. Carrera, V. Beltran, M. Gonzalez, X. Martorell, J. Torres, and E. Ayguade. Accurate energy accounting for shared virtualized environments using pmc-based power modeling techniques. In *Proc. 11th IEEE/ACM Int Grid Computing (GRID) Conf*, pages 1–8, 2010. doi:10.1109/GRID.2010.5697889.

[3] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 147–158. ACM, 2010.

[4] Agner Fog. The microarchitecture of Intel and AMD CPUs. Available from: http://www.agner.org/optimize/microarchitecture.pdf.

[5] Hewlett Packard/Intel/Microsoft/Phoenix/Toshiba. *Advanced Configuration and Power Interface Specification*, April 2010. Available from: http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf.

[6] Google Inc. Protocol Buffers – Techniques. Available from: http://code.google.com/apis/protocolbuffers/docs/techniques.html.

[7] Intel Corporation. Reference for Processor Events. Available from: http://software.intel.com/sites/products/documentation/hpc/amplifierxe/en-us/lin/ug_docs/reference/index.htm#snb/events/about_events.html.

[8] Intel Corporation. Specifications of the Intel® Core™i7-2600 Processor. Available from: http://ark.intel.com/products/52213.

[9] Intel Corporation. *Hyper-Threading Technology*, February 2002. Available from: `http://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf`.

[10] Intel Corporation. Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor, March 2004. Available from: `ftp://download.intel.com/design/network/papers/30117401.pdf`.

[11] Intel Corporation. Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors, November 2008. Available from: `http://download.intel.com/design/processor/applnots/320354.pdf?iid=tech_tb+paper`.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 (Basic Architecture)* , May 2011. Available from: `http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-1-manual.pdf`.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B (System Programming Guide, Part 2)*, May 2011. Available from: `http://www.intel.com/Assets/en_US/PDF/manual/253669.pdf?wapkw=(915)`.

[14] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93. IEEE Computer Society, 2003.

[15] Simon Kellner. Event-driven temperature control in operating systems, April 30 2003. Available from: `http://www4.informatik.uni-erlangen.de/SA/pdf/SA-I4-2003-02-Kellner.pdf`.

[16] Chris Lomont. Introduction to Intel® Advanced Vector Extensions. Technical report, Intel Corporation, May 2011. Available from: `http://software.intel.com/file/37205`.

[17] Robert M. Love. *Linux User's Manual – taskset*, April 2003. Available from: `http://linuxcommand.org/man_pages/taskset1.html`.

[18] Andreas Merkel, Frank Bellosa, and Andreas Weissel. Event-Driven Thermal Management in SMP Systems. In *Second Workshop on Temperature-Aware Computer Systems (TACS'05)*, Madison, USA, June 2005.

[19] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In *Fifth ACM SIGOPS EuroSys Conference*, Paris, France, April 13– 16 2010.

[20] National Instruments Corporation. *NI USB-621x Specifications*, apr2009 edition, April 2009. Available from: `www.ni.com/pdf/manuals/371932f.pdf`.

[21] National Instruments Corporation. *NI USB-621x User Manual*, april 2009 edition, April 2009. Available from: `http://www.ni.com/pdf/manuals/371931f.pdf`.

[22] D. Snowdon. *Operating System Directed Power Management*. PhD thesis, The University of New South Wales, 2010.

[23] R Development Core Team. *R Data Import/Export*, version 2.13.1 (2011-07-08) edition, July 2011. Available from: `http://cran.r-project.org/doc/manuals/R-data.pdf`.

[24] Vince Weaver. perf_event programming guide. [Online; accessed 12-September-2011]. Available from: `http://web.eecs.utk.edu/~vweaver1/projects/perf-events/programming.html`.

[25] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–246. ACM, 2002.

[26] Wikipedia. Four-terminal sensing — wikipedia, the free encyclopedia, 2011. [Online; accessed 13-September-2011]. Available from: `http://en.wikipedia.org/w/index.php?title=Four-terminal_sensing&oldid=447874833`.

[27] Wikipedia. MMX (instruction set) — Wikipedia, The Free Encyclopedia, 2011. [Online; accessed 9-September-2011]. Available from: `http://en.wikipedia.org/w/index.php?title=MMX_(instruction_set)&oldid=443484618`.

[28] Wikipedia. Pearson product-moment correlation coefficient — wikipedia, the free encyclopedia, 2011. [Online; accessed 26-September-2011]. Available from: `http://en.wikipedia.org/w/index.php?title=Pearson_product-moment_correlation_coefficient&oldid=452476020`.

[29] Wikipedia. Streaming SIMD Extensions — Wikipedia, The Free Encyclopedia, 2011. [Online; accessed 9-September-2011]. Available from: http://en.wikipedia.org/w/index.php?title=Streaming_SIMD_Extensions&oldid=444134351.

Appendices

A Performance Event Selection and Description

This appendix lists all the performance events that form the energy model presented in this work. The descriptive texts are all taken from [7].

▷ `CPU_CLK_UNHALTED`

This is an architectural event that counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling. For this reason, this event may have a changing ratio with regards to wall clock time.

▷ `INST_RETIRED`

This event counts the number of instructions retired from execution. For instructions that consist of multiple micro–ops, this event counts the retirement of the last micro–op of the instruction. Counting continues during hardware interrupts, traps, and inside interrupt handlers. Notes: `INST_RETIRED.ANY` is counted by a designated fixed counter, leaving the four (eight when Hyper–threading is disabled) programmable counters available for other events. `INST_RETIRED.ANY_P` is counted by a programmable counter and it is an architectural performance event. Counting: Faulting executions of GETSEC / VM entry / VM Exit / MWait will not count as retired instructions.

▷ `BR_INST_RETIRED:FAR_BRANCH`

This is a non–precise version (that is, does not use PEBS) of the event that counts far branch instructions retired.

▷ `DSB2MITE_SWITCHES`

This event counts the number of the Decode Stream Buffer (DSB)-to-MITE switches including all misses because of missing Decode Stream Buffer (DSB) cache and μ–arch forced misses. Note: Invoking MITE requires two or three cycles delay.

▷ `DSB_FILL:ALL_CANCEL`

This event counts the number of times when a valid Decode Stream Buffer (DSB) fill has been actually cancelled not because of exceeding the way limit. Cancelling Decode Stream Buffer (DSB) fill may also result, for example, from Decode Stream Buffer Queue (DSBQ) snoop hits. This is because the Decode Stream Buffer (DSB) full hit is guaranteed to delivery from Decode Stream Buffer (DSB). In the B step a four–bit counter will count the number of cancel operations and will reverse the priority upon look ing up the same set.

▷ `ILD_STALL:IQ_FULL`

This event counts stall cycles when instructions cannot be written because IQ is full. Note: If there is no Resource Allocation Table (RAT) stalls, it indicates the decoders issue.

▷ `L2_RQSTS:PF_HIT`

This event counts the number of requests from the L2 hardware prefetchers that hit L2 cache. LLC prefetch new types

▷ `LD_BLOCKS:ALL_BLOCK`

Number of cases where any load ends up with a valid block–code written to the load buffer (including blocks due to Memory Order Buffer (MOB), Data Cache Unit (DCU), TLB, but load has no DCU miss)

▷ `LD_BLOCKS:DATA_UNKNOWN`

This event counts the number of load operations delayed due to store buffer blocks, preceding store operations with known addresses but unknown data. Counting happens according to the final blocking codes. This does not include inline wakeups.

▷ `UOPS_DISPATCHED:STALL_CYCLES`

This event counts cycles during which no uops were dispatched from the Reservation Station (RS) per thread.

B File Formats

Data Point Files

For the definition of the Protocol Buffers used, see below. For the definition of the Protocol Buffer's language see their web site.

```
byte 0x03;                          # \
byte 0x01;                          #  THESE BYTES FORM THE FILE
byte 0x86;                          #  FORMAT'S MAGIC BYTES
byte 0x01;                          # /

byte version;                       # THE FORMAT'S VERSION: 0x1

uint32_t header_length;             # LENGTH OF FOLLOWING PROTOCOL BUFFER IN
                                    # BYTES (ENCODING: LITTLE ENDIAN)

<Protocol Buffer type 'MessageData'>  # THE "HEADER" PROTO BUF

REPEATED FOR EVERY CHUNK:
    byte 0x03;                      # \
    byte 0x01;                      #  THESE BYTES FORM THE FILE
    byte 0x86;                      #  CHUNK'S MAGIC BYTES
    byte 0x02;                      # /

    uint32_t message_length;        # LENGTH OF FOLLOWING PROTOCOL BUFFER IN
                                    # BYTES (ENCODING: LITTLE ENDIAN)

    <Protocol Buffer type 'DataSet'>  # THE DATA OF ONE CHUNK
```

Generic Protocol Buffers

This file contains generic Protocol Buffers definitions. Its file path is `protos/generic.proto`.

```
message Timestamp {
    required int64 sec = 1;
    required int64 nsec = 2;
}
```

Data Points Protocol Buffers

This file contains the Protocol Buffers definition as used by LIBDATAPOINTS (see chapter 4.1.2 and 4.2.2). Its file path is `protos/measured-data.proto`.

```
import "generic.proto";

message MeasuredData {
```

```
    required string shot_id = 1;
    required double sampling_rate = 2;
    required uint32 channel_count = 3;
    required bool has_external_data = 4;
    repeated DataSet inline_data = 5;
    repeated string channel_names = 6; //ordered!
}

message DataSet {
    required Timestamp time = 1;
    repeated DataPoints channel_data = 2;
}

message DataPoints {
    repeated double data_points = 1 [packed=true];
    optional uint32 channel_no = 2;
    optional string channel_name = 3;
}
```

Performance Event Counter Protocol Buffers

This file contains the Protocol Buffers definition as used by DATADUMP, BUILDSLE and DATAEXPORT (see chapter 4.1.3 and 4.2.2). Its file path is `protos/perf-counters.proto`.

```
import "generic.proto";

message CounterData {
    required string shot_id = 1;
    required Timestamp start_time = 2;
    required Timestamp stop_time = 3;
    required uint32 cpu_count = 4;
    repeated CounterValue counters = 5;
    optional string benchmark_cmd = 6;
}

message CounterValue {
    required string counter_name = 1;
    repeated uint64 counter_value_per_cpu = 2; //ordered by cpu id
    optional uint64 global_counter_value = 3;
}
```