

Erfassung und Abrechnung des Energieverbrauchs in Verteilten Systemen

Studienarbeit im Fach Informatik

vorgelegt von
Torsten Ehlers
geboren am 26. August 1976 in Bassum

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

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

Beginn der Arbeit: 1. Juni 2003
Abgabedatum: 1. März 2004

Erklärung

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

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

Erlangen, 1. März 2004

Transparent Energy Accounting in Distributed Systems

Studienarbeit

by

Torsten Ehlers

born August 26th 1976 in Bassum

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

Advisors: Dr. Ing. Frank Bellosa
 Dipl.-Inf. Andreas Weiβel
 Prof. Dr. Wolfgang Schröder-Preikschat

Begin: June 1st, 2003
Submission: March 1st, 2004

Copyright © 2004 Torsten Ehlers.

Permission is granted to copy and distribute this document provided it is complete and unchanged.

Parts of this work may be cited provided the citation is marked and its source is referenced.

The programs described herein are also copyrighted by Torsten Ehlers. They are free software; you can redistribute them and/or modify them under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

Abstract

Power consumption is a crucial characteristic of modern hardware, both for mobile, battery-driven devices and for high-end servers. Servers are increasingly highly-integrated in modern data centers and the power density per unit area is rising. This also induces higher heat densities. Costs for electricity supply and for necessary cooling equipment are not insignificant anymore. Higher clock speeds and growing demand for always-on services will intensify this problem even more. Methods to account and limit power consumption on the application- or task-level for stand-alone hosts have been successfully adopted. However, those methods lack support for distributed systems.

This thesis introduces a transparent energy accounting scheme for distributed systems. The well-known abstraction of resource containers representing resource principals in a system is extended to global resource containers to allow accounting of energy dissipation across system boundaries. With this extension, energy consumed for the accomplishment of a certain task within a server cluster can be accounted to a resource container globally. When a server is working on behalf of a client, the server is bound to that client's resource container only until its work for this client is completed. Server and client processes do not have to reside on the same host for this scheme. Information is sent piggyback with normal IPv6 network traffic, transparently for the applications. This way it is possible to accurately account energy consumption to its originator, even if this originator does not exist on the same host. Limiting global resource containers and using them for priority models or thermal management of computer clusters is achievable as well and presented in this work.

As a prototype implementation a modified Linux kernel running on Intel Pentium 4 CPUs is presented and tested with several experiments that prove its effectiveness.

Contents

1	Introduction	1
1.1	The resource energy	1
1.2	The problem with energy accounting	2
1.3	Contributions of this work	3
2	Related work	5
2.1	Local energy accounting	5
2.2	Distributed energy accounting	5
3	Motivation	7
3.1	The need for energy accounting	7
3.2	Energy accounting in distributed systems	8
3.2.1	Load balancing in computer clusters	8
3.2.2	Accounting of energy usage	9
3.2.3	Thermal management of computer clusters	10
4	Design	13
4.1	Control of energy usage and temperature using resource containers	13
4.1.1	Accounting and control of energy usage	15
4.1.2	Temperature control	16
4.2	Distributed energy accounting	17
4.2.1	Extending the concept to cluster networks	17
4.2.1.1	Sending usage information between nodes	18
4.2.1.2	Extending resource containers by identifiers	19
4.2.2	Transparent energy accounting	19
4.2.2.1	The Internet Protocol version 6	20

5 Implementation	23
5.1 Aims of the implementation	23
5.2 Modifications to the Linux kernel	24
5.2.1 Global identifiers for resource containers	25
5.2.2 The network stack	26
5.2.2.1 Outgoing network packets	27
5.2.2.2 Incoming network packets	31
5.2.3 Accounting and throttling of resources	32
5.2.4 Changes to the kernel's API	33
5.3 User space programs	33
6 Evaluation	35
6.1 Testing environment	35
6.2 Tests of the implementation	36
6.2.1 Distributed compilations	36
6.2.2 Transitive energy accounting	37
6.2.3 Limiting energy consumption	39
7 Future work	41
7.1 Extending the infrastructure for transparent information exchange	41
7.2 Relocation of unused resources	42
7.3 Accounting of further resources	42
7.4 Profiling of resource containers	43
8 Conclusions	45
Bibliography	47

List of Figures

3.1	Typical data center cooling infrastructure [1] (adapted)	11
3.2	Temperature distribution along hot aisle before and after workload redistribution [1] (adapted)	11
4.1	Example scenario for a client connecting to a server cluster	18
4.2	Sequence of IPv6 Headers as defined in RFC 2460	21
5.1	Sample Destination Options Header carrying some TLV encoded options	30
5.2	Destination Options Header carrying only the Resource ID option	30
5.3	Handler for the option type Resource ID	32
6.1	CPU power consumption for Linux kernel build on host running distcc and remote hosts (top) and power consumption during local build (bottom)	38
6.2	CPU power consumption for Apache web server and PostgreSQL database during transitive energy accounting	39
6.3	CPU power consumption for Apache web server limited to 10 watts and unlimited	40

List of Tables

4.1	CPU usage in user and kernel modes for network intensive applications [2]	14
6.1	Cluster power consumption and time spent during distcc build of Linux kernel 2.6.1	36
6.2	Total CPU energy consumption for Linux kernel build on distributed hosts and on local host	38

Chapter 1

Introduction

This chapter starts off with a short description of the importance of energy accounting. It continues with a concise survey of the contents of this work.

1.1 The resource energy

Energy consumption of modern hardware is a crucial characteristic in many environments. For mobile devices, the importance of energy consumption is obvious. Those devices need to be operated by batteries and long battery lifetimes (i.e. the time a mobile device can be used before its battery has to be recharged) are desirable. Energy consumption is a concern for stationary devices as well. The high integration of modern server hardware in data centers with highly-densed form-factors leads to an increasing power density per unit area (from currently about $500W/m^2$ to about $3000W/m^2$ in the future) [3, 4]. Accounting and controlling energy consumption is of growing importance because higher energy consumption causes higher costs, shorter hardware lifetimes and more waste heat dispersion. To cope with the latter, costly cooling facilities are needed. The problem with waste heat is increasing with progressing hardware development and higher integration factors as well, for this induces higher heat densities that require more efficient cooling hardware. Furthermore, the growing importance of the Internet has led to a growing demand for always-on servers and the demand will increase further in the upcoming years. A forecast for the United States, for instance, predicts that new data centers being installed by 2005 will add 5 GW to the current power demand. This is approximately 10% of California's current power generating capacity [5].

Reducing energy consumption can be achieved by special hardware. Various low power devices such as central processing units for mobile devices are available. Apart from actually reducing the total amount of energy consumed, it is also of high interest to be able to control current power consumption and postpone energy-intensive tasks to a later time. A system to account and limit energy usage of certain tasks as a software-only solution is valuable because no expensive specialised hardware is necessary.

1.2 The problem with energy accounting

Common operating systems do not offer energy accounting possibilities. Accounting is usually limited to few resources such as CPU time and disk space. Moreover, resource accounting is focused on processes, which is inappropriate for certain use-cases. Modern web servers, for instance, are multi-threaded, i.e. a single process serves multiple connections using several threads. Threads achieve better performance because the costs for context-switching within the operating system are reduced. Process centric resource accounting cannot distinguish between those connections and hence, accounting of single connections becomes impossible. The underlying problem is that processes are the unit of protection domains and the resource principals, i.e. the entities to account resource usage to, at the same time. Banga [6] eliminates this problem by introducing *resource containers* as new resource principals and thereby allowing combination of several processes to one resource principal or several resource principals in one single process.

A resource container implementation that allows accounting of the resource energy for the Linux operating system by Waitz [7] is available. Resource containers can be assigned to certain activities, no matter how many activities a process works for. When a process works for some activity it is bound to the corresponding resource container. When it starts working for another activity this binding is changed and set to the respective resource container. The implementation detects client-server-relationships, i.e. a server's resource binding is reset when it receives requests from a new client. It is also possible to limit a resource container's usage in such a way that a process currently bound to a resource container that is out of resources needs to wait until new resources become available. This implementation does not, however, work in distributed systems. Client and server have to reside on the same host for the implementation to work. This is a major drawback,

because generally client and server will be working on different hosts, for example in a computing cluster.

1.3 Contributions of this work

This work describes an operating system concept to transparently account energy usage in distributed systems. This is achieved by using the resource container implementation by Waitz, adding a concept to make resource containers globally available within a server cluster and forwarding resource container identifiers between clients and servers over the network. Forwarding does not require extra messages but the information is sent piggyback with normal network traffic via IPv6 connections. The concept is transparent to the applications running on the modified operating system as well as to other operating systems communicating with this system. The concept provides a general infrastructure for piggyback messages and can be used for different messages as well.

Resource containers can also be limited to a certain power consumption per time interval. When a resource container is out of resources, processes that are currently bound to this container will be throttled until new resources become available. That way, certain tasks can be prioritised over others. Thermal management of computer clusters is possible as well because energy usage data from the resource containers can be mapped to heat dissipation. Thermal management is important because cooling hardware is responsible for a significant fraction of costs in modern data centers already [8]. The resource container concept allows limiting the maximum temperature of certain hosts by throttling the CPU and thus being able to cut off peak temperatures that might occur only rarely. That way, cooling hardware can be adapted to average heat dissipation, rather than to maximum heat dissipation.

A prototype implementation for the Linux kernel that runs on Intel Pentium 4 CPUs is provided, the kernel's system call interface was extended and user programs that allow management of the system are made available as well.

The rest of this work is organized as follows: Chapter 2 gives an overview of related work. Chapter 3 presents the motivation for this work. It describes why distributed energy accounting is useful and shows some examples where it can be

used sensibly. In chapter 4, the designs of the accounting system and the underlying systems are described and the reasons for choosing this design are explained. Chapter 5 describes the actual implementation while chapter 6 presents the results of some tests that show the operativeness of the implementation. It also gives some information about energy usage of certain tasks. In chapter 7, possible future extensions to the existing work are outlined and chapter 8 summarises the results of this work.

Chapter 2

Related work

This chapter gives an overview of related work. Starting with energy accounting for single machines it continues with methods for accounting of energy for several hosts.

2.1 Local energy accounting

The general concept of resource containers as an abstract resource principal has been introduced by Banga [2]. Banga's work uses web servers as an example for services that require this abstraction for correct accounting. Later, Bohrer et al [9] have shown that power management for web servers is worthwhile because they mainly dissipate energy by using the CPU. In this work, some web server experiments are performed as well.

Bellosa demonstrated in [10] that energy consumption can be derived from data gathered from the CPU's performance counters. Kellner [11] presented an implementation that allows mapping of performance counter data to CPU temperature as well. A resource container implementation that relies on this performance counter data to calculate energy consumption was presented by Waitz [7]. Both implementations are limited to single machines.

2.2 Distributed energy accounting

Another concept that uses resource containers to achieve performance isolation of certain tasks by setting limits to local containers are *Cluster Reserves* as presented by Aron et al [12]. Cluster Reserves are defined as the resource principal

in clusters and are therefore similar to the global resource containers introduced in this work. Cluster Reserves guarantee a certain minimal proportion of cluster resources to every class of requests or *service class*. This remains true even if the total system load induced by other requests is high. The distribution of resources to the Cluster Reserves is computed by solving a constrained optimization problem.

Chase et al [8] studied the management of energy and other resources in data centers using an economical approach with their architecture *muse*. Services are considered customers that *bid* for resources while a central management instance, the *executive* balances the cost of the energy usage against the benefit of employing it. Servers under control of the executive can be turned on and off automatically to adjust to current resource demands. Muse uses resource containers and can limit maximum load. This allows adaption of cooling hardware to average cases rather than to maximum load. Muse has been implemented as a kernel module for the FreeBSD operating system [13]. Another possibility to avoid peak loads and therefore decrease the requirements for cooling hardware is dynamic voltage- and frequency-scaling, as described in [14].

The tradeoff between power and performance has been studied by Pinheiro et al [15]. They examined spreading the work evenly over available cluster resources (*load balancing*) and concentrating the work on fewer nodes to be able to turn the idle nodes off (*load unbalancing*).

Finally, Pinheiro et al have developed an operating system for load balancing in computer clusters called *Nomad* [16]. This system disseminates the load information across the cluster without extra messages by sending the information piggyback with distributed file access messages. Nomad has been implemented by modifying an existing Linux kernel.

Chapter 3

Motivation

This chapter characterises the motivation for this work. The need for energy accounting is illustrated and situations in which a concept for distributed energy accounting can be of use are outlined.

3.1 The need for energy accounting

In recent years the number of mobile devices has risen significantly. Additionally, those mobile devices have become more and more powerful over time. For instance, today's mobile phones are not only for making phone calls but also for taking pictures, sending and receiving multimedia-messages etc. The introduction of the new standard Universal Mobile Telecommunications System (UMTS) will add even more possibilities mobile phones can be used for. A major problem of all these devices is their energy consumption. While more and more possible fields of application cause increasing energy consumption, batteries cannot keep up with the growing demand satisfactorily. Nevertheless, the battery lifetime is an important factor, and therefore it is necessary to reduce the energy consumed by those devices.

While the need for reducing energy consumption is obvious for mobile devices, it is not restricted to them, though. Because less energy consumption also means less waste heat it is a matter of personal computers and server farms, too. While the former can be operated with fewer cooling devices and therefore work more silently, the latter can reduce costs for cooling of server racks.

Since it is desirable to reduce energy consumption, it is necessary to have a possibility to account it. Accounting of energy consumption makes it possible to give less priority to tasks that already consumed more than others or to stop those tasks until new energy is allocated to them. The gathered information could be used to charge users for their energy consumption as well. It is also possible to give more energy to certain types of tasks than to others.

Support for the accounting of energy consumption in modern operating systems is generally poor. The Linux 2.5 operating system, for example, offers a possibility to measure the time spent in user-space and kernel-space with the command `time` (that uses the `wait3` system call). But measuring the energy consumed in this period cannot be accomplished with standard system calls.

3.2 Energy accounting in distributed systems

In distributed systems the task of accounting energy becomes more complex. Energy consumed for the accomplishment of a certain task needs to be monitored on different systems and accounted correctly. Therefore it is necessary to keep a record of the task that a certain process is currently working for. Synchronized task lists on different machines are needed so that jobs computed on different machines that are working for the same task are accounted to this task correctly.

3.2.1 Load balancing in computer clusters

A **cluster** is a type of distributed system that consists of a collection of interconnected stand-alone computers and is used as a single, unified computing resource [17]. Such a cluster can be used in a client-server-context to serve requests from the outside world. One of the cluster computers acts as a dispatcher while the other machines perform the actual work. Then, the server cluster looks like a single machine with a single IP address (the dispatcher's IP) to the clients. After serving the request, the cluster computer that did the actual work can send its reply either to the dispatcher that in turn forwards the reply to the client or the reply can be sent directly back to the client, bypassing the dispatcher. Load balancing techniques such as those given in the Linux Virtual Server project [18] allow distribution of incoming requests to the cluster machines. Usual techniques include round-robin distribution or directing new connections to the machine that is currently serving the least number of requests (*least connection scheduling*).

However, only considering the current number of connections might not be appropriate since requests may differ significantly in the amount of energy dissipated. Energy accounting in distributed systems allows a way of *least energy scheduling* by directing new connections to the server that is currently using the least amount of energy and thereby levelling the energy usage within the server cluster and avoiding peak energy usages of single machines. This is possible because current local energy usage data can be made available on each system [7]. A dispatcher can collect these numbers and dispatch the current job to the system with the least usage. Machines that have entirely used up their current energy limit can even be taken out of the dispatcher's list of target machines completely until new resources become available on those computers.

Load balancing is not restricted to dispatchers like in the example above, though. Cluster computers that hand over parts of requests to other hosts might use load balancing techniques as well. More importantly, this is of interest for systems that do not have a designated coordinator at all (e.g. peer-to-peer systems). Peers could store the energy that they consumed on behalf of other peers and the energy that other peers consumed on behalf of them. This information could be used for deciding where to direct the next request to. That way, energy could be borrowed and returned between peers.

3.2.2 Accounting of energy usage

Apart from balancing requests regarding energy aspects within the cluster it could also be necessary to grant different amounts of energy to different types of requests from outside the cluster. A possibility would be to divide client computers into different request classes with respect to their IP-address. Assignment of energy classes to different types of clients also means it is not sufficient to account energy per process. Consider a server process within the cluster serving a request for a client. A typical process-centric view would account the energy used by the server process to this very process, while it was actually the client that initiated the request and is responsible for the work done by the server. Again, this server could also query some other server (i.e. it could be a client as well) to serve the initial request, for example query a DNS-Server. Therefore, we require both requests to be accounted to the initial client's energy class, not to the server processes.

Incoming requests can be tagged with their respective energy class at the dispatcher so that cluster computers performing the actual work will account the work to the relevant class. For example, computers accessing a web server from within the intranet can be favoured over computers accessing from elsewhere. Furthermore, it is desired to be able to set an upper limit of energy usage for certain request classes. Those limits need to be valid within the whole cluster. Moreover, collected data about energy usage of certain tasks can be used to make clients pay for their energy usage rather than their CPU time spent. Since energy usage also results in extra waste heat and therefore may require expensive cooling-strategies, charging the amount of energy used could be a better way of accounting.

3.2.3 Thermal management of computer clusters

Rising energy consumption also leads to rising amount of waste heat. This poses a major problem as costly cooling methods need to be applied. In server farms certain upper temperature bounds need to be met constantly to ensure hardware integrity. Sharma et al describe in [1] the nature of this problem. Sharma et al state that computing units are more and more stored in high-density racks while those racks are in turn stored closely together in data centers. In addition to this, increasing power of modern processors intensifies the waste heat problem as well.

A typical data center cooling scheme is based on under-floor cold air distribution. The racks are situated in rows, taking in the chilled air from one side (*cool aisle*) and exhausting the hot air to the other (*hot aisle*). Chilled air is produced by *computer room air conditioning units (CRACs)*. Such a typical cooling scheme is shown in Figure 3.1. The goal is to keep the temperature in the system below the upper bound at all times. However, due to non-uniform load-distributions and irregular air flow between the racks, temperature distribution is not uniform either and so-called *hot spots* are generated. If a CRAC fails, uniformity is completely lost and upper temperature bounds cannot be met around the failed device anymore.

Sharma et al show that it is necessary to monitor temperature distribution within the computer room (using a large number of temperature sensors) and to use work-load placement policies in order to correct thermal imbalances. The temperature distribution and especially temperature peak values can be predicted by monitoring the server utilization and by monitoring the current temperatures measured by the sensors.

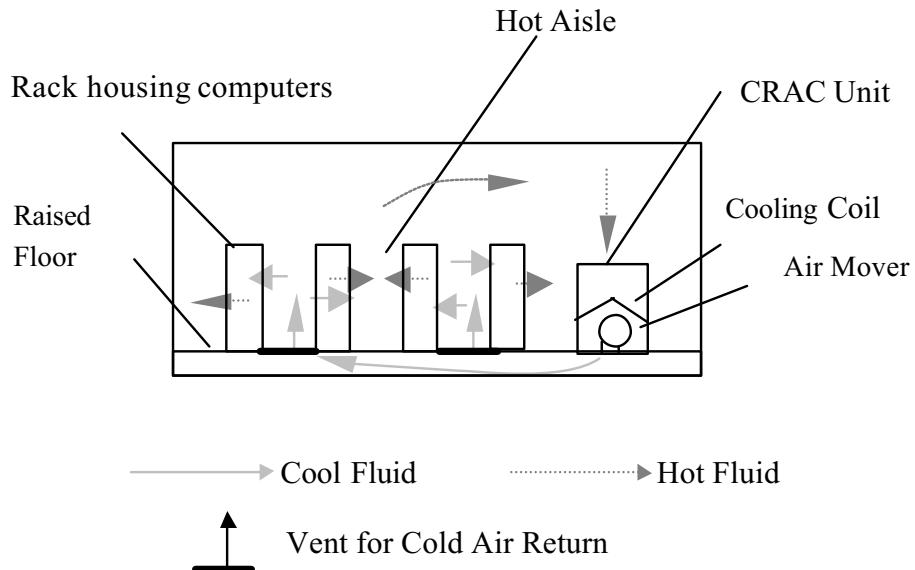


Figure 3.1: Typical data center cooling infrastructure [1] (adapted)

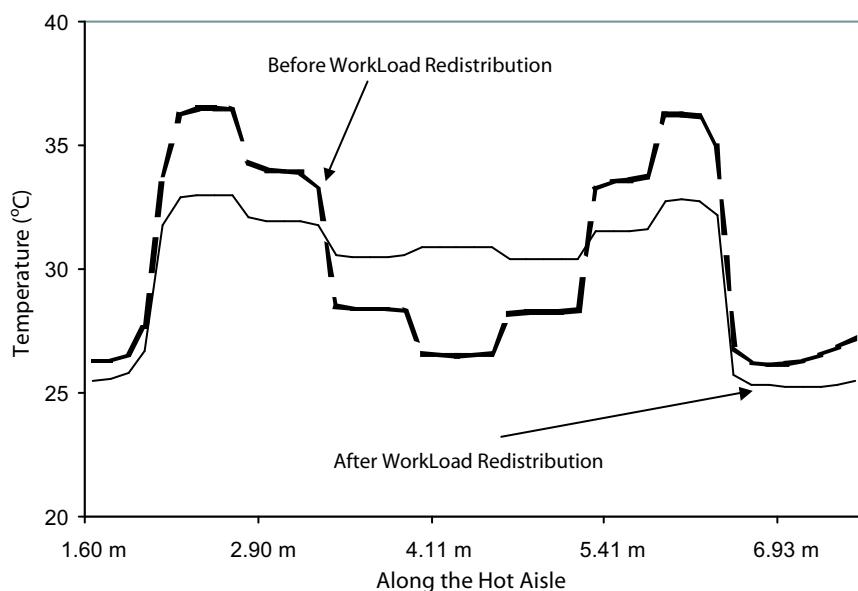


Figure 3.2: Temperature distribution along hot aisle before and after workload redistribution [1] (adapted)

In [1] it is shown that high temperature variations of up to 10 degrees Celsius are present although workload was distributed equally among the servers. Uniformity of temperature can be improved by redistribution of workload based on the measured temperatures. Figure 3.2 shows temperature distribution before and after workload redistribution in the example data center.

The work also shows that in the event of failure of a CRAC the temperature in the surrounding area rises dramatically and reaches peak values above hardware upper temperature bounds within 90 seconds. This, however, could be handled by workload redistribution as well.

Thermal load balancing can be achieved with the concepts shown in this work. Kellner provides a concept to map energy consumption to temperature in [11]. His work allows the enforcement of an upper temperature limit for a single machine by throttling the CPU. With the extensions described in this work several machines could be throttled to keep temperature evenly-distributed and below an upper limit. Non-uniform temperature distribution due to irregular air-flow can be controlled by using temperature factors for each machine depending on that machine's position in the data center.

Thermal management could use priorities like the accounting model in section 3.2.2, too. Request classes that have low priorities could be throttled first when temperature rises above a critical level. While temperature is below this level, no throttling would be performed.

Chapter 4

Design

In the previous chapter I have shown different fields where a system for distributed energy accounting and limiting would be of interest. In this chapter I will dig deeper into this system, describing how the system is built and why it was built this way.

4.1 Control of energy usage and temperature using resource containers

The usual entity of accounting resource usage by the operating system is the process. Consider the resource CPU time, for example. The CPU time that is used by a certain process will be accounted to this process and the resulting data will be used by the operating system's scheduler for its selection which process is to run on the CPU next. For many applications that only use a single, independent process and do not require the operating system kernel to perform larger amounts of work for them, this model is sufficient. However, there are scenarios where several processes work on behalf of a single task or a single process works for different tasks. An application might consist of several processes or — even worse for accounting — a server application that consists of a single process works successively for different clients and therefore it may work for different tasks. Additionally, code executed within the kernel cannot be reliably accounted to the initiating process. Server applications, for example, are network-intensive and require relatively high portions of CPU time spent in kernel mode. To receive a network packet, for instance, a process has to issue the `read` system call. The work that is done within the kernel for that system call is accounted to that process but the actual receiving

of the network packet in the lower layers is not. This is because the incoming network packet was received by the kernel earlier and held in a queue where it can be collected by the process later by issuing the system call. The same is true for outgoing network packets. Table 4.1 gives an overview of user/kernel mode CPU usage for some network intensive applications.

Hence, the process centric accounting model is not suitable for many situations. Process centric accounting would not be appropriate for energy accounting either. Consider the example of energy accounting within a server cluster in section 3.2.2. An accounting of energy usage of the cluster computers' server processes would be useless if usage were accounted to the server processes instead of being accounted to the clients.

To overcome this problem, Banga et al [6] have developed an abstraction entity called a *resource container*. A resource container accounts a certain resource (either energy or possibly other resources such as CPU time) that is consumed for the accomplishment of a certain activity, no matter if several, one or only part of a process is working on its accomplishment. This includes accounting of resource usage within kernel mode. A process gets bound to a certain resource container when it starts working for a certain task and the binding is changed afterwards. Using resource containers can guarantee a certain amount of some resource (for instance 60 per cent of the total CPU time) to a certain activity by putting all other activities on that host into another container that is limited to 40 per cent of CPU time. This concept of a resource container will be used below.

Application	% CPU (user mode)	% CPU (kernel mode)
ftp (transferring data at 92 Mbps)	10	90
Netscape (downloading at 320 Kbps)	31	69
Squid (120 requests / second)	22	78
thttpd Web server (400 requests / second)	16	84
gcc (compiling via NFS)	79	21
RealPlayer (30 frames / second)	32	68

Table 4.1: CPU usage in user and kernel modes for network intensive applications [2]

4.1.1 Accounting and control of energy usage

Martin Waitz developed an energy accounting system based on resource containers [7] as an extension to the Linux operating system kernel [19]. With this extension the operating system administers resource containers that account and limit energy consumption of certain tasks. Processes are bound to resource containers dynamically and energy consumed on behalf of these processes is accounted to the respective container. The computer does not need to be equipped with energy meters or similar equipment — the calculation of energy consumption is based on data collected from the Intel Pentium 4 processor's performance counters [7, 11]. This calculation will be described in section 4.1.2.

The resource container that a certain process is currently bound to is called the process' *resource binding*. The key concept is to charge the process that is responsible for a certain energy usage by dynamically setting the resource binding to the responsible process. To realize this, the kernel extension detects client-server relationships so that the client can be charged for energy usage of the server. A client that sends expensive requests to a server process will eventually run out of resources this way (even if the requests are not expensive from the client's point of view), stopping it from sending further requests. Thus, system load can be reduced.

When a server opens a socket to listen for incoming connections a special flag (`O_SERVER`) is set for the socket to indicate that this server process' resource binding should be reset each time a new client connects to the server. This standard behaviour may be turned off by disabling the `O_SERVER` flag with the `fctl` system call.

When a client connects to the server process and the server receives data by using the `read` system call the server's resource binding is set to the client's. The resource binding is kept until the next client connects to the server and the server calls `read` again, replacing the binding. Thus, accounting energy usage to the invoking client rather than to the server process is achieved.

This concept also works in transitive cases. If a server process connects to another server (i.e. the first server becomes a client) to fulfill the initial client's request, the resource binding of the first client is passed on to the second server, too. This means, the client is not only accounted for energy usage of the server to which the client is directly connected, but also for energy usage of further server processes that are contacted in turn by the first server.

The resource containers are organized as a hierarchy to enable more complex limitation models. In this hierarchy the energy usage of all descendant containers of a certain resource container is also accounted to the parent container. This allows restricting the total resource usage of a group of processes on a higher level within the hierarchy while allowing the application to freely distribute its share among the descendant resource containers. The operating system realizes those restrictions by allocating energy allowances to the resource containers with respect to their energy share. Those allocations are refreshed after a fixed time interval has passed. Currently there are two different time intervals (1s and 100ms) that can be limited autonomously. If a process' energy usage is over limit its process state is set to `TASK_UNINTERRUPTIBLE` so that the the Linux scheduler does not select it for running anymore until its resources have been refreshed with the next interval.

4.1.2 Temperature control

Simon Kellner developed a model to calculate energy usage on a local machine based on the performance counters available in the Intel Pentium 4 CPU [11]. The data that is collected from the energy resource containers in Martin Waitz' kernel extension as mentioned before is based on results of this work. Kellner measured the energy consumed by the Pentium 4 under different load conditions. Measurement of the CPU alone is possible because the Pentium 4 receives its power supply from an exclusive 12V source. In his experiments he used different groups of test programs (e.g. programs that only work on registers opposed to programs that need to access memory frequently). The data from the performance counters was collected and compared to the actual energy consumption derived from the meters. Using a linear approach he found an approximation mapping from performance counter data to energy consumption. Since energy consumption cannot be calculated exactly but only approximated from the limited data that is available from the performance counters, his design is optimized not to give an approximation that is lower than the actual value, i.e. only over-estimations of energy usage should be possible, not under-estimations. A test with real-world applications such as the Mozilla browser, Open Office or a Linux kernel build showed that energy estimations errors were between -0.56% and +6.09%. A more detailed explanation of this approximation can be found in [11].

In section 3.2.3 I presented an example why temperature control in clusters can be useful. To achieve temperature control on local machines a mapping from cur-

rent energy usage to current processor temperature is needed. Therefore Kellner's concept also allows calculation of processor temperature from the energy usage approximation. Again temperature approximation is required to never be less than actual temperature values. The model given showed in tests that it does not underestimate the CPU temperature while it does not over-estimate the temperature by more than 10%.

4.2 Distributed energy accounting

Section 4.1 showed possibilities for the use of resource containers for energy accounting and temperature control. However, both systems have one major drawback: they only work on single machines and cannot be directly applied to distributed systems. The reason for this problem is simple. A resource container is a data structure in kernel memory. A process' resource binding is saved as a reference to such a structure. Of course, those references are not valid across machine boundaries. Hence, to extend the concept of resource containers to more than a single machine, an abstraction from these references on local machines is necessary so that energy accounting and limiting can be achieved in distributed systems as well. Section 3.2.2 showed an example for such a scenario. In addition to the requirement to extend resource containers to be able to exist across machine boundaries, possibilities to collect usage data for a given container from the machines that share resource containers, to create new containers and to set limits for containers would be useful.

Not breaking the transparency with this extension is of great importance for the applications running on selected machines. An extension that relies on every single application being changed to adopt to the new environment is inconceivable. The kernel needs to take care of sending and receiving information that carries resource information without the applications' intervention, instead.

4.2.1 Extending the concept to cluster networks

Consider the following scenario: a cluster consisting of a dispatcher and several computing nodes is depicted in figure 4.1. Clients from outside the cluster send requests to the dispatcher. One of those requests, a HTTP GET-request, is marked (1) in the picture. The dispatcher forwards this request to a web server running on cluster computer A (2). To fulfill this request, the web server on node A sends

another request to a database server running on node B (3). The arrows 4, 5 and 6 show the replies sent back to node A, the dispatcher and the client, respectively. The work done on the dispatcher and nodes A and B to fulfill the initial request (1) needs to be accounted to the same resource container.

4.2.1.1 Sending usage information between nodes

To account and/or limit the energy used for a certain resource container cluster-wide, the nodes need to exchange information about upper limits and current usage data somehow. In a straightforward approach the dispatcher could assign an upper limit to the client's request (depending on the type of the request, the type of the client etc.) and add this information to the request before sending it on to another cluster node. To account the actual usage the dispatcher would also need to add information about its own energy usage to the request. Hence, every request would be extended by two extra data fields, `Energy_Max` and `Energy_Used`. The cluster node that serves the request could calculate the remaining energy left for use (`Energy_Max - Energy_Used`) and set its own resource container appropriately. When querying node B it would add its own energy usage so far to `Energy_Used`. Node B would return its reply with `Energy_Used` increased by its own usage and node A would again add its own usage and return this information to the dispatcher.

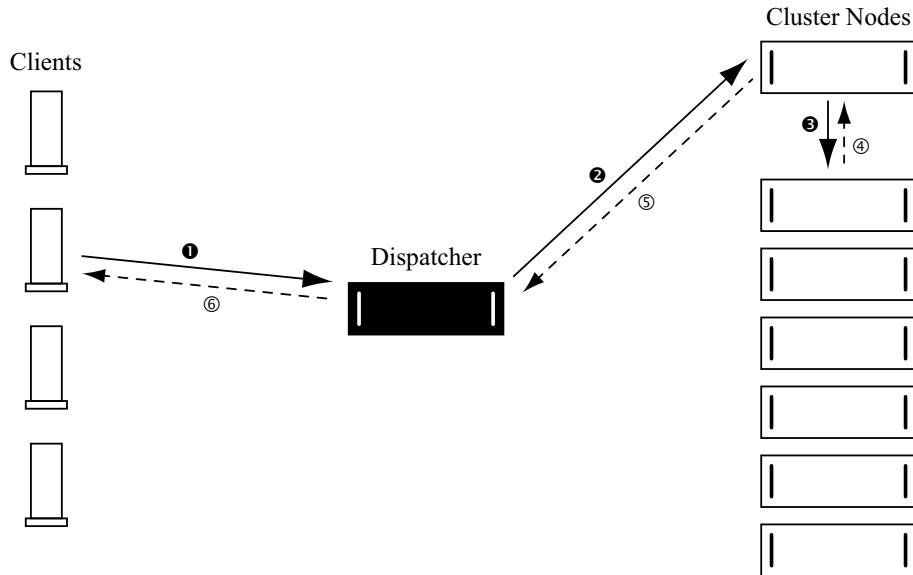


Figure 4.1: Example scenario for a client connecting to a server cluster

This approach is simple because it does not require altering the resource containers. Containers exist independently on different cluster nodes, only limits in the containers need to be adjusted on arrival of requests or replies. However, this concept does not allow to group different activities to be accounted to the same resource container, for there is no way to identify the resource container on another machine. For the same reason it is not possible either to give additional energy to a resource container later.

4.2.1.2 Extending resource containers by identifiers

To overcome the shortcomings of the approach described before, a different approach is used. The resource containers are extended by a global identifier that is valid not only on a single cluster node but within the whole cluster. In this case the nodes only need to add this resource container identifier to their messages. For each energy class that can be set by the dispatcher a resource container with a unique ID is created on every cluster node including the dispatcher. When the dispatcher assigns a resource container to an incoming request, it adds the ID of this container to the request before sending it to cluster node A. This ID is known on every node and the receiving node A can bind the receiver process to the local resource container with that ID accordingly. When sending another request to node B that same ID is added to the request. The same is true for the respective replies. With this concept it is possible to assign different requests to the same resource container (by using the same ID) and to change limits for a container cluster wide. It is still necessary to collect information about resource usage from the cluster nodes but this is independent from the requests and can be achieved by periodically obtaining usage information for each resource container ID from every cluster node. The data can be collected on the dispatcher and could be used for further distribution of energy resources among the resource containers.

4.2.2 Transparent energy accounting

In the previous section I described the information that needs to be added to the requests exchanged between the cluster nodes. However, a key requirement for this concept of energy accounting is to preserve transparency for the applications involved. The server process that receives a request needs to receive the same request as it would without distributed energy accounting. That is to say, the data belonging to the application layer protocol cannot be altered. This only leaves the

lower layers of the TCP/IP protocol stack for alteration: the data link layer, the network layer or the transport layer. While usually protocol headers of any protocol cannot be changed without breaking transparency either (a modified kernel would not be able to communicate with other kernels that use an unmodified version of this protocol) the Internet Protocol version 6 (**IPv6**) offers optional headers that can solve this problem. No extra messages are needed but the information can be contained in normal request/reply packets. By defining a custom optional header for exchanging resource container IDs between cluster computers it is possible to achieve transparency on kernel level as well as on application level. The application will generally not be aware of any headers of the network protocol layer and an unmodified kernel that does not know the custom optional header will simply discard the optional header but receive the packet normally.

4.2.2.1 The Internet Protocol version 6

The IPv6 protocol (formerly known as IPng for Internet Protocol next generation) is the successor of the existing IPv4 standard (the name IPv5 was already given to another protocol in the seventies). IPv6 was recommended by the IETF [20] in RFC 1752 [21] in July 1994. The current standard definition can be found in RFC 2460 [22]. The main reason for the proposal of a new standard was the threat of address shortage within the existing IPv4 standard. IPv4 addresses have a length of 32 bits. Theoretically this allows 4.2 billion different addresses but in practice many addresses are unusable because large subnets have been assigned to many organisations making them unavailable to others. Not only the growing number of computers connected to the Internet need unique addresses but other devices (such as mobile phones, printers etc.) have growing demand for assignment of IP addresses, too. Although there are techniques such as NAT¹ and Dynamic IP addresses for dial-up connections, IP addresses have become scarce in some areas already. To avoid address shortage for a long time, IPv6 introduces a new address length of 128 bits. That means there are $3 \cdot 10^{38}$ addresses available or $6 \cdot 10^{23}$ addresses per square metre on the Earth's surface.

A larger address space is not the only advantage IPv6 has over IPv4, though. IPv6 simplifies network administration by introducing automatic host configura-

¹Network Address Translation. A technique that allows connecting multiple computers to the Internet with only one IP address.

tion, making protocols such as BOOTP and DHCP² superfluous. Methods to provide secure communication (authentication headers, encapsulating security payload) and Quality-of-Service capabilities are also being added by IPv6. Yet, the most important issue for the extensions presented in this work: it also adds support for new options that are not part of the IPv6 standard but can be added in the future.

As IPv6 addresses are four times as long as IPv4 addresses and because new options (see above) are introduced with the new standard, the header is not defined to be of fixed length and contain room for all possible options. Such a header would be very large resulting in far higher protocol overhead compared to IPv4. Instead, the IPv6 header contains only the most necessary fields and defines additional headers that can be added between the IPv6 header and the transport protocol header (e.g. the TCP header) although this is not necessary. The sequence of possible IPv6 Headers is shown in Figure 4.2. One of those headers is the *Destination Options Header*. This header is used to carry optional information that need be examined only by a packet's destination node [22]. RFC 2460 defines two destination options headers, one that is located before and one that is located after the routing header. The latter is for options that need to be processed only by the packet's destination, not by intermediate destinations listed in the routing header. Intermediate destinations are not necessary in this concept, so the second destination options header (after the routing header) is chosen to carry the additional information. So

²Bootstrap Protocol and Dynamic Host Configuration Protocol. Protocols that allow assignment of an IP address to client computers remotely (among other things).

IPv6 header	Hop-by-Hop Options header	Destination Options header 1	Routing header	Fragment header	
	Authentication header	Encapsulating Security Payload header	Destination Options header 2	upper-layer header	

Figure 4.2: Sequence of IPv6 Headers as defined in RFC 2460

far there are no destination options defined except for two padding options used for data alignment of other options. For this work I have defined a destination option for the exchange of resource container IDs. Outgoing network packets that are sent by a process with a resource binding to a global resource container are extended by a destination options header containing the container's ID. On the receiver's side, incoming network packets with this new destination option are bound to the respective global resource container. A detailed description of the implementation of this concept can be found in the succeeding chapter.

Although most networks still use IPv4 it will soon be replaced by IPv6 because address shortage will force this step. The possibility to tunnel IPv6 packets through IPv4 networks makes it unnecessary to change to IPv6 at once but this can be done step by step. It is expected that large-scale IPv6 usage will start in Asia because IPv4 address shortage is worst on this continent. Europe will follow next and America has the least problems with address shortage so far. However, IPv6 is already in use in some areas and it will be the network protocol of the Internet within years so it is sensible to use it for the concept derived in this work.

Chapter 5

Implementation

This chapter describes the implementation of the distributed energy accounting scheme. The implementation is based on the Linux 2.5.65 kernel. Additionally, some user space tools were added. Starting with a description of the aims of this implementation the chapter continues with a detailed description of the modifications made to the Linux kernel including its API and an overview of the IPv6 data structures that were used for the implementation. Finally, new programs and changes to existing user space programs and libraries are listed.

5.1 Aims of the implementation

Apart from being suitable for its intended purpose the distributed energy accounting system was designed to fulfill the following ancillary aims, too:

- A main design goal is **transparency**. Any modifications that are made to the operating system need to be transparent to the normal application programs running on that system because it is impossible to change normal user applications for this kernel extension. Additionally, modifications performed on the kernel's network stack may not break transparency either. The modified kernel still needs to be able to communicate with hosts running unmodified kernels (or even other operating systems). The transparency of the system is vital for its application in real computing environments rather than restricting it to certain made up scenarios. Thus, I did not need to use own test programs written for the particular purpose of measuring results of the implementation in the following chapter, but I was able to test the implementation using com-

mon applications such as the Apache web server [23], the relational database PostgreSQL [24] and the distributed GCC version Distcc [25].

- **Stability** is a major design goal in every serious application. This goal is even more important when performing modifications to the operating system’s kernel. Bugs in the kernel have even larger impact than bugs in user level programs because the former will result in undesired behaviour of the whole system. Therefore, a design goal was the limitation of changes to the kernel code to as few and as small modifications as possible. Parts of the implementation that do not require to run inside the kernel were not included but developed as user space programs. Additionally, the changes to the kernel’s API¹ were limited. Certainly, user space tools need a way to communicate with the new kernel extensions. In order to keep the changes small, I did not add any new system calls but only extended the existing ones.

Additional goals such as good performance or most efficient code were not objectives of this work. Moreover, although running stable in the tests, the modified kernel has not been tested for a longer period or on many different systems and therefore cannot be compared to the well-tested original Linux kernel. The implementation described in this chapter is not optimized for these aims and improvements are probably possible.

5.2 Modifications to the Linux kernel

The implementation of my kernel extension is a modification to the existing energy resource container implementation for the Linux operating system by Martin Waitz [7]. This implementation is a modification, too. It was built using the Linux 2.5.65 kernel. The original kernel can be found at [19], the modified one at [26]. Martin Waitz’ implementation relies on the performance counters of the Intel Pentium 4 processor. Using data gathered from these counters, resource usage is computed. Therefore, this implementation is dependent on those counters too and has to be run on Pentium 4 CPUs.

¹Application Programming Interface. The kernel’s API is the interface for user space applications to communicate with the kernel, i.e. the set of system calls.

5.2.1 Global identifiers for resource containers

The existing resource container implementation represents a resource container by a structure in kernel memory. Data stored in this structure includes references to the container's parent and children, a list of processes that are bound to this container, a reference counter that counts the number of times that this container is currently referenced, references to the current resource usage count of this container etc. Extending resource containers by a global identifier and adding this identifier as an extra field to the kernel structure seems to be an obvious option. Doing so would not be the best choice, though. Mapping a resource container structure to its ID and vice versa is done very frequently during normal operation. Every time a packet is sent out the ID for a resource container has to be looked up and every time an ID is received the corresponding resource container needs to be found.

Resource containers are stored in a hierarchy in kernel memory. The pointer to the top of the hierarchy (the *root container*) is known. From there on, the hierarchy can be traversed by following the root container's references to its children. Those children in turn contain references to their children etc. Traversing the entire container hierarchy every time a container with a certain ID needs to be found would be too time consuming. That especially applies to an ID that does not exist in the system. It is not necessary for each container that exists in the system to have a global identifier. In fact, most resource containers will not have one but usually only a few cluster-wide containers are needed.

I therefore chose a different structure to save the identifiers. The array `rc_table` maps IDs to containers (and vice versa). This array does not affect the original hierarchical resource container representation. It does not even require more kernel memory than putting the ID into the existing structure would. In the array a reference to the respective resource container is stored for each entry. On the Intel Pentium 4 32-bits architecture this requires only 4 bytes. This insignificant amount is even outweighed by not reserving storage space for an ID in every resource container but only for those that actually need it. The size of the array can be adjusted to actual needs.

To add new entries to `rc_table` the function `rc_table_add_id` is provided. This function is called when registering a new ID, see section 5.2.4. To obtain

an ID for a given container as well as a container for a given ID, the functions `rc_table_get_id` and `rc_table_get_addr` are used, respectively.

Some applications create extra processes to perform their work using the `fork` system call. When a process that is bound to a resource container forks off children processes, those children will automatically be bound to new resource containers that are children of the initial process' container in the container hierarchy. That way, all their resource usage is accounted to the parent container as well. If such an application is bound to a globally known resource container it is vital that packets which are sent from that process' children are also accounted to the globally known container of the parent process. However, for the computer receiving those packets it is not important to know what actual process sent the packet - it actually does not even need to know that the sending process forked off extra processes at all. It only requires information about the sender's global resource container ID. Thus, the same ID needs to be sent with those packets, no matter if they are sent by the parent or by one of its children. The children containers cannot be assigned the same ID like their parent, though, because in that case multiple resource containers with the same ID would exist in the system. That would not allow to unambiguously map an ID to a resource container on that system (which is necessary when receiving packets with an ID, see section 5.2.2.2).

Hence, the children containers do not get an ID at all. However, the function `rc_table_get_id` takes care of returning the correct ID. Whenever it is called for a resource container that does not have an ID, it ascends in the container hierarchy until it either finds a container that does not have a parent anymore (the root container) or until it finds a container that has an ID. Only in the former case no ID is returned, otherwise it returns the first ID found. This way, subprocesses do not cause a problem for the ID system. If necessary it is even possible to assign a single ID only to the root container so that all processes on that machine will be sending out network packets containing that same ID.

5.2.2 The network stack

As illustrated in section 4.2.2 the optional IPv6 headers are used to transport resource container identifiers across the network. The kernel on the sending side is responsible to write the identifier of the global resource container into this header if the process that sent the network packet (or one of its parents, see section 5.2.1)

is bound to a container that has such an ID. Conversely, the kernel on the receiving side has to extract this information from the header (if it exists) and map it to the local resource container that corresponds to the global ID so that the receiving process can be bound to the container.

The IPv6 specification [22] defines two different optional extension headers that allow sending of user-defined options: the Hop-by-Hop options header and the Destination Options header. While the Hop-by-Hop options header's purpose is to carry options that are examined by every node on the packet's path up to its final destination (i.e. not only the packet's destination but routers too), the Destination options header is for carrying information destined only for the packet's final destination. Therefore, Destination Options headers are used.

Unfortunately, support for Destination Option headers is not fully implemented in the Linux Kernel 2.5.65. The reason is that no Destination options have been officially defined yet (except for two padding options used only for internal data alignment of other options). The kernel can handle reception of Destination Option headers and can scan them for options contained (although the handling of actual options is missing of course, as they have not been defined yet). It does not, however, offer a possibility to add new destination options to outgoing network packets. This was added in this implementation.

In the Linux operating system network packets are represented by the `sk_buff` data structure. This structure is used for outgoing packets as well as incoming ones. It also represents the network packet on every layer of the network stack; header information is added to the beginning of the `sk_buff` during send and removed and analyzed during reception. This way, copying of data between the layers is avoided. The `sk_buff`-structure can contain a reference to a resource container. When a packet is sent by a process bound to a certain container, the `sk_buff`'s reference is set to the sending process' resource binding. On reception of a packet the reference in the `sk_buff` can be read and used to set the receiving process' binding.

5.2.2.1 Outgoing network packets

Whenever a user space application wants to send data across a network it has to get a socket as a communication endpoint from the operating system first. The

socket system call is used to acquire a socket. The application needs to specify the protocol family used for the application (local communication, IPv4, IPv6 etc.). To be able to benefit from ID transfer it has to choose the IPv6 protocol. Additionally, the type of communication needs to be specified. For the TCP protocol, `SOCK_STREAM` is used.

After opening a connection on the socket the application can send data to the socket. Depending on the type of socket the data will be passed to a function that handles the sending for the corresponding protocol. For the TCP protocol on top of IPv6, the data is copied into an `sk_buff` and passed to the `tcp_v6_xmit` function. This function in turn calls the `ip6_xmit` function that passes the buffer to the network device layer. Finally, the packet is sent out over the network.

The `ip6_xmit` function was modified to add a destination options header to the outgoing packets. Within this function a destination options header is created and added as an extra header to the `sk_buff` data structure if a resource binding to a globally known resource container exists. The contents of a destination options header are defined in [22]. The header starts with a field `Next Header` identifying the type of the next header following this destination options header, i.e. the header of the transport layer (e.g. TCP). The next field is called `Hdr Ext Len` and defines the complete size of this header (including this and the `Next Header` field) as a multiple of eight bytes not including the first eight bytes. These two fields have a size of one byte each. The actual options immediately follow those header fields.

Destination Options headers can contain a variable number of *type-length-value* (*TLV*) encoded options, consisting of three consecutive type, length and value fields. Those tlv options have the following format:

- The Option Type is a one byte identifier. So far, only two option types exist (the `Pad1` and `PadN` options with option type 0 and 1, respectively). Those padding options allow insertion of either *one* or *n* padding bytes into the header to ensure correct alignment of special options that have certain alignment requirements. Since no destination options have been defined yet, alignment of options is not an issue here. The highest-order two bits of the option type define how a receiving node needs to act if it receives a packet with such an option but does not know this option type. Setting those two bits to *00* instructs the node to skip this option and continue processing this header. All the other possibilities for the highest-order bits instruct the node

to discard the packet and possibly return an ICMP message to the sender to indicate the problem. The third-highest-order bit indicates if the Option Data of this option may change on the packet's way to the recipient. A 0 indicates that this option will not change.

- The length of this option is encoded in the next byte, the field Opt Data Len. This defines the length of the subsequent Option Data in bytes.
- Following those two fields is the actual Option Data. The data is specific to the option type.

To send resource container identifiers within those headers I introduced a new option type 20, the Resource ID option. The option type 20 (bit representation *00010100*) is a number with its three highest-order bits set to 0, specifying that nodes that do not understand this option (i.e. nodes that do not have this kernel extension) will just skip this option and continue processing the packet normally (required for transparency) and that this option data will not change on the packet's way to the destination node. This Resource ID option is used for container ID exchange. For the actual ID, one byte (and therefore 256 possible IDs) are absolutely sufficient. As a destination options header's size needs to be a multiple of 8 bytes (see above) but this header is only 5 bytes long, the rest of the header is filled with a PadN option (3 bytes). Sample destination options headers filled with some TLV encoded options and filled with only one Resource ID option are displayed in figures 5.1 and 5.2, respectively.

To add a destination options header in the `ip6_xmit` function, the following steps need to be taken:

1. If the packet does not contain a resource binding, skip this packet (note that 'skip' in this context means that the next steps of this list are not taken for this packet – it does not mean that the packet is actually skipped by the `ip6_xmit` function). If it does contain a resource binding, check whether this binding references a container that has been made globally available by assigning an ID to it (or one of its parents has). This is achieved by calling `rc_table_get_id` for the resource binding and check whether this function returns a positive number identifying the ID. If it does not, skip this packet. If an ID is returned, continue.

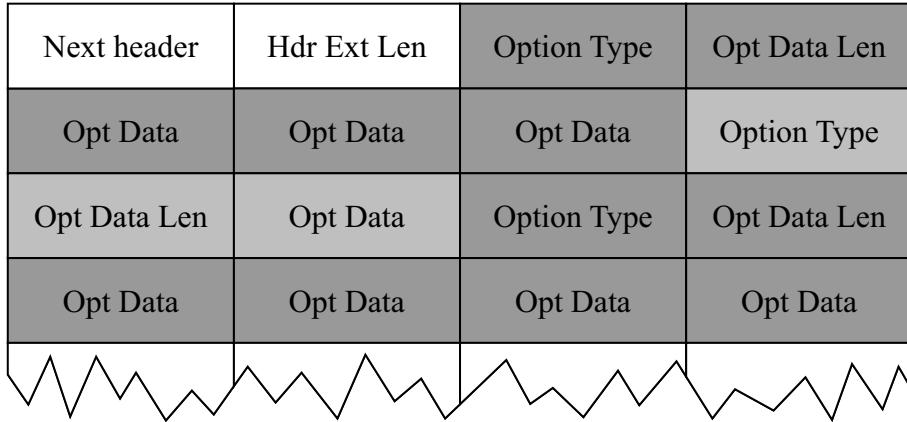


Figure 5.1: Sample Destination Options Header carrying some TLV encoded options

2. Allocate memory for the new destination options header from the socket's memory buffer using `sock_kmalloc`.
3. Enter the `tlv` option containing the current ID into the new header.
4. Adjust the length information of the optional header to the correct value.
5. Add the newly created header to the `skb`.
6. Free the memory allocated above.

Identifiers need to be added to every packet that is sent (and bound to a global container). At first glance it might be possible to send an ID only with the first packet for each container ID and each destination address and port (i.e. each server). This server will be bound to the resource container with this ID on reception of the first packet from node A and will continue to be bound to this ID until it receives a packet with a different ID. However, the kernel on node A does

6 (Protocol: TCP)	0 (0 more octets)	20 (Resource ID)	1 (1 byte data)
x (ID)	1 (PadN)	1 (1 byte padding)	0 (padding)

Figure 5.2: Destination Options Header carrying only the Resource ID option

not know whether there are other nodes sending packets with possibly different IDs to the same server process. In case of another node B that sends packets with a different ID, the server's binding will be changed to that other ID even if node B only sent a single packet and the work performed by the server for this request is completed very quickly. The work performed by the server for following requests from node A would also be accounted to this node B's ID. To avoid this problem, it is necessary to send the ID with every request (i.e. every packet), no matter how many packets have been sent to this server already.

To allow certain applications to define that they do not want this destination options headers to be added at all, the `setsockopt` system call has been expanded by another option, `IPV6_SEND_OPTS`. This option can be explicitly turned off for a socket, preventing all packets sent via this socket from being handled by this procedure.

Sending further options within the destination options headers described above is also conceivable. Nodes might send a new limit together with a resource container ID that is to be set on the receiving node. Servers might also return their usage data—taken from their current resource binding—to the client node that way. Those and other possibilities can be easily realized by either extending the newly defined `Resource ID` option to hold more than only the ID or—more obvious and implemented more easily—by defining extra options for this extra data to be exchanged. The code in the `ip6_xmit` function would have to be extended. This function would need to add these extra options to the packet. Furthermore, new handlers in analogy to the existing one for handling reception of those options (see section 5.2.2.2) would have to be added.

5.2.2.2 Incoming network packets

Kernel modifications for the reception and interpretation of destination options headers containing the newly defined option type 20 `Resource ID` (see section 5.2.2.1) take less effort than the modifications performed for outgoing network packets. The Linux Kernel 2.5.65 already has a system that handles incoming destination options headers, scanning them option for option and calling a handler for each option type (in case such a handler does not exist, it reacts as specified by the highest-order two bits of the option type as described in the previous section). Those handlers get passed the complete `sk_buff` and a pointer to the beginning of

the option (i.e. to the field Option Type). Hence, it suffices to add a new handler for the option type 20 (and possibly more handlers for further options if such an extension is desired, see section 5.2.2.1) and reference this handler in the data structure `tlvprocdestopt_lst[]` that maps option types to handler functions.

The handler for the Resource ID option is straightforward. It starts with checking the option length. Then it proceeds to acquire the ID from the option data and to call `rc_table_get_addr` for this ID. This function returns the address of the corresponding resource container (or a null pointer, if no such container is found). This address is stored in the `sk_buff`'s resource container reference from where it will be assigned to the receiving process later. The source code for this handler is shown in figure 5.3.

```
static int resource_id_handler(struct sk_buff *skb, int offset) {
    unsigned char *cp;
    int length;
    int rcvd_id;
    struct rc *rcvd_addr;

    cp = &skb->nh.raw[offset+1]; // offset points to option type
    length = (int) *cp; // option length
    if (!length) return 0; // option length is zero

    rcvd_id = (int) *(cp+1);
    if ((rcvd_id > 0) && (rcvd_id != 255)) { // id 255 is reserved
        rcvd_addr = rc_table_get_addr(rcvd_id);
        skb->rc = rcvd_addr;
        if (rcvd_addr) rc_get(rcvd_addr); // increment rc's refcount
    }
    return 1;
}
```

Figure 5.3: Handler for the option type Resource ID

5.2.3 Accounting and throttling of resources

The actual accounting of resources consumed on behalf of a local resource container has been already implemented by Martin Waitz using the Pentium 4 performance counters as described in section 4.1.1. No changes are needed in the accounting system—the distribution of identifiers allows global resource containers consisting of local resource containers that exist on every single machine. Account-

ing of those local containers is already provided by the existing work and there is no need for further changes to the kernel. The same is true for throttling processes that are bound to resource containers which have no resources left. However, to aid distributed accounting, user space tools are provided (see section 5.3).

5.2.4 Changes to the kernel's API

The kernel extensions add some functionality to the system call interface of the existing kernel. To keep changes to the API small, no new system calls were added but only existing ones were extended. The following extensions were made:

- As mentioned in section 5.2.2.1, the `setsockopt` system call has been extended to allow an application to control the sending of destination options headers in outgoing network packets via one of the application's sockets. A new option `IPV6_SEND_OPTS` was added to the `SOL_IPV6` options level of this system call that toggles a flag in the socket. This flag is checked by the modifications in the `ip6_xmit` function every time before starting their work.
- The system call `resource_info` was extended by two new options. This system call was introduced in Martin Waitz' kernel extension and is used to get and set options of existing resource containers, such as query or modify the current limit or usage of this container. It gets passed a resource container and a variable number of options. The two new options assign an ID to an existing container or get its ID if one exists. These options are named `RCI_SET_ID` and `RCI_GET_ID`, respectively. The necessity of the former option is obvious, identifiers must be assigned to resource containers to make them global. The latter option is for interaction of user space tools with the kernel extension (see section 5.3).

5.3 User space programs

The most important user space program to make use of the kernel extension is the program `mrcid` (**M**ake **R**esource **C**ontainer with **ID**). This program creates a new resource container by cloning its own container. This container is assigned an identifier passed as an argument to `mrcid` using the new `RCI_SET_ID` option of the `resource_info` system call. A limit that can be passed as another argument

to `mrcid` is set for this container. Finally, if the user starts `mrcid` with further arguments, they are interpreted as a program name and its arguments. This program is executed within the current process context (using the `execvp` system call), i.e. the program is bound to the newly created resource container with the given ID and the resource limits apply. If a client is started in such a way its ID will be transferred to server processes across the network.

Two more user space programs are the tools `globalrcd` (**global** resource container **d**aemon) and the corresponding client `globalrc`. Those tools are provided for easier use of this concept within a server cluster. The daemon process is started on each cluster node and listens on a specified port for incoming requests from the client `globalrc`. Of course, client and daemon communicate using the IPv6 protocol although resource accounting is not required for those tools' communication. But usage of IPv4 would require the cluster nodes to be connected via IPv6 and IPv4 and this can be avoided by completely using IPv6. The client needs to know the addresses of all nodes that are part of the cluster and are therefore running a `globalrcd` daemon. This information can be given to the client via a configuration file. Then, the client can communicate with the daemons on all the nodes and access or modify some of the resource container data on the local machines. The following functionality is provided:

- Create a new global resource container with a given ID
- Set or modify the resource limit of a given global resource container
- Collect the current usage data of a given global resource container

Finally, the existing user space tool `viewrcs` that prints the current resource container hierarchy as well as the limits and the usage data for each container was edited to print out the identifier (if existent) of each container as well (using the `RCI_GET_ID` option). Also, the header files of the user space tools had to be edited to support the new options for the system calls.

Chapter 6

Evaluation

In the previous chapter I described the prototype implementation of a distributed transparent energy accounting scheme. In this chapter some experiments that show that this prototype implementation actually works are described. The power consumptions of resource containers in different scenarios is measured. Starting off with a description of the testing environment the chapter continues with the actual experiments and their results.

6.1 Testing environment

All tests were performed with a homogeneous cluster consisting of three cluster nodes. Those nodes were equipped with Intel Pentium 4 2000MHz CPUs and 512 Megabytes RAM. All machines ran the modified kernel described in the previous chapter. They were connected by a 100 MBit switch and they communicated using the IPv6 protocol. The machines will be referred to as *cluster1*, *cluster2* and *cluster3* below. A further machine outside the cluster acted as client (*clientnode*) and ran the same modified kernel as the cluster nodes.

The software that I used for the experiments was chosen to be either IPv6 compliant or at least ready to use within an IPv6 network with few modifications. The Apache web server 2.0.44 [23] together with PHP 4.3.4 [27] and the PostgreSQL 7.4.1 relational database [24] are IPv6 compliant. I also used the Distributed GNU Cross Compiler distcc 2.12.1 [25] that is not IPv6 compliant but has an option to use ssh [28] for communication so that it can communicate using IPv6. I also used Hammerhead 2.1.3 [29], a tool for stress testing web servers, that can be configured to send a specified number of requests to a web server a configurable number

of times per second. In order to be able to use Hammerhead, I modified its source code to make the connections via IPv6.

6.2 Tests of the implementation

6.2.1 Distributed compilations

Distcc is a program to distribute builds of C programs across several machines on a network [25]. While usual tests only consider the time speed-up achieved by distributed compilation, I used the kernel extension that I created to measure the energy consumption during a distributed build. I also ran a local build of the same code on only one of the cluster machines using the same configuration otherwise¹, so that the results are comparable. The code used for compilation was the unmodified Linux kernel 2.6.1 with the standard configuration options. Table 6.1 shows the time that was needed for local and distributed compilation of this code. The average power consumption was measured for the whole cluster, i.e. all three cluster nodes were connected to a single meter. During the local build the two other nodes were turned on but idle. If those machines are not needed otherwise they could of course be turned off, leading to a more distinct difference between the power consumption of the local and the distributed build than the difference given here.

	Average power consumption [W]	Time [s]
Cluster idle	214	—
Distributed build	295	263
Local build	250	330

Table 6.1: Cluster power consumption and time spent during distcc build of Linux kernel 2.6.1

As table 6.1 shows, the distributed build was 67 seconds or 20.3% faster than the local build. This is a rather disappointing speed-up compared to noticeable better results with respect to speed described on the distcc web site [25]. I presume this result was partly due to the usage of ssh (instead of direct communication using the distcc daemon), which adds a usually unnecessary overhead to distributed

¹To force local compilation, the DISTCC_HOSTS environment variable was set to localhost only.

compilation time, but was necessary here to allow IPv6 communication (see section 6.1). Additionally, the Pentium 4 2GHz CPUs in use here were probably too fast for distcc to have a better effect on speed — the time needed for distributing part of the C code files to other machines and returning the corresponding object files via the same channel is not significantly shorter than the time for local compilation of the complete set of C code files on a fast CPU.

The relatively small gain in speed leads to an even higher interest in the question of how much extra energy is needed for distributed compilation as opposed to local compilation. The new kernel extension allows fine-grained measurement of CPU energy consumption for the kernel build only. To measure this consumption, I set up a global resource container in the cluster and started the distcc build bound to this container on one of the cluster nodes (this host will be called *distcc host* from here). The kernel extension takes care of binding the compilations performed on the other cluster nodes (the *remote hosts*) to this global container, enabling accurate energy measurement of the whole compilation process. Figure 6.1 shows the CPU power consumption for the compilation process on all three cluster nodes and the power consumption on one machine for the local build. The consumptions were measured once per second (data has been normalized for this figure). Note that the power consumption on the distcc node alone in the distributed case is slightly higher than it is for a local compilation, even without considering the extra power consumption on the two remote hosts. That is, compiling only part of the files locally and sending and receiving further C-code- and object-files across the network requires more power than compiling everything locally in this case.

However, because of the shorter compilation time in the distributed case the total energy consumption has to be considered rather than only looking at the power consumption at a specific point in time during the build. Table 6.2 shows the total CPU energy consumptions measured by the global resource container for the distributed and the local kernel build. In this example, the 20% speed increase of the distributed build needed about twice as much energy than the local build.

6.2.2 Transitive energy accounting

The next experiment demonstrates that the implementation also accounts transitive client-server relationships on different nodes correctly. The Apache web server

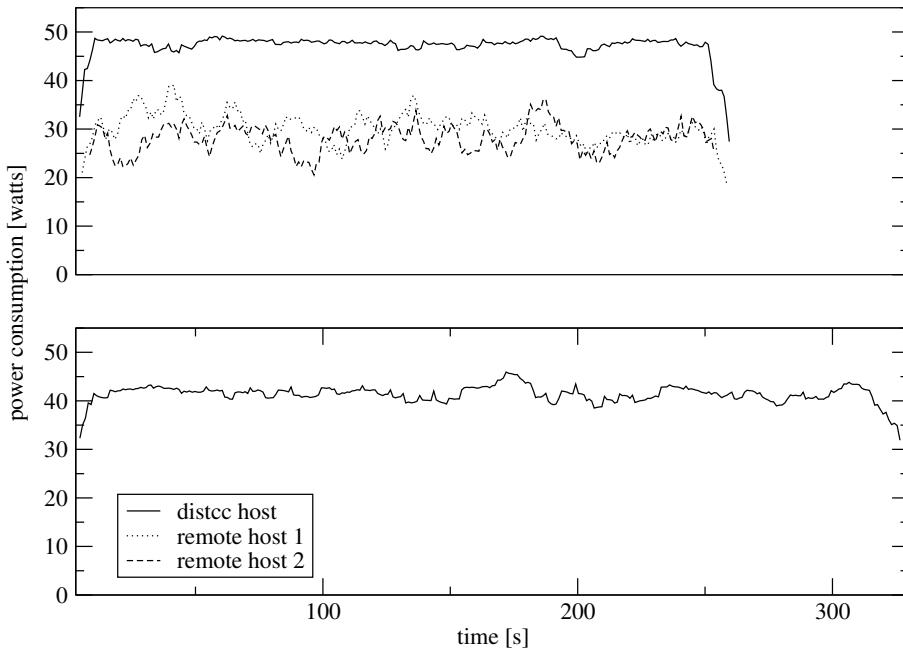


Figure 6.1: CPU power consumption for Linux kernel build on host running distcc and remote hosts (top) and power consumption during local build (bottom)

Host	Time [s]	Energy consumption [J]
Distributed build		
distcc host	263	11991,7
remote host 1	261	7576,4
remote host 2	253	6905,4
total	263	26473,5
Local build		
local host	330	13237,5

Table 6.2: Total CPU energy consumption for Linux kernel build on distributed hosts and on local host

was running on cluster1 and the PostgreSQL database on cluster2. A client from outside the cluster sent queries to the web server on node 1 for a dynamic HTML page that required a database lookup on the database server on node 2. The lookup was performed via the PHP PostgreSQL API. By not generating the requests manually with a web browser but by using the open source web server stress testing tool Hammerhead I obtained a fixed and sufficiently high number of requests per second. Though not IPv6 compliant, very small changes to Hammerhead's code

were sufficient to be able to connect via IPv6.

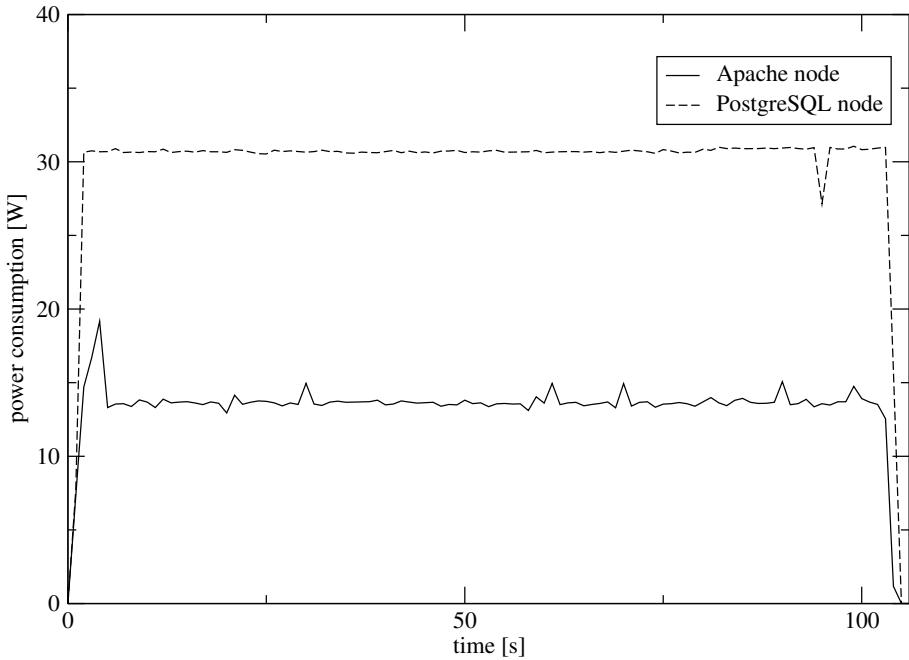


Figure 6.2: CPU power consumption for Apache web server and PostgreSQL database during transitive energy accounting

Hammerhead was bound to a global resource container and set up to send 500 requests per second to the Apache web server. Each of those requests caused Apache to send an own request to PostgreSQL. Hence, work performed on cluster nodes one and two for the initial Hammerhead request was accounted to the same resource container. Figure 6.2 shows the CPU power consumption (one value per second) on the affected cluster hosts measured by the local containers.

6.2.3 Limiting energy consumption

This final experiment shows that limiting energy consumption for requests sent from remote machines is possible. Again, the Apache web server was running on a cluster node while connections were made from a client computer outside the cluster. To simulate two different request classes, two Hammerheads were running on the client node. The Hammerheads were started bound to a global resource container each. One of those resource containers was limited to 10 watts per second on the cluster machine running the web server, the other container was not limited.

The actual power consumption of both resource containers is plotted in Figure 6.3 (one value per second).

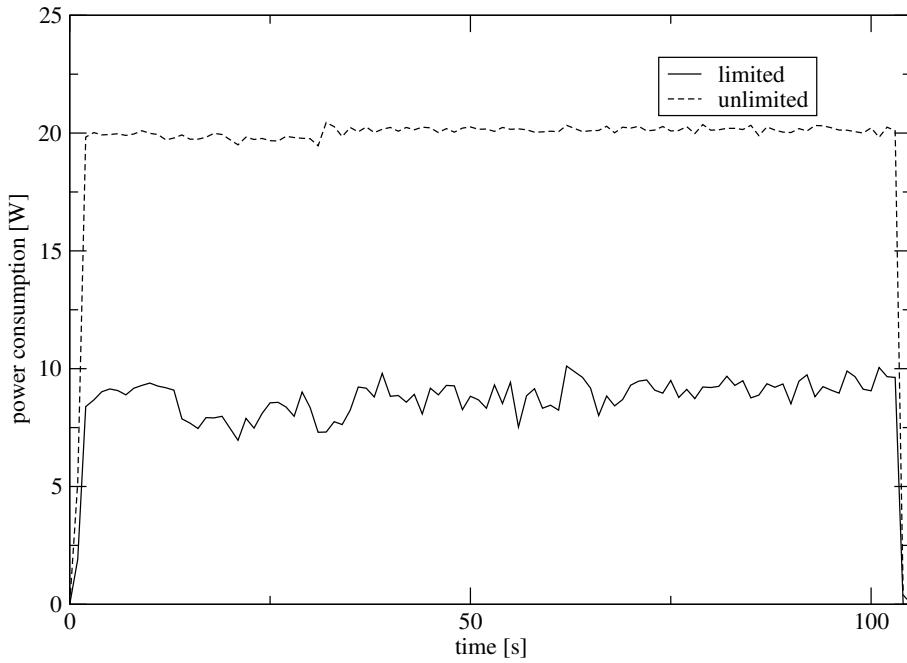


Figure 6.3: CPU power consumption for Apache web server limited to 10 watts and unlimited

Chapter 7

Future work

This chapter gives a concise survey of possible extensions to the concept of transparent energy accounting described in this work.

7.1 Extending the infrastructure for transparent information exchange

This work has introduced a new option type for destination option headers, the Resource_ID option. As already mentioned in section 5.2.2.1, this does not need to remain the only possible option. As this work implements an infrastructure for sending and receiving of piggyback information in destination options headers transparently for the application, there is obviously the possibility to send other information via this way as well. Further options could be added easily.

Yet, the number of possible options is limited. The Option Type is an 8 bit field, but the first 3 bits have a special meaning and cannot be chosen freely (see section 5.2.2.1). Furthermore, two option numbers are taken for the padding options. This leaves 30 different options types. Considering that option numbers for destination options headers might be used for other concepts in the future as well, there might not be enough options left for all conceivable applications. In order to circumvent this problem, extra options could be limited to a single option type and a protocol for the contents of the Option Data field would have to be defined to regularise the communication between sender and receiver. A straightforward approach would use another way of tlv-encoding inside the Option Data field.

7.2 Relocation of unused resources

Global resource container usage data is collected by installing local resource containers on each host and adding up usage data of those containers. When containers are limited, the same limit is set on every node. If some nodes do not exhaust their limit, this share of the limit is simply lost and not relocated to the other hosts.

This is not a problem for accounting-only scenarios or for thermal management either, because each node can be limited freely. However, this can be a problem in scenarios in which customers pay for a certain share of energy in advance, for instance, and fractions of this share are lost. Aron et al have shown in [12] that equally distributed resource limits on every cluster node suffice to achieve performance isolation if the requests bound to each resource container are load balanced across the cluster nodes. For scenarios in which this assumption does not hold, the energy accounting and limiting concept developed in this work could be extended to a system that collects unused resources from local containers and relocates those resources among corresponding containers that belong to the same global container and have exhausted their share already.

Implementation of such an extension is not straightforward, though. The current local resource container implementation that this work is based upon [7] does not store unused resources but old resources are lost with every refresh cycle. Thus, those containers would need to be extended first. Moreover, relocation of resources among the nodes would require extra network traffic, especially because efficient relocation would require to quickly and therefore frequently redistribute unused resources. Consequently, a significant overhead for such a relocation system is likely. It needs to be studied whether or not this overhead is outweighed by the expected advantages.

7.3 Accounting of further resources

Of course, the concept of resource containers is not limited to accounting CPU energy usage. Further resources such as memory or network-bandwidth could be accounted, too. Likewise, accounting of energy usage could be extended to other hardware components in the system other than the CPU. This would either require

that those components are capable of delivering similar data like the Pentium 4 CPU does from its performance counters, or certain fixed energy amounts could be assigned to special events. For instance, the sending of a network packet induces energy usage within the network interface card to actually send the packet. Unless the energy consumption of this card during the sending of a packet is unpredictable, fixed amounts could be accounted for such events.

7.4 Profiling of resource containers

Keeping a history of a resource container's usage data could be of interest for allocation of processes to cluster nodes. In that respect, processes bound to resource containers which have shown the need of only little CPU time could be combined on one machine and the clock speed could be reduced to save energy without or with only little performance penalties.

Chapter 8

Conclusions

This work has introduced a concept for task-specific accounting of energy dissipation in distributed systems.

Limitations of the existing energy resource container implementation have been overcome by extending the concept of a resource container to a *global resource container* that is valid across system boundaries. This has been accomplished by adding globally available identifiers to the resource containers and sending those identifiers between clients and servers that reside on different hosts. Transmitting extra messages has been avoided by sending the identifiers piggyback with normal network traffic. The use of the new Internet Protocol IPv6 is a sensible choice as this protocol allows optional and definable headers, hence identifiers can be transmitted transparently for the applications involved. Communication with other operating systems is not obstructed either, although energy accounting will not be in effect on those systems, of course.

Several applications of the given accounting scheme have been proposed. Together with the limitation of resource containers and the mapping of energy usage data to heat dissipation of CPUs, further scenarios have been outlined.

A prototype implementation of the concept has been presented as a modification of a Linux kernel together with user-level programs. Several experiments have shown that the implementation is effective and that valuable energy usage data can be collected on clusters for different scenarios.

Finally, the use of energy usage data obtained from this concept for relocation of unused energy portions to other systems has been suggested. While not necessary in all situations, this extension could enhance the existing concept further.

Bibliography

- [1] Ratnesh K. Sharma, Cullen E. Bash, Chandrakant D. Patel, Richard J. Friedrich, and Jeffrey S. Chase. Balance of power: Dynamic thermal management for internet data centers. Technical Report HPL-2003-5, HP Labs, February 2003.
- [2] Gaurav Banga. *Operating System Support for Server Applications*. PhD thesis, Rice University, May 1999.
- [3] Chandrakant D. Patel. A vision of energy aware computing from chips to data centers. In *International Symposium on Micro-Mechanical Engineering*. HP Labs, December 2003.
- [4] Fred Stack. Feeling the heat. *dcm magazine*, July/August 2003.
- [5] Jeff Chase and Ron Doyle. Balance of power: Energy management for server clusters. In *Proceedings of the Eighth Workshop on Hot Topic in Operating Systems HotOS'2001*, May 2001.
- [6] Gaurav Banga, Peter Druschel, and Jeffrey Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating System Design and Implementation OSDI'99*, February 1999.
- [7] Martin Waitz. Accounting and control of power consumption in energy-aware operating systems. Master's thesis, Department of Computer Science 4, University of Erlangen-Nürnberg, January 2003. SA-I4-2002-14.
- [8] Jeff Chase, Darrell Anderson, Prachi Thakur, and Amin Vahdat. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating Systems Principles SOSP'01*, October 2001.

- [9] Pat Bohrer, Mootaz Elnozahy, Mike Kistler, Charles Lefurgy, Chandler McDowell, and Ramakrishnan Rajamony. The case for power management in web servers; P. Bohrer, et al. In Robert Graybill and Rami Melhem, editors, *Power Aware Computing*. Kluwer Academic Publishers, 2002.
- [10] F. Bellosa. The case for event-driven energy accounting. Technical Report TR-I4-01-07, University of Erlangen, Department of Computer Science, June 2001.
- [11] Simon Kellner. Event-driven temperature-control in operating systems. student thesis SA-I4-2003-02, April 2003. Department of Computer Science 4, University of Erlangen-Nürnberg.
- [12] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Measurement and Modeling of Computer Systems*, pages 90–101, 2000.
- [13] FreeBSD operating system. <http://www.freebsd.org>.
- [14] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings Of The Seventh International Symposium On High-Performance Computer Architecture (HPCA'01)*, January 2001.
- [15] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Dynamic cluster reconfiguration for power and performance. In Luca Benini, Mahmut Kandemir, and J. Ramanujam, editors, *Compilers and Operating Systems for Low Power*. Kluwer Academic Publishers, 2002.
- [16] E. Pinheiro and R. Bianchini. Nomad: A scalable operating system for clusters of uni and multiprocessors. In *Proceedings of the 1st IEEE International Workshop on Cluster Computing*, December 1999.
- [17] Gregory F. Pfister. *In search of clusters*. Prentice Hall, 2nd edition, 1998.
- [18] Wensong Zhang. Linux virtual server for scalable network services. In *Proceedings of the Ottawa Linux Symposium 2000*, July 2000.
- [19] The Linux Kernel Archives. <http://www.kernel.org>.
- [20] The Internet Engineering Taskforce. <http://www.ietf.org>.

- [21] The Recommendation for the IP Next Generation Protocol. RFC 1752.
- [22] Internet Protocol, Version 6 (IPv6), Specification. RFC 2460.
- [23] The Apache HTTP Server Project. <http://httpd.apache.org/>.
- [24] PostgreSQL Relational Database. <http://www.postgresql.org/>.
- [25] Distcc - Distributed GNU Cross Compiler. <http://distcc.samba.org/>.
- [26] Bitkeeper repository holding the sourcecode of Martin Waitz' implementation. <http://tali.bkbits.net/>.
- [27] PHP Hypertext Preprocessor. <http://www.php.net>.
- [28] OpenSSH - Secure Shell. <http://www.openssh.org/>.
- [29] Hammerhead web server stress testing tool.
<http://hammerhead.sourceforge.net/>.

Erfassung und Abrechnung des Energieverbrauchs in Verteilten Systemen

Die Entwicklung hin zu immer leistungsfähigerer Hardware hat dazu geführt, dass Energieverbrauch und Abwärme und die mit letzterer verbundenen Ausgaben für Kühlung immer mehr zu einem entscheidenden Kostenfaktor werden. Dieses Problem wird durch den zunehmenden Trend zu höher integrierter Hardware auf engem Raum in Rechenzentren und damit der Zunahme der Leistungsaufnahme pro Quadratmeter Stellfläche noch weiter verschärft. Die Erfassung, Abrechnung und Limitierung des Energieverbrauchs ist deshalb von zunehmender Bedeutung.

Möglichkeiten zur Erfassung des Energieverbrauchs werden von aktuellen Betriebssystem nicht angeboten. Es gibt jedoch existierende Betriebssystemerweiterungen, die eine derartige Funktionalität bereitstellen. Eine Erweiterung für das Linux Betriebssystem von Martin Waitz verwendet das bekannte Konzept der *Resource Container*, um eine Erfassung des Energieverbrauchs bestimmter Aufgaben, d.h. über Prozessgrenzen hinweg, zu gewährleisten. Prozesse werden dabei an bestimmte Resource Container gebunden, um zu kennzeichnen, für welchen Resource Container ein Prozess im Moment arbeitet. Diese Bindung ist dynamisch, d.h. wenn der Prozess seine Aufgabe wechselt, so wechselt auch die Resourcenbindung. Allerdings ist diese Implementierung nur auf Einzelrechnern einsetzbar.

Diese Arbeit stellt ein auf obiger Implementierung aufbauendes Konzept vor, das den Einsatz von Resource Containern über Rechnergrenzen hinweg, z.B. in Rechnerverbünden, ermöglicht. Hierzu werden Resource Container um einen global gültigen Identifikator erweitert. Dieser wird bei Anfragen an andere Rechner mitübertragen, um eine Zuordnung zum korrekten — nun global gültigen — Resource Container zu ermöglichen. Wird etwa ein Server-Prozess für einen Client-Prozess aktiv, so wird der Server-Prozess für die Dauer dieser Aktivität an den

Resource Container des Clients gebunden und somit der vom Server ausgehende CPU-Stromverbrauch für die Dauer dieser Bindung diesem Resource Container zugeschrieben. Bei einer Anfrage des nächsten Clients ändert sich diese Bindung entsprechend. Client und Server können sich dabei auf unterschiedlichen Rechnern innerhalb des Gültigkeitsbereichs des globalen Resource Containers befinden. Eine Begrenzung des Stromverbrauchs für globale Container sowie die Verwendung dieses Konzepts für die Temperaturregelung in Rechnerverbünden sind ebenfalls möglich.

Das Übertragen der zusätzlichen Identifikatoren erfolgt dabei transparent für die beteiligten Anwendungen und ohne zusätzlichen Netzwerkverkehr über IPv6 Verbindungen. Hierzu bedient sich die Implementierung einer von IPv6 angebotenen Möglichkeit, selbst definierte Optionen in zusätzliche, optionale IPv6-Header einzufügen. Die Pakete bleiben dabei absolut standardkonform, so dass ein Datenaustausch auch mit anderen, nicht modifizierten Betriebssystemen weiterhin möglich ist.

Zur Umsetzung dieses Konzepts wurde eine prototypische Implementierung für das Betriebssystem Linux erstellt. Sie besteht aus einer Kernerweiterung und aus verschiedenen Hilfsprogrammen zur Verwaltung. Mehrere Experimente, bei denen der Energieverbrauch zur Bewältigung verschiedener Aufgaben innerhalb eines Rechnerverbunds gemessen und limitiert wird, zeigen die Funktionsfähigkeit dieser Implementierung. Die Experimente umfassen Messungen mit einem Webserver, der zum Bearbeiten einer Anfrage nach einer dynamischen Seite auf einen Datenbankserver auf einem anderen Rechner zugreift, sowie Messungen bei der verteilten Übersetzung eines Programms auf mehreren Rechnern. Abschließend werden Ansätze zur weiteren Verbesserung des Konzepts vorgeschlagen.