Workload Characterization for Predicting Energy Efficiency of Servers

Philip P. Moltmann

System Architecture Group (IBDS Bellosa)
Department of Computer Science
Universität Karlsruhe (TH)

DIPLOMA THESIS
DIPLOMARBEIT

Advisors:	Prof. Dr.-Ing. Frank Bellosa
	Dipl.-Inform. Andreas Merkel
Begin:	March 30, 2007
Submission:	September 28, 2007

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

Karlsruhe, September 28, 2007. Philip P. Moltmann

Abstract

The energy consumption of servers has become a major factor in their total costs of ownership. In 2005 the average server in the US consumed energy for US$ 264. However, there is has been few research on predicting the energy efficiency of servers.

I argue that energy efficiency cannot be expressed as a single metric because a server's energy consumption is mostly determined by the workload. In this thesis I describe a system predicting the power consumption from an abstract description (profile) of a previously recorded workload. With this profile a company specifies the real workload of one of their servers. By replaying it on several machines it can be measured which machine is the most energy efficient one for the company's workload.

If this process is applied when a new server is bought, it leads to a much more energy efficient infrastructure. Consequently my system can be used to reduce the company's energy costs and save the environment.

I created an exemplarily implementation of this system for Linux 2.6 and the Pentium 4, but it can easily be ported to other platforms. It showed a prediction inaccuracy of < 4 % for test cases in my scope, making it a proper tool for it's purpose. For intentionally unsuitable test cases the prediction error was 12 %.

Contents

1 Introduction

In February 2007 Jonathan G. Koomey published a widely recognized study about the total worldwide energy consumption of servers [Koo07]. He stated that the total amount of energy used grew from 3.3 gigawatts in 2000 to seven gigawatts in 2005. This is equivalent to seven average sized nuclear power plants.

1.1 Energy usage as a problem

The heavy energy consumption of servers is problematic in several different ways:

- Computer components consume direct current. Usually a server uses a power supply unit (PSU) to transform the alternating current supplied by the power socket into the current needed by the components. Most PSUs built into servers have a loss of about 10 % to 40 %. The PSU's maximum efficiency is usually reached at about 90% load. Under non-lab conditions load varies heavily. Redundant power supplies have a maximum load of 50 %. This means that with heavy, unpredictable energy consumption comes a serious amount of wasted energy by the PSU.

- Another task, that has to be addressed, is that servers are usually packed very densly into 19" racks. This increases the power density to a hardly manageable level. Usually datacenters feature big air conditioning systems using about the same energy used by the servers directly to keep the air temperature at an adequate level [GMT$^+$06]. This increases the energy usage of servers including energy needed for cooling by 100 % to 14 GWs or 14 nuclear power plants [Koo07].

- The running energy costs for all servers worldwide summed up to 7.2 billion US$ per year in 2005 or US$ 264 per machine $\cdot$ year [Koo07]. Note that this values reflect the state of 2005 when many of the servers where based on relatively frugal CPUs as the Pentium III. If the lifetime of a server is about five years the energy costs are a mayor part in the total ownership costs. Due to continuesly rising energy costs this will become an even more serious issue in the future.

- Last but not least, most of the energy is currently produced by nuclear power plants or by burning fossil fuels. Both kinds have a negative effect on the environment and therefore need to be reduced as far as possible.

Big datacenter-operators like Google or server hosters can test new servers whether they are energy efficient for their workload or not, because they usually buy

servers at big scale. But small and medium sized companies buying only one or two servers at a time usually have no ability to test their servers ex ante. They have to rely on measurements supplied by the manufacturers usually – if available – giving only a very rough hint about the real energy usage at the company's workload. Therefore energy consumption, opposed to its importance, is still seldom regarded when acquiring new hardware.

This leads to the need for a simple and industry-standard way to help small and medium companies to estimate energy costs for their new-to-buy servers.

1.2 The challenge of defining energy efficiency

Usually energy efficiency is defined as:

$$energy\ efficiency := \frac{amount\ of\ work\ done}{energy\ used} \tag{1}$$

or catchier:

$$energy\ efficiency := \frac{performance}{watt}$$

This definition is evident but not sufficient to solve the problem because it does not define a protocol how and what to measure. Nontheless it is useful to compare the yields of different energy saving features or to compare very similar systems with exactly the same workload. Therefore, the main focus when extending the definition of Equation 1 should be on defining such a protocol.

Important to be considered when measuring energy is that the load usually varies over time. Most servers are most efficient at near maximum load and are very inefficient near zero load because they still draw much power while they are idle. For example a HP ProLiant DL380G4 draws 134 W while idle even in the most energy saving configuration [Hew07].

Additionally a workload may have heavy I/O and another one may be only using the CPU. This means that the type of load generated has to be taken into account. Small servers are usually more efficient (according to Equation 1) for low load workloads and bigger multi-socket servers are more efficient for high load workloads. Consequently one cannot calculate power usage for a workload from a single energy efficiency metric.

CPU performace still rises exponentially [Sch97]. As I/O performance rises slower a benchmark, that is now CPU bound, may become I/O bound in the furture. With more processing power new features are introduced into existing software. For example e-mail servers are now equipped with spamfilter modules. This completely changes the characteristics of the software from a relatively simple I/O bound send-and-receive workload to a heavily CPU bound parse-and-filter load.

Thus it is nearly impossible to define a set of standard workloads that covers all types of current and future workloads.

Therefore approaches based solely on benchmarking [SPE07, Ene07] or classifying servers by performance metrics (e.g. CPU speed) [Ene07] need to be adapted regularly to new hardware and software.

1.3 Thesis overview

In this thesis I will describe how to design (Chapter 4) and implement (Chapter 5) a protocol and all needed tools targeted to provide assistance to small and medium sized companies when acquiring new energy efficient servers. This protocol measures the company's workload by logging relevant hardware and software counters. These logfiles (or profiles) will be used to replay the workload on another system and measure it's power consumption. I will describe how this approach can be extended to give customers a hint about power consumption of servers for their workload without workload replaying or power measurement (Chapter 7).

My approach is new as it does not define any standard-benchmark to measure power consumption but tries to resemble the real workload. This results in more accurate predictions. The main challenge is to resemble the workload in such way that the server consumes the same power as it would have running the real workload.

I will provide an overview about current research in the areas of benchmarking, workload characterization and power benchmarking if relevant to this thesis in Chapter 2. I will evaluate the accuracy and applicability of my approach in Chapter 6. I will conclude in Chapter 8.

A priori needed knowledge that might be unfamiliar to some readers is briefly summarized in Chapter 3.

As I experienced problems with inaccurate defined metrics while working on this thesis I will give an in depth definition of all metrics used by me in Appendix A.

2 Related work

This work is based on three interrelated fields of research.

First, my thesis is based on benchmarking as it measures power consumption while playing a workload. There has been much research on benchmarking processing performance and power consumption of individual components. I will give a brief overview on energy benchmarking in this chapter.

Secondly, it is also based on workload characterization. Workload characterization is a subdivision of benchmarking. The focus is on describing a complex workload with as few values as possible. My system is based on workload profiling, modeling, and replaying. Therefore it uses workload characterization as described in Chapter 4.

Third, it is an energy benchmark. Compared to energy benchmarking of single components, energy benchmarking of full systems is a relatively new field of research with few research until now. The available approaches have a different focus than mine. They are similar as they try to solve the same problem: defining whether a server is energy efficient or not.

2.1 Benchmarking

There are two kinds of benchmarks. Microbenchmarks measure the performance of a single function of a software or hardware. For example measuring the speed of context-switches would be a microbenchmark for operating systems. As a software or hardware product usually features a lot of functions these benchmarks do not describe the overall quality of the whole product. Anyway, these benchmark are suitable tools to find bottlenecks or while optimizing.

Macrobenchmarks try to evaluate whole products or big parts of them. They are based on generating a certain workload and measuring it's performance. In this thesis I will focus on this kind of benchmarking.

A benchmark has to fulfill three goals: It has to be repeatable, comparable and relevant [Mog99]. Relevant means that the resulting metric of the benchmark gives a good hint on the real performance of the measured system. For a macro benchmark this is the most complicated of the three conditions. To solve this issue properly, an energy benchmark has to give a good assumption on the energy consumption of real programs.

Metrics Making a macro benchmarks relevant means choosing a load that represents the real workload. Steven L. Gaede described three ways to find a relevant workload each guided by another type of metric [Gae81].

A corse grained view on a workload would be to monitor the input and output of a server, e.g., e-mails received and sent. This is shown in Figure 1 as requests.

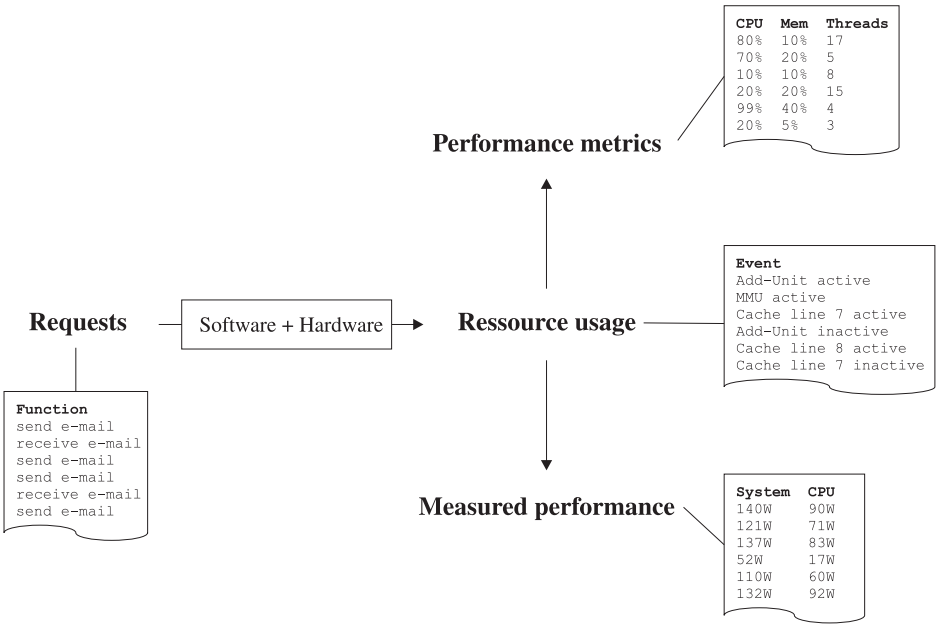

Figure 1: Metrics as used by Gaede [Gae81]

Gaede calls this *functional approach*. The assumption is that a workload is representative if it performs the same functions as the original workload. The functions of an e-mail server are sending and receiving e-mails. The functions of a database server are the transactions. Gaede states that the main drawback of this approach is that the problem would be transformed from researching the representativeness of programs and commands executed to researching the representativeness of functions performed [Gae81]. In the case of an e-mail server the problem would be to find representative e-mails to be processed.

The other extreme is to focus on the resources consumed by a workload. In the case of an e-mail server this would be the usage of the units of the processor, memory, and other components of the server. Figure 1 calls this resource usage. This approach - *resource consumption approach* - assumes that a workload is representative if it consumes the same resources at the same rates as the real workload [Gae81]. The disadvantage in this case is that it has to be specified which resources to monitor and that their behavior is very machine dependent.

Gaede favors the *performance oriented approach*. It assumes that a workload is representative if it generates the same performance characteristics as the original one [Gae81]. The performance metrics have to been chosen to reflect the important metrics of the software benchmarked. Gaede gives an example: In case of compar-

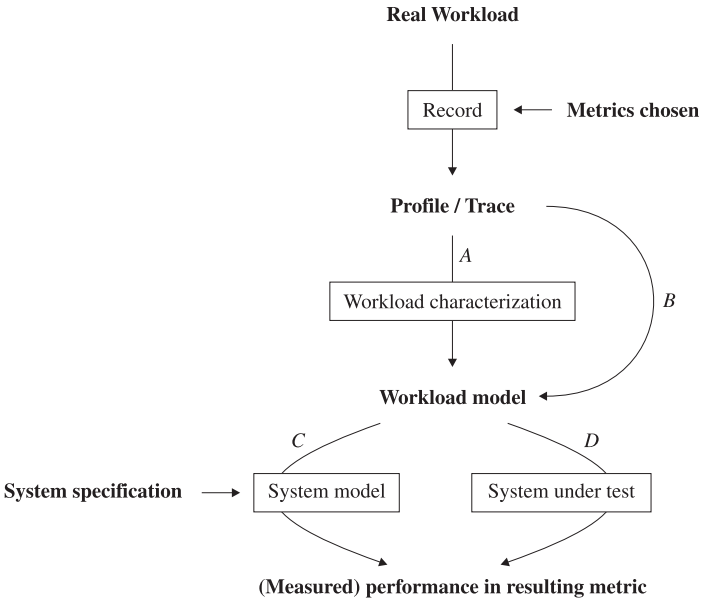

Figure 2: Benchmarking is based on a system model or on testing. The benchmarks creator can either choose path *A* or *B* and either path *C* or *D*. Choosing path B surpasses workload characterization by assuming that one trace is relevant. Path *C* is only chooseable if an accurate system specification is available.

ing disk scheduling algorithms, performance metrics would be queue length and access time.

The three ways proposed by Gaede can be seen as different levels of abstraction. The resource consuming approach is the most detailed view on the workload and functional one is the most abstract one. This means, selecting the right metrics is in fact choosing the level of abstraction. When benchmarking more than one component at a time, the best levels of abstraction for them can vary.

2.2 Workload characterization

Workload characterization is a part of benchmarking focusing on the minimization of data needed to describe the workload (see Figure 2 path *A*). Calzarossa et al. and Elnaffar et al. wrote reviews about the state of the art in 1993 and 2002 [CS93, EM02]. They partitioned the techniques developed into either static or dynamic.

Static techniques A workload characterization technique is called static if it describes more a characteristic of a workload than a behavior. Often datamining technologies (clustering, principal component analysis, etc...) are used. For

example a histogram describes the devision of measurements taken but not when they were taken and if there are dependencies among them. The advantage of this kind of characterization is that it produces very small and easily comprehensible results.

Dynamic techniques Dynamic techniques are used if there are dependencies in the behavior of a metric or if one measurement influences another. In a SMTP-server an incoming e-mail is likely to cause an outgoing mail. Mostly technologies developed for artificial intelligence research as neuronal nets or Markov models are used to describe those dependencies. Using AI technology is understandable because dynamic workload behavior is usually caused directly or indirectly by humans. Even if dynamic techniques appear to be more powerful they have the disadvantage of producing usually complex, hard to understand and sometimes indeterministic models.

Again there is no best solution for all cases. When designing a new benchmark the creator has to choose from the available techniques the one that fits best to the workload and the resulting metric.

2.3 Energy estimation

Energy can be estimated either by creating a model of the component and replaying the workload on this model or by benchmarking.

2.3.1 Power modeling for single components

There are power models for about every kind of component. These models are build upon a specification of the component (see Figure 2 path *C*).

CPU The most prominent project for CPU power measurement is the Wattch project [BTM00]. It extends the simplescaler framework, a low level microprocessor simulation toolkit, by power measuring [BAB96]. Because of this it needs a low level description of the benchmarked CPU. There are also black box online power estimation solutions [BKW03], haveing an error of about 10 %.

Network Orion is a power simulator for network components [WZPM02]. Similar to Wattch, it simulates the microarchitectural behavior of the component and estimates power consumption from it.

Memory Mempower is a set of tools using memory traces to estimate the power consumption [Raw04]. Again this is an offline power simulator using low level hardware information as input.

Disk There is also a simulation based approach for disk power consumption [ZSG+03]. Like Mempower this approach uses traces to estimate energy resulting in the same drawbacks.

2.3.2 Energy Benchmarking of full systems

There are different ways to deal with energy consumption of full servers. Goal of all approaches is to provide hints to the buyer of a server to choose the most efficient one for his needs.

Energy Star Program Requirements for Computers In 2007 Energy Star published a scheme for desktop computers, workstations and small servers classifying each computer as either energy efficient or not [Ene07]. Energy Star is a cooperation of the US department of energy and the US environmental protection agency. They define standards to give customers of electronic components (as refrigerators, TVs or computers and serves) an idea, whether the product is energy efficient or not.

Servers that can be classified with this scheme have to be desktop-derived, i.e., they have to be regular desktop PCs used as servers. Whether a server is efficient or not is only dependant on their energy usage while in standby mode and while idle. Energy Star also describes standard environment conditions and power-meter requirements for their tests.

Comparable approaches As far as I know there are two approaches comparable to mine. SPEC Power is not a product of academic research but a tool by and for the IT-industry. It is important because it is very likely to be widely used. Zesti is the academic approach most closely comparable to mine. Even if the goal is different the methodics are similar.

SPECpower *This section reviews the information avaliable to the public. For members of the SPEC being eligible to access the SPEC Powers development wiki, an extended version of this section is available. Please contact me or the System Architecture Group of the Universität Karlsruhe (TH) to receive the extended chapter.*

SPECpower is a power benchmark of the Standard Performance Evaluation Corporation (SPEC). The difference to my approach is that they try to define

energy efficiency in spite trying to circumvent this – in my opinion unsolvable – problem. As described in Equation 1 any energy efficiency definition has to respect computational performace proportional to energy consumption.

> [..] SPEC wants to define server energy measurement standards in the same way we have done for performance. Development of this benchmark provides a means to measure energy use in conjunction with a performance metric. This should help IT managers in their acquisitions to consider power efficiency along with other selection criteria to increase the efficiency in data centers. [1]

A single power efficiency metric needs a predefined workload. Their scope is broader than mine in hardware components because they do not specify any hardware but they are very specific in workload because of the tight connection to Java.

> The committee develops a means to fairly and consistently report system energy use under various usage levels.
>
> The initial benchmark addresses one subset of server workloads; the performance of a server side Java. It exercises the CPUs, caches, memory hierarchy and the scalability of shared memory processors (SMPs) as well as the implementations of the JVM (Java Virtual Machine), JIT (Just-In-Time) compiler, garbage collection, threads and some aspects of the operating system. [1]

My approach requires no additional hardware or software at the company because this can annihilate the money-saving effect of my approach. The creators of SPECpower follow the same thoughts.

> The benchmark needs to run on a wide variety of OS and hardware architectures and should not require extensive client or storage infrastructure. Also the usability needs to be high, the setup relatively easy, and the run-time needs to be reasonable. [1]

An important advantage of the SPEC's benchmarks is that all stakeholders took part in the development process. This assures that it will be well accepted. It is likely to become an industry standard.

[1] From SPEC Power's homepage, `http://www.spec.org/specpower`, September 7th 2007

Zesti Zesti is a system that enables estimating the power consumption for a full server [DER06]. It can either be used online or to estimate power offline from a workload profile. Basically the system profiles a set of metrics m_i and uses a linear function $E(t)$ to estimate the power consumption for the measurement point. Each metric is multiplied by an energy weight a_i and the results are summed up with an additional energy consumption a_0 related to no metric.

$$E(t) = a_0 + \sum^{\forall i} a_i \cdot m_{i,t} \tag{2}$$

The authors of the paper recommend their system for two purposes: to find power bottlenecks of the system or as hint for the scheduling of the CPU and other ressources.

Zesti is interesting in the scope of this thesis because of two similarities.

1. Power estimation from high level metrics as done by Zesti is a way to conveniently circumvent the time-consuming replaying process (as described in Chapter 4). It determines the power consumption for each measuring point. Summing up these values gives a reasonable estimation of the power consumption of a workload. I described my ideas about a power estimation extension of my system in Section 7.1.

2. Zesti is the only system known to me that tries to estimate power for a full server from high level metrics. The main difference to my approach is that they limit their estimation to one machine. The calibration of the a_i and the choice of the metrics is done for a single machine. This makes them free of the invariance-constraint I have to obey (see Section 4.3). Therefore they can use more metrics as I can.

The resulting metric, i.e. power consumption, of Zesti is estimated by $E(t)$. I measure the power consumption. Hence it is hard to compare the quality of the approaches. Their system is targeted to engineers trying to optimize the system and mine is built for supporting the server-acquisition process.

3 Prerequisites

This chapter is a summary of all knowledge necessary to understand the following sections.

3.1 Terminology

In computer science terms operating systems have had a very long evolution. Due to this some terms used for important abstractions have become indefinite or inaccurate. In this thesis I use the following terms for the subsequent abstractions:

Thread A thread is a single flow of execution. It technically consists of a stack, a set of registers (loaded into the real registers or stored into main memory), and status information. It can be most closely compared to a lightweight process in Linux. A thread is created in Linux by the `clone()` system call.

Task A task is an address space with all threads running in it. It technically consists of a pagetable, at least one thread, and some status information. It is compareable to a threadgroup or (in case of only one thread) to a traditional "heavyweight" process in Linux terminology. Executing `fork()` in Linux creates a new task.

Working set The main memory recently used by a specific task is called working set. "Recently" is defined by the memory modeling in Section 4.5.2. Note that this definition might change with other memory modeling implementations.

3.2 Energy saving technology

To unterstand how to design a system that measures energy efficiency it is important to understand how energy is handled in current computers and what kinds of energy saving technologies are available. Therefore I'll give a short overview about the current state of art in energy handling and energy saving technology. I will neither provide much detail nor list all available technologies. The next section is intended to give a rough impression how energy saving usually works.

Energy handling technology can roughly be divided in two categories. Low level or hardware technologies cover abilities of the hardware to save energy and high level software technologies cover mechanisms and policies that use knowledge about the current runtime behavior in order to adapt the hardware to the current workload.

3.2.1 Hardware technologies

There are two kinds of hardware power saving features. The first kind is saving power independly of software and is called passive technology. The other one needs active support of software to save power. As more and more features are implemented in hardware this section reflects the state of the art in August 2007 and is likely to become incomplete soon.

Passive power saving technology Most of the components in recent servers try to save as much power as they can without user or operating system interaction.

CPU There has been much effort to reduce the power consumption of CPUs because they still draw most power in the majority of servers. Smaller structure sizes result in less power consumption. Usually CPUs with smaller structures are more advanced meaning more transistors / units and higher frequencies rising the power in turn. In the last years leakage power has become a major problem so the manufacturers implemented on-chip mechanisms to automatically power off parts of the CPU when they are not used. For example Intel has developed a policy that can adaptively power off single cache lines [AGVO05].

One can see attempts to use the CPU more effectively as a kind of power saving technology. Hyper Threading or dynamic cache sharing leads to a higher performance and therefore to a better energy efficiency according to the definition in Equation 1.

PSU Power supply units are still contributing a lot to the total power consumption of a server. The PSU consist of AC to DC transformers and onboard DC to DC converters used by the components (Disc, RAM, CPU, Fans). Current converters operate most efficient at near maximum load. This means that there are two possibilities to save energy in PSUs.

The first one is to use better, more efficient (and more expensive) PSUs. Modern PSUs as recommended by the 80 PLUS program and the Energy Star requirements for Desktops [Ene07] reach up to 90 % efficiency at 20 % - 100 % load.

Secondly, choosing correctly sized PSUs would also save energy as it rises the load percentage.

Active power saving technology Many hardware components support several run modes, i.e., several performance levels with different energy usage.

ACPI The *Advanced Configuration and Power Interface* is an interface standard developed by major IT-companies to enable easy configuring and power

management of hardware ressources [Hew06]. ACPI provides a table containing all hardware components in a system and the available performance/energy states per component. This table also contains code fragments that enable the operating system to switch between the states.

CPU Much research has been done in creating feasible active power saving features for the CPU [BM01]. Current CPUs support four of these features. They enable trading in execution performance for energy consumption.

1. The operating system can execute the `HLT`-instruction to set the CPU in a low power mode. If it is a multicore CPU the power mode depend on whether all cores are executing `HLT` or if one is still running. Since executing the `HLT`-instruction means that no work is done, it one makes sense to execute it, if there is no work left.

2. Clock modulation – initially introduced as safety technology against termal emergencies – temporarily stops the CPU's clock signal. It is more efficient to use `HLT` + fullspeed than using clock gating [Mol06]. Therefore clock gating should only be used if no other power saving feature is available.

3. Instruction decode throttling can be used to limit the amount of instructions decoded. This results in a lower utilization of the pipeline.

4. Frequency scaling reduces the CPU's frequency. While running at a lower frequency the voltage can be lowered. As CPU power can be computed as $P = x + y \cdot \textit{frequency} \cdot \textit{voltage}^2$ this is a very efficient power saving technology. The drawback of this feature is that it takes some time to switch between the states, e.g., Intel's Enhanced Speedstep Technology switches the state in 10 microseconds [GRA+03].

Memory RDRAM memory chips support low-power states when they are not accessed. The power can be reduced to between 10 % and 1 % of the power consumed while active. Switching back to active mode is time consuming [HPS03].

Disk It is usually not feasible to completely shutdown the disk in servers, because of the long spinup times. To solve this problem multiple rotation speed disks have been developed [GSKF03]. Lower rotation speeds can currently only be used when the disk is idle; for serving requests the disk has to be spun up to full speed.

Wired network Most energy used by network adapters is used to drive the long CAT5/7 cables. The cable has to be constantly driven so there are currently no low power modes available.

Consolidation Consolidation of disks, services or complete servers is a yielding way to save power. The reason is that the load is concentrated and computer components are generally more energy efficient at higher load. Servers can be consolidated by using virtualization technology. Using blade-servers colsolidates servers because the individual blades share the PSU, high speed disks and the blade-server architecture enables lower power network cards as shorter cables are used. Disks can be consolidated by so called storage area networks.

3.2.2 Policies

Active power saving technology relies on software that sets the components in lower power states. It dependent on a policy that tries to predict the best available state.

Break even approaches Such policies calculate a so-called break even point. It marks the time a component has to be idle before switching to a low power mode becomes yielding. This is bigger than zero because state changes usually take time and uses additional energy, e.g., if the disk has spun down. Dependent on the access history and the break even point the system predics if switching to another power mode saves energy.

In it's most simple form this approach means that a component is set to a low power state after it is not used for a predefined time. This policy is used for disks in current operating systems like Windows Vista.

Workload characterization approaches Sometimes energy can be saved without any impact on processing performance. For example, it can be utilized that memory references are rather slow compared to other instructions. If the workload is memory bound setting the CPU to a lower frequency may result in the same speed but in less energy consumption, because the CPU spends less time waiting for the memory subsystem [Bel01].

Energy estimation Energy estimation approaches are based on a mechanism that determines the energy usage of a component. Overheating of a component needs to be avoided and the system is not allowed to use more power than the PSU supplies. These approaches switch to a lower power state until the crisis is solved [BKW03].

Load balancing Energy aware load balancing is done to keep energy consumption of similar components at the same level. This can either be used for components as CPUs or memory modules or can be applied to full servers. The advantage is that the more homogeneous heat dissipation makes cooling much simpler. There is also a break even point for these techniques because migrating load usually includes overhead [Mer05].

It can also save power to keep the load inhomogeneous. Transferring the load from one node to another in a cluster enables powering down some of the nodes and run the others at higher efficiency resulting in an overall power save [PBCH01].

Energy management support from user level Because the behavior of user level programs is best known by themselves their energy saving abilities are the best. If the user level program could tell the OS about the proposed idle time for a resource the OS can switch to the appropriate power state. Additionally user level programs could be modified to make accesses to a component burstier [HPB02].

3.2.3 Energy saving technologies summary

There are mainly two approaches for power saving technologies. Technologies of the first kind try to make the power consumption more homogeneous to simplify cooling and enable certain low power features like frequency/voltage scaling. The other technologies try to make the load burstier to achieve longer idle times to power down components. The appropiate way is dependent on the features supported by the hardware.

This implies for a workload resembling the power consumption characteristics, that all features and policies mentioned in this section must behave as similar as possible compared to the real workload. In particular the workload must have the same burstiness and homogeneous characteristics.

3.3 Measurement

3.3.1 Power Measurement

DC power measurement Direct current power consumption is measured directly at the power supply. Most voltages U_U supplied by the PSU are rather stable with minor inaccuracies at high load. Current I is computed from the voltage change U_I at a small high precision resistor R_I as in Figure 3. With these values power P is

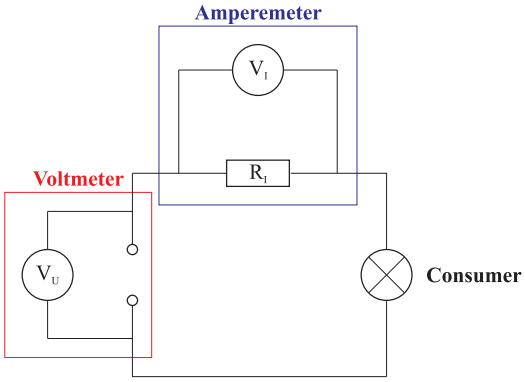

Figure 3: Power measurement network

$$
\begin{aligned}
P &= U \cdot I \\
I &= \frac{U_I}{R_I} \\
\Rightarrow P &= U_U \cdot \frac{U_I}{R_I}
\end{aligned} \tag{3}
$$

and energy consumed is

$$W = P \cdot t \tag{4}$$

For some components as CPU, disk and network adapter, power consumption can be easily measured. Measuring memory consumption requires modified mainboards as it cannot be traced directly to one source at the PSU.

AC power measurement AC power measurement uses the same measuring circuit as DC power measurement. Since the voltage and current are alternating these measurements cannot directly be used as values. In electrical engineering two powers are used to describe the power usage of an AC device. The so called *reactive power* periodically adds and subtracts a fixed amount of energy to the power measurement. As the consumed power is recompensed it can be seen as unrelevant error. The power really used by the component is called *real power* and is reported by most AC measurement devices. These usually report about one value per second. Appendix B explains AC power measurement in more detail.

PSMI If the PSU supports the *Power Supply Management Interface*, it can be used to monitor the PSU's state [Ser05]. All input and output voltages and currents

can be monitored. If the system under test contains a PSMI capable PSU, no further measurement equipment is needed.

3.3.2 Performance counters

Performance counters are model specific registers of CPUs. They were introduced to give the programmer a more accurate view on their programs' behavior. The counters can count several different events each in different modes. The events countable were chosen by the chip's manufacturer to provide help for optimizing the execution speed of programs [Int03, Int04].

Despite they were not originally designed to measure energy consumption it has been shown that they can give a reasonably good estimation for it [BKW03, Kel03]. Therefore they can also be used to create an energy profile for a CPU.

To estimate power consumption from performance counters a set of performance counters is chosen. It is assumed that each counted event describes a real event inside the CPU. Therefore it can be assigned to a small amount of energy consumed. This is called energy weight. To estimate the power consumption for the full CPU each counter has to be multiplied by its energy weight.

Most server's CPUs support performance counters but as they are model specific the interesting counters have to be selected again for each model. In this thesis I will focus on the Pentium 4 and derived CPUs as the Xeon and Pentium D. There are libraries to unify the access to performance counters, e.g., PAPI [BDG+00].

Performance counters can only be written in kernel mode. Therefore I used Linux's MSR-module to set up the counters. They can be read either by setting bit 8 of the CR4 register and read them by `RDPMC` or by reading them with help of Linux's MSR-module.

3.3.3 Linux /proc filesystem

The Linux proc filesystem was introduced in the kernel to enable a simple sharing of kernel data with the userland. It enables the user to access data that is usually only available to the kernel while maintaining security restrictions. For example it enables reading the state of a process but not modifying it.

The data available is eclectic. I use data from the `/proc/<pid>` directories to determine the thread count and CPU and memory usage. For information about network usage I use data from `/proc/net`.

Most of the data is collected on read so the data is accurate in timing. There are some metrics that have inaccurate values. For example, the CPU share is measured in jiffies which correspond to the amount of timer interrupts happend while this process was running. If a program gets blocked or yields the metric gets inaccurate because partial jiffies are not counted.

3.3.4 Linux VFS instrumentation

The virtual file system is a thin layer of software multiplexing and unifying filesystem accesses. Unfortunately, there is no information about VFS usage in the proc filesystem. As I use VFS accesses as a metric I had to collect the data in the VFS and expose it to the user-mode profiler by using the proc filesystem. The changes I had to apply to the Linux kernel are described in Section 5.2.

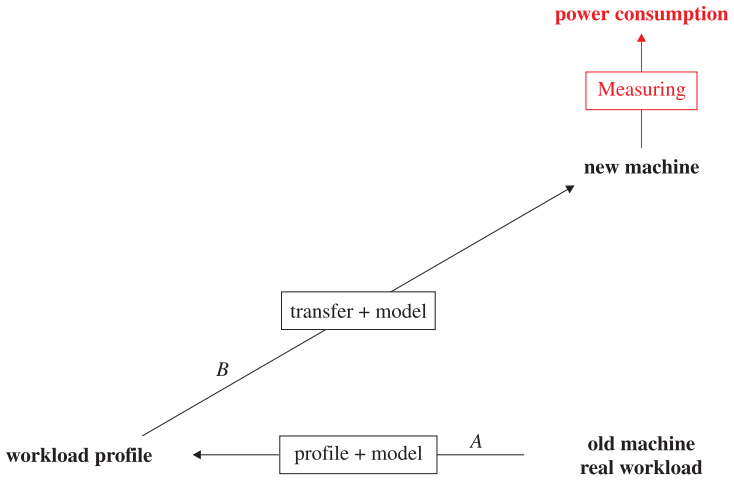

Figure 4: Three points design. The four point extension (Figure 19) is described in Section 7.1.

4 Design

The intended application of the software described in this thesis is to support small and medium sized companies to choose energy efficient machines when acquiring new servers. There are several different ways to define energy efficiency, leading to a set of currently available approaches. My basic assumption is that it is not possible to define energy efficiency as a single metric. There are workloads that are I/O-dependent, compute-dependent, containing many or few requests, homogenous or inhomegenous etc... Each of them performs differently on different machines. Sometimes a machine fits to one workload and uses few energy while executing it, while another machine uses less energy than others for another workload. A single metric can only give *one* hint for the energy efficiency of a server. But since there are different types of servers targeted to different workloads a machine cannot generally be described as energy efficient. Therefore, a single metric for energy efficiency can only cover a small scope of workloads.

Consequently, I will not define energy efficiency as one metric. The companies want to know how much energy a new server would use. My system tries to predict the power consumption of the server from an abstract description of the companies' real workload. I propose that this approach solves the companies' problem.

4.1 Design overview

The power prediction follows a three steps design displayed in Figure 4.

Component	Scope
Sockets	1 to 2
Cores	1 to 8
Architecture	IA-32 or AMD64/Intel64
Disk type	private, directly attached disks
Memory architecture	UMA
Network adapter	one or two 1 Gbit/s network adapters

Table 1: Hardware scope

1. Initially, the real workload on the old machine has to be profiled. To do this, the system administrator has to run the *profiler*-application on the old machine. It creates the profile by recording a set of metrics and saving their description (workload profile, path *A*). The profile consists of a list of measuring points including the metrics and a timestamp. To convince the company that no data is leaked, this profile has to be clearly defined and easily viewable. I use a XML-document and a small XSLT-script to transform it into a HTML-document. The profiler's source code is available as well.

   As the profile has to be generic for all operating systems and machines, creating it includes modeling the not directly measureable characteristics. E.g., it is not possible to measure each thread's CPU load in a Linux system without modifying the kernel. I model this metrics from the task's CPU load and it's thread-count.

2. After recording the profile it is sent to the manufacturer of the new machine. The *player*-application takes a profile as input and replays it. This process includes a lot of modeling because there are dependencies among the individual metrics profiled and some metrics need many assumptions to be replayable. This is described as path *B*.

3. For each measuring point the player-application generates the load described by the profile. The manufacturer measures the power consumption of the replayed profile on the new machine and reports the result back to the company. If this process is done for several servers, the company can decide which server is most energy efficient for their workload by comparing the reported values.

Sections 4.4, 4.5 and 4.6 explain how the recording, modeling, and replaying is done in detail.

Type of server	Example	Ressource heavily used
Web	Apache	
Web plugin	PHP	CPU
E-mail	imapd	CPU
Fileserver	samba	memory (as disk cache), network, disk
Application	JBoss	CPU
Remote	Citrix	network
Database	Oracle	memory, disk
Groupware	Lotus	
Domain controller	Windows Server 2003	

Table 2: Software scope. An exemplary overview about software considered. Not mentioned resources are used regularly.

4.2 Scope

The servers in my scope are either desktop derived (i.e. ordinary office computers) or small servers with one or two sockets. They are equipped with inefficient PSUs, IDE disks, often desktop processors and desktop style memory and chipsets. Current small servers have one or two sockets and consequently up to 8 cores running the mainstream IA-32 or AMD64/Intel64 instruction set. Specialized CPUs as the Sun UltraSparc T1 have 4 cores each processing 8 threads simultaneously and storage area networks change the power consumption characteristics of disk load fundamentally. I consider those specialized components, but the main scope will be on servers as described in Table 1.

Small companies often use standard software like web servers or databases. Some types of software are heavily using a specific component, e.g., fileservers are dependent on network and disk performance. Table 2 shows some examples. I do explicitly exclude high performance computing from my scope. The power consumption of such a server can be determined from the maximum power consumption often reported by the sever manufacturer.

The type of workload characteristics of the servers in the scope is eclectic. Most of the servers load is driven by external requests. A database server can have a high load during business hours and close to zero load in the night. An e-mail server's load is determined by spam attacks which may not be related to business hours. The load of a heavily used groupware server is homogeneous; the load of a webserver can be very inhomogeneous. Some load is I/O dependent, some is CPU dependent, even of the same program, e.g., a webserver can either be misused as file-transfer server (high network load) or used via https (high CPU load). Therefore it cannot be predicted how the load behaves for a component because it is determined by the software used and the requests submitted. Consequently I have to respect all kinds of load.

4.3 Design space

Generally speaking I follow the performance oriented approach as described by Gaede (Section 2.1 on page 5). It is based upon the assumption that benchmarking is realistic if the metrics correspond to the performance (in my case power consumption) and if the they are replayed at the same rates as in the original workload.

4.3.1 Choosing the metrics

The most basic decision is which characteristics of the original workload have to be replayed. As the metric, that is important to users of the system, is consumed power of the new machine, all characteristics chosen have to be related to power consumption. The other way round, the power consumption of a machine has to be traced to it's sources, i.e., metrics describing the power characteristics of the machine have to be found. Both ways have to be considered when choosing the metrics to log and deciding how to replay them.

As my system is profiling and replaying on different machines there is a special constraint for the metrics. My metrics have to be invariant to machine changes. This means that every feature of a component that might change shall have no effect on the metrics. E.g. cache effects, despite being important for the energy consumption, are not allowed as metric.

The effects of taking a non invariant metric is inadequate treatment of certain components. If L2 cache misses would be taken as metric there would be unrealistically many misses if the new machine uses a CPU with a bigger L2 cache than the profiled one. The real workload would have benefited of the big cache and the executing speed would be much higher and the memory and northbridge would use less energy.

There are some cases where the machines are so different that it becomes hard to find any invariant metric. For example if the instruction set architecture on the profiled and the new machine differ, it is not possible to find any invariant metric for CPU usage. To solve this case the only possibility is to alter the profile according to rules of thumb, gained by observation of example profiles. This processing makes the results become less significant. In this work I assume the ISA, word-size, type of disk, type of network (Ethernet, Token ring) and page size as constant.

Invariant metrics loose this characteristic if a component is overloaded. If the CPU is working at maximum capacity it cannot process any more instructions. If a server is overloaded often it becomes unusable. It can be assumed that most servers are replaced before they become unusable. Consequently overloading is seldom a problem.

4.3.2 Choosing the hardware

Although the servers in my scope seem to be similar on a first view, they have a wide variety of hardware components all contributing to the power consumption. Some servers are featuring special service processors and a lot of fans, others have northbridges with internal memory controller (as in use for Intel IA-32 CPUs) and some have raid-controllers with private memory. I therefore have to choose a set of hardware components that is available on most of the servers and also reflects the power consumption of the whole machine. A component becomes interesting to my system if the whole machine's power consumption is dependent on its load.

I chose the following components:

- The CPU is still contributing most to the power consumption. Hence, I have to take special care to accurately replay its power consumption. The runtime behavior and power consumption of multi-core CPUs or servers with more than one socket is dependent of the multithreading behavior of the workload.
- Memory power consumption is hard to measure and memory load is hard to trace, because there are no suitable built-in counters for the memory subsystem on the Pentium 4. But as it is a major contributor to the server's consumption it has to be taken into account.
- Database and fileservers have a set of large, high performance disks sometimes consuming more power that the main memory of the system.
- Fast network cards contribute a lot of power, too. Even though the power characteristics of current network adapters are very simple (no power saving features) it is likely that there will be power saving features in future hardware.

There are also some important metrics that cannot be assigned to a hardware component. Threads are important as they define how the CPU load can be partitioned onto several cores. Tasks are important as thread containers and since memory usage can only be assigned to tasks.

4.3.3 Replaying the profile

After choosing the metrics and how to measure them it has to decided how to replay the profiled information.

To replay a profile the load measured on old machine has to be generated on the new one. The main goal is to generate load in such way that the profiler would record the same profile again. Inaccuracies are allowed if it is assured that they don't affect the energy consumption of the whole server.

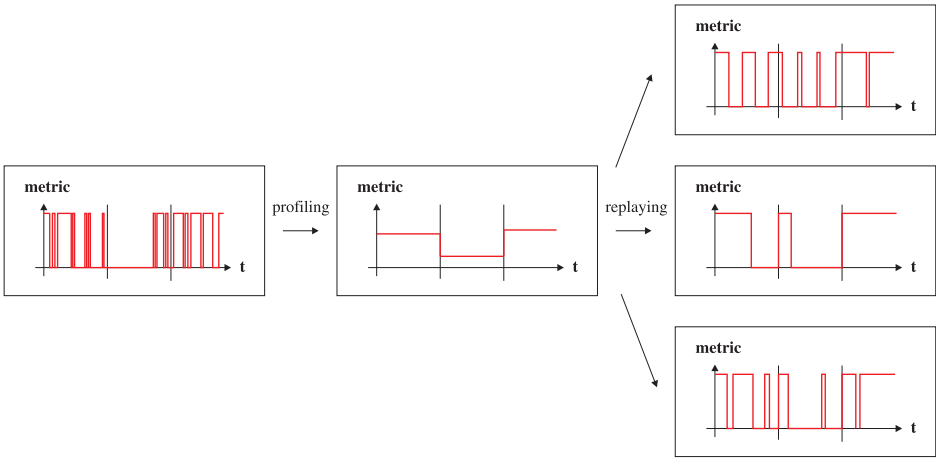

Figure 5: Three different ways to replay a profiled workload on a component with only one active performance state. The first box shows the real usage, the second the profile and the three boxes on the right show replays, that would create the same profile as in the second box.

Power management and therefore power consumption is dependent on timing characteristics (see Section 3.2 on page 13). The events described by the metrics cannot be accounted to an exact point in time, rather than to a timeframe. Consequently, it is important to choose the length of the measurement interval. It is also important at what kind of timing behavior the load is replayed. There are several different ways to replay the workload all valid by the performance oriented approach. Three different ways are displayed in Figure 5.

The easiest way to generate load is to generated it separately for each component. But, e.g., generating load for the network inducts some load to the CPU. This overhead load has to be subtracted from the load generated solely for the CPU. If one component generates overhead load for another and the other is generating overhead load for the first one, circular dependencies have to be solved.

Another way to replay the profile is to create a set of small programs voluntary generating load for more than one component. Then the player-application decides what mixture of these programs have to be run to reach the targeted load. This approach automatically solves the problem caused by overhead load. The disadvantage of this approach is that it results in less control over the replaying process.

In the system developed in this thesis each component's load is generated separately in order to have more control of what happens. I use the small-program-approach to generate the load for the CPU because the two metrics of the CPU are circularly interconnected.

Despite of all attempts to be realistic, the creator of a benchmark – sooner or later – comes to the point, where he has to choose out of several possibilities to

design the workload replaying process. This *benchmarking magic* is needed when the workload is not described accurately enough. For example, it is not possible to log all names of the files touched because it is far to much data. Whenever benchmarking magic is used there are different ways to design it. The creator of the benchmarking system has to claim that his decision resembles reality most closely. The most extreme form of benchmarking magic is to define one fixed workload and state that it is realistic. Least benchmarking magic is used when the real workload can be identically replayed but this is not in the scope of this thesis.

When I use benchmarking magic, I try to use it to simulate the behavior or the software in my scope. If there is no suitable behavior that can be generated I will rather generate no load than doing something wrong. This is because no generating is predictable but some unmotivated load is be hard to predict.

While designing my system I therefore had to respect the rules of Table 3.

Rules for profiling

1. Choose metrics that are related to the power consumption.
2. The metrics have to be invariant.
3. They have to be directly measureable or easily computable from measured metrics on a Linux 2.6 system.
4. If possible do not modify the kernel. If a modification is necessary it must be minimal.

Rules for replaying

1. Generate the same metrics at the same rate. (performance oriented approach)
2. Inaccuracies are only allowed if they do not affect the power consumption.
3. Use benchmarking magic if a behavior can be assumed for a large part of the software scope.
4. Better do nothing than something wrong.

Table 3: Design rules. All design decisions are based on these rules.

4.4 OS abstraction modeling

Abstractions of the operating system (like threads) are not directly responsible for any power consumption but some of them influence the load characteristics. And since the power consumption of a component is dependent on the load characteristics, they also have to be taken into account when profiling a workload.

The most important effect on power consumption is that threads enable the OS to spread the CPU load to different cores. This may have positive or negative effects on the power consumption dependent on the CPU's model and the OS' scheduling policy. Tasks are important as they represent address spaces. Task switching implies a TLB flush and some scheduling policies try to keep the threads of a task on one CPU because of cache effects. The TLB and the caches contribute heavily to the CPU's power consumption.

Recording The Linux proc filesystem (Linux kernel at `fs/proc`; also referred to as *procfs*) gives a lot of information about the currently running processes (see Section 3.3.3).

There is a directory for each Linux process that started a new task having it's process ID as name. So each directory having a number as name represents one task. This directory contains virtual files reporting a lot of information from the kernel.

As processes cannot be mapped directly to threads and tasks, the information how to do this mapping sometimes becomes short. The procfs contains information about the number of threads in the tasks and how many jiffies the task was scheduled. The jiffies-value for the process is incremented for each scheduling slot it is running. As jiffies is an integer value, half used scheduling slots lead to inaccurate accounting. There is no information about how long the individual threads were running.

Modeling For each directory in the procfs one task is created in the profile. If the directory is not there anymore the task ends. It is very unlikely that a directory (representing a task) is deleted and created with the same ID during one measuremet cycle as Linux reuses process IDs as late as possible.

As only the number of threads is available, the CPU usage of the threads must be estimated. Therefore I have to decide which multithreading pattern to use. I choose to use dispatcher/worker. In this scenario a main thread (dispatcher) dispatches each request to a new thread (worker). For example a webserver creates a new thread for each file request. In this scenario it is likely that each thread uses a more or less equal CPU load share. Consequently the threads of a task get an equal CPU load share of the CPU load of it's task divided by their number. In other

multithreading scenarios the CPU load will be spread differently to the threads but dispatcher/worker is the pattern most often used by the software in my scope.

The procfs provides no information whether a thread has been created or has been finished. I only assume a thread to be created when the thread count in a task increases and to have finished when the thread count decreases. As in a dispatcher/worker-scenario the lifetime of a thread can be very short, this does not reflect reality. It is irrelevant whether a thread or its descendent executes the CPU load but there is some load in the kernel for creating and deleting threads. The load generated by thread creating/deleting is not considered because it is irrelevant compared to the load generated by the programs running. Ergo there is no cutback in accuracy if only the thread count is modeled correctly.

For each measuring point, a full list of tasks with their IDs, threads with their IDs, and a mapping tasks $\leftrightarrow$ threads is stored into the profile file. If the operating system of the profiled machine provides more information about threads than Linux does, there is no problem to respect it, as each thread is stored separately.

The amount of tasks and threads is controlled by the application. Hence, it is invariant to machine type and operating system. If there are multithreading policies that are dependent on cores available or system load they create a relation between machine type / operating system and tasks and threads. This situation is not solved by my approach.

Replaying For each task the player-application determines when the task has to start. For each thread it is determined how long after it's enclosing tasks it has to be executed. The main player task executes the task at it's start-time and the tasks starts its threads at their time. The threads terminate when there is no more CPU load left to generate and a tasks finishes after all it's threads have terminated.

4.5 Component load modeling

All components are replayed for their self. If the generation of load for a component inducts load on another, the overhead load must be respected while replaying the load of the other component. This process is described in Section 4.6.

4.5.1 CPU

The CPUs are still the main power consumer of a server. Current CPU cooling solutions have been built to handle heat dissipation of up to 130W. Also the power consumed by a CPU varies heavily. This implies for my system that modeling the CPU load accurately is the most important part of this work.

CPU metrics	Description	Granularity
Instructions	Counts all instructions executed in between two measuring points including branches. Speculatively executed instructions during a branch misprediction are not counted because they are not invariant to machine changes.	thread
Conditional branches	Counts conditional branch instructions (`JE`, `JNE`, etc...)	thread

Table 4: Metrics used for CPU profiling

Recording The metrics supplied by Linux are CPU load percentage and jiffies used by each task. Both metrics are dependent on the execution speed of the CPU making them not invariant to machine changes. In 2003 Bellosa et al. described a system to estimate energy consumption from performance counters of a CPU [BKW03]. Many of these counters are machine dependent (e.g. L2 cache misses) but there are also some invariant. Out of the Pentium 4's counters instructions executed and branches executed are invariant to machine changes because they are determined by the application program itself. If the metrics are restricted to user-level they are also OS invariant. A minor error in accuracy can be inducted by OS-near code that is dynamically loaded by the program as for example the glibc, the Java virtual machine or user level OS-components (e.g. user level drivers).

Bellosa et. al. use seven metrics to characterize the workload of a CPU but as only two invariant metrics are available my system will be less accurate.

When changing to another ISA the recorded values have to be preprocessed. Changing from IA-32 to AMD64/Intel64 would need a small correction of the instructions executed because 64-bit operations can be processed faster. When the ISAs differ more basically, (e.g. CISC and RISC) further measurements are needed to get hints for preprocessing those metrics.

I use the msr-module of Linux to setup and read the performance counters. This standard kernel module allows writing to and reading from special registers of the CPU as, e.g., the performance counters. According to the CPU share of the threads the instructions/branches are accounted to the individual threads.

If none of the threads have any CPU share and there were some instructions used, this is caused by the inaccuracy in CPU share computation. As each instruction consumes energy, I chose to respect them by accounting them equally to all threads. Another way would be to assign them to the last thread, that was running. In this case no behavior can to be favored without more or more detailed metrics.

Modeling Each instruction uses a distinct set of CPU units. The usage is also influenced by the state of the pipeline, TLB and the caches. The power consumption is the sum of the consumptions of the currently active units. This means, that the current power consumption is dependent on the recent instruction history.

The information from the profiler gives an outline for the load generation. The task for modeling CPU load is to fill it with code that is realistic in the recent instruction history compared to real server applications. It is not possible to generate branches and instructions independently. Each branch is an instruction and while generating instructions there has to be a conditional loop. This makes these two metrics interconnected. As it is not possible to replay them separately I have to use the small program approach. Also, dividing branches from instructions would not be realistic as the recent instruction history of a real program usually contains both.

The small program approach uses a set of program fragments (in my case functions) executing each of them several times to generate a given load target. The programs have to be as realistic as possible and also have to cover all combinations of values of the metrics. As I use only two metrics the only combination is

$$branchiness := \frac{branches}{instructions}$$

According to measurements the branchiness of real programs is between 0.2 and 0.04 for highly optimized code. Therefore I use a set of functions with branchiness varying from 0.2 to 0.04. The small programs have to reflect the instruction mix of the software in my scope. Most of the softwares are using the C/C++ standard-library and only a few are using floating point instructions. Therefore my programs mimic these instruction mix. For each function it is measured how many instructions and branches they use. This is stored into the vector $functionLoad_x$.

The problem to be solved is

$$targetLoad + error \stackrel{!}{=} \sum^{\forall x} runs_x \cdot functionLoad_x \tag{5}$$

with *targetLoad* and $functionLoad_x$ being twodimensional vectors. The *error* in Equation 5 has to be as small as possible.

$runs_x$ describe how often a function has to be executed as an integer value. The computation of the optimal $runs_x$ is theoretically a multi-dimensional knapsack problem. But as *targetLoad* is very big and the $functionLoad_x$-s are small a minimally nonoptimal error for $runs_x$ can be tolerated. This makes it possible to use the simplex algorithm that returns the optimal (rational) values for $runs_x$ which will then be rounded. A description of the simplex algorithm can be found in most text books on optimization theory [Rao96]. The maximal difference between the optimal solution and the solution returned by the simplex algorithm is

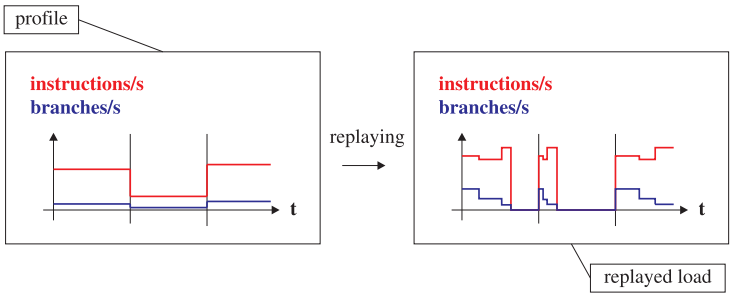

Figure 6: CPU load generation for metrics instructions and branches using three small programs.

$$\max error = \frac{1}{2} \sum^{\forall x} functionLoad_x$$

This system can easily be extended to more CPU load metrics by adding more small programs that cover all possible combinations of values of these metrics.

Replaying For each thread and each measuring point $runs_x$ is calculated. When the load is due each function is run as often as calculated. After that the thread is paused until the next measuring point. The load profile looks as in Figure 6. It is unlikely to happen that in between the measuring points more load should be generated than possible, because new machines are usually faster than old ones. If there is more load than generateable the next load is postponed until the former one is generated.

4.5.2 Memory

Recording The load of a memory system is basically determined by the access pattern to the caches and finally to the memory modules. The only invariant metrics can be recorded at application level. In between physical usage and application there are the following levels, each affecting the characteristics of the access pattern [AGVO05]: Application → OS's virtual to physical address translation → L1 cache → L2 cache (potentially shared between one CPU's cores) → sometimes L3 cache → physical memory.

Additionally, there are no feasible measurement facilities for online memory pattern profiling on the Pentium 4. TLB references for instruction fetching as available as performance counters only reflect a very small part of memory usage and this part is also cached well. Cache misses are not invariant and therefore cannot be used as metrics.

Memory metrics	Description	Granularity
Memory size	KB allocated by the task including, e.g., memory used by code, data, heap and stack.	task
Working set	KB recently used. “Recently” is not further defined.	task

Table 5: Metrics used for memory profiling

Linux reports two metrics to the procfs: The recent set is the sum of the currently loaded memory pages for each task. The total memory size is the sum of recent set and paged out pages. Total memory usage is invariant, but the recent set is determined by the size of the installed memory in the machine. Nonetheless these two metrics are the only two giving a hint on the memory usage pattern.

These two metrics are very high level. This means, that they cannot directly be traced to power consumption. The reason for including them into my system is, that the memory usage of applications narrows the disk's cache. A smaller disk cache has severe influence on power consumption.

I take the recent set as estimation for the working set. The recent set is bigger than the working set because otherwise the memory subsystem would have to use heavy paging which makes servers unusable. The other way round the amount of memory that is in the recent set but not in the working set does not affect the memory behavior because the new machine will usually have more memory than the old one. These two assumptions permit the usage of the recent set as estimation for the working set.

An interesting aspect of the memory metrics is that they are not event specific. The metrics do not count events, they describe a state. While event specific metrics have to be fully generated, state-specific ones have to be adjusted. The working set as naturally being state specific becomes partially event specific by the need of touching each chunk once in a while.

The two metrics do not cover all aspects that influence the memory power consumption. Therefore memory power estimation is rather inaccurate.

Modeling To model which part of the total memory is in the working set I treat it as a list, adding the new chunks at the beginning and removing the chunks at the end. The working set consists of the newest data. Another possible model is to add a certain percentage of the memory at the beginning and the other part at the end. The problem with the suggested approach is that it generates more CPU load due to the deciding function.

I do not model or replay any memory reference pattern, because without any information about it creating one is pure guessing.

Replaying The total memory size is adjusted at every replayed measuring point by simply calling `malloc()` and `free()`. The working set is adjusted by touching two bytes of each KB-chunk of the working set. This assures that the memory stays in the working set.

4.5.3 Network

It does not affect the networks adapter power consumption which thread of which task sent a package. Therefore the network load can be generated for the whole system by one thread.

Recording When writing this thesis there were no power saving features available for network adapters. This lead to two power modes. The first on is while the network card is idle, transferring no data. In this state most energy is required to drive the long network cables (0.7W on an Intel e100 Network adapter at 100 Mbit/s, own measurement). For processing and sending packets additional energy is needed (0.5W). The power consumption of Gbit network adapters is much higher.

Some major network component manufacturers have announced to enable to adapt the network speed for a network adapter without forcing the adapter to reconnect. As it takes less power to drive the cables with lower speed this can be used as power saving feature. Then power consumption will become dependent on amount of data transmitted.

Generating network packets implies some generation of CPU load in user- and kernel mode. The load in user mode is automatically subtracted from the CPU load of the measurement point but the instruction mix of the CPU load changes, because of the special mix of network load generating program. The generated CPU load is dependent on the size of the packets sent/received. Consequently the load has to be divided into packet-count and packet-size.

Linux provides information about network load in `/proc/net/dev`. It is categorized by network adapter and transmit-direction. As the networking infrastructure cannot be rebuilt at the manufacturer, I ignore the information about which adapter sends what percentage of the load. This leads to the metrics of Table 6. The values reported in the procfs counts the transmitted bytes including the header of the network-protocols. A TCP/IP on Ethernet stack adds about 42 bytes to the payload data. The exact overhead depends on network interna.

Network metrics	Description	Granularity
Bytes sent	Bytes send to any computer, including all headers.	system
Packets sent	Packets send to any computer.	system
Bytes received	Bytes received from any computer, including all headers.	system
Packets received	Packets received from any computer.	system

Table 6: Metrics used for network profiling

Modeling All packets have the same size in my system. This is because of the assumption that the packet-size does not matter for the CPU load of the operating system. The randoming system – creating different network packet sizes – would cause too much and hard to predict CPU load. This prediction is necessary because the CPU load caused by the network load generation has to be subtracted from the CPU load to be generated.

Similar to the CPU load the network load is replayed with maximal burstiness, meaning the packets are sent as fast as possible. If there are no packets left in a measurement point the replaying waits until the next packets are due. The burstiness could be reduced by dividing each measuring point into several new measuring points.

Replaying The network data must be generated at two sources. The first source is the new machine itself, sending packets to a client program running on another machine. The client is connected via a local area network. The profile of the incoming data is sent to the client before replaying starts. The client sends the packets back to the new machine when they are due. The packets are using UDP and the network lag will be corrected. The timing depends on the local systems clock so the clocks must be kept in sync (e.g by using NTP).

4.5.4 Disk

Linux provides no invariant metrics on disk usage. Those available cannot be used because they do not respect disk caching, being an important performance and power saving feature. Systems with bigger disk-caches usually behave much faster and need less energy for the disk. Therefore disk profiling and replaying needs an extension to the Linux kernel. This makes the design space for the disk much wider.

Recording The disk's electronic components and on board buffers draw power when not inactive. The rotating of the platters will cost extra watts when the disk is active. When serving requests positioning the actuator will draw power too. Some disks support different active states with different power usage and wake up times. This leads to the following power states for a disk:

Inactive The disk is shut off. It can be activated in runtime, but this takes very long.

Not active The disk's platters are not rotating. The mechanical components are inactive but the electronic ones are not.

Active idle The disk has spun up but no data is currently transferred to or from the disks. Some disks do support lower rotation speeds while in this mode.

Active The disk is serving requests, moving the actuator to the file's track and transferring the data to the memory. There are currently no disks available supporting lower rotation speeds at this mode but future hardware will support this feature.

Whether to switch to the first three modes or not is dependent on the time between the accesses to the disk. This is not equal to the time between the write/read requests of the application because of the disk-cache that buffers the data recently read or written, if there is another request for already known data. The following informations are needed to cover this behavior: How many requests are issued and to which files do they apply to.

For disks with different rotation speed the amount of data read and written is important. This also influences fixed speed disks as some files of the disk may be fragmented and therefore need several movements of the actuator to serve a long request.

These constraints lead to the need for the metrics of Table 7. I count the open/close requests to have a hint on how many files are touched. To perfectly replay the disk-load (1) the filenames of each request, (2) the location and (3) size of the file on the disk are required. The second information will change on the new system and the first one is not recordable because it would generate far to much data. The third results in the most extreme case in an overview on the company's data which makes recording this data prohibited due to privacy concerns. The other way round generating the disk-load from these six metrics includes a lot of benchmarking magic.

Modeling The read and write requests are modeled similar to the network requests. The two possibilities are to have all requests have the same size or picking

Disk metrics	Description	Granularity
Opens	Files opened	system
Closes	Files closed	system
Bytes written	Bytes written to any location. Cached bytes are also counted	system
Write requests	Write request issued	system
Bytes read	Bytes read from any location. Cached bytes are also counted	system
Read requests	Read request issued	system

Table 7: Metrics used for disk profiling

a size randomly according to a given distribution. In my implementation I used equally sized requests for one transfer direction because of the same reason as with the network modeling

To recreate a file usage profile from the open/close profile I apply a multi-step approach.

Creation of the processes The basic abstraction in file handling is an access to a file: open → reads or writes → close. I call this abstraction a disk-load-process – not to be confused with a Linux process. It consists of several bursts of load that can be different in request count and bytes served.

For each measuring point it is calculated from the open/close values how many processes have to be open at it's beginning. If at one instance more processes have been closed then are open these are assumed to have been open initially. Then the new processes are created. Some of them are marked as "read" and some as "write"; It is tried to achieve a equal request-size for each program. If there is heavy read load in the measurement point many new read-processes are opened to keep the read byte count per process load low. Afterwards some processes are closed to perform the close-events.

This practice leads to a set of processes with an open- and a close-time, a load for the time in between and a marker, whether it is a read or write process. There are no processes that generate read and write requests as this is uncommon in reality. Reading and writing to the same file is modeled by separate processes. Some processes are shown in Figure 7 as blue (read) and red (write) lines.

Finding the parameters of the *filesAt(t)-function* At the beginning of the profiling process many unknown files are used. In succession more and more already touched files are reused. In Figure 7 a process reading an unknown file

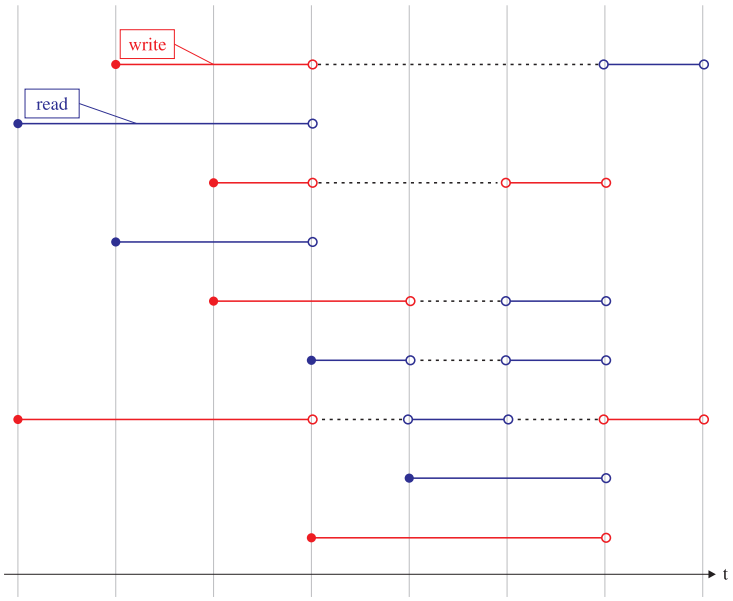

Figure 7: Disk usage modeling. Each line corresponds to one file. Each colored line represents one process. Multiple processes can be assigned to one file.

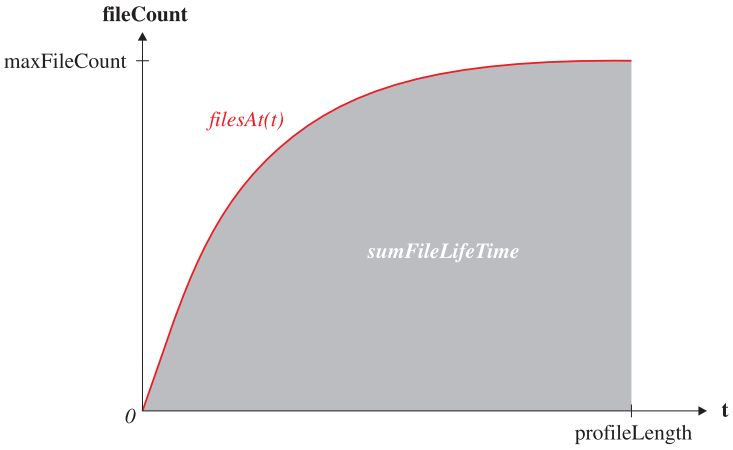

Figure 8: *filesAt(t)*

begins with a filled dot and processes using already known files begin with a empty dot.

The files that have been used at a point of time t are shown in Figure 8 as *filesAt(t)*. For various softwares in my scope this behavior reflects the behavior of real workloads. There are two constants (*fileUsageQuota* and *fileReqritePossibility*) that are used as parameters in this formula. This makes my decision adaptable to several kinds of behaviors.

There are different kinds of loads having a completely different *filesAt(t)*; E.g., a streaming server is reading the same source file again and again. If the real workload of the company has a behavior not conform to my *filesAt(t)*, the formula must be adapted.

I use the following formula for *filesAt(t)*:

$$\mathit{filesAt}(t) := \mathit{maxFileCount} - \mathit{maxFileCount} \cdot \mathrm{e}^{\frac{-5t}{\mathit{profileLength}}}$$

profileLenght is the total length of the profile, i.e., the time between the first and the last measuring point. *maxFileCount* is dependent on the predefined constant *fileUsageQuota* because a lower quota means more files.

$$\begin{aligned} \mathit{sumFileLifeTime} &= \int_0^{\mathit{profileLength}} \mathit{filesAt}(t)\ dt \\ &= \frac{\mathit{maxFileCount} \cdot \mathit{profileLength}}{5} \cdot (4 + \mathrm{e}^{-5}) \end{aligned} \tag{6}$$

sumFileLifeTime is the the maximum available time files can be used by a process (grey in Figure 8). The length of all processes accumulate to *sumProcessLength*.

$$\mathit{sumProcessLength} = \mathit{sumFileLifeTime} \cdot \mathit{fileUsageQuota} \tag{7}$$

Ergo *maxFileCount* is calculated as

$$(6, 7) \Rightarrow \mathit{maxFileCount} = 5 \cdot \frac{\mathit{sumProcessLength}}{\mathit{profileLength}\ \cdot \mathit{fileUsageQuota} \cdot (4 + \mathrm{e}^{-5})}$$

Figure 7 is a magnification of a part of the grey area of Figure 8. *fileUsageQuota* determines what fraction of the grey area would be colored blue and red and can be adjusted to achieve different modeling characteristics.

Assignment of processes to file-objects In this step the processes are assigned to files-objects. The file-objects are bound to real files in the next step.

First, the write-processes are handled. Processes are assigned to already available file-objects, as often as determined by the predefined constant *fileReqritePossibility*. The others are assigned to file-objects that are newly created at the beginning of the process. The processes in the rewritten file-object must not collide. Two processes collide, if they are generating load in the same measuring point.

Second, the read-processes are randomly assigned to available – created by write-processes – or new file-objects. The program takes care that in each measurement point the total amount of file-objects is at *filesAt(t)*. Again processes can only be assigned to file-objects containing no colliding process.

Creation and finding of real files for the file-objects For each file-object that is never written, a real, already on the disk existing file is searched. If no such is available a new one is created. This is done to achieve a distribution of the files over the disk.

Replaying In one measuring point the load for a file has to be generated by a distinct thread to enable concurrent accesses to different areas of the disk. To keep the amount of threads used for disk load replaying as small as possible, each thread can handle several files if they are accessed one after the other. Additionally, each thread is started at the beginning of the replaying process and sleeps until the first file is to be handled. This keeps the CPU-load generated by the thread's creation low.

4.6 Cross load effects elimination

When replaying the loads of different components concurrently they influence each other. For example all load replaying functions generate CPU load. Therefore each load generator has to report how much load for the other components it generates.

This overhead load is dependent on the amount of load to be replayed and the type of system. To predict the amount of overhead load, it is quantified during an extra calibration step. Not every combination of input values to the load generator can be measured. Therefore, I have to interpolate between the calibration measurements. As most loads are defined by several metrics (e. g., instructions and branches for CPU load) the interpolation often becomes multidimensional. If there is more load generated as overhead than should be replayed, it is regarded to in the next measuring point.

Another cross-effect that happens, even if the previous cross effect has been respected, is that the characteristics of load generated changes. E.g., the CPU load

is usually replayed by a set of small programs; If the memory load generates some CPU load as cross-effect less of these small programs are run. The memory load replaying function has not the same CPU load characteristics as the ones it replaces. Therefore the characteristic (instruction history, etc ...) of the CPU load changes, even if the metrics are correctly replayed.

If there is heavy memory access in the real workload the CPU characteristics also differ from the characteristics of light memory access periods. This makes this cross effect a good effect as it also happens in the real workload.

4.7 Design summary

In this thesis I describe a system that profiles the real workload of a company and replays the workload's profile on a new machine. From measuring the power consumption of the new server replaying the profile energy consumption for the real workload can be estimated.

The workload modeling process for all components follows the same scheme:

First, the metrics have to be identified. They have to be related to the power consumption and have to be invariant to machine and operating system changes. This constraint is needed because otherwise the workload would be influenced by characteristics of the old machine. There are no invariant metrics for the file and disk-subsystem in Linux 2.6. I therefore created a set of metrics that fits to the rules. Some metrics have to be postprocessed to achieve a homogeneous profile from different operating systems. E. g., in Linux the thread's CPU usage metric is calculated from values of the procfs' tasks status.

As the metrics do not completely describe the behavior of the workload so called "benchmarking magic" is needed to build a new workload that can be replayed. Benchmarking magic means that the creator of the workload replaying system has to choose from the available designs, while not being able to prove that the chosen one is right. No benchmarking magic is correct for all possible workloads.

The workload for some hardware components can be replayed without too much benchmarking magic. Memory replaying is very difficult, because the metrics only give a very rough outline of it's real workload. Disk load needs a lot of benchmarking magic too, because it is not possible to store information about the exact file-usage pattern. In my system I model the workloads to match the behavior of the majority of server applications in use by the companies in my scope. Other models for a different scope produce different results. Less benchmarking magic can be achieved by better metrics but some issues of the disk load replaying cannot be solved by the approach chosen in this thesis. When replaying the workload, some cross-effect happen and are counterbalanced if possible.

4.8 Other usage scenarios

Although, being developed for the main usage scenario of predicting the power consumption of a workload my system can be used to solve other problems.

Consolidation A big energy efficiency improvement is to consolidate two servers onto one machine. This can either be done by moving the applications or by virtualization techniques. To estimate the power consumption of the consolidated workload a consolidated profile has to be generated.

All metrics used by my system are invariant to OS changes. Among other things this means that they describe the user-level behavior of the workload. A main function of an operating system is to make sure, that user level programs do not influence each other. The only influence appears when ressources run low and two programs compete for them. Therefore, the consolidation of two workloads can be simulated by summing up their profiles.

As the profiles are dependent on time, it has to be respected when the profiles were measured. If the profiles were recorded at different times of the day the measurement points of the first have to be added to the measurement points of the second at the same time of day. For some workloads it is also important on what weekday the measurement points were recorded. In this case this timing behavior has to be respected, too.

Non-energy benchmarking Macro benchmarking is based on replaying a specified workload. Usually a new program is created to generate the load. My system can be used to generate custom workloads for different kinds of benchmarks.

There is a drawback of using my energy benchmarking system for benchmarking other metrics, like processing performance or network speed. I build my system according to the design rules made for achieving the best results for the main usage scenario. Using other metrics or using other benchmarking magic give more suitable benchmarks for non-energy metrics.

Application profiling The profiler-application creates a profile of all programs running on a machine. If there is only one program running and the input data, i.e., network requests, are known, the profile gives a description of what happens for which kind of input. This can be used to optimize the behavior of a single program. In this scenario no replaying is done and the invariance-constraint becomes unnecessary. Although using the profiler as described would give a reasonable good description of the behavior, choosing other metrics would result in a better description.

5 Implementation

I created two programs, a Linux kernel entension, and a small utility program to implement the three design steps (profiling, modeling, and replaying). All programs were written in C++ using a lot of OOP. Figures 9, 10, 11 and 12 give an overview of the important parts of their class-structure.

As my benchmarking magic does not fit to all real workloads, my system was built to be easily extendendable. This can be done by modifing functions of the player responsible for the modeling process. If other operating systems than Linux 2.6 or other hardware platforms as the Pentium 4 need to be supported the programs have to be modified. In Section 7.2 I describe what needs to be done to port the system to other platforms and what has to be adapted if a new modeling or new metrics should be introduced.

5.1 Profiler

The purpose of the profiler-application is to create a profile of a currently running workload. It is run by the company wanting to profile it's workload. It's execution can be divided into three steps.

Measuring Figure 9 gives an overview over the profiler application. The main class used in this step is the System class or one of its subclasses. System provides a standardized interface to the systems state. At each call of `updateState()` the state of the class is adjusted to the real state of the system. Event-specific metrics as branch-instructions are counted between two updates. With calling `getState()` the current state can be externalized from a System-class, while `setState(state)` sets a System-class to a previously saved state.

Reading metrics from a System includes some overhead. Therefore I decided to not do this at recording time to keep the profiler-application's overhead low. Ergo, it calls `getState()` periodically during the recording and stores the resulting states.

Processing After finishing recording, the raw data in the states have to be transformed into metrics. The subclasses of StatReader (statistics) are responsible for reading the metrics of one component or an OS-abstraction. The ReaderContainer class is a composite of readers and the TimeStampReader is a ReaderContainer adding a timestamp.

For each measuring point the state is loaded back into the System-class. Then `ReaderContainer::read()` is called, calling the `read()` method of each contained StatReader. The read calls return a tree of Stats for this measurement

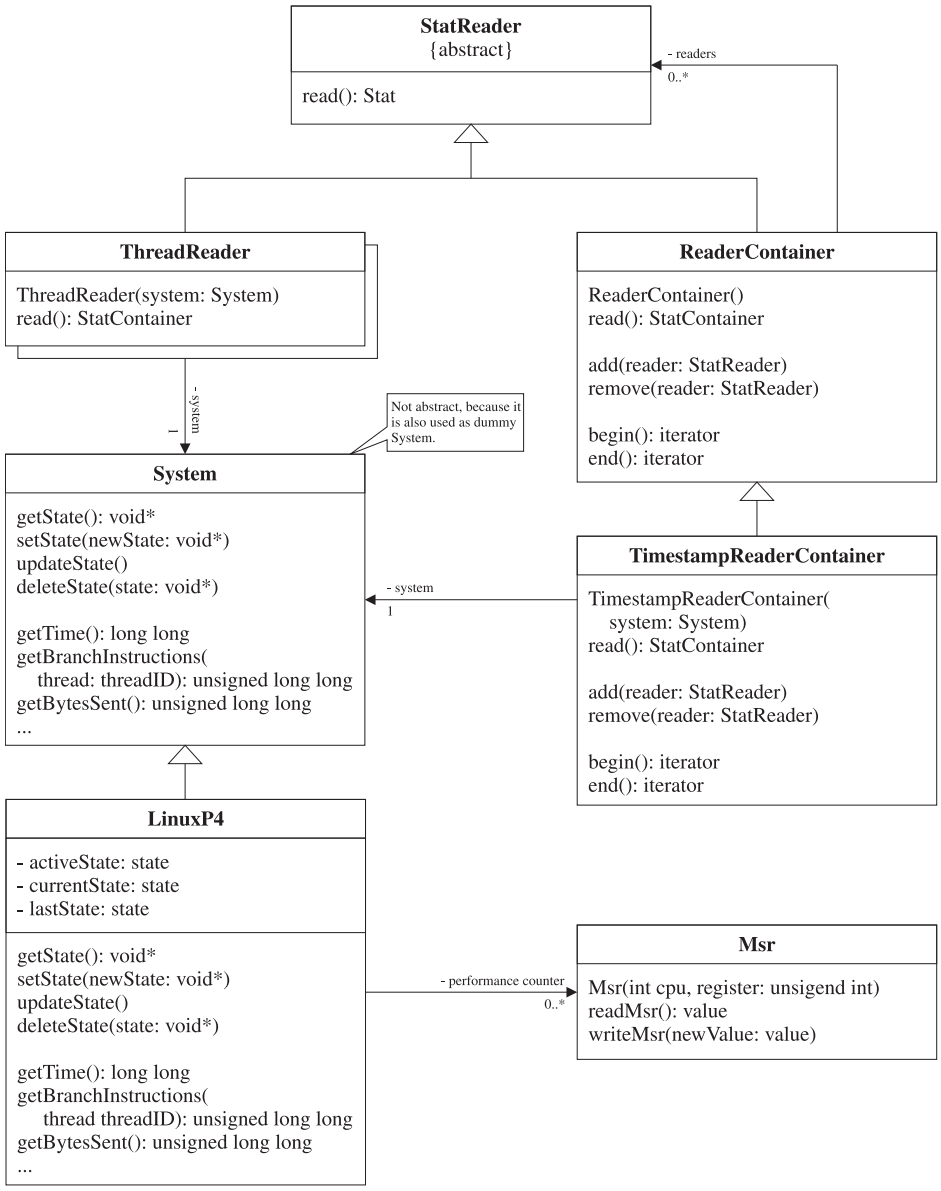

Figure 9: Class diagram of the profiler-application. There are subclasses of StatReader similar to ThreadReader for each component or OS-abstraction. If another operating system or CPU model should be supported a subclass of System has to be added.

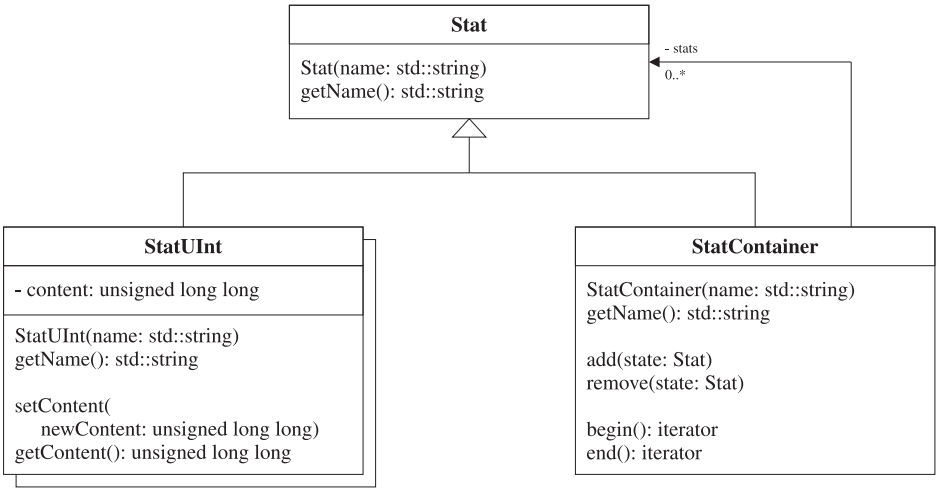

Figure 10: Class diagram of stats. There are subclasses of Stat for each type of value.

point. Due to the similarity of the StatReader- and the Stat-structure (see Figure 10) no further work needs to be done.

As my system should be as OS-generic as possible, I use the System class as interface to a class doing the real recording. In my prototype I implemented such a class for Linux and a Pentium 4. This class retrieves the metrics from the procfs, the VFS extension (Section 5.2) and the Pentium 4's performance counter via a Msr-class and the Linux msr-module shipped with the standard kernel.

In the LinuxP4 class most metrics are directly read from the raw state data. The methods for reading thread-related metrics as, e.g, `getBranchInstructions (thread)` are based on a internal modeling of the threads' state. As described in Section 4.4 the CPU load of the threads is calculated from the CPU load of the thread's task and it's thread-count. This is done once per state when needed the first time.

Saving the profile The file containing the profile is a XML file resembling the structure of the stat tree created during the processing step. As XML is a tree based data description language the stat tree can identically be written to the file. For each Stat a new XML-tag is opened with its name. If the Stat is no container the content of the tag is read via `getContent()`. In the case of a container the content of the tag is a collection of the tags of the Stats in the container.

Each `<measuring_point>` describes the load of one state. There is a tag for each OS-abstraction or component in each `<measuring_point>`. These tags contain either the metrics (`<disk>` and `<network>`) or in case of the OS-abstractions (`<tasks>` and `<threads>`) tags for each single abstraction

(<task> and <thread>). The <task> and <thread> tags contain their metrics and the mapping thread → task.

5.2 VFS /proc extension

An extension to Linux' procfs is needed to profile the disk load. It is not a part of any standard kernel. I implemented an example what is to be done to record the needed metrics. The purpose of the extension is to expose disk metrics that are invariant to machine changes from the kernel to the user. This means that they have to be measured above the disk cache.

In Linux the *VFS* (virtual file system) layer is used to unify file accesses of different file system. It is an interface translating system calls into calls to the file systems' read, write, etc... functions. As I retrieve the metrics directly from the system calls they are invariant of filesystems and machine specifica.

There are four different ways to access files in Linux [BC05]. Direct I/O is only used by a few high performance applications. It enables those programs to directly access the files without interference of the operating system's disk caching. Memory mapped I/O is also used seldomly. Asynchronous I/O can also be used by the user directly but is mainly used by direct and memory mapped I/O. The vast majority of file accesses are regular synchronized I/O requests.

Therefore I decided to instrument the synchronized I/O. Basically there are four system calls: `open()`, `close()`, `read()` and `write()` (and the multiread and multiwrite calls `readv()` and `writev()`). All syscalls use a file handle that is translatable to an unique file descriptor. The read and write calls also specify read or write size. The `seek()` system calls are not interesting as they do not affect the disk directly. They become important if a file gets larger than one block and the filepointer has to be used to determine which block to read.

It is not possible to expose all information collectable to the user and keep an acceptable performance because all data means one integer per open / close and three per write / read request. I decided to store the amount of opens, closes, read requests and write requests each in a separate counter and to sum up all bytes read and written in another two counters. These six counters give a reasonable accurate overview over the VFS requests as discussed in Section 4.5.4.

5.3 Player

The player-application is run by a server-manufacturer to measure the power consumption of a profiled workload on a specific machine. The machine has to be equipped with an AC-measurement device (described in Section 3.3.1 on page 17). The measured energy consumption is reported back to the company.

After reading the profile, the load generating classes are created. The classes for CPU, network and memory replaying are subclasses of WorkLoadGenerator. As the disk load modeling is more complex it is modeled separately.

Calibrating All load generators have to be calibrated to serve two purposes. First, some need to be calibrated to know how to replay the load. For example the CPULoadGenerator measures how many instructions each small program generates. The second purpose is to measure how much overhead load for other components a generator produces. This is needed for the `getRealGeneratedLoad()` method, reporting how much load a WorkLoadGenerator really generates. The returned value includes volitional load and overhead.

Modeling In the next step the profile has to be transformed into a playable structure. It looks as shown in Figure 11 and 12. These structure or model is build out of a set of Runnable classes. There are `Threads` and `Tasks`. A `Executor` is a class starting other Runnables at specified times.

CPU, network, memory For all components beside the disk the following is done to build be playable model: For each measuring point the loads are read from the profile and assigned to the a new load generator for each component. The overhead load is increased by the load specified by the profile and decreased by the really generated load. The network load is added to a global PlayerThread. For each task that is new a new PlayerTask and for each new thread a PlayerThread is created. The threads are assigned to the tasks as appointed by the profile. Each task contains an utility thread used to replay the memory load. The CPU load of a thread can directly be assigned to the thread by adding the CPULoadGenerator to the thread by calling `addWorkload(load, delay)`. The delay specifies how long after the creation of the thread the generater shall generate the load.

Disk The disk load modeling function needs to know about all load that should be generated before assigning load to a file. This makes it impossible to generate the load for each measuring point separately. Therefore there is specialized modeling system for the disk.

While setting up the other loads this system records which disk loads need to be generated. After collecting all loads the disk modeling is started. The result of the modeling is a set of DiskLoadPlayerThreads containing the DiskLoadGenerators. For each measuring point one thread is responsible for generating the load for one file. This file is stored in a `std::fstream*` passed to the generators. If a generator is generating a file-close this pointer is set to `NULL` and if it generates a file-open the pointer is replaced by a pointer to the file's stream.

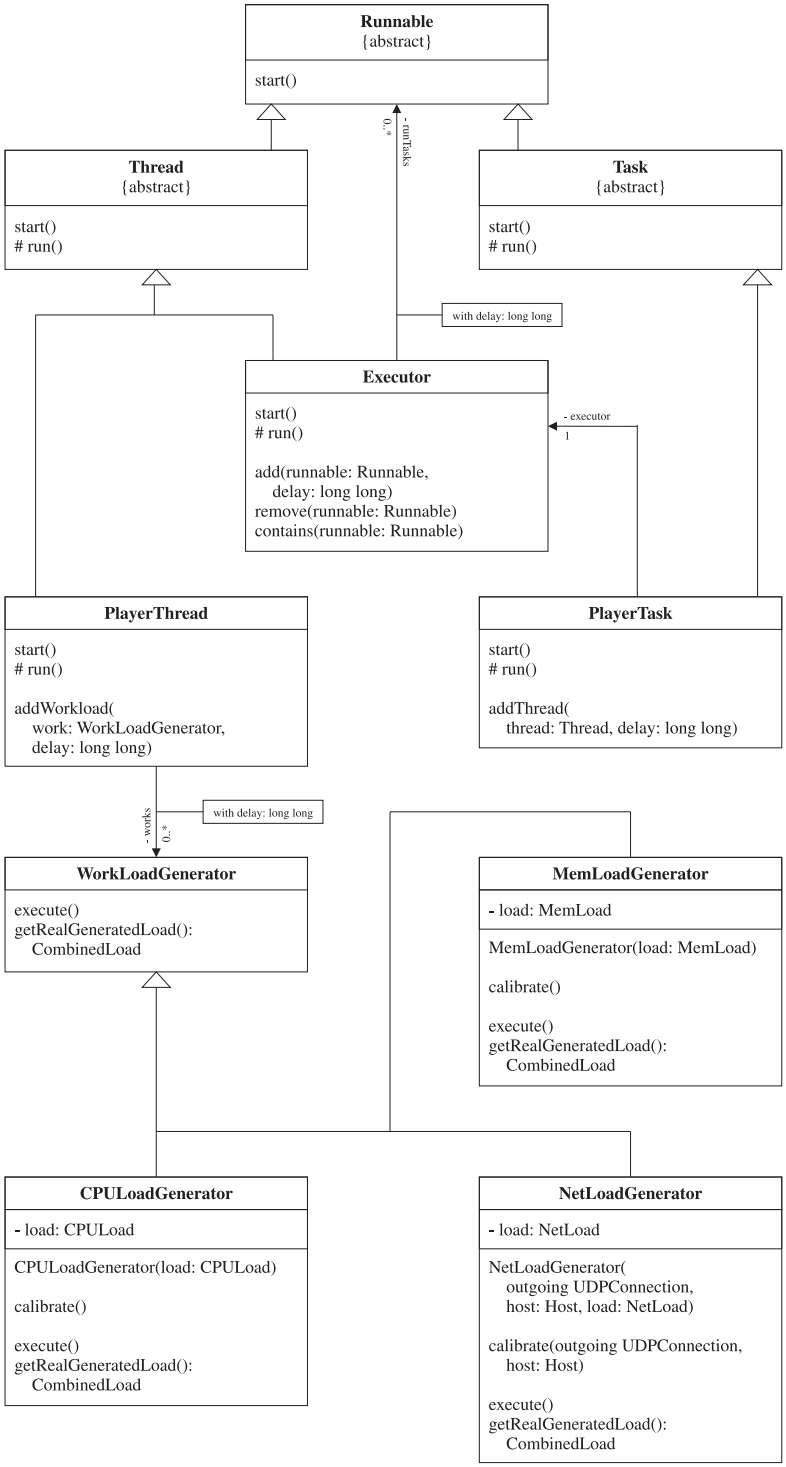

Figure 11: Class diagram of the runnable model. The runnable disk model is described by Figure 12.

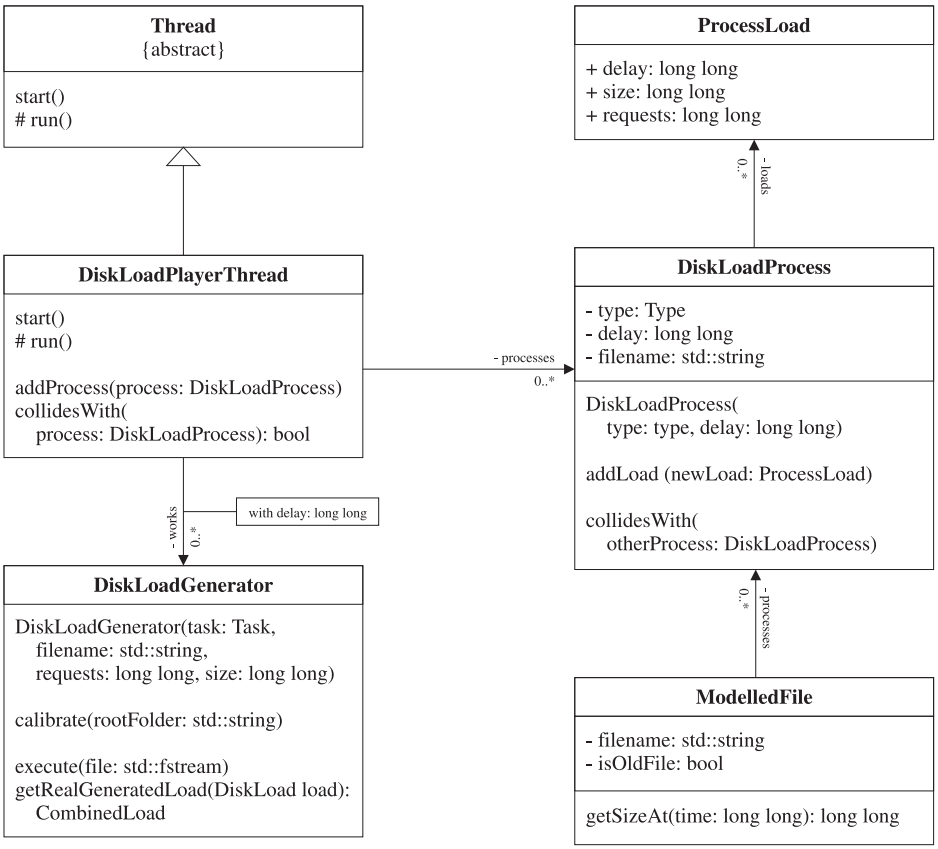

Figure 12: Class diagram of the runnable disk model. The model of the other components is Figure 11. The Thread-class is the same as in Figure 11.

The DiskLoadProcess is an abstraction for the disk-load-modeling's process. It's and the DiskLoadPlayerThread's `collidesWith(process)` functions assure, that at no point of time two processes are assigned to the same file or thread. The process contains a set of consecutive DiskLoadProcessLoads describing the load to be generated during the lifetime of the process. When adding the process to a thread by calling `addProcess(process)` this information is translated into an open-generator, a set of read- or write-generators and a close-generator. The type of generator is stored in the generators `Task`-property.

In the end all threads modeling the disk load are added to the `Executor` of the main player task. If there are many files used in the original workload this results in a lot of threads. Most operating systems limit the thread count. If some threads cannot be created the replaying process fails.

Playing To replay all tasks and threads, all modeled tasks and the utility threads for disk and network replaying are added to a main executor in a main task. Then it's `start()` function is called, starting the tasks and threads after their delay. Each task contains it's own executor starting the memory utility thread and it's CPU load threads. After all loads has been generated the main tasks finishes.

5.4 Player network client

Network load is generated as UDP traffic between two computers. The outbound traffic is generated on the system under test itself and sent to a client computer. This computer is running the player network client. It is a very simple program replaying network load sent to it by the player. The communication between the player and the player network client is outlined in Figure 13.

After connecting to each other the client is calibrated. From this calibration the client determines the network delay. As UDP is used for the calibration this step has to be explicitly be finished by the player. In the next step the NetworkLoad-Generator of the player-application is calibrated. Before replaying the workload the incoming network load is sent to the client. It stores this information to send it back during replaying. Finally the start time of the replaying process is sent to the client and both programs wait to replay the load. During the replaying process both programs behave independant.

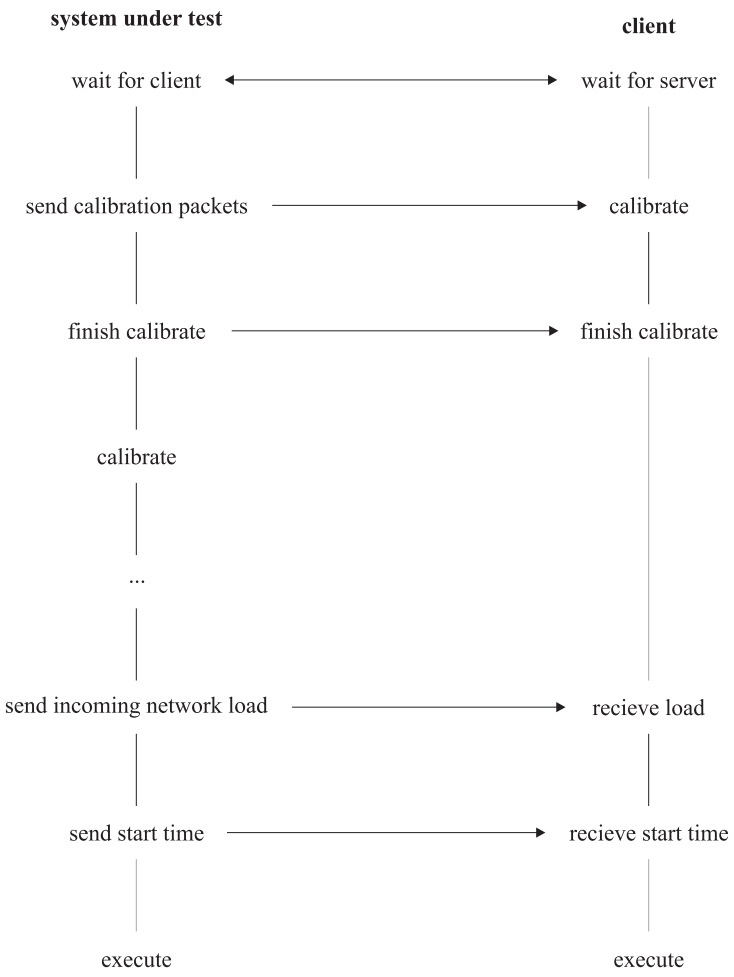

Figure 13: Communication between client and player-application

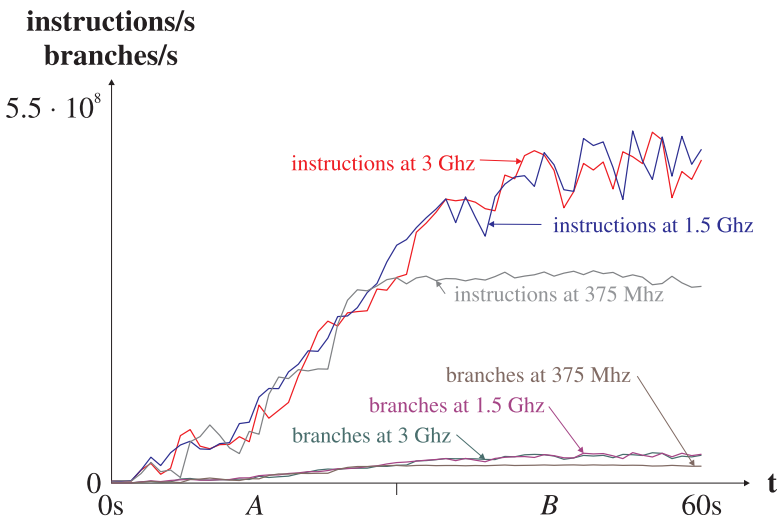

Figure 14: Three profiles of CPU load behavior on different machines. The profile of the 375 Mhz run is not invariant. The profiles were recorded using a deterministic workload on a CPU at 3 Ghz, and the same CPU clock modulated to 1.5 Ghz, or 375 Mhz. Clock modulation regularly suspends the program execution for short time interval. This results in a "fake" lowered frequency.
Hardware CPU: Intel Pentium D 3 Ghz (clock modulated to 3 Ghz, 1.5 Ghz, or 375 Mhz), Memory: 2 GB, Network: Intel e100. Disk: Samsung HD080HJ *Software* Apache2, PHP 4.4. *Load* Deterministic access to a small set of dynamic web pages.

6 Evaluation

To evaluate my system I attached an AC-measurement device to a desktop derived server. The measurement device was a *Voltcraft Plus Digital Multimeter VC-940* having a measurement accuracy of $\pm$ 1.2 % for 230V AC current. The system under test contained an Intel Pentium D 3 Ghz, 2 gigabyte of DDR2 memory, an Intel e100 and an e1000 network adapter, and two disks (Maxtor 6Y160P0, Samsung HD080HJ). The operating system used was a Debian sid running a Linux 2.6.22.1 kernel patched with my VFS extension. The AC measurement device reports the current power consumption of the whole machine once per second. These values were recorded using Labview 7.1. All graphs in this section are using linear scales.

6.1 Invariance

Each component can handle a certain maximum capacity of load. If more requests are issued than can be handled the surplus is either postponed or canceled. If one of these distorted profiles would be replayed on another machine the power

consumption would differ from the consumption of the real workload, i.e., the distorted profile is not invariant (as shown is Figure 14, section *B*).

A profile distorted by overloading has the following characteristics:

- The load rises to the maximum capacity of the component and remains there.
- If the outstanding requests are canceled because of timeouts, less work executed. This will result in a decreased performance of the server. As energy efficiency is dependent of the performance, this effect makes two machines incomparable by my system.
- If the requests are not canceled, they are processed later. When the load to be handled decreases the piled up requests will be served. This will spoil the timing behavior of the profile. As timing behavior has a strong influence on power management techniques the power estimation process becomes inaccurate.
- The overloading of a component also influences the metrics not directly related to it. If the CPU gets overloaded less requests are processed and therefor less answers are send back to the client. This results in less network packets sent.

Therefore the workload has to be profiled on a machine having no or only very short phases of overloading.

6.2 Workload replay

According to the performance oriented approach the workload is replayed accurately if the metrics are replayed at the same rate at the same time. The player-application of my system replays the disk, memory, network and OS-related metrics straight forward. E. g., network requests are directly issued by the player with the size determined for the current measuring point. The only metrics not directly replayed are the CPU metrics. I use the small program approach to replay the load (Section 4.3.3 on page 25).

This works well in most cases (see section *A* and *D* in Figure 15). The small programs have to be able to be combined to all different types of workload. As I use only two metrics to describe the CPU load the type of load is described by their quotient $\frac{branches/s}{instructions/s}$. My system can create load with a quotient between 0.2 and 0.04. If there are measuring points having another quotient as much load as possible is replayed without using more instructions or branches as described in the profile. Consequently, the replaying becomes inaccurate (see sections *B* and *E* in Figure 15).

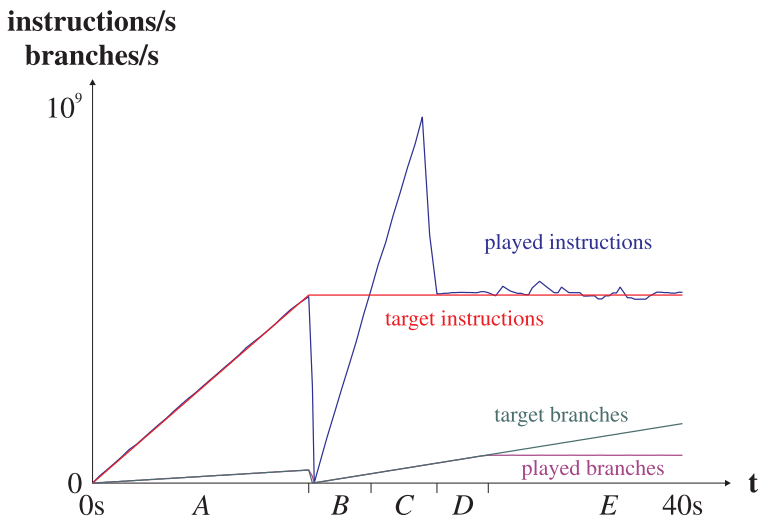

Figure 15: Generating CPU load. Replaying an artificial profile with occasional very high (0.3) and low (0) $\frac{branches/s}{instructions/s}$ quotient.
Hardware CPU: Intel Pentium D 3 Ghz *Load* Artificial load.

There are two ways to handle this situation. The unreplayable surplus can be discarded or replayed afterwards. As each instruction and branch uses a certain amount of energy, discarding them results in an inaccurate power consumption. My system stores the surplus in a special "non respected load" value and tries to replay is as soon as possible. E.g., in section *B* of Figure 15 some instructions could not be generated and are kept back. In section *C* they are replayed. The non respected load collected in section *E* is not replayed as the profile ends before it can be replayed. In this case keeping the timing behavior is more important.

6.3 Power prediction

The profiling and replaying process are accurate, beside the limitations described above. I used a lot of so called benchmarking magic – i.e. rules of thumb – to bridge the gap between these two steps. I tried to make my system as universal as possible, but if I had to make a decision I tried to mimic the behavior of the software im my scope (see Table 2).

The goal of my system is to replay a previously (on another machine) recorded profile with the same power consumption as if the real workload would have been executed. To evaluate the prediction accuracy I profiled the workload on a machine; I measured the power consumption of the real workload on it and compared it

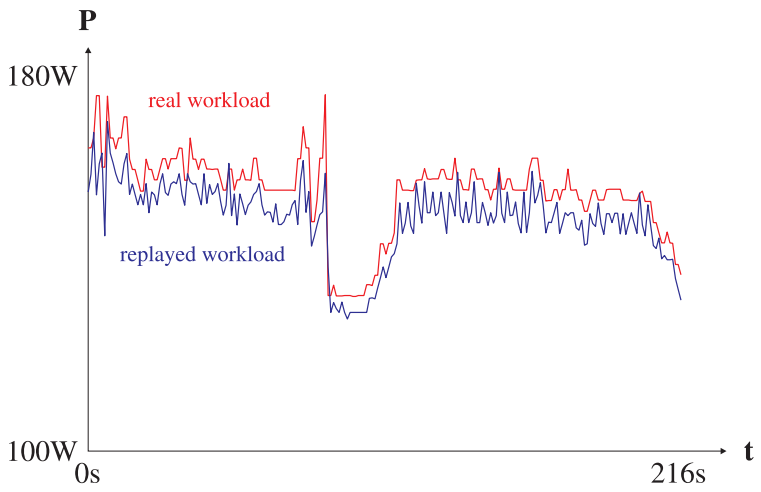

Figure 16: Power consumption of a webserver. Prediction error is 3.7 %.
Hardware CPU: Intel Pentium D 3 Ghz, Memory: 2 GB, Network: Intel e100. Disk: Samsung HD080HJ *Software* Apache2, PHP 4.4. *Load* Random access to a small set of dynamic webpages.

to the power consumption when replaying the profile. I aim to keep total power consumption accurate but also the behavior of the power consumption over time has to be considered.

Figures 16 and 17 compare the power consumption of the real workload to the replayed. For both workloads the power prediction error is below 4 %. The minor inaccuracy is caused by the insufficient memory modeling (see Section 4.5.2 on page 32). The precise power prediction proves the benchmarking magic being suitable for these workloads.

There are also cases where the benchmarking magic does not fit to the real behavior of the workload. Figure 18 shows the power consumption graph of a video transcoding and streaming workload. Video transcoding algorithms are special in several ways: They are highly optimized for optimal caching behavior and use special multi media instructions as MMX, SSE, etc... My benchmarking magic is using a lot of C++ std-lib code, especially the container-classes. Those are not used by the transcoding algorithms using their own custom tailored memory management. The timing behavior of a streaming server is replayed correctly, but the prediction inaccuracy for the streaming test case is 12 %.

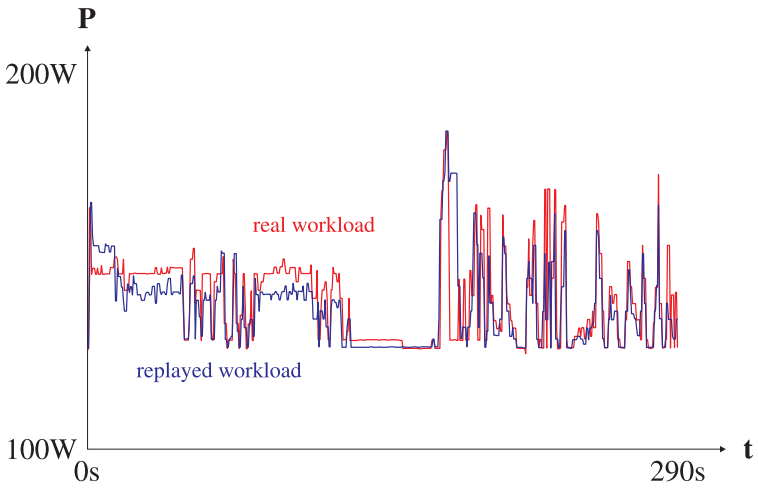

Figure 17: Power consumption of a file sync. Prediction error is 1.4 %.
Hardware CPU: Intel Pentium D 3 Ghz, Memory: 2 GB, Network: Intel e100. Disk: Samsung HD080HJ *Software* `rsync` with compress option. *Load* Syncing and resyncing two linux kernel source trees and 4 big files.

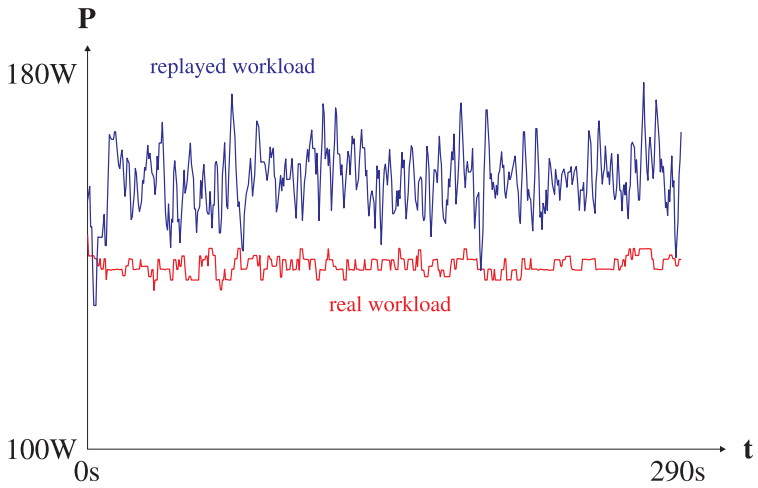

Figure 18: Power consumption of a video stream. Prediction error is 12 %.
Hardware CPU: Intel Pentium D 3 Ghz, Memory: 2 GB, Network: Intel e100. Disk: Samsung HD080HJ *Software* VideoLanClient, ffmpeg. *Load* Transcoding and sending a video.

6.4 Dodging mispredictions

My system does not work well if the real workload has certain characteristics. Whether a system is overloaded or not can easily be determined at run time. If it is overloaded the real workload should be profiled on a different, more powerful machine. Long phases of uncommon CPU load can be detected by investigating the profile. Replaying them correctly can be achieved by adding another small program with the adequate $\frac{branches/s}{instructions/s}$ to the CPULoadGenerator class.

Whether the benchmarking magic used in my implementation suits the behavior of the workload or not, cannot be detected without measuring the power consumption of the real workload. If the power consumption of the real workload can be measured and if there is a system available for replaying the workload, the accuracy of the benchmarking magic can be appraised and altered if available.

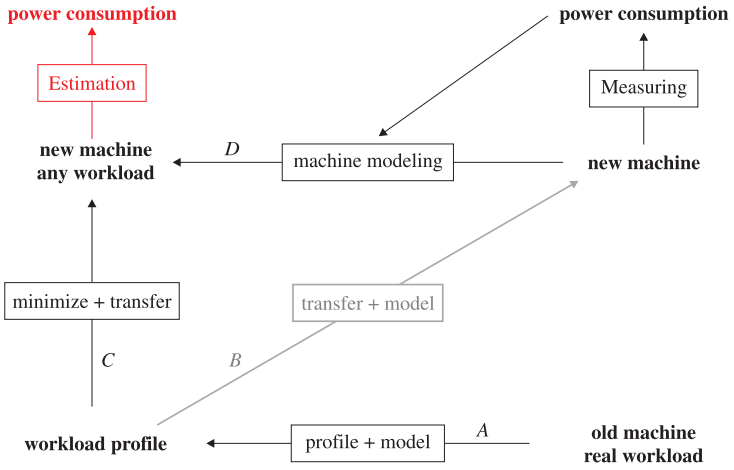

Figure 19: Four points design. The final power estimation is printed in red.

7 Future Work

To make my system more universal and to increase it's usage comfort I propose two extensions. First, my current system can be supported by a power model for the new machine. This will reduce the effort for the power estimation from a measurement to a computation. Second, if the system should be extended to more operating systems and hardware platforms the profiler and player application needs to be extended.

7.1 Power consumption estimation by system modeling

Adding a machine model to the three points design developed in Chapter 4 will result in a design as shown in Figure 19. The resulting metric is power consumption, too, but it will be retrieved from the previously created model for the full machine (new machine, any workload). Using this model makes it possible to estimate the power consumption of a profile without time consuming power measuring for each workload. Dependent of the type of machine model the estimation will be more or less accurate. The disadvantage of using a system model is that creating it (path *D*) can be complex and that the estimation will be less accurate.

Extending Zesti Zesti is a system following the four point design as in Figure 19. As described in Section 2.3.2 (page 9), the old machine has to be the same as – or at least very similar to – the new one. However, Zesti could be (mis-)used to estimate energy for other machines. To do this, the metrics have to be chosen according to my design rules (Table 3) including the invariant constraint.

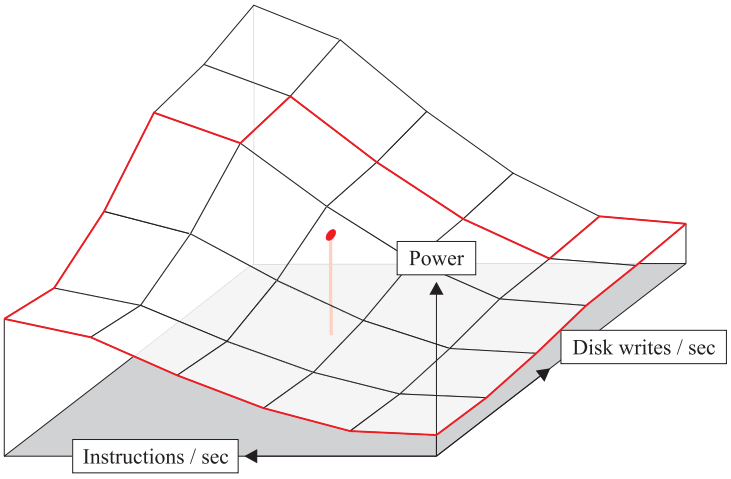

Figure 20: Energy estimation from minimized metrics. Possible combinations are framed in red. The red dot is the energy usage at a specific combination of the two metrics.

Zesti was not designed to use invariant metrics and it can be assumed that the estimation accuracy will suffer from this constraint. As it's machine model (Equation 2 on page 11) only depends on the current measuring point and does not respect the usage history of a ressource, many power saving technologies depending on long break even times cannot be respected. It might be possible to make the model dependent on a history of measuring points. This would make the energy weight retrieving process much more complicated and will still not assure that all power saving features will be respected.

System model based on workload characterization Energy saving features have a big impact on the power consumption of lightly loaded components. Most workloads in my scope have phases of light load. As the extension of Zesti could be complex, I propose another type of model. This new full system power model uses workload characterization to estimate the power consumption of a profile.

The model takes so called *minimized metrics* as input. These are single-value characteristics of a metric. The average instructions/sec are such a metric, because all measurements of the metric in the whole profile are contracted in one value. Such metrics can be produced by workload characterization techniques (Section 2.2).

For each combination of the values of the chosen minimized metrics, the energy consumption is measured. If there are two metrics and five steps per metric should be used to create the model, there are 5^2 possible combinations. This leads to a multidimensional graph as in Figure 20 describing the power consumption for all profiles. There are some combination of values that are impossible and do not need

to be measured. If the minimized profile's combination of values was not measured it has to be interpolated from the nearest measurements.

The advantage of this approach is that it respects most power saving features. The drawback is that it needs many measurements. If there are five minimzed metrics and five values per metric are measured and each measurements take five minutes, the model creation process takes $5^5 \cdot 5\ \textit{min} = 11\ \textit{days}$.

7.2 Generalization

This section outlines what has to be done to extend my system to support more operating systems, more hardware platforms and additional kinds of workload.

Software The profiler encapsulates all OS-specific parts in subclasses of the `System` class. For each new operating system one of these subclasses has to be added. It is responsible for reading the metrics from the OS or computing them, if they are not directly readable (as thread load in Linux).

The problem with my current implementation for Linux is it's dependency on the procfs-extension. It is not part of the standard or any distributation's kernel. So it would not be used in any productive environment. If a major server manufacturer wants to use a system as mine, there are two possibilities: Either the procfs has to be extended by my patch, or an additional, better structured, and more precise documented procfs has to be created.

The player is dependant on the POSIX thread library and POSIX style file access, the Linux style directory browsing, and network handling. Most of these functions are encapsulated in the thread and net libraries of my project and can conveniently be replaced.

Hardware The prototype currently uses only the Pentium 4's performance counters. All other server processors also feature performance counters that report similar metrics. As they are accessed differently an abstractionizing infrastructure is needed. Libraries such as PAPI have already solved this issue [BDG+00]. They are built for computation performance measurement not for energy measurement. So these libraries have to be revised if they obey to my design rules.

Load Despite all attempts to record and replay the load as identical as possible I had to use a lot of benchmarking magic. I tried to make my system fit to as much loads as possible, but there are some loads that are replayed badly. Adapting or replacing the modeling can be yielding if the type of workload to be replayed is known. If nothing is known about the profiled workload adapting the model

would lead from benchmarking magic to another – maybe worse – magic. The only enhancement for all types of workload is to find more suitable metrics.

8 Conclusion

The direct and indirect power consumption of servers summed up to 14 gigawatts in 2005. This consumption does not need to be that high. Most of the servers used are equipped with cheap or badly sized components.

To support the acquisition process to choose well sized, energy efficient servers it has to be described how energy efficient a server is. This can either be done by providing one value describing the energy efficiency of a server (as by SPECpower) or by estimating the energy efficiency for a given load (as by Zesti or my system).

In this thesis I described how to design and implement a system built to predict the power consumption of a server from a profile of a real workload recorded on any other server. I propose this system to be used by server manufacturers to give small companies a hint about the power consumption of various servers for the company's real workload. By comparing the reported consumptions the company can choose the most energy efficient server. The scope of this work were small and desktop derived servers running standard server applications as web- and email servers.

8.1 Energy prediction process

I propose an energy prediction process that can be divided into three steps.

Profiling The real workload has to be profiled. I basically followed the performance oriented approach. I record only those metrics obeying to a set of rules. They have to be related to the energy consumption of the server and have to be invariant to machine or OS specifica. The invariance constraint ensures that the profile recorded on one machine is the same one as if it would have been recorded on another machine under the same workload. This is important because otherwise the profile would not be relevant for other machines.

If the profiled server has long phases of overload, invariant metrics loose this characteristic. Hence, if the original server is often overloaded the profile has to be recorded on another more powerful machine.

There are types of system differences making the invariance constraint too strict. If all types of instruction set architectures should be considered there is no invariant metric for CPU load. I therefore assumed that the profiled server and the server, the profile is being replayed at, do not differ too much. The following characteristics have to be equal: ISA, word width, type of disk (network or directly attached), type of network (Ethernet, Token ring) and page size. The profile has to be postprocessed to adapt the profile to, e.g., a new word width.

Modeling The power prediction is made by measuring the power consumption of the replayed workload on another machine. I created a set of workload generating classes for each component in the server. If the profile did not describe the workload to be generated accurately enough, I had to make some assumptions on the workload's behavior. I refer to these assumptions as benchmarking magic, as it cannot be proved that they suit to every kind of real workload.

Replaying and power measurement There are some cases when even the outline of the workload cannot be replayed accurately. This happens if (1) a component's load shows an untypical behavior and (2) it's load is generated by the small process approach. To solve this issue the load generating program must be extended to support the additional behavior.

8.2 Achievements

In this work I showed that it is possible to predict the energy usage of machines from a profile recorded on another machine. My system can accurately predict the power consumption of a server for the real workload of a company. This prediction helps a company to aquire well sized and energy efficient servers. Using these server will lead to a reducement of the total costs of ownership of the machines and will also help to protect the environment.

To evaluate the total prediction accuracy of my system, I compared the power consumption of a real workload with the consumption of the replaying of it's profile.

Two workloads showed a prediction inaccuracy of below 4 %. The workload of these test cases were a webserver and a file synchronization. These two softwares are part of my software scope so the benchmarking magic worked well for them. I also measured a workload that is not in my scope. Consequently, my benchmarking magic did not suit to it. Transcoding and sending a video stream uses a lot of specialized instructions and a custom tailored memory management. The prediction inaccuracy of this test case was 12 %.

My system worked well for the softwares in my scope. For softwares being not in my scope the results showed that the prediction accuracy, despite being guided by the profile, is still dependent on the type of benchmarking magic used. This can be solved by finding more invariant metrics. These will describe the workload in more detail and allow to use less benchmarking magic.

Appendix

A Definition of metrics

OS-abstractions

Metric	Description	Granularity
Tasks	Currently running tasks including all utility tasks of the operating system (like cron, the shell, etc...) and the profiler.	system
Threads	Currently running threads including all threads of all OS-utility tasks and the profiler thread.	task

CPU

Linux 2.6 does not report CPU load in thread granularity. Therefore, a task's CPU load is assigned equally to it's threads.

Metric	Description	Granularity
Instructions	Counts all instructions executed in user mode in between two measuring points. Speculatively executed instructions while a branch misprediction are not counted. On a Pentium 4 (and similar CPUs) this information is read by setting up the CRU_ESCR0 (0x3B8) with `2<<25 + 0xF<<9 + 4` and setting up the IQ_CCCR0 (0x36F) with `3<<16 + 4<<13 + 1<<12`. The instructions can then be read from IQ counter 0 (0x30C).	thread
Conditional branches (also referenced to as branches)	Counts conditional branch instructions (`JE`, `JNE`...) in usermode in between two measuring points. Speculatively executed branches while a branch misprediction are not counted. On a Pentium 4 (and similar CPUs) this information is read by setting up the TBPU_ESCR0 (0x3C2) with `4<<25 + 2<<9 + 4` and setting up the MS_CCCR0 (0x364) by `3<<16 + 2<<13 + 1<<12`. The branches can then be read from MS counter 0 (0x304)	thread

Network

Metric	Description	Granularity
Bytes sent	Bytes send to any computer by any network adapter including all headers.	system
Packets sent	Packets send to any computer by any network adapter.	system
Bytes received	Bytes received from any computer by any network adapter including all headers.	system
Packets received	Packets received from any computer by any network adapter.	system

Disk

The profiler only counts accesses to the real filesystem. It records all accesses independently if they are served by the disk cache or the disk itself.

The LinuxP4 class and the Linux VFS extension counts accesses to filesystems that have set the FS_REQUIRES_DEV flag. Virtual filesystems as the proc- and sysfs do not set this flag.

Metric	Description	Granularity
Opens	Counts all opens of real files on any disk. Using the LinuxP4 class and my VFS extension this metric counts occurrences of the POSIX command `open()`.	system
Closes	Counts all closes of real files on any disk. Using the LinuxP4 class and my VFS extension this metric counts of the POSIX command `close()`.	system
Bytes written	Counts all bytes written to any location of any disk. Using the LinuxP4 class and my VFS extension this metric counts the bytes written by the POSIX commands `write()` and `writev()`.	system
Write requests	Counts write requests issued to any file of any disk. Using the LinuxP4 class and my VFS extension this metric counts the POSIX commands `write()` and `writev()`.	system
Bytes read	Counts all bytes read from any location of any disk. Using the LinuxP4 class and my VFS extension this metric counts the bytes read by the POSIX commands `read()` and `readv()`.	system
Read requests	Counts read requests issued to any file of any disk. Using the LinuxP4 class and my VFS extension this metric counts the POSIX commands `read()` and `readv()`.	system

Memory

Metric	Description	Granularity
Memory size	KByte currently allocated by the task. This includes automatically allocated regions (code, data, stack) and volitionally allocated memory (heap, libraries). Libraries shared by more than one task are only counted once.	task
Working set	KByte recently used. “Recently” is defined by the implementation of the profiler and player. The LinuxP4 class reading the metric on a Linux system reports the recent set instead of the working set (see Section 4.5.2). The player touches all pages at each measuring point.	task

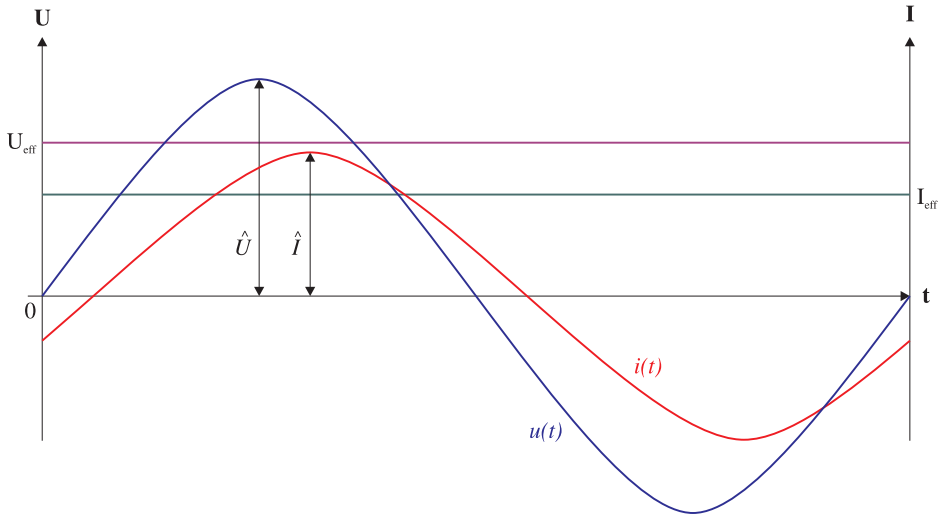

Figure 21: Alternating current: $p(t) = u(t) \cdot i(t)$

B AC power measurement

The alternating current in Europe usually looks like $u(t)$ in Figure 21 with an effective voltage U_{eff} of 230V and a frequency of 50Hz. The measuring device takes samples of $u(t)$ and $i(t)$ to compute the used power $p(t)$ for a specific time as

$$p(t) = u(t) \cdot i(t)$$

As $u(t)$ and $i(t)$ are dependent on the frequency ω and the phase angle φ they can be expressed as

$$\begin{aligned} u(t) &= \hat{U} \cdot \cos \omega t \\ i(t) &= \hat{I} \cdot \cos\left(\omega t + \varphi\right) \end{aligned}$$

Consequently

$$\begin{aligned} p(t) &= \hat{U}\hat{I} \cdot \cos\left(\omega t + \varphi\right) \cdot \cos \omega t \\ &= \frac{\hat{U}\hat{I}}{2} \cdot \cos \varphi \cdot (1 + \cos 2\omega t) - \frac{\hat{U}\hat{I}}{2} \sin \varphi \cdot \sin 2\omega t \end{aligned} \tag{8}$$

Equation 8 can be split into two parts

$$p_{real}(t) = \frac{\hat{U}\hat{I}}{2} \cos\varphi \cdot (1 + \cos 2\omega t)$$
$$p_{reactive}(t) = \frac{\hat{U}\hat{I}}{2} \sin\varphi \cdot \sin 2\omega t$$

The reactive power is swinging around 0 and the maximum energy contributed is

$$\max_{t_0,t_1 \in \mathbb{R}} \int_{t_0}^{t_1} p_{reactive}(t)\, dt = \frac{\hat{U}\hat{I}}{4\omega} \cdot \sin\varphi$$

Consequently $p_{reactive}$ can be seen as a nonrelevant error and the energy consumed W is

$$W = \int p_{real}(t)\, dt$$

AC power meters as used by SPEC Power [SPE07] and in this thesis report one value per second.

List of Figures

List of Tables

References

[AGVO05] J. Abella, A. González, X. Vera, and M.F.P. O'Boyle. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):55–77, 2005.

[BAB96] D.C. Burger, T.M. Austin, and S. Bennett. *Evaluating Future Microprocessors: The SimpleScalar Tool Set*. University of Wisconsin-Madison, Computer Sciences Dept, 1996.

[BC05] D.P. Bovet and M. Cesati. *Understanding The Linux Kernel*. O'Reilly, third edition, 2005.

[BDG+00] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. *Supercomputing, ACM/IEEE 2000 Conference*, pages 42–42, 2000.

[Bel01] Frank Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. Technical Report TR-I4-01-11, University of Erlangen, Germany, December 14 2001.

[BKW03] Frank Bellosa, Simon Kellner, and Andreas Weissel. Event-driven energy accounting for dynamic thermal management. Technical Report TR-I4-03-02, University of Erlangen, Germany, July 30 2003.

[BM01] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. *hpca*, 00:0171, 2001.

[BTM00] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, 2000.

[CS93] M. Calzarossa and G. Serazzi. Workload characterization: a survey. *Proceedings of the IEEE*, 81(8):1136–1150, 1993.

[DER06] Christos Kozyrakis Dimitris Economous, Suzanne Rivoire and Parthasarathy Ranganathan. Zesti: Full-system power modeling and estimation. Under review, available from Parthasarathy Ranganathan's webpage, 2006.

[EM02] S. Elnaffar and P. Martin. Characterizing computer systems' workloads. *Submitted to ACM Computing Surveys Journal*, December 2002.

[Ene07] Energy Star. *Energy Star Program Requirements for Computers 4.0*, July 20 2007.

[Gae81] S. L. Gaede. Tools for research in computer workload characterization and modeling. *Experimental Computer Performance Evaluation (North-Holland, 1981)*, pages 235–247, 1981.

[GMT+06] S. Greenberg, E. Mills, B. Tschudi, et al. Best practices for data centers: Lessons learned from benchmarking 22 data centers. *ACEEE Summer Study on Energy Efficiency in Buildings*, 2006.

[GRA+03] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R.C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–36, 2003.

[GSKF03] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: dynamic speed control for power management in server class disks. *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 169–179, 2003.

[Hew06] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd. and Toshiba Corp. *ACPI Specification 3.0b*, October 10 2006.

[Hew07] Hewlett-Packard Corp. *HP Power Calculator revision 0.05*, 2007.

[HPB02] T. Heath, E. Pinheiro, and R. Bianchini. Application-supported device management for energy and performance. *Proceedings of the 2002 Workshop on Power-Aware Computer Systems*, pages 114–123, 2002.

[HPS03] H. Huang, P. Pillai, and K.G. Shin. Design and implementation of power-aware virtual memory. *Proceedings of the USENIX Annual Technical Conference*, June 9 – 14 2003.

[Int03] Intel Corp. *Events on Intel Pentium 4 Processors, Including Hyper-threading Technology-Enabled Processors*, July 2003.

[Int04] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2004.

[Kel03] Simon Kellner. Event-driven temperature control in operating systems. Study thesis, University of Erlangen, Germany, December 1 2003.

[Koo07] J. G. Koomey. Estimating total power consumption by servers in the US and the world. Technical report, Lawrence Berkeley National Laboratory, February 2007.

[Mer05] Andreas Merkel. Balancing power consumption in multiprocessor systems. Diploma thesis, System Architecture Group, University of Karlsruhe (TH), Germany, September 30 2005.

[Mog99] J.C. Mogul. Brittle metrics in operating systems research. *The Seventh Symposium on Hot Topics in Operating Systems*, 1999.

[Mol06] Philip P. Moltmann. Energy estimation for cpu-events dependent on frequency scaling and clock gating. Study thesis, System Architecture Group, University of Karlsruhe (TH), Germany, September 1 2006.

[PBCH01] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, September 2001.

[Rao96] S.S. Rao. *Engineering Optimization: Theory and Practice*. Wiley-IEEE, 1996.

[Raw04] F. Rawson. MEMPOWER: A simple memory power analysis tool set. Technical report, IBM Austin Research Laboratory, January 2004.

[Sch97] R. R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[Ser05] Server Systems Infrastructure. *Power Supply Management Interface Design Guide Revision 2.12*, 2005.

[SPE07] SPECpower_ssj2008 user guide, scheduled for the second half 2007.

[WZPM02] H.S. Wang, X. Zhu, L.S. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. *35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 294–305, 2002.

[ZSG+03] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. *Proceedings of Second Conference on File and Storage Technologies*, March 2003.