Diplomarbeit

Energy/Utility for L4Re

Measurements and Theoretical Observations

Marcus Hähnel

28. März 2012

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuende Mitarbeiter:	Dr.-Ing. Marcus Völp
	Dipl. Inf. Björn Döbel

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 28. März 2012

Marcus Hähnel

Thanks and Acknowledgments

First I want to thank Professor Härtig for giving me the opportunity to write this thesis, and for the trust placed in me. His vision of Energy/Utility is the foundation of this thesis. I also want to thank Björn Döbel and Marcus Völp, who were always available for advice on short notice and provided valuable input and helpful ideas and critique. A lot of help and information were provided by Daniel Hackenberg and Robert Schöne of the Center for High Performance Computing (ZIH), who invested time and provided me with an instrumentation setup, and a way to take high resolution external measurements. Benjamin Engel and Michal Sojka advised me on ideas to generate meaningful cache benchmarks. Further I want to extend my thanks to Alexander Warg and Adam Lackorzynski for sharing their deep knowledge of L4Re and the Fiasco.OC microkernel. Michael Roitzsch provided a patch for ffmpeg that instrumented the decoding path, which was very useful and saved me a lot of time. I also want to thank Waltenegus Dargie for his explanation of his measurement setup and the opportunity to take a closer look at his instrumented computers.

I also want to thank my parents, who funded my many years of study and provided moral support all the time.

Many thanks also to the people in the OS Students Lab, you know who you are, for the pleasant company and the many interesting discussions and helpful input.

Last, but definitely not least, thanks go to Katharina for the patience, and doing many chores alone when I spent weekends in the lab fixing broken measurements and benchmarks.

Contents

Introduction **1**

A Measuring Energy **3**

1 Related Work on Energy Measurements **5**

2 A Measurement Framework for L4Re **7**
2.1 Implementing a Performance Monitoring Framework 7
2.1.1 Introduction to x86 Performance Counters 7
2.1.2 HAECER — A Generic Performance Measurement Framework . 11
2.1.2.1 Core Library Functionality 12
2.1.2.2 Giving Applications Access to MSRs 12
2.1.2.3 The API 14
2.1.2.4 The Event Counter Database 15
2.1.2.5 Evaluating the Performance Counter Implementation . 15
2.1.2.6 Avoiding System Management Mode 17
2.1.2.7 RDMSR vs. RDPMC Overhead 17
2.2 Ferret for L4Re and HAECER 18
2.2.1 Modifications Needed 19
2.2.2 Integration into HAECER 20
2.3 Introducing Energy Counters 21
2.3.1 Extending the Library for Energy Counter Support 22
2.3.2 Evaluating the Energy Counters 22
2.3.3 Trying to Improve Counter Resolution 24
2.3.3.1 Implementation 26
2.3.3.2 Analysis of Results 27

3 Evaluating the Energy Counters **29**
3.1 Examples of Different Loads 29
3.2 Alternative Ways to Measure Energy 31
3.2.1 Challenges 31
3.2.2 Comparing the Energy Counters to External Measurements . . . 32

4 Summary **37**

B A Novel Approach to Energy Modeling **39**

1 Related Work on Energy Modeling **41**

2 A New Energy Modeling Framework - Energy/Utility Functions **43**
2.1 An Introduction to Energy/Utility Functions 43
2.2 A Hierarchy of Resources and Services 44
2.3 Formalization 46
2.4 Composing Workloads 49

3 Characterizing Resources **51**
3.1 Devices 51
3.1.1 CPU/Caches 51
3.1.1.1 Design of the Microbenchmark 52
3.1.1.2 Measurement Results 53
3.1.2 Example: Display 54
3.1.2.1 The Measurement 55
3.1.2.2 Measurement Results 55
3.2 Application Utility - The Video Player 56

4 Preliminary Usage Scenarios - Future Work **59**
4.1 Scheduling 59
4.1.1 Energy Optimized 59
4.1.2 Quality Assuring 60
4.1.3 Combination 60
4.2 Informed User Decisions 60
4.3 Energy Accounting 60
4.4 Global Energy Modeling 61

Conclusion **63**

Glossary **65**

List of Listings **67**

List of Tables **67**

List of Figures **69**

Bibliography **71**

Introduction

In the past years, there has been a steadily increasing attention toward the energy consumption of computer based systems in research as well as in system's development. The motivation for this interest is that energy consumption is increasingly becoming a key metric for the quality of the design of a computer system. A number of aspects are strongly related to the energy consumption.

- **Heat** is proportional to energy consumption and influences the needed cooling capacities or even the choice of materials for a given system design. Cooling might again inflict a secondary energy requirement.
- **Mobility** is largely determined by the weight of a device and its runtime. Both of these aspects are influenced by energy consumption with a larger battery providing for longer life-time but larger overall weight.
- **Power costs** are constantly rising and the energy pricing is often of major importance in large scale systems. Energy costs usually exceed the actual cost for the hardware.
- **Supply** lines for delivering energy to the processor depend on the amount of energy that needs to be transferred.
- **Selective powering** of only a subset of the units in a device will enable future devices that have more units than can be powered at once. A power envelope determines the amount of units that can be powered simultaneously [EBSA+11].

Depending on the concrete system type the relevant aspects may differ, but for virtually all computer systems today, energy plays an important role. Portable devices depend on mobility while nonmobile embedded devices have stringent heat dissipation requirements. Those are rooted in the often limited cooling possibilities of the machines in which they are used. For servers, data centers, and home use, the cost of energy and cooling are key factors for any new system [dcc].

The modeling of energy usage and its optimization are state-of-the-art problems on which I will attempt to provide a new perspective. Despite the existence of a huge body of related work on energy modeling [MAC+11] [Sno10] [Bel00] [BGN+10], a uniform metric that provides a correlation between energy consumption and the benefit the user gains by investing this energy, *and* that is applicable for multiple layers of a whole system is missing. We seek for a metric that is applicable for all layers of a system and not just individual parts of it. The often used metric of performance per watt [RAK+11] [SCS+08] [KF09] is not sufficient for modeling energy either. A higher number of clock cycles, which most authors associate with performance, may not provide a

better performance for the user. In today's systems, performance per watt cannot even be expressed as a scalar value. The power draw changes depending on the workload type. Further, this metric is not applicable to adaptive applications that can operate at different performance levels with varying quality of service levels.

My goal in this thesis is to introduce a new method to characterize energy usage, which does not exhibit these shortcomings. I will propose a model, which characterizes energy demand in terms of **Energy/Utility Functions**, and provides the same basic mechanism for all layers of a system. My approach enables seamless modeling of any system, from small embedded chips to complete data centers. Utility was chosen as a universal measure for the benefit to the user, because it is generic enough to model the highly application specific notion of usefulness. This measure is also in resemblance to Time/Utility functions, which is well-known from real-time systems research [Jen92] [JLT85] [RJL05a].

The concept uses the general idea that each component in a system provides a benefit to the user or to other components. Once the resource is used by a higher layer in the system, it provides a *utility* to this higher layer in a form that depends on the type of the component. As any part of a system requires energy to operate, the energy consumed by the resource to provide this utility can be described as its *energy demand*. I will express the relation between utility and energy demand as **Energy/Utility Functions** and use these functions to characterize systems. In the future, we will extend these functions to drive energy-related decisions in the system.

Our approach is highly beneficial for adaptive workloads. They provide a way for the user, or operator, to weigh the usefulness that they get out of the system against the energy they are willing to invest for the benefit.

To be able to characterize the system, I implemented a measurement framework, which exploits the newly available energy counters on current Intel Sandy Bridge cores [RNA+12], as well as the performance counters found in many of today's processors. My example application for characterizing Energy/Utility will be a video player running on top of L4Re [l4r], a small multiserver OS layer running on top of the Fiasco.OC [fia] microkernel.

I split this thesis in two parts. The first part deals with the implemention of an energy and performance measurement framework for L4Re. I will evaluate the precision of the energy measurements taken with this framework and performe some example measurements. In the second part, I will present my approach to energy modeling. After first highlighting current and past works on optimizing energy usage, I will then present the concept of Energy/Utility Functions as a way to model system energy requirements, which builds on these existing works. It extends them to provide a more user centric and universal approach to enabele the decomposition of a system into multiple components. This approach will also consider gating effects that are often not discussed in traditional modeling methods. I will use the results of the first part of the thesis to provide the application and component characteristics needed to model energy consumption and show that such results are obtainable. I conclude this thesis with a short outlook on preliminary ideas for energy aware scheduling based on Energy/Utility Functions.

Part A

Measuring Energy

1 Related Work on Energy Measurements

Until recently, the only option for measuring energy of system components was by externally instrumenting special measurement boards with measurements points or by tapping the power supply. The boards that were used for these measurements range from embedded systems like the Intel XScale PXA255 based on an ARMv5 RISC core [Con05] [SPH07] over ARM7 cores [LEMC01] to modern boards with Intel Pentium 4 processors [IM03] and current Intel Sandy Bridge processors [Wei]. These measurements usually have a low time resolution and rely on the repeated execution of microbenchmarks to profile the power draw of events. While most of these works provide interesting results, which would also benefit my work, few provide details on the implementation.

Since the advent of in-processor energy-accounting, live measurements can be performed directly on the processor. These features were introduced in Intel Sandy Bridge cores in 2011 [RNA+12] and AMD Llano cores in 2010 [amd10].

On IA32 architectures, counting performance events has been possible since the Pentium processor and a number of frameworks exist that can measure a number of performance events for applications. The most prominent of them is the PERFTOOLS toolkit [per] for Linux. While this toolkit is incredibly powerful and supports a large range of performance events, no instrumentation of individual parts of an application is possible as no library or external API exists. Only the whole application may be measured and it is difficult to trace back a large amount of cache misses to a specific function. Also the perf tools are highly specific to Linux, with support for many software events that are generated by the Linux kernel. While a kernel module to access the energy counters in recent Intel hardware exists [rap11], it is, to my knowledge, not used by any tool like the perf utilities.

To monitor performance counters in the CPU, Intel provides a tool termed *Intel performance counter monitor* [int10], which has a user API and can be used in applications. But it has no support to monitor the energy counters of the processor. Further, there is no simple way to determine which counters to use. Per default the tool only counts cache events, and there is no simple way to only monitor one specific event type.

For monitoring the energy counters Intel only provides the *Intel power gadget* [int11a] that is only available for Microsoft Windows and Mac OS X. No source code for it is provided. It also performs only rudimentary logging for the whole system and not for individual applications or parts of an application.

For runtime monitoring of L4 applications a framework was developed by Martin Pohlack at the Operating Systems Group of TU Dresden. The framework named FERRET [PDLD06] [Poh10] provides a low overhead, real-time capable mechanism to insert instrumentation points into applications and is provided in the form of a library. While it provides a limited amount of support for hardware performance counters, it only enables this by a kernel hack that needs to have support built in for each instrumented

architecture. It further only enables access to performance counters using the RDPMC instruction of the processor, which cannot read all performance counters.

I improve on this work by providing a library that makes instrumentation easier by moving the actual hardware dependent part out of kernel space into a user space library. This user space library also aims to abstract from the actual hardware implementation to make the instrumentation of programs easier. This is especially important when a low overhead of the instrumentation methods is needed. It also enables the use of counters not accessible by RDPMC.

2 A Measurement Framework for L4Re

In this chapter, I will present my framework for instrumenting applications with energy and performance measurement points. The framework is needed to measure energy and performance characteristics of applications, and provide this data to other programs. After a first introduction to performance counters, I describe an extensible library that can be used to instrument programs without the need for specific knowledge of the monitoring facilities on different processor cores. I will then proceed to extend this library with monitoring capabilities to communicate monitoring results to other processes, and methods for energy accounting. The special features of the power metering interfaces will be discussed and their usefulness and precision is evaluated.

2.1 Implementing a Performance Monitoring Framework

The first step to my approach to energy modeling was the development of a versatile, and easy to use measurement framework that enables the user to get live measurement data during program execution. These facilities were developed for applications running on top of the Fiasco.OC [fia] microkernel, developed at the Operating Systems group at the Computer Sciences Department of TU Dresden. The user land that is available to applications for Fiasco is called L4Re [l4r] and aims to provide a reasonable POSIX subset.

For the development of an application's energy profile, measuring energy is not the only point which has to be taken care of. To also be able to map energy to the runtime behavior of the program, this behavior must be observable as well. As mentioned in the section on related work, FERRET already provides a method to monitor runtime behavior, but is limited in its capabilities to monitor hardware events. To keep instrumentation simple, and not burden the programmer with complex performance counter mechanisms, another framework is needed that enables the collection of hardware generated events, such as cache or memory statistics, and makes such events accessible to the one instrumenting the application. The framework should provide a reasonable abstraction.

2.1.1 Introduction to x86 Performance Counters

Performance counters existed in x86 CPUs as far back as the Pentium I [Mat99], but they were not officially documented back then. Nowadays, virtually every CPU, from the small embedded ARM core, to the latest generation of desktop and server processors by Intel and AMD, contain these mechanisms for instrumenting the CPU. These performance counters are often the only possibility to observe what happens deep inside the processor. They can monitor everything from cache characteristics, over the prefetchers,

to individual usage of certain units inside the CPU. Performance counters are extremely low overhead and, when the only other possibility to monitor behavior is to simulate the CPU, the fastest possibility to obtain performance characteristics of an application.

The other possibility to capture, for example, cache characteristics, is by using a cache simulator like Cachegrind [Net04] that traps every memory access and simulates the cache hierarchy of the processor. However, because the detailed eviction strategy and the prefetching strategy of modern CPUs are often regarded as strict company secrets, as they determine to a large degree how well the CPU performs, simulation can always only approximate these algorithms instead of directly implementing them. Performance counters provide a way for the CPU manufacturer to let developers monitor application behavior without exposing the details of those more delicate areas of CPU design.

The problem when designing a system that is supposed to provide live measurements in a program is that hardware counters differ significantly between architectures. Not only between the different instruction set architectures (ISA), such as ARM, MIPS, or x86, but also between the different CPU vendors of an ISA — like AMD and Intel for the x86/AMD64 architectures — and even between the different generations of cores from a single vendor. Intel and AMD implement mechanisms for performance monitoring that use programmable counters. A limited number of these counters can be programmed to an often large number of specific performance events. These events differ between cores, some of them significantly, due to their different hardware architectures. But not only the events that can be measured differ, but also the facilities available to monitor them. While the Intel Pentium 3 had only two programmable performance counters available, the Intel Core CPUs of the Sandy Bridge hardware architecture posses up to eight of them. The way to set up these counters has also changed over time and there is no universal way to program them. Table 1 provides an overview of the number of programmable counters present in five CPU generations of x86 CPUs from Intel and AMD. As can be seen from the Pentium 4 example, it is not even guaranteed that the number of counters available increases with every CPU generation.

	Intel Pentium 4	Intel Sandy Bridge	Intel Pentium 3	AMD Llano	AMD Bull-dozer
Programmable Counters	18	8+2 (4+2 in HT)	2	4	6+4
Fixed-function Counters	n/a	3+1	n/a	n/a	n/a
Programmable Events	45	62+6	73	114	80+29
Architectural Events	n/a	6 (of 7)	n/a	n/a	n/a
Bit-width	40 Bit	48 Bit	40 Bit	48 Bit	48 Bit

Table 1: Available counters in various CPUs. Sources: [Int11c], [AMD12], [AMD11]

The numbers after the plus sign for the Sandy Bridge architecture designate values for the uncore performance monitoring facilities, which are available per C-Box. A C-Box is Intel's term for a unit in the uncore also called the coherence engine. It is responsible

for the interface between the processor core and the last level cache [xeo10]. Events were counted by type and there are more available when considering individual umasks that filter out specific sub-events. Intel itself speaks of 100 programmable event types [?]. For AMD Bulldozer the second number shows the values for northbridge performance monitoring.

To add to these possibilities Intel did also introduce so called "fixed-function counters". These count event types that are present on close to all CPUs that implement these counters, because they are intrinsic to the x86 CPU design and are often needed in performance analyses. To not take up one of the few precious programmable counters Intel invested 3 fixed-function counters that always count the same type of event and can otherwise be used in a similar way as the regular programmable counters. The fixed-function counters, which are available since version 2 of the Intel performance monitoring facilities, have to be enumerated before they can be used to ensure that they are available. The general facility to enumerate counters or to get details of the CPUs performance counter implementation is by accessing the CPUID leaf 0Ah. This leaf also indicates the version of the performance monitoring facilities and gives information on the bit width of the counters and the number of counters available. Table 2 lists the fixed-function counters that are defined for version 2 and 3 of the performance monitoring facilities.

Event Name	Description
Any retired instruction INST_RETIRED_ANY	Counts all instructions that are retired and not discarded because of failed speculation
Unhalted Core Clocks CPU_CLK_UNHALTED_CORE	Counts the number of clock cycles during which the CPU was not halted or in one of its sleep modes
Unhalted Reference Clocks CPU_CLK_UNHALTED_REF	Counts the reference clock, which is not subject to frequency scaling

Table 2: Fixed-function counters in current Intel architectures. Source: [Int11c]

A number of events are reoccurring on most processors and will not change between core generations because they are intrinsic to Intel's architecture concept. Intel decided to declare seven such counters as *architectural counters*. These share the same event IDs across all Intel cores that support architectural performance monitoring, and are guaranteed to be available under this ID — if they are available at all. The availability of these counters needs to be queried using the CPUID instruction just as described in the previous paragraph. The actual fixed-function performance counters, mentioned there as well, are also considered architectural, as their meaning is not changed over CPU generations and because they can, just like the architectural counters, be enumerated by the software. The available architectural counters can be seen in Table 3.

Retirement means that the instruction executed and the result was not discarded because of false speculation. For branches this signifies that the branch was taken, and not later discarded due to a wrong branching prediction by the processors branch

Event Name Event Symbol	Description
UnHalted Core Cycles UNHALTED_CORE_CYCLES	Cycles that the CPU clock was running (not halted)
Instructions retired INSTRUCTIONS_RETIRED	Counts the number of instructions at retirement
Unhalted Reference Cycles UNHALTED_REF_CYCLES	Counts the number of reference cycles when the core was executing. The reference clock is independent of the actual CPU frequency
Last Level Cache References LL_CACHE_REFERENCES	Hits in the last level cache, includes speculation and L1-prefetcher hit
Last Level Cache Misses LL_CACHE_MISSES	Misses in the last level cache, include speculation and L1-prefetcher hits
Branch Instructions Retired BRANCH_INS_RETIRED	Counts branch instructions at retirement
All Branch Mispredict Retired BRANCH_MIS_RETIRED	Mispredicted branch instructions that are retired with the branch later not taken

Table 3: Architectural counters in current Intel architectures. Source: [Int11c] Page 1141

predictor. The last event in Table 3 samples these branches that were taken but are later discarded because they were mispredicted.

The description up until here already provides a glimpse into the complexity and high hardware dependence of the performance monitoring facilities, even though most of the details were only from strictly Intel CPUs of the past few years. It becomes clear that these facilities need to be abstracted by an intermediate layer and be handled transparently to the programmer using performance monitoring. He neither cares for what counter register to program an event in nor the architectural delicacies of the different versions of performance monitoring facilities. Checking the availability of the counters that he wants to use is, due to the associated complexity, also not something a user wants to performan manually. He should be able to use a simple, and straight forward API that handles this initialization for him — a library that acts just like a driver for performance monitoring.

I limit myself to Intel CPUs for this thesis because at the time of writing these were the only recent CPUs available to me. While these only present a subset of the possible performance monitoring techniques, they already introduce quite a lot of complexity and the implementation will demonstrate the versatility of the library nonetheless.

The requirements I determined for this library are the following:

1. simple interface
2. check for availability of counters
3. choose the best counter type possible
4. allow monitoring of an application by another process

5. easily extensible to newer CPUs

The first of these goals is also the main reason to implement this library — to simplify the usage of performance monitoring facilities. The library should check the availability of counters, how many of them can be programmed, and their bit width upon startup and handle this information transparently for the programmer. It should also try to use the best possible counter, so that if a fixed-function counter exists for the same event type this counter is used rather than one of the limited programmable ones.

Another feature that would be important and will be tackled by the fourth requirement, is to make the measured results available to other programs in a low overhead fashion. This communication of the values is especially beneficial, if the evaluation of the measured results is a more complex calculation. Using external evaluation of measured data makes the actual instrumentation of an existing program easier by just requiring the insertion of a sampling point. It would also be best, if other, non hardware counter, performance and statistical data could be transmitted the same way so that only one library is needed for instrumentation purposes.

The fifth point is a long term goal for the library. It should be designed versatile enough that new architectures and performance monitoring techniques could be introduced easily enough, and especially transparent for the user. The program should simply compile with a new version of the library and then be ready for the new architecture. I did not implement this in the library because I did not have an AMD board available for testing. The structure of the library that I will introduce in the next section is versatile enough to be adaptable without changes to the user API.

2.1.2 HAECER — A Generic Performance Measurement Framework

The most important part in this library is the external API visible to the programmer. Making it easy to use, and at the same time also adaptable to the multitude of largely differing performance counter facilities available, is a challenge. I did want an API that is stable and did not require the programmer to rewrite the sensors and instrumentation points in his application when a new CPU architecture is released and is included into the library. All that should be needed is a recompilation — or even only a relinking against the library. Still, the abstraction should not come at the price of a significant runtime overhead or introduce too many additional events. The way to achieve this goal is to remove all visible occurrences of concrete event numbers and registers from the sight of the user of the library, but provide them to the library through its own type that contains all the relevant information to execute fast reads on performance counters. The functions should then only need this struct and be able to perform fast and low overhead counter reads. The API described in the next section relies on such a structure type.

I called my program HAECER because it will be used in the HAEC project for Highly Adaptive Energy-efficient Computing (HAEC) and will contain a method to read energy values. So HAECER is short for HAEC Energy Reader.

2.1.2.1 Core Library Functionality

The library that implements the abstraction of the performance counter hardware has an initializer function that detects the CPU type and vendor and from this deduces the performance monitoring facilities available. This assumes that a CPU that implements the CPUID instruction is used, which is valid for all x86 processors since the first Pentium released back in 1993. It is a safe assumption to make that every CPU still in use today supports the CPUID instruction.

If either no Intel CPU is found, or any other of the hardware monitoring detection steps fails, the library will put itself into a disabled state. All function calls to setup and management functions of the library will fail gracefully in this case

The CPU family and model are stored in the library's internal management structures, because they are used later on to select the correct event counters based on a list of available counters for each CPU model. This list of counters is maintained in the form of the so called *event counter database* and compiled into the library itself. The selection of the correct counter is only performed at run-time. This run-time selection ensures that the system can be started on any compatible core and continue functioning as normal, as long as an event type counting the same event is also available on the new architecture. More details on this can be found in the following API section on the user API for the library.

After this initial detection routine the global structures of the library are initialized. They indicate whether a counter is used or not and support an in principle unbounded number of performance counters, only limited by available system memory.

2.1.2.2 Giving Applications Access to MSRs

Management of performance counters as well as the actual counters are practically always implemented as machine specific registers (MSR). Because a number of these MSRs are also controlling security and performance critical aspects of the processor, access to them is generally reserved to code running at the highest privilege level (CPL=0). In order for a program using the HAECER library to use the performance counters it needs the ability to write to those MSRs to select events or perform actions on the counters. They also need the ability to read them to retrieve the results of the performance measurements from the programmable or nonprogrammable counter types. Although the actual reading of performance counters via the RDPMC instruction can be enabled for user level applications by setting the *Performance Counter Enable* bit in CR4 machine register, the same is not possible for access to MSRs. Due to the security implications of access to MSRs no flag to enable direct, unguarded access to the WRMSR and RDMSR instructions is present on the processor. Further, RDPMC will also not enable access to all performance monitoring registers. The energy counters discussed later, as well as half of the Sandy-Bridge performance monitoring registers, are not available via RDPMC according to the Intel manual [Int11c]. I see two possibilities to enable programs with user level privileges to securely access the machine status registers:

1. trap and execute the instructions

2. implement a system call

The first option would let the program just use the instructions to access machine status registers. As the CPU will refuse to execute them for a user level application it will trap with a general protection fault. This fault could then be caught by the kernel, and, depending on the register that was accessed, either the RDMSR/WRMSR instruction is executed on behalf of the user program by the kernel. If the wrong register was accessed the application could be forcefully terminated due to the protection fault. This has, however, one major drawback. The policy on which registers are accessible by an application would have to be implemented inside the kernel. Because, according to generally accepted microkernel design principles, policies do not belong into the kernel, but into userspace, this is not acceptable. The MSR access instructions could only be executed by a governor enforcing the policy, but the kernel cannot reliably determine what program executed the trapped instruction and decide if it was allowed to do so, or not. This makes trap and execute difficult to implement in a security preserving way.

A system call, which can be guarded by a capability, is more versatile in this regard. The MSR access capability might only be given to a server that provides access to the MSRs to others and checks the machine status registers accessed by a client against a white-list of allowed registers. This moves the policy enforcement into userspace and restricts direct access to machine status registers to specific applications. Due to the limited amount of time available I decided to leave out the filtering of the registers, and only implemented a prototype version that hooks into an already existing system call. The implementation of a capability and the corresponding policy enforcing userland program is left for future work.

The implementation of the system call is based on a previous patch to the `thread_object` class of the Fiasco kernel by Björn Döbel. This version was limited to specific MSRs and I extended it to be able to access any number of machine status registers. The system call that the extra functions were inserted in is called `sys_thread_stats`. By providing specific parameters to the standard `sys_thread_stats` function in the UTCB, programs can now either read or write a specified machine status register and return the result in case of a read to the caller.

Because this implementation is only temporary, and will be replaced with a complete API in time, a macro was used to wrap accesses to the modified system call. It automatically chooses the correct parameters to pass to the `sys_thread_stats` function based on its own parameters and provides a clean interface for the user. A version using an MSR access server implementing policies or another way to access the machine specific registers may use these interfaces allowing existing software to run seamlessly with a new implementation.

While I access all performance counters for the majority of the remainder of this thesis using this interface, it might improve performance to selectively access performance counters that are available using the RDPMC instruction directly and not via the detour through kernel mode.

2.1.2.3 The API

There are basically two types of functions that are accessible to a user of the HAECER library. Those that are used for allocating and freeing of the performance counters and those that are inserted into the code to instrument, which provide the instrumentation points. The former of these can take all the time they need to execute and are not at all sensitive to execution time or introduced overhead. They are usually only called during the setup phase of the application and should perform all the necessary checks on counter availability, and perform efficient counter placement depending on the counter type. Currently a separate setup function for each type of counter is used. The functions available to set up the performance monitoring counters of the processor can be seen in Listing 1.

```
PerfCounter setupArchCounter(ARCH_counter ctr, int flags, char mask);
PerfCounter setupFFCounter(FF_counter ctr, int flags);
PerfCounter setupNonArchCounter(event ev, umask um, int flags, char mask);
```

Listing 1: Methods to set up performance monitoring counters

```
void disableCounter(PerfCounter *ctr);
void enableCounter(PerfCounter *ctr);
void resetCounter(PerfCounter *ctr);

void freeCounter(PerfCounter *ctr);

unsigned long long readCounter(PerfCounter *ctr);
void simultaneousReadCounter(PerfCounter *ctr, unsigned long long* buffer,
                             int count);
```

Listing 2: Operations on performance counters

Counters can be disabled, enabled and reset to zero using the corresponding functions detailed in the header file. This enables pausing the measurement and a later continuation, and can be used to cumulatively measure a recurring function or part of a function in a program without reading the counter value each time.

Freeing a counter is another feature that is needed. As counters are a limited resource, they should be freed by an application as soon as they are no longer needed so that others may access them. Even if they are multiplexed, which makes applications independent in their counter usage, freeing them might be needed when different parts of an application require different performance instrumentation and the number of available counters is not enough to provide all these event types at the same time.

The last two functions, given in Listing 2, are the core of the actual instrumentation. They can be used at any point in the instrumented program and provide a low overhead way to extract performance measurements. The first function is the obvious solution for reading a single performance counter and returns the counter value as a 64 bit unsigned integer. The second function can be used when many counters should be read at once. The advantage of this method is that the library is able to disable the counters all at the

same time, yielding results that are closer together in time. It reenables the counters again afterward, so this handling is completely transparent to the user. The overhead of reading the counter, which is included in the measurements, is minimized using this approach and fewer events are added to the counters due to the multiple readCounter() calls.

These functions handle invalid performance counters gracefully. If the application does not deal with failed counter setup there will be no negative side-effects by any of the library calls. The read functions return minus one for failed counters and the other functions just return without performing any action on the invalid counter item.

2.1.2.4 The Event Counter Database

The event counter database stores information on all known nonarchitectural performance counters and is the heart of the library. When I started my thesis I only had access to an Intel processor with a Westmere core and already knew that this would only be temporary. Because of this the database was designed to be easily adaptable to the Sandy Bridge core that was later used for most of the measurements in the thesis. The database consists of a file that uses a number of preprocessor macros to build a structured representation of the event counters that also holds the information on which Intel cores the corresponding event is available. The counters are divided in so called *Event classes*, which loosely correspond to individual functional units of the processor. The database is written as a Class structure and can be used directly in C++. For usage in C applications the header file is converted to a number of constants, which can then be used in C files. The user does not need to care for what counter structure to include, as this detection and selection is done automatically based on the language used. The only relevant header file for the user is `haecer/perfmon.h`.

The database currently contains information for counters found in Intel Westmere and Sandy Bridge processors, as they were available for me to test.

This concludes the introduction into the basic framework that can be used to instrument all kinds of applications and that all the extensions that are discussed in the following sections will use.

2.1.2.5 Evaluating the Performance Counter Implementation

	normal	without SMM
min	50	56
avg	58	56
max	**316306**	**394**

Table 4: Overhead of calling the RDTSCP instruction

The next step, before any measurements will be discussed, is to establish the cost of the measurements and thus the overhead introduced by my instrumentation. There are two aspects of overhead introduced that I want to highlight, and which I will come back to when I extend the library. The first is introduced clock cycles. Each measurement

takes time and this time can be measured. Of course, the measurement of time itself also introduces overhead, but this overhead can easily be calculated by performing two successive time reads. The difference between these reads should represent the overhead introduced by measurement well enough. In Table 4 the overhead for reading the timestamp counter, the method I choose for time measurements, is displayed. For the left column the measurement was taken in a tight loop reading the TSC value by using the RDTSCP instruction about ten million times.

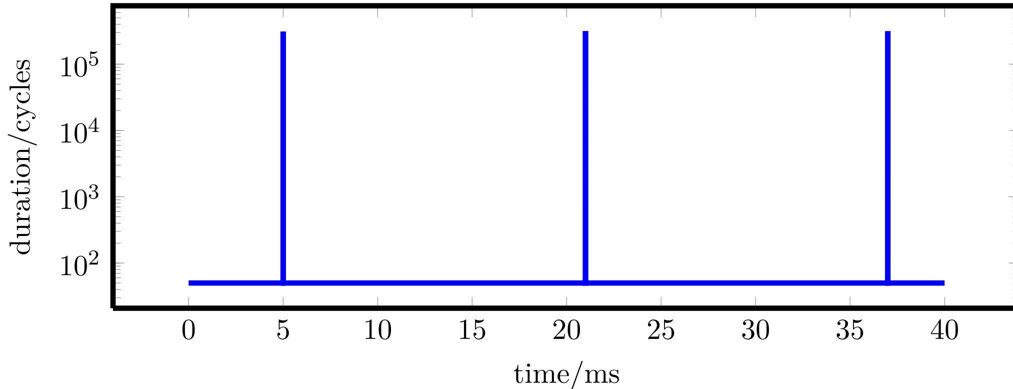

Figure 1: RDTSCP overhead over time

The left bars, which show minimum, maximum and average overhead, suggest that the reading of a timestamp can take up to 310k clock cycles, which is equal to about 0.125 milliseconds on the measured 2.5GHz Intel Core i5 processor. This duration is way too long for a simple read of the timestamp counter and the overhead introduced by this would be unacceptable. The lower average did also suggest that those high times between updates were isolated events. A recording of the individual time differences can be seen in Figure 1. Each measurement point was a tuple (tsc_1, tsc_2) that was created by two consecutive reads of the time stamp counter. The y axis is the value calculated from $tsc_2 - tsc_1$, which denotes the number of clock cycles a single read takes. The x axis displays the time of the second counter read tsc_2 relative to the first data point, which is aligned to zero. The x axis is in milliseconds, with the value calculated for the 2.5GHz Core i5 used in my measurements. It is clearly visible that the reading of the TSC takes a significantly longer time about every 16 milliseconds. I tried eliminating all possible causes for this by disabling all BIOS options possible, including the integrated graphics chip, USB, the keyboard and other on board devices — only excluding the network card used for booting, but not during OS runtime, and the serial port that was needed to transfer measurement data — and even moved the measurement into the kernel to rule out other side effects. None of these changed the outcome of the measurement. This left only the System Management Mode (SMM) as the source of the introduced *"missing"* clock cycles. This internal routine of the system, setup by the BIOS at boot time cannot be easily disabled. It is responsible for managing overheating protection, fan control and other recurring jobs that are not delegated to the operating system because a failure to execute them correctly might lead to hardware failures.

2.1.2.6 Avoiding System Management Mode

Although work on exploiting and reprogramming the system management mode exists [BSD08][ccFW09], it was not feasible to implement this in my measurement system. The effects of the system management mode further influenced the design decisions I took when extending the measurement framework. For the overhead of the RDTSCP instruction I decided to filter out the SMM influenced measurement results. These filtered results can be seen in the right column of Table 4.

To not run into the same problem with system management mode in the future I chose to employ the method to wait for an SMM to occur before each sampling. The code to accomplish this can be seen as pseudo-code in Listing 3. This was placed before every measurement and ensures that I definitely have just shy of another 16ms until the next SMM occurs and skews the results again. As none of the measured code sections in the microbenchmarks in the section exceeded those 16ms the results are not influenced by system management mode overhead.

```
curTSC = readTSC
do
        oldTSC = curTSC
        curTSC = readTSC
while (curTSC−oldTSC < 100000)
```

Listing 3: Pseudocode for synchronization to SMM

The 100000 clock cycles as a threshold seen in the listing was chosen based on experience. The System Management Mode took always approximately 300-400k cycles. Because this time is usually significantly longer than the dozens to hundreds of cycles a RDTSCP takes, a good measure would be to filter out all results larger than a hundred times the arithmetic mean. The relation between SMM duration and the frequency of its appearance (always less than 1 in a million measurements), guarantees that this only filters large effects such as the SMM and not the normal jitter in RDTSCP duration.

2.1.2.7 RDMSR vs. RDPMC Overhead

The graphs in Figure 2 display several performance characteristics of the library. The time overhead for reading performance counters either via the machine status register (readCtr) or directly from userland with the RDPMC call (RDPMC). Additionally the overhead for the simultaneous reading of multiple counters is displayed for each reading method (sim*). For each method to read the counters four bars are given with the leftmost representing the reading of a single counter and the rightmost displaying the overhead of reading four counters successively. Each bar has three values presenting, from bottom to top, the minimum, average and maximum introduced overhead. They are stacked in that the average part is the *additional* time overhead for the average case over the minimal case.

As can be seen reading of counters is significantly faster when RDPMC can be used. In the same time that three counters can be read successively, four of them can be read simultaneously. The simultaneous version of reading performance counters is always

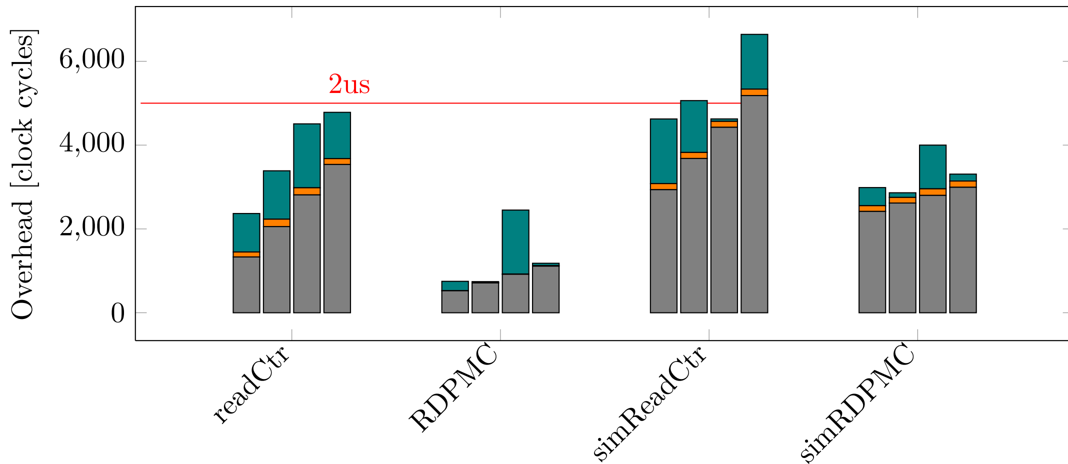

Figure 2: Performance counter reading overhead

slower, because it needs two additional accesses to machine status registers to enable and disable the counters at the same time. These actions are *always* accesses to MSRs and cannot be performed from userland. The influence of these two accesses can also be seen in the graph where a reading of three successive counters takes about as long as reading a single counter using the simultaneous reading mechanism. Because of this the simultaneous reading of performance counters is always slower than the corresponding successive readings, and should only be performed if it is important to have all counters stop and start counting at the same time.

Even though the RDPMC instruction is significantly faster than the RDMSR version, as it is not inflicting a switch to kernel mode and back, it does not support all types of counters. Especially the energy counters, which are important for my work, are not accessible using this more efficient method.

2.2 Ferret for L4Re and HAECER

In my initial analysis of the requirements of the library, point four mentioned a method to communicate results to other processes. This communication is desirable not only to avoid implementing complex analysis logic in the program that is instrumented, but may also serve to record a history of events and adapt hardware configurations according to the past behavior of the application and its past history of performance events. These performance events can not only be generated by hardware as low level events like the ones described in the previous sections, but may also be generated by an application itself indicating certain behavioral patterns that are specific to the application. Because of this, it is important that a universal communication layer exists to bring hardware and software events alike to a monitoring process. This layer should introduce as little overhead as possible to not further disturb the actual program that is measured.

As was already mentioned in the related works section of this part, there already exists a low overhead framework for communicating software performance data across

processes. It is called Ferret [PDLD06] [Poh10] and was developed by Martin Pohlack in 2006 for the L4Env software environment as part of his PhD thesis. It offers a way to instrument a system without requiring any system calls and as such costly kernel entries at runtime and provides continuous monitoring for several different application types with only one mechanism in all places enabling the collection of all monitoring data with one framework [PDLD06].The ferret framework also features sensors that provide automatic aggregation of measurement results, reducing the amount of data transferred between programs, and with this the associated overhead.

Ferret is exactly the type of application that is needed to communicate instrumented performance results, and, because it can also be used independent of the performance monitoring framework, it is also suitable to collect software performance events. The low overhead makes it the perfect fit for the goals of low intrusion performance counter monitoring.

L4Env was the previous user level environment available for the Fiasco microkernel and has since been superseded by the now current L4Re. The newer L4Re is a capability based system and because of this a part of the functionality of ferret had to be ported to the new system. For example, communication between processes is now guarded by a capability and only programs that have the capability to a communication channel may access it. Most of the porting work that consisted of translating the primitives from L4Env to equivalent mechanisms in L4Re had already been done by Björn Döbel, however, no extensive testing had been performed. During my integration I found a bug in the translated code that I had to fix to make ferret perform as expected.

2.2.1 Modifications Needed

While integrating the ferret monitoring framework into haecer three bugs were discovered in the list producer and sensors. The implementation assumed that a list would only consist of `_num_elem` list elements of `size _elemsize`. But this was incorrect because ferret performs optimizations like alignment to cache lines inside the list structure and this alignment space was not respected in the calculation of the required space.

The original equation for the amount of space needed to store a list calculated a lower amount of required space and thus access of the last elements of the list would overflow into unallocated memory. Because, depending on the page layout of the buffer, this might not lead to immediate page faults the error was quite difficult to track down and mostly lead to malformed values inside the list. Lists also contain a header that provides a per list element index element and global header data, including pointers to the next element to be written or read. When the list's buffer wrapped around it would overwrite these pointers and make the future handling of the list exhibit random behavior. This resulted in wrong values being read or written with no effect. I developed a patch that fixed the issues and that enables normal list operations again.

2.2.2 Integration into HAECER

In the interest of both easy instrumentation and high performance, I decided to directly implement access to FERRET into the HAECER measurement framework. To enable FERRET integration additional steps have to be undertaken during initialization of the library, which are completely transparent to the user. The library needs to acquire the `ferret` capability on startup, which is needed to communicate with the server and map the shared memory. If it fails to aquire this capability it behaves as if it was compiled without FERRET support.

Because the use of FERRET introduces a slight amount of overhead during measurements there is a compile option that guards whether FERRET may be used or not. It is only compiled if the constant `USE_FERRET` is defined. If the constant is not defined the lighter functions without FERRET support are used. The integration of FERRET only works if both the library and the application are built with ferret support. Functions that setup FERRET return a failure if the library was compiled without this support and the measurement functions do not incorporate methods to communicate the results. A mix between a library and a program, where one is FERRET enabled and the other is not, is generally handled gracefully with no side effects.

A performance counter can be attached to a FERRET monitor by calling the function `int setupFerretMonitor(PerfCounter *ctr, uint16_t major, uint16_t minor)`. This function creates a new FERRET monitor that is currently fixed to a buffer of 3000 list elements. This may be extended or made configurable by a simple change to the setup function. The parameters `major` and `minor` describe the identification information for the monitor, which can be used by the application that consumes the events produced by the instrumented program. By providing the consumer application with these major and minor numbers it attaches to the same shared memory region and is able to extract the data from this buffer using all facilities provided by FERRET. A pointer to the new FERRET object is stored in the PerfCounter structure for fast access when performance data is written. The function returns the return value of the `ferret_create()` function and the user of the library should check this return value to be sure that the attachment process has succeeded.

Whenever a `readCounter()` or `simultaneousReadCounter()` operation is performed, in addition to returning the read values they are also written directly into the shared memory buffer. They use the `data64[]` field of the `ferrret_list_entry_common` structure. The first element of this array is the value that has been read. The second value is the ID of the performance counter. This can be used to determine what counter the PerfCounter is associated with, but applications may not rely on this as it may be counter specific. Instead each counter should be attached to its own monitor. The third 64 bit field contains the UNIX timestamp of the measurement. While FERRET itself also timestamps data that is inserted into the buffer, this timestamp can be regarded as more precise, because it is taken as close as possible to the actual measurement.

If there is no monitor attached to the performance counter the additional overhead is as low as a pointer comparison for NULL for a FERRET enabled library. On a library without FERRET support the only change by this extension is the addition of two fields to

RAPL_PKG	The whole CPU package
RAPL_PP0	The processor cores
RAPL_PP1	A specific device in the uncore
RAPL_DRAM*	The DRAM domain of the processor

Table 5: Measurable power planes available in Intel processors, [Int11c]

the PerfCounter structure. This prevents the optional FERRET features from disturbing measurements in non-FERRET applications scenarios.

2.3 Introducing Energy Counters

There are at least two ways to measure a system's energy consumption. The first method is to externally instrument the whole system. The board and all the hardware that is to be measured needs to be outfitted with measurement points and then be attached to power meters that have a sufficiently high sample rate to capture as fine granular events as possible. Then these power meters need to be read by a computer, and their values must be mapped to individual sections of the code using a synchronization mechanism.

The other possibility is to use the so called *Energy Counters*. These performance monitoring events, which are available in the Intel Cores, starting with Sandy Bridge [Int11c], as well as AMD processors, starting with Llano [AMD11], allow online power metering directly in the CPU. For Intel processors these facilities are called *RAPL Counters*, and I will make use of them in this thesis, as an Intel Sandy Bridge CPU was the platform available to me for my work. The acronym RAPL stands for *running average power limit*, and can be used to measure the power consumption of individual components of the processor. I will focus this section on implementing support for these hardware counters. I will compare these counters to external hardware measurements at the end of part A and detail the difficulties in instrumenting and measuring a complex system correctly.

With the Sandy Bridge processors, which were introduced in January 2011 [Int11b], Intel delivered its first x86 processors that provided a facility to measure energy directly on chip. For these RAPL facilities on the processor die, the energy measurement mechanisms are only a by-product of the power limiting facility. As the name implies this part of the processor hardware can be given a power envelope in which the processor must stay. The RAPL facility calculates the average energy consumption over a user defined time window and throttles the processor should it exceed this limit. The throttling aspect of the RAPL facilities was not further pursued in this work.

The measurements update an energy register in the processor, which is exposed to software solely as a machine status register, and not readable via the RDPMC instruction. Three to four of these energy registers are available for the different power planes. An overview over them is given in Table 5. The power plane marked with the asterisk (*) is only available on newer generation server processors of the Intel Xeon brand. No such processor was available during the development of the HAECER library and because of this it was not implemented.

The counters calculate the energy consumption continuously, but only update the energy register every millisecond. The value of the registers is always increased by the amount of energy consumed since the previous update. The registers unit of measurement is joules and it is multiplied with a fixed multiplier that can be found in a configuration register that is available for each RAPL domain.

In principle the RAPL energy counters are just another form of performance counters, and as such fit perfectly into the design of my HAECER library.

2.3.1 Extending the Library for Energy Counter Support

The HAECER library was extended by two additional functions to make energy counters available to the userland. These can be seen in Listing 4

```
PerfCounter setupEnergyCounter(RAPLCounter type);
double getEnergyUnit(void);
```

Listing 4: HAECER functions to support energy monitoring

The first function is similar to the other setup functions that exist for general purpose function counters, but it takes less parameters. Energy counters are a lot less configurable than traditional performance counters They count continuously and can be neither reset, nor enabled or disabled. They also count independently of whether the processor is executing in user or kernel mode. All these features would have to be simulated in software to make them available to user programs. While such a simulation is entirely possible, it was not needed for my measurements, and, because of this, is left for future work. The second function is essential to retrieve the unit of the measured data. The double value returned needs to be multiplied with every measurement result obtained by a call to the general purpose `readCounter()` and `simultaneousReadCounter*()` functions. Calls to enable, disable or reset the energy counter have currently no effect on the readings.

Like all other performance counter types, energy counters can also be attached to FERRET sensors. The semantics of the measured data is the same as described in the previous section on FERRET. Nevertheless, there is one aspect that differs from the other counters. For energy counters, the second byte in the `data64[]` payload array of FERRET list entries is the exponent to the unit calculation. The energy in joules is calculated by $e_{ctr} \cdot 2^{-x}$ where x is the value returned in this second array field. The result of the equation 2^{-x} is also the value returned by the `getEnergyUnit()` function.

2.3.2 Evaluating the Energy Counters

The first step in evaluating the new energy counters was to determine how reliable the update interval is. Because the update interval of 1 ms takes approximately 2.5 million clock cycles, the measurement resolution is quite coarse grained for sampling short time events such as locks or individual parts of the decoding process of a frame in a video decoding application.

The results of such a measurement can be seen in Figures 3 and 4.

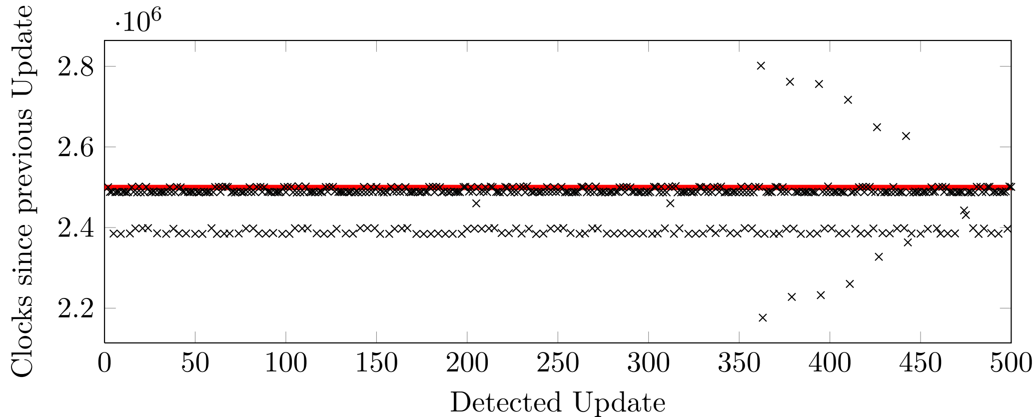

Figure 3: Analysis of energy counter update behavior (update distance)

The first of those figures shows the time between individual updates of the counter. This has been measured by continuously reading the counter in a tight loop and taking a timestamp each time a change in the counter value was observed. The left figure shows the difference to the previous TSC timestamp in clock cycles for each of the individual samplings. While there are only 500 samples displayed the general trend is the same over longer time periods. As can be seen updates tend to come in before the 2.5 million cycle mark, which corresponds to a millisecond on our processor, but on average come at an interval of 2499394 cycles, which is just below one milliseconds. On the right part of the graphics the effects of system management mode can be seen again. When interrupting the measurement loop right before an update is detected, the SMM inserts its approximately 310k cycles. The update appears to come late, but because it had actually happened earlier but was not detected, the next updated appears to come early effectively canceling out the effect. This can be seen by the negative spike that directly follows a positive one. Depending on when exactly the update happens, an effect of synchronization between update and SMM interval can be witnessed, leading to the decrease in the effect over SMM intervals that can be observed toward the end of the graph. Such a spike comes at about every 16 energy counter updates, when the SMM interval is in sync with or close to the energy counter update interval. In the beginning of the figure, the two are not in sync and are synchronizing up until the first spike can be seen.

The Figure 4 displays the amount of cycles the update was away from the exact millisecond mark. The values are relative to the average update point and not relative to the previous update seen. For convenience a line was inserted that shows the mark of 20us, indicating that the result was 2% too late or too early. All updates, except those where the measurements were influenced by the system management mode fall inside this time-frame. Here the effect of SMM is not canceled out, because the next update would not be expected later, but to still be on time. Because of this, only the upper part of the anomaly can be seen in Figure 3.

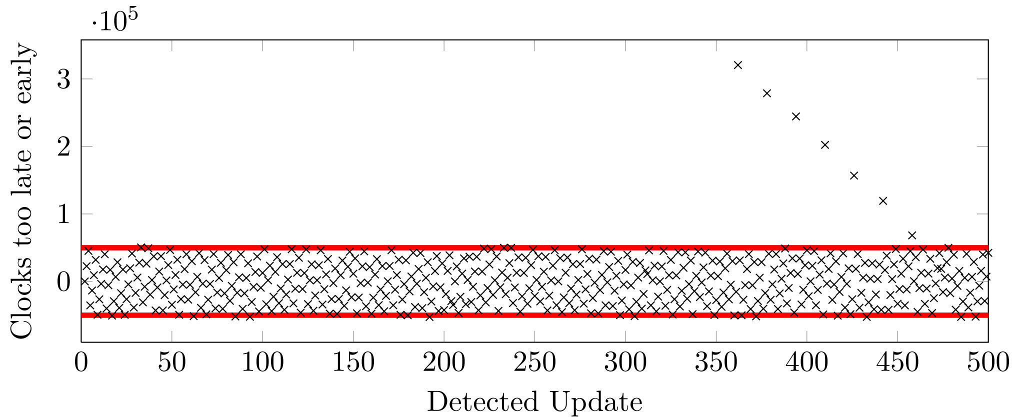

Figure 4: Analysis of energy counter update behavior (1ms mark)

2.3.3 Trying to Improve Counter Resolution

The one millisecond resolution of the power metering facilities is too coarse grained to capture shorter events. According to the Nyquist-Shannon sampling theorem [Sha49], only effects with a frequency of more than twice the sampling rate will be reconstructable from the sampled data. With a sampling rate of 1 KHz the energy counters only allow a reconstruction of signals with a frequency of less then 500 Hz, meaning that any signal that has an amplitude of less than two milliseconds is not guaranteed to be reconstructed correctly.

The illustrations in Figure 5 demonstrate the problem. The black function in the left graph, which displays the energy characteristic of a workload that has a runtime of only 0.8 ms then 0.6 ms idle time before it runs again for 0.8 ms, is to be sampled at the points indicated by the triangles. The squares indicate the points when the energy counter is updated. These later points are strictly in 1 ms intervals. When a sample is taken, the value that is read is the energy value *at the previous sampling point.* The resulting function, which can be recreated from the measurements, is shown in red in the right graph. This happens because, although the sampling captures the correct total energy consumption, it can only capture this data for the whole 1ms interval. The area under both graphs is exactly the same, but the consumed energy cannot be correctly attributed to the relevant parts of the program. If we now try to reconstruct the function, by assuming that the energy consumption between our measurement points would be exact we get the dashed function in the left graph of Figure 5. The total measured energy consumption is lower than the actual one, because the sampling points are not corresponding to measurement points. The lower energy consumption that can be seen in the original diagram is not visible in the measurements.

The problem can be divided in two parts. The first part is that the start of a measured block will not necessarily fall together with the time the energy counter is updated. This leads to a miscalculation of the start energy value. As the energy counters are always increasing this leads to the lower value of energy consumed in the counter register than what has been used before the start of the measurement. When we now sample our event

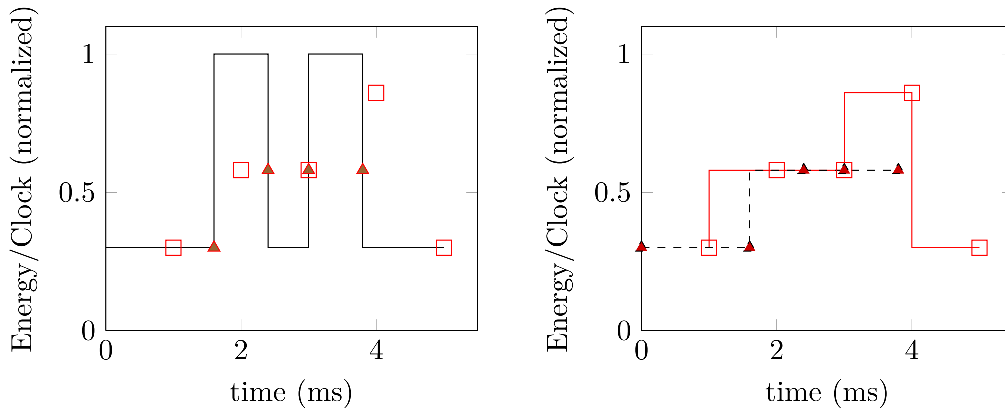

Figure 5: Sampling a short time event at 1ms intervals

at the *end* of the measurement period, even assuming this falls precisely on the point of a counter update, the value we calculate from the differences is not the amount of energy consumed in the measured time period but in the time since the previous update plus the time period of the measurement. The following equations should illustrate the problem

$$
\begin{aligned}
&r_{t=2.4\,\text{ms}} = e_{t=2.0\,\text{ms}}\\
&r_{t=3.0\,\text{ms}} = e_{t=3.0\,\text{ms}}\\
&r_{total} = r_{t=2.4\,\text{ms}} - r_{t=3.0ms} = e_{t=3.0ms} - e_{t=2.0\,\text{ms}}\\
&e_{total} = r_{total} - (\mathbf{e_{t=2.4\,\text{ms}}} - \mathbf{e_{t=2.0\,\text{ms}}})
\end{aligned}
$$

The variables $r_{t=x}$ are the read energy values at time x, the variables $e_{t=y}$ stand for the energy that was consumed at point y. The first two equations show that the read energy is always that of the previous full millisecond. When the total amount of energy is calculated, it contains the energy consumed between $t = 2.0\,\text{ms}$ and $t = 2.4\,\text{ms}$, distorting the result. The real amount of energy, e_{total} cannot be calculated, because the bold part of the equation cannot be measured using the energy counters.

This part of the problem is the simplest to solve. It can be taken care of by waiting for the energy counter to update as soon as a measurement is started. While this introduces energy consumption, it is not added to the measured part, because the additional energy is only consumed *before* the measurement point, and because of this included in the measurement of the energy before the start of the instrumented function. The only aspect that is changed is to move the start point of the function to be measured into the future, to sync it up with the actual update of the counter. In the left graph in Figure 6 the dashed lines represent the time introduced between the start signal of the measurement (triangle) and the actual update point of the counter (square). The energy level is arbitrary, as we don't care for it. This effectively eliminates the bold part of the previous equations.

The second problem is more delicate. At the end of the measurement, another sample needs to be taken and this often is also not at the point of a counter update. The obvious

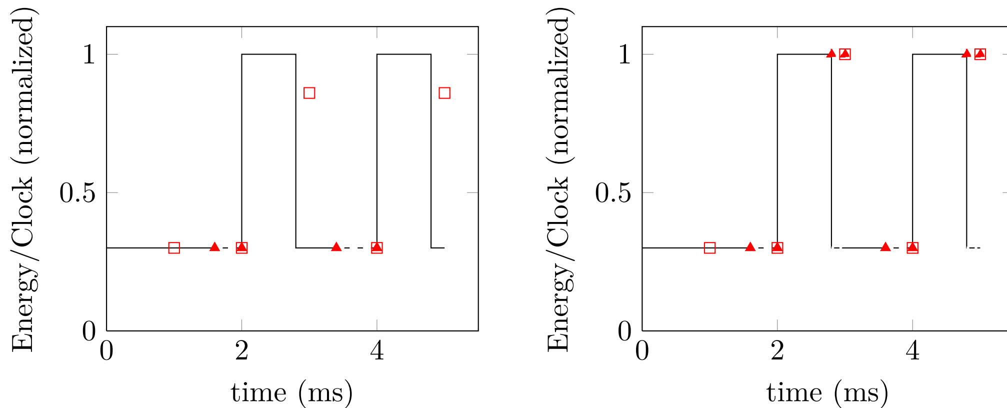

Figure 6: Counter synchronized measurements

solution, of also introducing cycles until the energy counter is updated again changes the amount of energy that is consumed during the measurement interval. Because both executing instructions, or even only waiting for the update, consumes energy that is measured and thus influences the result. The key to the solution here is to insert instructions that have a known amount of energy consumption and count them. By including this feature in the framework, as is shown in the right graphic of Figure 6, we are able to calculate the energy consumed by the following equations.

$$r_{t=1.6\,\mathrm{ms}+0.4\,\mathrm{ms}} = e_{t=2\,\mathrm{ms}}$$
$$r_{t=2.4\,\mathrm{ms}+0.4\,\mathrm{ms}} = e_{t=3\,\mathrm{ms}} - (\mathbf{e}_{\mathbf{t}\geq\mathbf{2.8}\,\mathrm{ms}\wedge\mathbf{t}<\mathbf{3}\,\mathrm{ms}})$$
$$r_{total} = r_{t=2.0\,\mathrm{ms}} + r_{t=2.8\,\mathrm{ms}} = e_{total}$$

The bold part in this equation is the part that must be known to improve the counter resolution. If we take the timestamp at the end of the instrumented section, and at the time we seen an update in the energy counter, we know the number of cycles executed during this time. When we know the cost of one such cycle (on average) the total amount of energy consumed during the introduced time period is know and can be inserted into the calculation.

The obtained measurement graph, with the introduced delays again displayed as dashed lines, can be seen at the right side of Figure 6.

2.3.3.1 Implementation

The main difference to the other performance counters is that we now need additional functions to tell the start of a measurement and the end apart. I introduced the functions in listing 5 to serve these purposes into HAECER.

```
void startEnergyRead(RAPLCounter);
double endEnergyRead(RAPLCounter);
```

Listing 5: Functions for precise short time measurements

When a short section should be instrumented, the start of this section is marked by the `startEnergyRead()` function. This function returns no value but just synchronizes execution to the energy counter updates, as described in the paragraph on the first problem in the previous section, and stores the current counter value internally. The synchronization to the energy counter is achieved by continuously reading its value until it is updated. Once the section to be instrumented is finished, the `endEnergyRead()` function is called and returns the energy consumption during the runtime of the part of the program that is executed between the two function calls. As discussed before, the later function needs to execute instructions of known energy consumption until the update to the energy counter register happens. To detect such an update, I repeatedly read the register until it changes. Timestamps are taken at the start of the `endEnergyRead()` function ($t_{endStart}$) and once the update to the energy counter has been detected ($t_{detected}$), and the time spent is multiplied with the energy cost per clock cycle of looping over the register ($t_{ePerTSC}$). This energy is then subtracted from the energy difference of the two readings, yielding the net energy of the measured function. The values returned are already in joules. The following equation documents the calculation.

$$e_{total} = e_{start} - e_{end} - (t_{endStart} - t_{detected}) * e_{perTSC}$$

To know the cost e_{perTSC} an additional calibration was added to the initialization of the library. Here the first update of the energy counter is detected and timestamped. Then the counter is continuously read until the next update and timestamped again. The difference of the two measured energy values divided by the difference of these two timestamp counter values comprises the cost for looping over the energy counter per clock cycle. The calculation formula is as follows:

$$e_{perTSC} = \frac{e_{end} - e_{start}}{t_{end} - t_{start}}$$

2.3.3.2 Analysis of Results

To analyze the quality of the results obtained by these short time measurements I benchmarked the library. The first run was performed to get a comparison result. The system was fed with load comprised of looping over the `gettimeofday()` function for 10 seconds. The energy consumed during this time was recorded and used as a reference. This measurement was **not** taken using the previously described method, but by reading the energy value directly. Only a synchronization of the start of the measurement was performed. Even if the measurement was up to 1 ms away from the next sampling point (worst case scenario), due to the duration of the whole measurement this would only cause an error of 0.01 %. This was then used as a reference to read the same load pattern with a steadily reduced load time. The smallest measured time was 1 s. The deviation from the expected result, which is the corresponding part of the reference energy, was recorded. However, this measurement yielded no useful results. The measurements resulted in estimated *negative* energy consumption, suggesting that

the energy consumed during the one millisecond interval was less than the cost for looping over the counter.

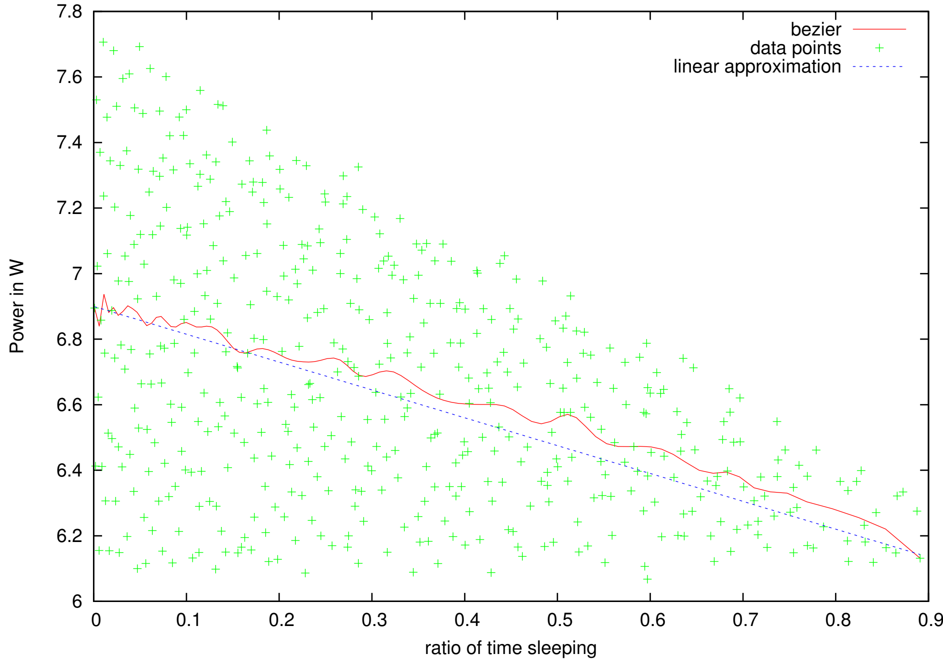

Figure 7: Energy counter benchmark for changing load/idle in one ms interval

The results show that the precision is not increasing for measurements shorter than 1 ms. The reason for this may be that the precision of the energy counter is not high enough. A further example illustrates the problem with the counter precision. When dividing the 1 ms measurement interval into n percent of idle time and $100 - n$ percent of loop time the energy consumption should see a straight linear curve between the minimum and maximum points. But the benchmark showed the results shown in Figure 7. The x axis is the ratio of time spent idling in the 1 ms interval and the y axis shows the energy consumption in watt for this one millisecond. The blue dashed line is the expected result — a linear correlation between the energy spent when only idling and when only looping. Instead, the energy fluctuates highly between the measurements. While it will not go significantly below the base energy spent when only sleeping, the measured values fange from just below the base line to a maximum value that is decreasing with the increase of idle time.

The red line is a bezier curve over the data points, and shows the approximate average. It is close to the expected line, showing that the counter is correct on average, but a single value fails to provide enough information to facilitate short time measurements.

To measure short term events, we have to repeatedly measure their energy to average out the fluctuations. For short term events that are not easily reproducible in a microbenchmark or for online measurements of short events, the fluctuations prevent precise measurements results.

3 Evaluating the Energy Counters

Now that an implementation that enables me to measure energy exists, I performed example benchmarks that generate different CPU loads and microbenchmarked individual instructions to test the possibilities of profiling applications with the help of the HEACER framework.

3.1 Examples of Different Loads

The first example is the difference between the load and idle case of a CPU. The idle case is represented by the `sleep()` function, whereas CPU load was simulated by a continuous read of the time using the `gettimeofday()` function. The graph in Figure 8 shows the data that has been retrieved by a function that samples a one second load, one second idle pattern. The measurement thread sampled the data every 100 ms. The benchmark was executed on the Core i5 Sandy Bridge processor with a clock frequency of 2.5 GHz. All benchmarks were performed with only a single core active, unless noted otherwise. Available system memory was 8 GB but limited to 3 GB due to the 32 bit version of Fiasco being used. Features such as frequency scaling or hyper threading, which could influence energy measurements, were not enabled.

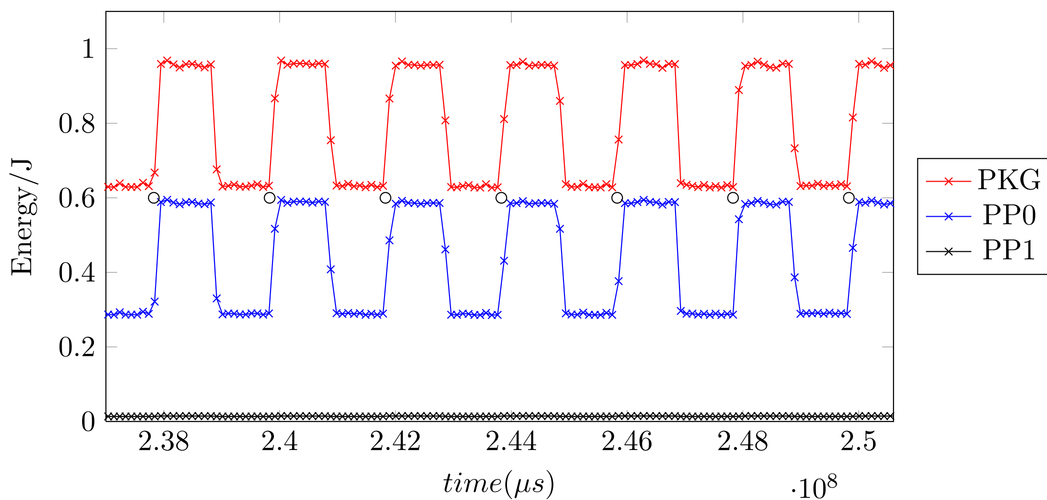

Figure 8: Energy difference of between load and idle (10 Hz sampling freq.)

The graph displays each of the three power planes available on the processor, with the PP1 plane barely visible at the bottom. The individual sampling points of the measurement thread can be identified by the x marks, and the circle marks identify

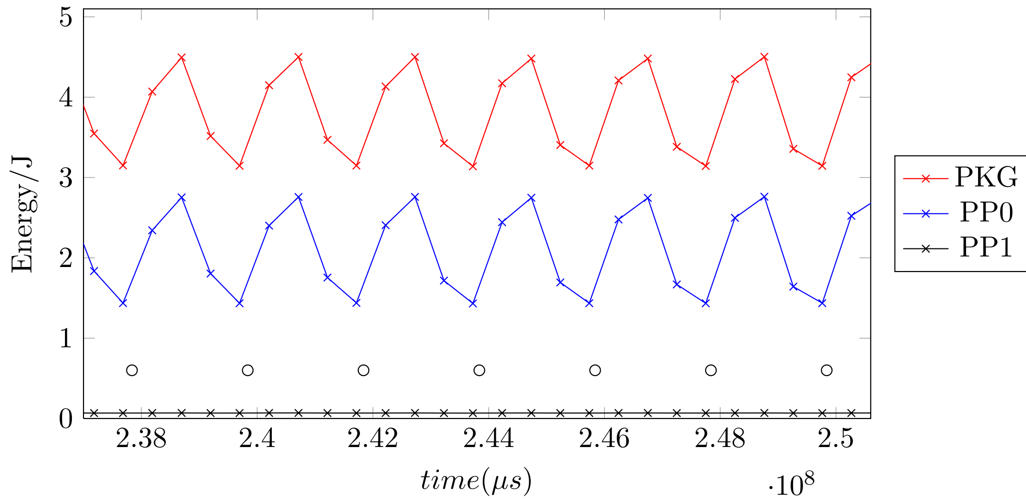

Figure 9: Energy difference of between load and idle (2 Hz sampling freq.)

the point in time when the measured thread changed from idle (sleep) to load. If the sampling frequency is reduced to half the frequency of the load/idle cycle the data seen in the graph in Figure 9 is measured. While the actual graph is recognizable, the pattern is way more sawtooth like than rectangular.

The y-Axis shows the energy consumed **since the previous counter read**. Because of this, the consumed energy in the second figure is approximately five times as high as in the previous figure. The values need to be converted to watts to have a measure that is independent of time.

I further measured energy usage data on individual instructions. They were executed in a tight loop 750 million times and statistics like the number of clock cycles needed for the instruction and the energy per clock cycle are displayed in Table 6. The first entry shows the overhead of the measurement loop, which can be subtracted from all the following values to get the net data. An energy per TSC value, although it can be calculated is not meaningful, because the TSC count only refers to the overhead loop in which no instructions are executed. Subtracting the Energy per TSC from other values will yield skewed results.

One result that can be seen is that longer executing instructions tend to have less energy consumption per clock cycle. While these microbenchmarks are interesting, they do not have much value in real world applications. In such applications the context an instruction is executing in is most important, as cache hit/miss ratios, prediction and prefetching techniques by the processor, and memory throughput play a major role in the actual energy consumption caused by an application. As such, these measurements are also highly specific to the processor on which they are taken.

Instruction	Duration (Clocks)	Energy	Energy per Clock
[Overhead]	0.00016059	0.045pJ	n/a
MOV EAX,EAX	1.57812686	5.136nJ	3.2533nJ
AND EAX,EAX	1.57811981	5.139nJ	3.256nJ
MOV EAX,ECX	0.52611966	2.011nJ	3.822nJ
DEC EAX	1.57812006	5.152nJ	3.265nJ
BSWAP EAX	1.57826240	5.140nJ	3.257nJ
RDTSC	44.18880477	141.071nJ	3.1924nJ
ADD EAX,1	1.57837791	5.135nJ	3.253nJ
RCL EAX	3.25062335	11.051nJ	3.399nJ
MFENCE	52.47508494	157.51nJ	3.002nJ
ADDPD XMM4,XMM0	4.75165067	14.943nJ	3.145nJ
MOV EAX,mem	1.35946055	4.978nJ	3.662nJ

Table 6: Energy and execution statistics for individual instructions

3.2 Alternative Ways to Measure Energy

Besides using the Intel energy counters that are embedded into the processor, there is also the possibility to perform energy measurements on the power supply or the board level.

External measurements of the platform have a number of advantages over the Intel power metering facilities. They are able to not only capture the power consumption of the processor itself, but can measure any point where means to insert a power meter exist. This extends measurement to RAM, disks, peripherials, and plug-in cards, giving unpreceded detail in the measurement of the distribution of power on a board that is under load. This, however, comes at a price. I did not venture into instrumenting a board myself, but used a prototype made available to me by the Center for High Performance Computing (ZIH).

3.2.1 Challenges

The first idea of just cutting some cords and inserting some cables to instrument the system is, although easily executed, problematic on many levels. At first direct measurement of power, meaning a simultaneous sampling of current and voltage, can often only be performed by devices with a low time resolution. Even the most expensive devices like the ZES LMG 450 or 550 only support a sampling rate of up to 20Hz. Other high resolution devices like the NI-PCI 6255 can provide sampling rates as high as 200k Samples, but have two shortcomings. First, they are only able to measure voltage. For measurements of current to be exact a high precision shunt has to be used. The voltage that is seen on this special type of resistor can then be registered by the card and the correct current can be calculated by the application of the $I = U/R$ formula. The second problem is that it is not possible to sample two sensors, one for voltage and one for current (via a shunt) at exactly the same time.

While the application of a measurement resistor seems easy for normal electronics applications, it is nowhere as simple for high precision microelectronics, where stable voltages and currents are of paramount importance. For precise measurements and especially for measurements of extremely short time periods, the simple resistance of the shunt is not enough for a correct calculation of the current. The transition resistance also starts being increasingly important the more precise the measurement is supposed to be.

The ZIH is currently in progress of instrumenting a measurement board by inserting instrumentation points into the layout of the board, between voltage converters and other components. But even the shortest introduced wire would cause the board to no longer boot. Because of this there are non invasive measurement methods that are currently explored. They involve measuring the electromagnetic field around a conductor and do not exhibit the problems of introducing resistance into critical energy supply paths of the board. But this process is time consuming and the instrumentation itself is extremely expensive. Because of this no such instrumented board was available at the time of this writing.

Another problem that occurs with all methods of external measurements is the synchronization to the individual parts of the program. Timestamping introduces the problem of synchronization between the clocks of the system being instrumented and the system taking the measurements. Just measuring directly on the system being instrumented only solves part of this problem, as the latency of the measurement still leads to an offset between getting the measured data and the time the measurement was taken. It further introduces another problem: The instrumentation itself consumes energy. Especially with high sampling rates this amounts to a nontrivial amount that significantly influences the real energy consumption of the system. Because of this, measurements taken by another computer or other external recording device are still the best options to get results that are as precise as possible.

But all external measurements exhibit another problem that is caused by the the large amount of capacities built into a modern mainboard. These capacities essentially act like a low-pass filter, smoothing the measured values and leading to delays between the start of an observed high power event and the time the higher power consumption is visible on the measurement point. They also lead to the effect that higher sampling rates may not yield higher resolution results, as short peaks are absorbed by the capacities in the board.

To evaluate these concerns and compare the results of my RAPL measurements to external measurements, a first crude implementation of a measurement setup was provided to me by the ZIH.

3.2.2 Comparing the Energy Counters to External Measurements

The measurement setup provided by the ZIH staff consisted of an instrumented Intel Xeon E31280 CPU running at a frequency of 3.50 GHz. The board contained one CPU, and of this one CPU a single core was used. Hyperthreading had been disabled in the BIOS as have been all power saving mechanisms. The instrumentation was inserted between the mainboard and the PSU by inserting a shunt in the 12 V rail. This shunt

was then measured using one of the voltage probes of the NI-PCI 6255 data aquisition card. It is important to note that this 12 V rail supplies power not only to the CPU but also the the RAM, which cannot be easily seperated from the CPU power consumption using this setup. This setup is different from the data measured by the RAPL counters, which, for the package counter, only contain the power for the memory controler, which is contained in the CPU, but not the power supply of the memory itself.

The data acquisition card was plugged into another measurement computer that had the software reading the values running. This software could be instructed by a library to start the measurement and write its data at 200k Samples per second to a local buffer. The library communicated with the measurement server over the network. Once the measurement was finished the data could be acquired from the buffering computer over the same network connection. The data points consisted of a Unix time stamp that indicates the time of measurement and a measured energy value in watts. The individual data points were exactly 5us apart, indicating the correct sampling rate of 200k samples per second.

This setup was used to run the load/idle test, which was described before in the measurement section, on the Intel Xeon processor. The data captured by the data acquisition card and the data of the RAPL counters were both recorded. They were not synchronized but this was not needed as the main goal was to get a quantitative result on the accuracy of the RAPL counters versus that of the external measurement card.

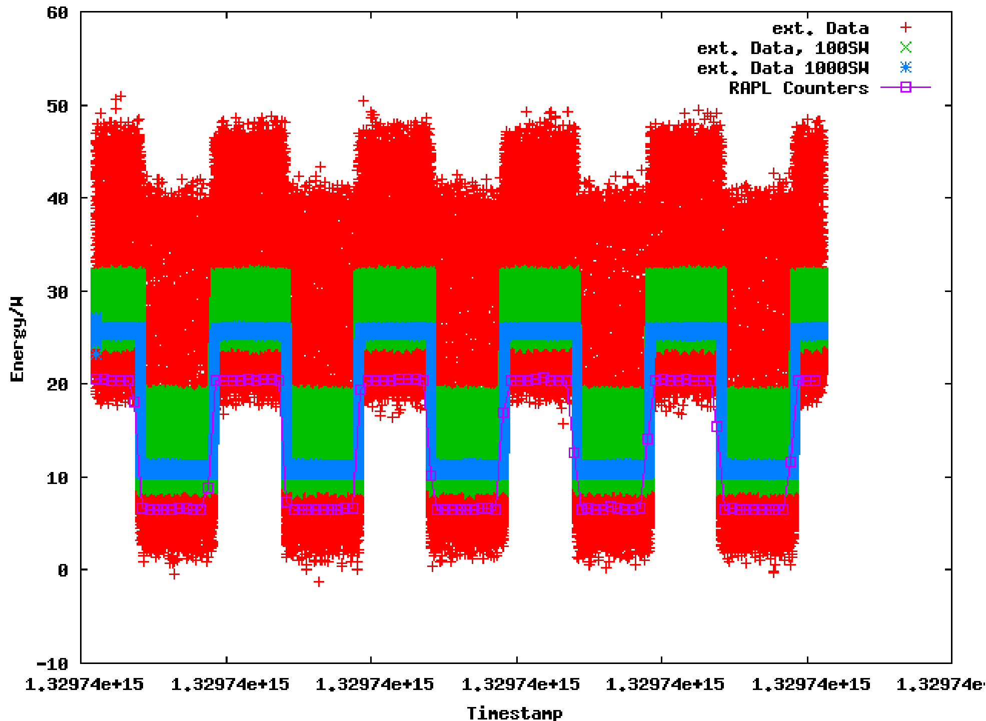

Figure 10: External measurement of load idle case

The graph in Figure 10 is the result of this measurement. The red measurement data is the raw data captured with the NI-PCI 6255 card. The 100SW an 1000SW values are the values averaged over a sliding window of 100 and 1000 values. The purple curve are the RAPL measurements taken at the same time. The RAPL values have been converted to watts for better comparability.

The figure shows two interesting aspects. The first is that the measurements have a high variance between the individual data points. While the load/idle cases are still visible it is unclear what the actual power consumption during these cases is. The sliding window averages indicate that there are only few data points in the higher range, and quite a lot in the lower range of each bulk of measurement points. A zoomed in view on one of the load and one of the idle cases is shown in Figure 11. The figures show the deviation from the average in each case. The whole peak was averaged and the deviation of each value from this average is plotted. The same was done with the idle region.

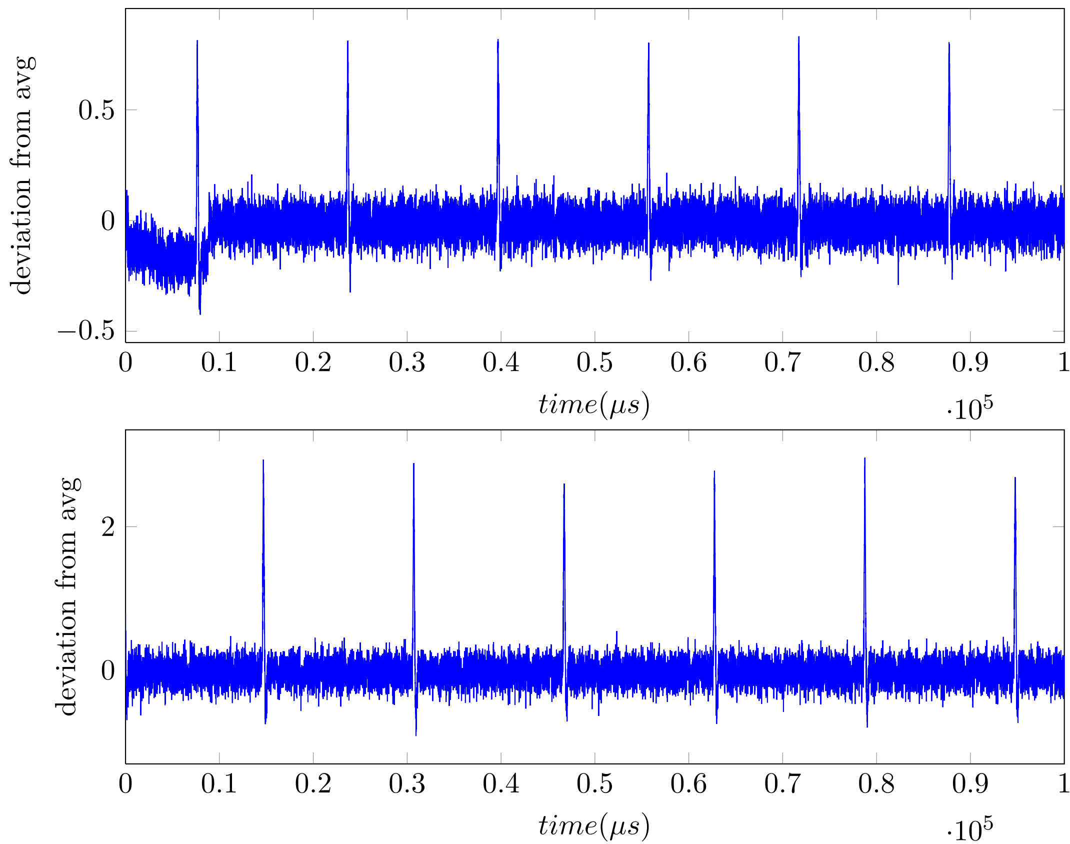

Figure 11: Deviation from average energy consumption. Zoomed in view of load (top) and idle (bottom) cases of the external measurement.

The distance of the spikes is about 16 milliseconds. The spikes are clearly visible above the noise that is usually within +/-30% of the average power consumption. This happens independent of whether the CPU is under load or idle. This fits perfectly well

with the interval of the previously observed system management mode, which now also manifests in the external measurements. With the average power consumption of 10.7W if the system is idle, and 25.2W when it is under load, the peaks amount to an additional 31W in the idle and 23.4W in the load case.

An even more zoomed in view of the spikes can be seen in Figure 12. A detailed analysis of a program would have to filter out these noise functions.

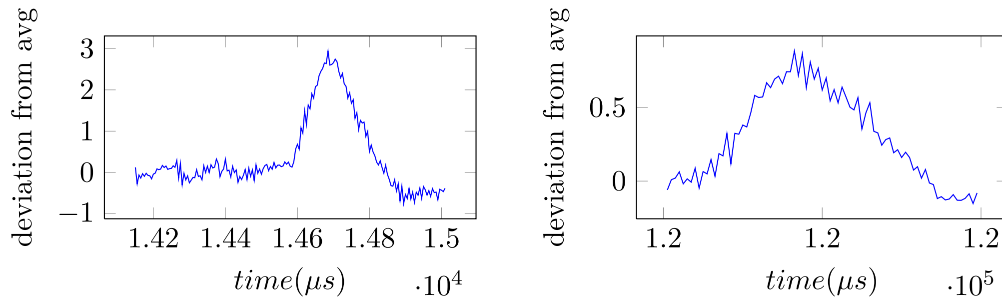

Figure 12: High detail view of SMM energy on low (left) and high (right) power cases

The remaining noise that is +/-30% of the power level in the idle case and +/-20% in the load case makes a correct analysis difficult. This noise may have different roots. Causes of error may be the assumption that the voltage on the 12V line is constantly and exactly 12V, or imprecise measurements on the shunt. Another source of the large variation in measurements may be that parts of the CPU are clock-gated often leading to this fluctuation. But then the idle measurements should not exhibit this behavior making this error source unlikely. Further measurements are needed to determine the precision of the instrumentation.

4 Summary

In this part of my work I have shown different ways to measure the energy consumption of a computer system. After having implemented a performance monitoring system for L4Re, I extended this system with support for energy monitoring and result communication. I then evaluated the implementation using simple examples.

Whereas the internal performance counters yielded usable results, they were limited in two aspects: time resolution and the devices they can capture. They were further disturbed by events like system management mode. I showed that I can reduce the influence of system management mode on these measurements, even removing it from measurements shorter than the SMM interval. The time resolution could not be improved and I showed that the reason for this is the instability of measurements of loads that vary significantly in the 1ms measurement interval.

I further tried to mitigate the missing ability to measure devices other than the CPU by measuring power consumption externally in high resolution, with the help of the Center for High Performance Computing (ZIH). While these results showed a high variance in the measurements, a part of this variance is caused by the system management mode recurring periodically and can be filtered out. Other variances may be caused by measurement inaccuracies and need further investigation. The high precision that the system management mode interrupts were resolved with is promising good results, although they are skewed by the low pass filter characteristics of the board.

The implemented library gives users of a system the possibility to instrument applications and generate profiles of system components. I will use these capabilities of the system to propose an energy model in the next section. This energy model relies on the features provided by the library and needs detailed application and system component profiles.

While the instrumentation I implemented can still be improved by a better use of RDPMC and might still need some adaptations for other architecture's performance measurement facilities, the introduced API is versatile enough to enable this. Because the library suited the needs for this thesis no such enhancements were implemented and they are left for future work.

Part B

A Novel Approach to Energy Modeling

1 Related Work on Energy Modeling

There is an extensive body of related work in the area of energy efficiency for systems. Energy modeling in particular is a topic on its own and discussed in papers by Snowndon [SPH07] or Bellosa [Bel00]. They recognize that energy models, and scheduling algorithms based on them, that are purely based on race to idle, or on constant execution at low powers, are not enough to accurately determine the best operation point for energy-efficient computing.

In their survey paper on advances in Time/Utility real-time scheduling Ravindran, Jensen, and Li give an overview of scheduling mechanisms for various kinds of Time/Utility Functions [RJL05b]. The work of Wu, Ravindran, Jensen and Li [WRJL06] is cited as a new approach for modeling energy aware Time/Utility functions, which can be step functions or non increasing. No known approach for arbitrary functions or unimodal functions is mentioned. However, the cited work does assume a CPU model that is no longer valid in today's processors. The clock gating effects of modern processors and, because of this, the high dependency on the instruction and application mix is not recognized. But their model of a Utility/Energy ratio is a promising approach that we will lean on in our work.

The team of Gernot Heiser and David C. Snowdon developed a system called Koala [SLSPH09], which is one of the few to model the trade-off between energy consumption and system or application performance. To do this it uses pre-characterized models of components at runtime to predict the energy characteristics and the performance of software [SLSPH09] [Sno10]. The Koala system has been implemented in Linux and evaluated to reduce energy consumption in some benchmarks by 30% at only a 4% performance degradation [SLSPH09]. This result suggests that there is a need for a model that can directly give the user the trade-off between energy and the performance of the system. But Heiser and colleagues do not provide a generic framework for energy modeling. Their approach centers on DVFS techniques that are available for CPUs but often not for other components of a system. We will extend on this approach by providing a more generic framework for modeling a systems energy requirements that works across many layers of a system.

In their paper on a currentcy model to enable easier energy accounting and admission as well as scheduling of energy consuming resources, Zeng and colleagues [ZELV03] provide a model for efficient and predictable energy based scheduling that enables granting different shares of energy to applications. While an interesting approach, the methods discussed in their paper are no longer valid for recent systems. Their assumption of static energy for CPU operations or an energy consumption per block access, although this may be true on average, do not hold in a real world scenario with modern clock gated processor components and disk seek times that are heavily dependent on data

position and workload characteristics. Still the model is interesting and would be worth to include in any energy scheduling mechanism.

2 A New Energy Modeling Framework - Energy/Utility Functions

In the previous part, I provided a monitoring framework that enables the measurement of energy characteristics of a resource based on its usage pattern and the current load. The monitoring library further enables the creation of energy profiles for applications by providing lightweight instrumentation mechanisms. With the knowledge that these profiles are obtainable I will proceed to propose an energy modeling framework based on Resource/Utility and Energy/Utility Functions.

2.1 An Introduction to Energy/Utility Functions

Our concept of Energy/Utility Functions is loosely based on Time/Utility Functions (TUF) [RJL05a] [JLT85]. They describe a relation between the value a result has for the user and the time at which the result was available. For example, the utility of decoding a frame of a 25 fps video is high, if the job is finished before the frame should be displayed. The utility decreases afterwards as a later result introduces more and more stuttering or drooped frames in the video.

A similar function can be specified for energy consumption versus the perceived quality. For a video player application, the quality of the result is influenced by a multitude of factors. Not only the time a frame needs to be decoded (and thus the frame rate) contributes to the perceived quality, but also the display brightness, the resolution of the video, and the number and kind of post processing steps do change this subjective measure. Further, the encoding format of the video may have an impact on the quality. Modern codecs have a lot of attributes whose settings influence the total energy consumption of a computing platform. A highly packed video stream, such as x264 needs large amounts of raw computing power, increasing the energy consumption of the CPU required for a certain level of quality. Conversely an MPEG-2 encoded video is less compute intensive but has a larger storage footprint that manifests itself in higher network bandwidth required and thus a higher power drawn by the network card. For non-streamed video the same effect can be seen in the power consumption of the storage controller.

In Figure 13 a Energy-Utility Function for a video player is displayed. It illustrates the basic characteristics that can be seen in any Energy/Utility Function. The first such characteristic is that a so called *base energy* is needed to get any useful result from the system. This energy is consumed to get any picture at all, to have the dimmest possible display setting, and the lowest possible load on the network or storage controller. We call this power that is consumed to get the resource operational the *ramp up power*. It is required to power up the devices and keep them running without any workload running

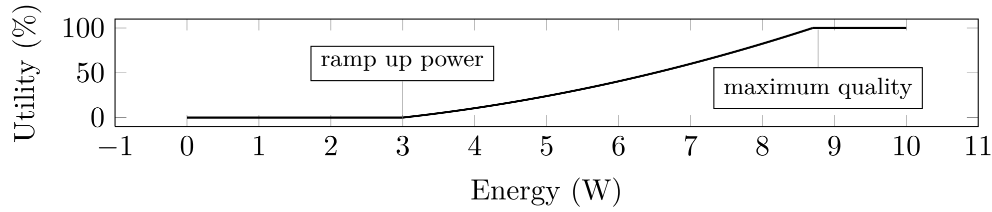

Figure 13: A basic Energy/Utility Function

on it. However, a video display would not be possible at this energy level, as the player cannot put any workload on the CPU without increasing its energy usage. Because of this, the utility for the user at this point is still zero. Below this the utility is also zero, because the energy is not enough to power up the required resource.

Adding to this base energy is a workload dependent curve that usually increases monotonically, because an increase in energy translates into more resources allocated to the application, and thus more compute power, more storage or network bandwidth, or a brighter display. The actual energy utility function might even change for individual parts of an application, depending on whether they are compute, memory, or I/O bound or on their level of data locality.

The simple mechanism as shown until here, while ideal for visualizing the general concept, is not applicable in real world systems. The energy that is needed by a computer to provide a specific service quality or utility is highly dependent on the underlying hardware as well as the application mix running on said hardware. A database application that provides a utility in the form of transactions to the user at an energy cost, cannot specify such a function without the knowledge of the hardware it will run on. Because of this specifying a generic Energy/Utility function for an application directly is not only hard but practically impossible.

The solution we investigate is to introduce a hierarchical ordering of resources and services, where each level is described in terms of the utility in provides given a certain utility of the resources it receives.

2.2 A Hierarchy of Resources and Services

The situation is not as grim as described above. There is one Energy/Utility Function that can directly be given: that for the power supplies. The power supplies are the lowest level components in a computer system that consumes energy and usually have an energy conversion efficiency that is in most cases even given explicitly by the manufacturer. All further hardware components use the energy provided by the power supplies and provide a service to other layers of the system. A hardware component of a system usually has a direct relation between the service it provides (its *utility*) and its energy consumption. For a network card the energy consumption scales with throughput or bandwidth [HGSW10], for a display with brightness. This hardware layer can be seen as the bottom of our hierarchy. Hardware components can also be modeled in more

detail, down to individual circuits, with the help of Energy/Utility functions. For the purpose of this work, we assume the hardware layer as the lowest level.

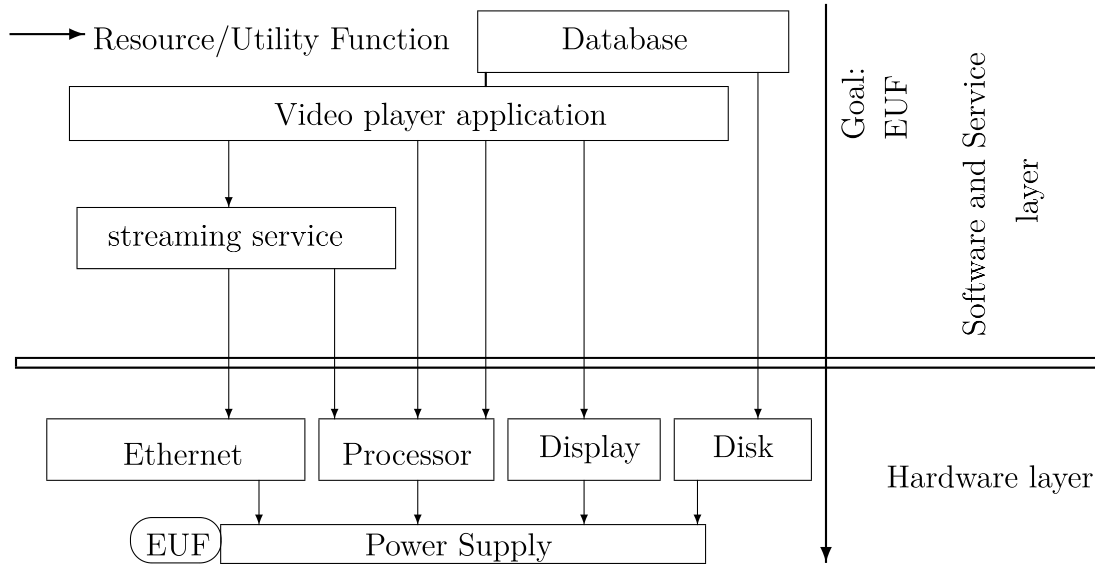

Figure 14: Hierarchy of resources and and services

All services and applications that run above this have, either directly or through several indirections, hardware components as a dependency. Figure 14 demonstrates this hierarchy for a network video player using a streaming service and a database. The components in the hardware layer each have their individual Energy/Utility Function which either needs to be measured or provided by the manufacturer. However this approach is impossible for the higher layers. They do not have a specific Energy/Utility function, because this function depends on the application mix in the system and the specific energy characteristic of the lower level resources. But higher layers have a specific *Resource/Utility Function* (RUF) signified by the arrows in Figure 14. This function determines how much utility from a number of resource is needed to provide a certain level of quality. For network this might be bandwidth or latency, whereas for the display brightness can be taken as a measure. For the hardware components themselves energy supplied at a certain voltage is the resource provided by the power supply.

Resource/Utility functions are n-dimensional functions that map the utility of the n required resources to the utility of the service. Resources for higher level applications might be low level resources from the hardware layer or other services. For the view inside the computer most applications will require at least the CPU resource. But they might require both, hardware resources and other services as a resource. The streaming video player is one such example that requires a streaming server and the processor as well as the display.

Our model has another advantage. It is quite simple to model different hardware serving the same purpose but at different quality and with different energy requirements. An example for such a setup can be seen in Figure 15. Here two network cards are

present. One might be an optical interconnect and the other a simple gigabit Ethernet connection. While the former can provide high throughput when needed, the latter might provide a low energy consumption for slower network speeds. A virtual network card, which incorporates those two interfaces can now expose an Energy/Utility Function that models both cards, always using the most energy efficient for each quality level. The VNIC driver can decide which card to use based on this model and this handling is completely transparent to the user who might only see a steep increase in the energy part of the Energy/Utility Function once the powerful optical interconnect is the more efficient device.

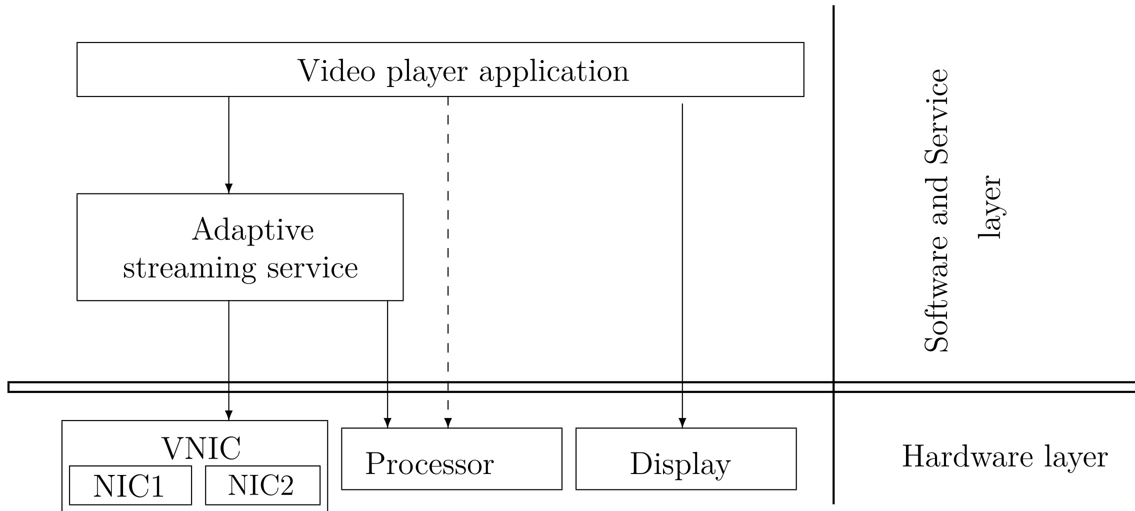

Figure 15: Hierarchy of resources and and services for composable devices

2.3 Formalization

In Figure 16 a sample device energy characteristic of an arbitrary component is shown. The figure displays the individual phases the component has, when it is used by an application. These phases can also be used for services and devices alike. For a device, there are energy costs involved in powering it up and down. Similar costs can be found for the startup or termination of a service or application. After the device is ready, the base energy, which is shared by all processes that intend to use the resource, is consumed. As soon as a resource is used by one or more clients it often consumes more energy as it provides its utility in the form of its service. For services that can be multiplexed the cost of this additional load needs to be divided between its users.

The individual energy costs for the different phases of a service that are

- For the power up of a resource
  $E_{up} = \int_0^{t_{up}} f_{up}(t)\,\mathrm{d}t$
- For the initialization of a resource
  $E_{pre} = \int_0^{t_{pre}} f_{pre}(t)\,\mathrm{d}t$

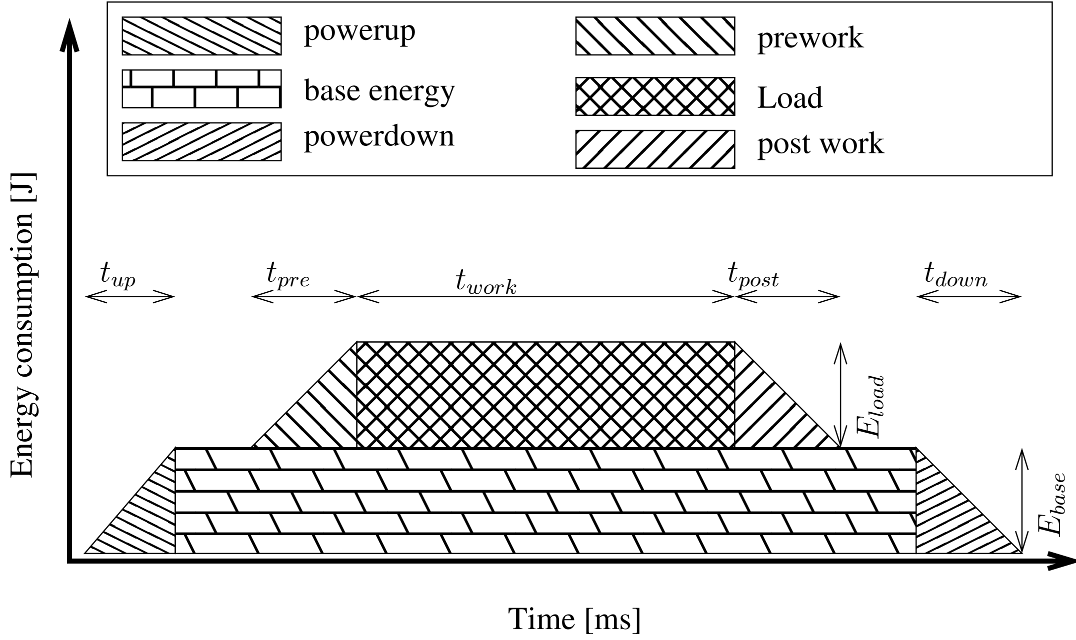

Figure 16: Simple energy characteristic of a resource

- For the work phase, during which the resource is used by clients, processes a request, or has a load for the time t_{start} to $t_{start} + t_{work}$
  $E_{wl} = \int_{t_{start}}^{t_{start}+t_{work}} f_{wl}(t, u)\,\mathrm{d}t$
  Where u is the utility required by the client user of the resource.

- For the deinitialization of the resource
  $E_{post} \int_{0}^{t_{post}} f_{post}(t)\,\mathrm{d}t$

- For the power down of the resource
  $E_{down} \int_{0}^{t_{down}} f_{down}(t)\,\mathrm{d}t$

If a client is the only user of a resource its energy requirements are the sum of all of these costs. Else the costs are shared between the clients.

Please note that there may be multiple work time spans for different numbers of clients. These time spans might also feature additional setup times for the additional client. These additional client costs are attributed to the new client.

The general energy requirements E of any resource for a utility of u may be expressed by the formula

$$E_res(u) = E_{rampup} + E_{wl}(u)$$

The energy of a resource is the sum of the ramp up energies of the transitive closure of the resources it utilizes (E_{ramp_up}) and a workload dependent part. I want to focus on the workload specific part, because the ramp up power is static.

To calculate the **Utility/Energy Function** $E_{wl}(u)$, which is needed later on, from a given Energy/Utility function $u = g(E)$ can be done by calculating the minimum energy for all energy values at a certain utility level. The function is calculated by

$$E_{wl}(u) = min(\{E|u = g(E)\})$$

The actual Energy/Utility function of the higher levels in a system can only be calculated using the **Resource/Utility Function (RUF)** of the service or resource. For an arbitrary resource that requires n other resources the utility u can be given as

$$u = f_{component}(u_{r_i}, \ldots, u_{r_n})$$

where the Resource/Utility Function f itself is application specific and defines the mapping between the utility of required resources and the utility u of the application.

If the Energy/Utility Functions should be derived from this RUF, then the corresponding Utility/Energy Functions $E_{wl}(u)$ of the lower level resources must be available, which is the case for the lowest level - the power supplies.
The **Resource/Energy Function (REF)** can now be computed by combining the RUF with the resources' individual UEFs.

$$E_{component}(u_{r_1}, \ldots, u_{r_n}) = E_{wl_{r_1}(u_{r_1})} \oplus \ldots \oplus E_{wl_{r_n}(u_{r_n})}$$

The combination of the resource energys is dependent on the individual Energy/Utility Functions. For linear functions the $\oplus$ can be replaced by a normal plus sign. This function maps for each combination of resource utilities the corresponding energy required to provide the utility point in the original RUF. However, this function is still multi dimensional and no direct mapping between the total energy consumption and the maximum attainable utility at this energy level. The determination of this direct mapping is a search problem. The formula to calculate the Energy/Utility function is

$$u(E) = max(\{u|u = f_{component}(u_{r_1}, \ldots, u_{r_n}) \wedge E_{component}(u_{r_1}, \ldots, u_{r_n})\})$$

This extracts the optimal Energy/Utility Function with regards to energy consumption. The equation first extracts the isoline of a specific energy from the resource's REF and then looks at the same points in the resource's RUF described. The point with the highest utility in the isoline is taken as the utility for this energy level. This approach is already simplified, as a complete approach would be to also punish switching modes too often along the Energy/Utility Function. An optimization problem that has those switching costs as a constraint would be the correct approach to solve this.

Using the same strategy, the Utiliy/Energy function, which is needed for other applications, which are using the currently inspected application as a resource, can be calculated as well. The equation for this is

$$E(u) = min(\{E|u = f_{component}(u_{r_1}, \ldots, u_{r_n}) \wedge E_{component}(u_{r_1}, \ldots, u_{r_n})\})$$

This modeling approach can also work with devices that have multiple discrete states, such as predefined frequency settings in modern CPUs. They may induce jumps in the Energy/Utility Functions.

2.4 Composing Workloads

i A further concept that must be developed for universal and comprehensive Energy/Utility Functions to work is that of composing different workloads on a system. Multiple applications may run at the same time or use time sharing techniques, depending on the number of available cores. They can also request the same resources at the same time. Resources in this context may be devices in the hardware layer or services in the upper layers of a system. A mechanism is needed to attribute the Energy costs to individual processes. There are three immediately visible costs when considering such systems

1. setup or ramp up cost
2. basic device operating cost
3. switching or multiplexing cost

The first point on this represents the cost for setting up the device or service. Because it is not possible to reliably tell which applications will need the resource in the future this cost could be attributed to the first user accessing the resource. The second point in the list is the cost to keep the resource operational without any utilization. This cost should be shared proportionally between applications using the resource. This modeling approach also has a major advantage. It makes process placement decisions directly possible using the Energy/Utility approach. As the cost is shared between processes it gets more attractive to move jobs to a system or core that already has this resource active. However, there are more energy considerations to take into account when considering consolidation of jobs using similar resources. One of these decisions is the so called switching or multiplexing cost. Once a device is no longer used exclusively by one application, or a service is shared by multiple clients, there are multiplexing costs involved [MSB10]. For devices, these are the costs of stopping current operations if necessary, storing the device state, switching to the state of the other process, and then starting the operation for this process need to be considered. In the case of services this is especially important for resources that keep a more complex internal state of the client. When another client uses the service, the state might not be in the cache and the warming up of the cache will again cost energy.

All these costs can also be modeled by Energy/Utility Functions. The first is a basic constant E_{setup} that represents the energy cost for getting the device or service in an operational state. The second is the resource's base energy E_{base}. The switching cost need to be measured on a per application or device basis, and should depend on the number of clients served by the resource. In a simplified view it can be seen as a function of the number of clients $E_{mux} = f(n_{clients})$.

The operational cost of resources with a Energy/Utility function is shared proportionall between users for resources where the utility can be multiplexed. The network card is one example of such a multiplexable resource. If one device needs 25% of the offered utility, and another uses 50%, then they get accounted 1/3 and 2/3 of the operational costs that are above the base energy. If the function describing the energy costs is not linear, than the operational costs of the resource might actually decrease or increase if another client uses the resource.

For devices where the utility is not multiplexed the accounting differs. For example, assuming a linear brightness/energy relation for a display, if one application requires a brightness of 50%, the other of at least 10%, the question is who gets attributed how much of the energy cost. The additional application requiring 10% display brightness should not pay for the 50% requirement of the other application, while it may still benefit from the higher display quality.

3 Characterizing Resources

In the first part of my thesis, I discussed the methods to measure energy consumption of devices. The preceding sections contained a description of a model for energy characteristics that can be used as a basis for accounting and scheduling purposes. But this model requires that energy characteristics of individual devices used by applications are known. In this chapter I want to show that it is possible to obtain such characteristics using the measurement framework introduced in part A and also by means of other measurement methods.

We can now replace the boxes from Figure 14 on page 45 with the Energy/Utility functions and Resource/Utility Functions they represent to get a hierarchy of functions. To get the low level functions of the hardware components the CPU and the display need to be modeled.

3.1 Devices

I modeled the energy requirements for two devices. The display using an external measurement method and the CPU using the measurement framework that was introduced in part one of the thesis.

3.1.1 CPU/Caches

The CPU is the main energy consumer in most of today's systems and, as I now have a framework that enables me to measure the CPU's power consumption, I can further characterize the precise power requirements of different types of instructions and loads.

The CPU is quite a complex device concerning its energy model. Because large amounts of the CPU's transistors are clock gated to save energy there can be major variations in the energy consumption even if no features like Dynamic Voltage and Frequency Scaling (DVFS) are used. The cache bandwidth and utilization, as well as the cache's hit/miss ratios are specific to an application and the cache architecture. They determine the CPU's energy consumption for a specific application mix. But also the type of instructions that are executed determines the amount of energy consumed by the CPU. The power consumption of a processor can be expressed as the sum of its static power E_s and its dynamic power E_{dyn}. The static power is the baseline of the processor when it is powered up and not performing any work. The switching power depends on the actual amount of transistors that are switched and can be calculated by the formula $E_{dyn} = f \cdot C_{sw} \cdot V^2$ **??**. Here C_{sw} describes the switched capacitance, which is dependent on the actually executed instruction mix and the usage of the different cache levels. The voltage (V) and frequency (f) can also be adapted on modern processors. This adaptation is known as dynamic voltage and frequency switching (DVFS).

The switched capacitance in a system C_{sw} can be calculated as

$$C_{sw} = g(L1_{HR}, \ldots, Ln_{HR}, L1_{bw}, \ldots, Lnbw, I_{mix})$$

with Li_{HR} describing the cache hit ratios for the individual levels of a n-level cache architecture. Li_{bw} describes the utilized cache bandwidth and I_{mix} is a description of the current instruction mix.

So in order to characterize an applications CPU usage we first need a complete model of the CPU's cache characteristics.

I designed a microbenchmark to measure the cache utilization, provide different hit ratios for the different cache levels of the processor and measure the energy consumed for the given hit ratios and bandwidths.

3.1.1.1 Design of the Microbenchmark

To sample a sufficient amount of cache misses I needed to work around the prefetcher of the CPU. The standard way to circumvent this piece of hardware is to traverse a large linked list of memory locations. The algorithm I employed first filled a 1 gigabyte array of pointers, with the addresses of the next array element. The algorithm choose a random memory location in the array but would check that no short loops exist in the linked list. The length of the linked list was steadily increased to increase the cache miss rate of the different levels of cache. The last array element would link back to the start of the list, effectively creating a cycle of length n.

Then a tiny assembler program, which traversed this list until it had read 4Gi array elements, would run. This large amount of data creates a large time window for the energy accounting to minimize the error of the power readings. The code for the assembler program can be seen in Listing 6. The address of the array is given in the ESI register and reading starts at the base of the array. The code always reads the next memory location from which it should read. To minimize the overhead by the loop, the instruction was replicated 1024 times.

```
__asm__ __volatile__ (".intel_syntax noprefix\n"
                "loop_me:\n"
                "           .rept 1024\n"
                "           MOV ESI,[ESI]\n"
                "           .endr\n"
                "           SUB ECX,1\n"
                "           JNC loop_me\n"
                ".att_syntax prefix\n"
                : /*out*/
                : /*in */ "S"(data), "c"(runs)
                : /*clobber*/);
```

Listing 6: Cache benchmark

While this approach was functioning, it yielded only very low memory bandwidths and, especially, only one bandwidth for each hit ratio. The reason for this low band-

width is that the next instruction cannot be executed until the previous instruction has finished, as it directly depends on its results. This effectively makes instruction reordering impossible and, more importantly, leads to pipeline stalls and because of this to an under-utilization of the processor. To fix this I introduced the concept of *streams*. The example in Listing 6 uses exactly one stream and only one MOV instruction can be in the pipeline at once. If another loop was added in the array, then this linked list loop could be traversed as well. Because of this the example was extended to five such *streams*. They start at the first five addresses of the array and are not intersecting. The individual addresses of the loops come from all the space in the array and do not segment it into five subarrays. The registers EAX,EBX,EDX and EDI were used to implement the other 4 streams. While this did not achieve the maximum throughput supported by the L1 cache, it enabled the easy implementation of variable bandwidth tests. Depending on the number of streams the bandwidth was up to five times as high as in the single stream version.

3.1.1.2 Measurement Results

In Figure 17 the energy consumption is displayed for different bandwidths and hit ratios of the L1 cache. All accesses that would not hit the L1 cache went into the L2 cache. The L3 cache and memory were not significantly accessed for this benchmark (less than 10^{-6}% of the accesses). Because the surface plot is interpolated for better visualization, I provide the raw data in Figure 18 so that the exact measurement points can be seen. All energy values display the additional energy consumption of the memory accesses. The base power of 6.314W was derived from the sleep power and is already subtracted to put more emphasis on the differences in energy consumption.

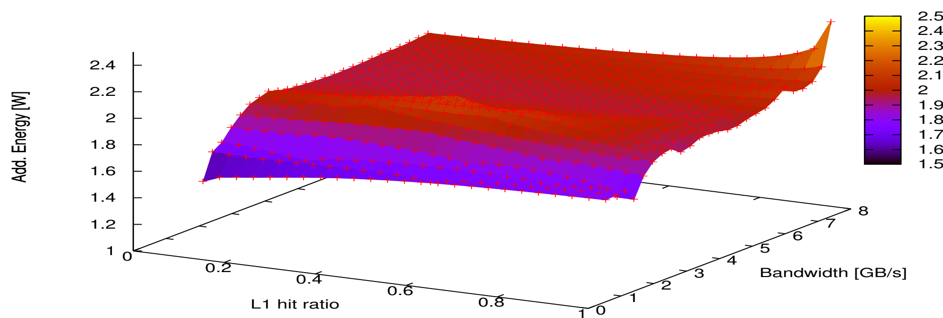

Figure 17: L1 cache energy characteristics

As can be seen the energy consumption mainly increases with the bandwidth usage of the processor, because then more transistors are active more often and not clock gated. The cache miss ratio also has a minor influence on the energy consumption, but mostly due to the higher bandwidth possible when more accesses go to the first level cache.

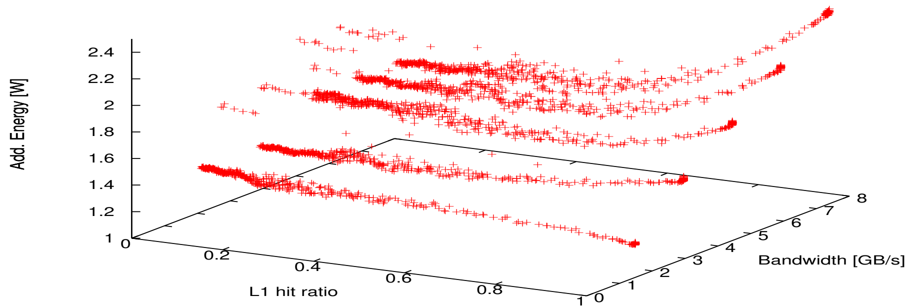

Figure 18: L1 cache energy characteristics (raw)

The surface in Figure 19 three shows the same graph as before but for the last level cache (LLC) *miss* ratio. All accesses that would not be cache misses and went to memory would go to the L3 cache. The L1 and L2 caches served up to 10% of the cache hits, which could not be prevented easily.

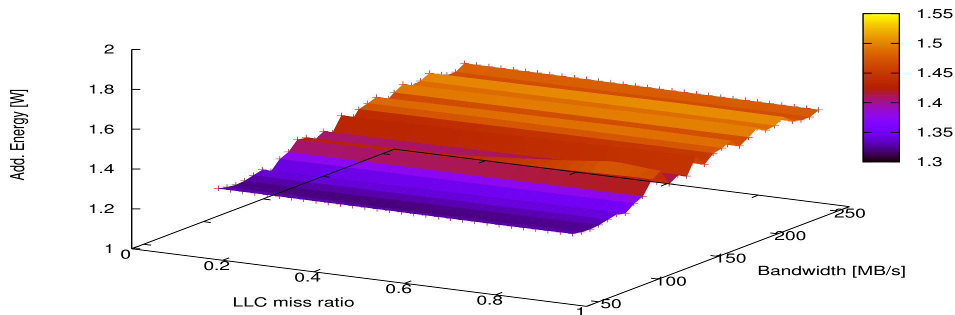

Figure 19: LLC cache to memory energy characteristics

The three figures show that the energy consumption of the CPU is dependent on the bandwidth that the memory is accessed with and the hit ratio of the caches.

3.1.2 Example: Display

The display is one of the more intriguing adaptive components. It is the only component where its utility might decrease if you put more energy into it. The reason for this decrease is that the display brightness, which is the dominating parameter for display energy consumption, can hurt the eyes if it is set too high. This scenario is especially

valid at night or if working in extremely dark environments. As such, this component was important to characterize to capture more energy consumption characteristics of the hardware components involved in a video viewing experience.

3.1.2.1 The Measurement

For the measurement of display energy consumption, a NEC MultiSync LCD1960NXi was used. According to the datasheet [LCD], the typical energy consumption for the monitor is 38W and it has a maximum display frequency of 75Hz. Its brightness ranges up to $225cd/m^2$ with a typical contrast ratio of 500:1. It is powered by an internal power supply and has built in speakers.

The device that was used to measure the energy consumption of the monitor is a Voltcraft EnergyCheck 3000 [Vol] that is plugged between the power outlet and the power cord of the monitor. The precision of the measurement device is +/-1% +/-1W.

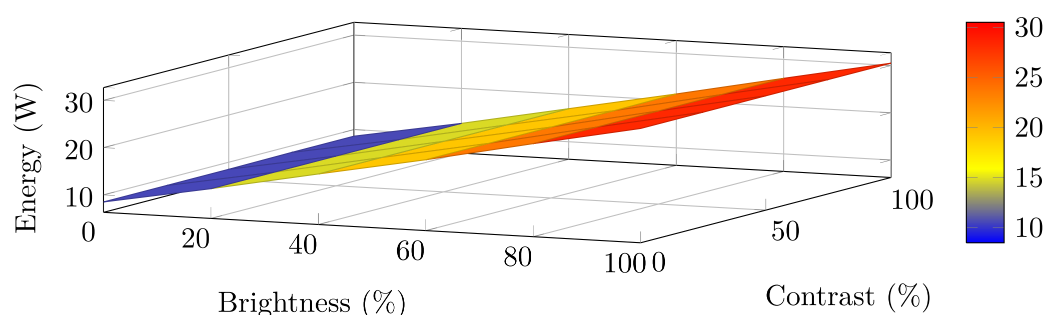

Figure 20: Display energy for a black picture

To measure the energy consumption of the monitor for various picture quality settings different patterns of pictures were displayed. The picture was fed to the monitors DVI input with the VGA input and the audio input disconnected. The brightness and the contrast of the device were varied for the different measurements. The brightness and contrast were varied between 100 and 0 percent as displayed on the devices OSD. The brightness values were changed in steps of 20 percent whereas the contrast was changed in steps of 25 percent. The value of the current energy consumption was read from the Voltcraft measurement device when the number shown on the Voltcraft's display had stabilized.

3.1.2.2 Measurement Results

A first measurement with a video revealed constantly changing energy consumption values. These changes were traced back to the display's energy consumption being dependent on the displayed content, and especially on the subpixels that are enabled. The two extreme cases were chosen for the benchmark, with a plain white picture having all subpixels turned on and a plain black picture having all turned off. An intermediate picture with the top half of the image white and the bottom half black was shown as well. The results of the black picture's measurement, which are the baseline for the

further graphs, are visible in Figure 20. The Figures 21 and 22 show the *additional* power consumption for a completely white picture and for the 50% white and 50% black image.

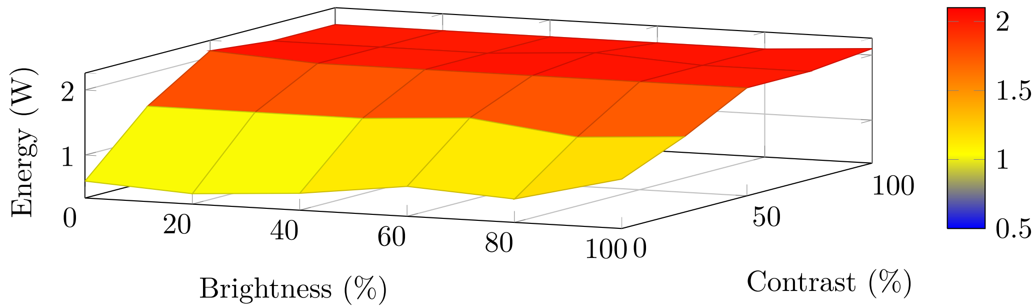

Figure 21: Additional display energy for a completely white picture

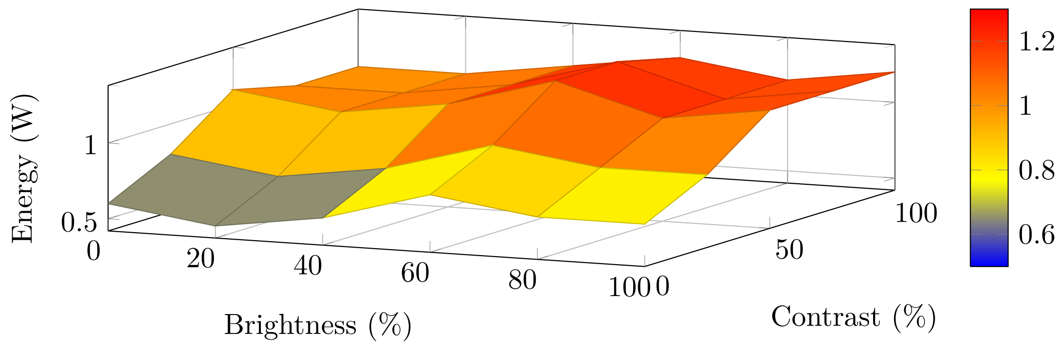

Figure 22: Additional display energy for white and partially white picture

The graphs show that the base energy consumption for a black picture is only dependent on the brightness of the display. The contrast setting has no significant influence on energy consumption. The consumed energy scales linearly with the display brightness. This changes once the pixels get color. For a 50% white picture the difference between 0% contrast and 100% contrast results in a 0.4W change in energy consumption. For a completely white picture the influence on energy consumption is even more significant, with a difference of 1 to 1.5 Watts between the highest and lowest contrast settings. The contrast only starts influencing the energy consumption noticeably when it falls below 50%, independent of the picture contents.

3.2 Application Utility - The Video Player

Besides the need for device profiles, which has been shown in the previous chapter, an application profile is needed as well. It must be possible to determine the amount of resource usage of an application to determine the current energy level it requires by use of the device's Energy/Utility characteristics. I want to show that it is possible to obtain such a profile using my framework using a real world example.

One real world example that was available for testing was the video player ffmpeg [ffm]. I instrumented this player in a way such that the energy consumption and the cache hit/miss data is sampled for each frame. This includes only the decoding time but not the time to wait until the frame should be displayed. The result of the cache analysis of the first 500 frames of the video Big Buck Bunny [bbb] can be seen in the top graph of Figure 23.

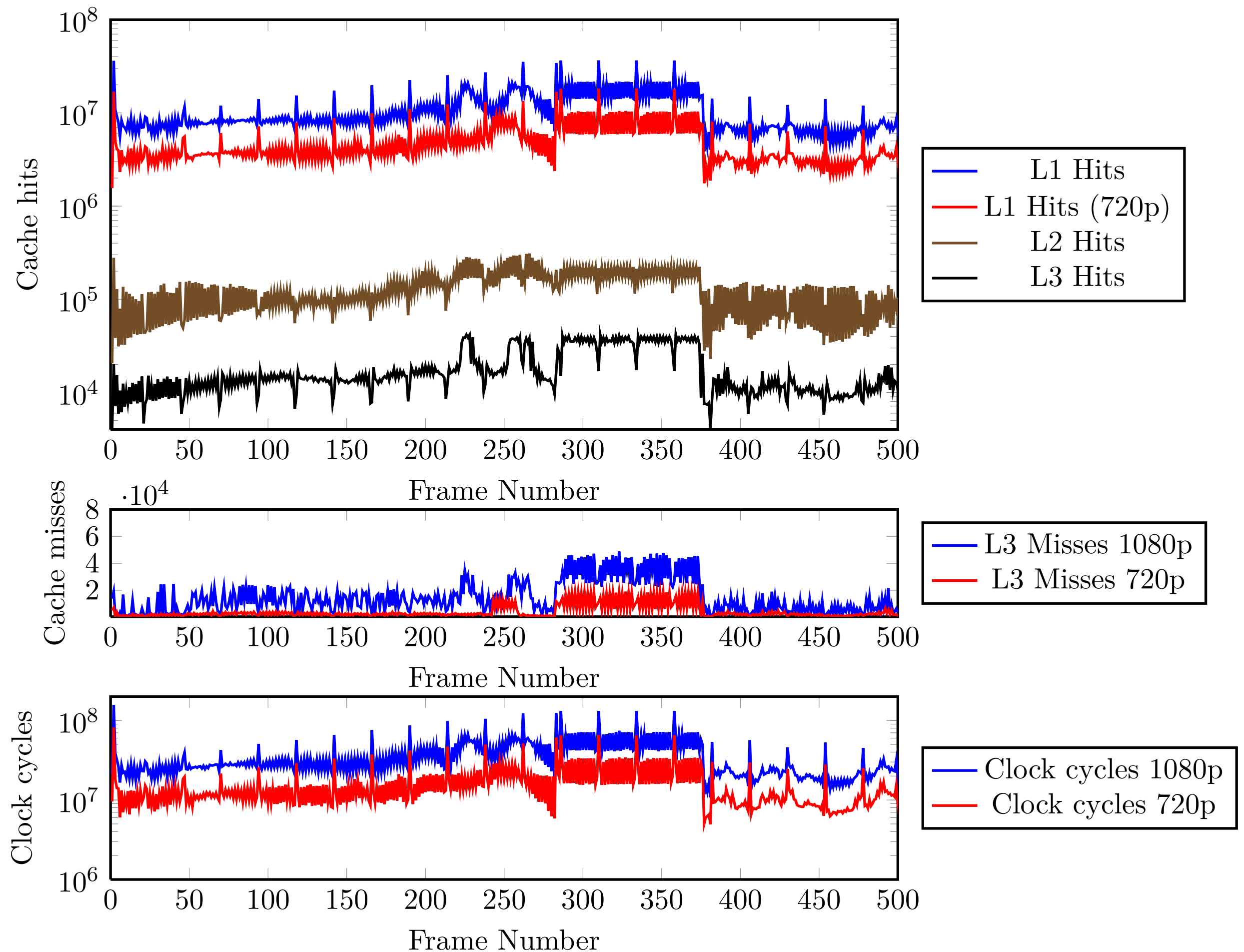

Figure 23: Cache profile of Big Buck Bunny on 1080p

I also tried measuring smaller units than frames such as slices or macro blocks, but as frames, in a fluently playing 25fps video, already only use a maximum of 40ms decoding time (and often a lot less), these measurement of sections well below the 1 millisecond mark yielded no usable results. The player was still instrumented by updating a patch, supplied by Michael Roitzsch, to the current version of ffmpeg. The video profile shown here is of the 1080p and 720p versions of the movie, because these had the longest processing times for the individual frames, and because of this the most precise results. While the 480p quality version took only about 2-6ms per frame to decode, in the 1080p decoding of a single frame took between 5ms to 30ms. Any load that is shorter than

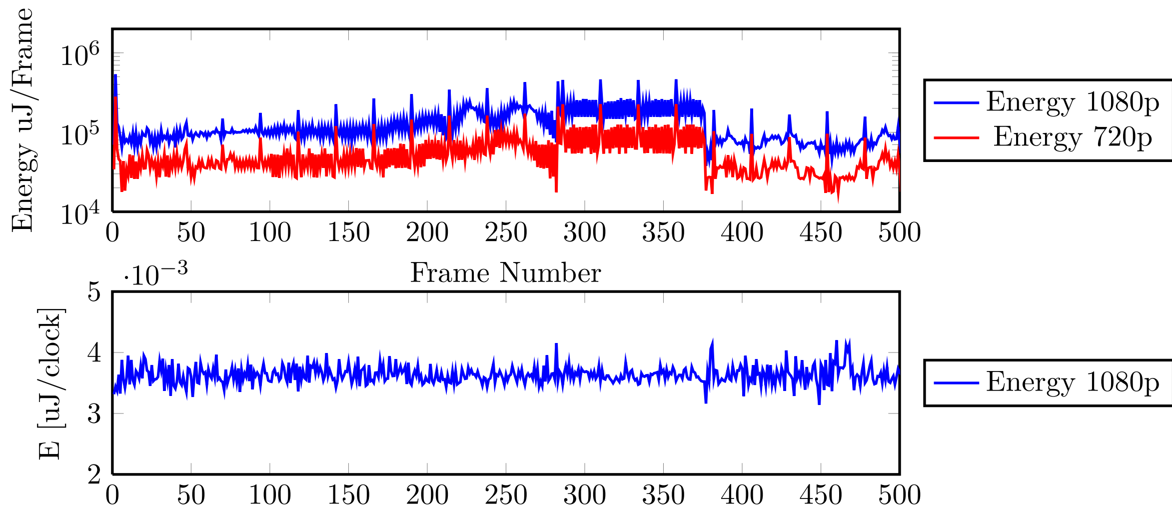

Figure 24: Energy profile of Big Buck Bunny on 1080p

2ms can currently not be sampled accurately using the energy counters, leading to the skewed results as demonstrated in the section on improving the energy counter precision. Graph 23 shows the number of clock cycles, the cache characteristics of the different cache levels, and the energy consumed.

I chose to measure the cache hit and miss ratios of the various cache levels, as they are amongst the most indicative performance counters when measuring energy consumption [SPH07]. They are closely correlated to the energy consumed per frame, as can be seen from the graphs in Figures 24 and 23. The obvious effect that shorter execution times are the dominant influence for lower energy consumption is also true for these videos. The energy per clock cycle that can be seen at the bottom graph, is not varying significantly over the frames and is largely independent of the video quality. The 720p version is not drawn in the graph, because it is virtually equal to the 1080p version. As can be seen by the high amount of cache hits, especially on the L1 cache this effect is inherent to the example. It executes many operations on small amounts of data, only rarely accessing the memory, which could stall the CPU and may lead to larger variations in energy consumption per clock cycle. It is also not using the L3 cache significantly. As a large part of the transistors in a modern processor are in the L3 cache a more significant usage would increase the switched capacitance. Other applications that process large amounts of data and only perform a limited amount of operations on each of them. Databases may provide such a usage scenario [tki12].

4 Preliminary Usage Scenarios - Future Work

In the previous chapters I introduced our approach to modeling energy characteristics of a system. I further showed that the we can use the relation between the service a component provides, and the resources it requires to do so, to model the Energy/Utility Function of the component. This depends on the availability of energy profiles for the lower levels of a platform, which I have shown to be obtainable by various measurement methods. The Energy/Utility Functions that are available for the different levels of the system can be used for different purposes.

4.1 Scheduling

The first possibility is a usage of the energy model for scheduling. With knowledge about the cost of placing an application on different hardware it is possible to determine the best placement from an energy point of view. The model further enables us to not only tell the energy impact a placement has but also its influence on the quality level of the service it provides - its utility.

Energy/Utility Functions do not only allow to model services and components themselves but also actions in the system. For example the decision to migrate a job to another system and the interconnect that is used to perform this migration can be weighted and optimized using EUFs.

We envision two basic modes for Energy/Utility based scheduling as well a combination of the two.

4.1.1 Energy Optimized

The first mode lets the user of a system or a service set an upper limit to the services power consumption [FS99]. This might be a peak power limit. This scenario is especially important if, for example, the power supply can only provide a limited amount of power that is not enough to drive all components at full power. For data centers it might be the limit of the ingoing high voltage power lines. Another possibility is to set the maximum average power consumption in a time window. While peak power might exceed the envelope, the energy scheduler should not let the system exceed the user set average power consumption. One system type where this might be important is mobile systems, where a guaranteed operation time of a device might be of value.

Our modeling approach allows the system to always scale to the highest possible quality while staying in the limits given by the user. If a possibility to look ahead, and

estimate future power consumption, exists, as does for video [Roi07], power envelope based scheduling is made easier and the minimization of switching is better enforced.

Especially switching overheads and cost, and the way to model them using our approach are subject of future work.

4.1.2 Quality Assuring

Another mode to let the operating system schedule system resources modeled using EUFs is to let the user set a minimum quality. We call this approach quality assuring energy scheduling. The goal is to give the user a guaranteed, user-defined, minimum quality and scale the systems energy consumption to be as low as possible while still granting the user the desired quality. We also envision an approach that lets the user set a preferred and a minimum quality level at the same time, which enables further sophisticated scheduling techniques.

4.1.3 Combination

A further possible scheduling technique is to let the user set a minimum quality level and a maximum power envelope and let him set a preference. On a mobile system the preference might be to enforce the power envelope so that a computation can finish, a video can be watched to the end, or the battery lasts for the whole workday. Whereas on another type of system the minimum quality might be more important.

4.2 Informed User Decisions

One big advantage of a system based on Energy/Utility Functions that we see, is the possible to tell the user the impact of his alterations to the system. We can tell the user in advance what the cost of reducing the energy consumption would be in terms of quality. And vice versa we can tell him the gains for increasing the allowed energy consumption. The same holds true for alterations to the minimum utility. The user can make an informed decision whether the increased energy cost of an increase in utility is worth it to him.

There are other advantages as well. The concept of Energy/Utility functions allows for dynamic recalculation of the system characteristics for changed hardware. The user may get information on the benefit another stick of RAM or a faster CPU will bring him. Conversely he can be told the impact of powering down a core on the system.

4.3 Energy Accounting

It is further made easy by this model to incorporate a model of currentcy [ZELV03] proposed by Zeng and colleagues. The model allows not only to use models like that of ECOSystem [ZELV02] but we envision that it could also be used to give the user knowledge of the benefit an increased budget of currentcy might bring to the application.

4.4 Global Energy Modeling

The Energy/Utility Model that was introduced in this thesis especially provides good scalability across system layers. It is usable for individual circuitry in the hardware up to a whole system and can even be applied for whole data centers. For hardware we envision scheduling individual parts of a core as proclaimed in the dark silicon paper [EBSA+11]. In a datacenter each server provides a service, and the network bandwidth is determined by the interconnects. All these components can be modeled using Energy/Utility and a service offered by a data center can require a certain utility from them. Our model can extend to the cooling, requiring lower cooling for throttled servers and enables data center operator to precisely model their system. A service might require the ability to serve thousands of clients at the same time, and the resources it requires are individual servers, load balancers, the network connection and the cooling.

This modeling approach might even in the long run be able to give a data center or HPC cluster operator information on the impact to the service and the total energy consumption that the removal or addition of servers or faster interconnects has. Especially for HPC centers the possibility to limit the maximum power and getting information on the performance impact of such a throttling will be worthwhile. It could tell operators the impact of the power limits and enables an informed decision when extending the cluster's power lines will be appropriate and be offset by the resulting increase in performance - and thus revenue.

Conclusion

In this thesis I at first implemented a measurement framework that makes profiling applications and hardware components easier. It provides methods to gather statistical and performance data from performance counters found in modern processors. This work was then extended to energy counters present in the latest generation hardware that make energy measurements and profiling easy. It enables the generation of energy profiles for applications and the CPU. The HAECER measurement framework incorporated the FERRET library that enables real-time capable communication and storage of measurement results. The aim was to enable low overhead measurements in userland and this goal was met, with measurement overhead in the negligible range of a few per mille for the profiles gathered for this thesis, due to the long runtime of the measurements.

I further evaluated the precision of the measurements by external instrumentation and discussed the difficulties in providing external measurements and correlating internal events. The problem of a correctly instrumented board, where the instrumentation does not influence the functionality or the measurement results has been discussed and the ZIH does currently work on providing such a board for future work. This will enable us to gather even more precise measurement for more devices in a system, as well as applications.

In the second part I introduced a new approach to energy modeling that aims to complement and enhance existing modeling methods. Our proposed Energy/Utility Functions enable modeling across all levels of a system and can be used to gather comprehensive information of a system. Contrary to existing approaches that try to model energy requirements of individual levels of a system or provide energy-aware scheduling methods for them, our approach encompasses all system layers and enables modeling of even the most complex systems. We require a mapping between the resources a services requires and the utility it provides based on the availability of these resources.

I showed that the required Resource/Utility Functions, which are the modeling of whole systems independently of the application mix and the underlying hardware, are obtainable using the profiling framework introduced in the first part. I have further presented two example profiles of system components, one for the CPU and one for the display. Additionally I showed for the example of high definition video that it is possible to determine the resource requirements of an application, for the example of its CPU and cache requirements. I showed that amount of a resource that is required by the application to provide its services also correlates with the energy consumption of the application.

Beside a formalization of the functions, and the introduction to the basic approach to get the current hardware and application mix specific Energy/Utility Function of a resource, I further discussed different approaches to scheduling that are enabled by

our model. Those approaches are again valid for all layers of a system and give the user detailed information on the influence of his choices. Our solution provides the first energy modeling approach that is independent of concrete hardware and special software and adaptable to wide range of situations, hardware and applications. It is especially well suited for adaptable applications.

As energy optimization becomes more and more important, with rising costs and increasing difficulties to supply enough power to small chips, we are convinced that approaches such as ours can enable future technology to be energy aware and at the same time provide high performance. This will make energy a true first-class resource of a system.

Glossary

CPU Central Processing Unit / Processor

DVFS Digital Voltage and Frequency Scaling

DVI Digital Video Interface

EUF Energy/Utility Functions

HAEC Highly Adaptive Energy-efficient Computing

HAECER HAEC Energy Reader

HPC High Performance Computing

ISA Instruction Set Architecture

LLC Last Level Cache

OSD OnScreen Display

POSIX Portal Operating Systems Interface

RAPL Running Average Power Limit

RUF Resource/Utility Functions

RDPMC Read Performance Counter instruction

SMM System Management Mode

TSC Time Stamp Counter

TUF Time/Utility Functions

VGA Video Graphics Array

ZIH Center for High Performance Computing

List of Listings

1	Methods to set up performance monitoring counters	14
2	Operations on performance counters	14
3	Pseudocode for synchronization to SMM	17
4	HAECER functions to support energy monitoring	22
5	Functions for precise short time measurements	26
6	Cache benchmark	52

List of Tables

1	Available counters in various CPUs	8
2	Fixed-function counters in current Intel architectures	9
3	Architectural counters in current Intel architectures	10
4	Overhead of calling the RDTSCP instruction	15
5	Measurable power planes available in Intel processors	21
6	Energy and execution statistics for individual instructions	31

List of Figures

1	RDTSCP overhead over time	16
2	Performance counter reading overhead	18
3	Analysis of energy counter update behavior (update distance)	23
4	Analysis of energy counter update behavior (1ms mark)	24
5	Sampling a short time event at 1ms intervals	25
6	Counter synchronized measurements	26
7	Energy counter resolution benchmark	28
8	Energy difference of between load and idle (10 Hz sampling freq.)	29
9	Energy difference of between load and idle (2 Hz sampling freq.)	30
10	External measurement of load idle case	33
11	Zoomed in view of load/idle cases for external measurement	34
12	High detail SMM view	35
13	A basic Energy/Utility Function	44
14	Hierarchy of resources and and services	45
15	Hierarchy of resources and and services for composable devices	46
16	Simple energy characteristic of a resource	47
17	L1 cache energy characteristics	53
18	L1 cache energy characteristics (raw)	54
19	LLC cache to memory energy characteristics	54
20	Display energy for a black picture	55
21	Additional display energy for a completely white picture	56
22	Additional display energy for white and partially white picture	56
23	Cache profile of Big Buck Bunny on 1080p	57
24	Energy profile of Big Buck Bunny on 1080p	58

Bibliography

[amd10] AMD finally outs the 32nm Llano core. http://semiaccurate.com/2010/02/10/amd-finally-outs-32nm-llano-core/, February 2010. 5

[AMD11] AMD, http://support.amd.com/us/Processor_TechDocs/41131.pdf. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 12h Processors*, June 2011. 8, 21

[AMD12] AMD, http://support.amd.com/us/Processor_TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf. *BIOS and Kernel Developer Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*, February 2012. 8

[bbb] Big buck bunny. http://www.bigbuckbunny.org/. 57

[Bel00] Frank Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *In Proceedings of the 9th ACM SIGOPS European Workshop*, 2000. 1, 41

[BGN+10] Josep Ll. Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 215–224, New York, NY, USA, 2010. ACM. 1

[BSD08] D0nAnd0n BSDaemon, coideloko. System management mode hacks. http://www.phrack.org/issues.html?issue=65&id=7, March 2008. 17

[ccFW09] core collapse (Filip Wecherowski). A real SMM rootkit. http://www.phrack.org/issues.html?issue=66&id=11, June 2009. 17

[Con05] Gilberto Contreras. Power prediction for Intel XScale processors using performance monitoring unit events. In *In Proceedings of the International symposium on Low power electronics and design (ISLPED)*, pages 221–226. ACM Press, 2005. 5

[dcc] Datacenter energy costs outpacing hardware prices. http://arstechnica.com/business/news/2009/10/datacenter-energy-costs-outpacing-hardware-prices.ars. 1

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer*

architecture, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM. 1, 61

[ffm] Ffmpeg homepage. http://ffmpeg.org/http://ffmpeg.org/. 57

[fia] The fiasco microkernel. http://os.inf.tu-dresden.de/fiasco/. 2, 7

[FS99] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 48–63, New York, NY, USA, 1999. ACM. 59

[HGSW10] Daniel Halperin, Ben Greenstein, Anmol Sheth, and David Wetherall. Demystifying 802.11n power consumption. In *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association. 44

[IM03] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society. 5

[int10] Intel Performance Counter Monitor - a better way to measure CPU utilization. http://software.intel.com/en-us/articles/intel-performance-counter-monitor/, 2010. 5

[int11a] Intel power gadget 2.0. http://software.intel.com/en-us/articles/intel-power-gadget/, December 2011. 5

[Int11b] IntelPR. Intel brings 'eye candy' to masses with newest laptop, PC Chips. http://newsroom.intel.com/community/intel_newsroom/blog/2011/01/05/intel-brings-eye-candy-to-masses-with-newest-laptop-pc-chips?cid=rss-258152-c1-263334, Jan 2011. 21

[Int11c] Intel®, http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-3a-3b-system-programming-manual.pdf. *Intel®64 and IA-32 Architectures Software Developers Manual, Volume 3A & 3B*, May 2011. 8, 9, 10, 12, 21

[Jen92] E. D. Jensen. Asynchronous decentralized real-time computer systems. In W. A. Halang and A. D. Stoyenko, editors, *Real-Time Computing*, the NATO Advanced Study Institute. Springer Verlag, October 1992. 2

[JLT85] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *IEEE Real-Time Systems Symposium*, pages 112–122, Dec. 1985. 2, 43

[KF09] Viren Kumar and Alexandra Fedorova. Towards better performance per watt in virtual environments on asymmetric single-ISA multi-core systems. *Operating Systems Review*, 43(3):105–109, 2009. 1

[l4r] L4Re – the L4 runtime environment. `http://os.inf.tu-dresden.de/L4Re/`. 2, 7

[LCD] NEC MultiSync LCD1960NXi Manual. `www.comiga.de/PDF/LCD1960NXi.pdf`. 55

[LEMC01] Sheayun Lee, Andreas Ermedahl, Sang Lyul Min, and Naehyuck Chang. An accurate instruction-level energy consumption model for embedded RISC processors. *SIGPLAN Not.*, 36(8):1–10, August 2001. 5

[MAC+11] John McCullough, Yuvraj Agarwal, Jaideep Chandrashekhar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh Gupta. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the USENIX Annual Technical Conference*, Portland, OR, June 2011. 1

[Mat99] Terje Mathisen. Pentium secrets. `http://www.gamedev.net/reference/articles/article213.asp`, Oct 1999. 7

[MSB10] 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. 49

[Net04] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, November 2004. 8

[PDLD06] Martin Pohlack, Björn Döbel, Adam Lackorzynski, and Technische Universität Dresden. Towards runtime monitoring in real-time systems. In *In Proceedings of the Eighth Real-Time Linux Workshop*, 2006. 5, 19

[per] perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/`. 5

[Poh10] Martin Pohlack. *Runtime Monitoring for Open Real-Time Systems*. PhD thesis, Technischen Universität Dresden, `http://os.inf.tu-dresden.de/papers_ps/pohlack-phd.pdf`, Juli 2010. 5, 19

[RAK+11] Rance Rodrigues, Arunachalam Annamalai, Israel Koren, Sandip Kundu, and Omer Khan. Performance per watt benefits of dynamic core morphing in asymmetric multicores. In *PACT*, pages 121–130, 2011. 1

[rap11] introduce intel_rapl driver. `http://lwn.net/Articles/444887/`, May 2011. 5

[RJL05a] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *IEEE ISORC*, pages 55 – 60, May 2005. 2, 43

[RJL05b] Binoy Ravindran, E. Douglas Jensen, and Peng Li. On recent advances in time/utility function real-time scheduling and resource management. In *ISORC*, pages 55–60, 2005. 41

[RNA+12] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power management architecture of the second generation intel core micro architecture - sandy bridge. *IEEE Micro*, 99(PrePrints), 2012. 2, 5

[Roi07] Michael Roitzsch. Slice-balancing h.264 video encoding for improved scalability of multicore decoding. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 269–278, New York, NY, USA, 2007. ACM. 60

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *In SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15. ACM, 2008. 1

[Sha49] C. E. Shannon. Communication in the presence of noise. In *Proceedings of the Institute of Radio Engineers (IRE)*, volume 37, pages 10–21, 1949. 24

[SLSPH09] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: A platform for OS-level power management. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, Apr 2009. 41

[Sno10] David C. Snowdon. *OS-Level Power Management.* PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Mar 2010. Available from publications page at `http://www.ertos.nicta.com.au/`. 1, 41

[SPH07] David C. Snowdon, Stefan M. Petters, and Gernot Heiser. Accurate on-line prediction of processor and memoryenergy usage under voltage scaling. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 84–93, New York, NY, USA, 2007. ACM. 5, 41, 58

[tki12] Thomas Kissinger. Personal Conversations, 2012. 58

[Vol] Energy-check 3000. `http://www.produktinfo.conrad.com/datenblaetter/125000-149999/125330-an-01-ml-ENERGY_CHECK_3000_de_en.pdf`. 55

[Wei] Johannes Weiß. Multi-core energy accounting. 5

[WRJL06] Haisang Wu, Binoy Ravindran, E. Douglas Jensen, and Peng Li. Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems. *ACM Trans. Embed. Comput. Syst.*, 5(3):513–542, August 2006. 41

[xeo10] *Intel Xeon Processor 7500 Series Uncore Programming Guide*. `http://www.intel.com/Assets/PDF/designguide/323535.pdf`, Marchs 2010. 9

[ZELV02] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. ECOSystem: managing energy as a first class operating system resource. *SIGOPS Oper. Syst. Rev.*, 36(5):123–132, October 2002. 60

[ZELV03] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: a unifying abstraction for expressing energy management policies. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC '03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association. 41, 60