Energy-Aware Memory Management
Energiebewusste Speicherverwaltung

Diploma Thesis

by

Jürgen Obernolte

born December 23th, 1974, in Marktredwitz

Department of Computer Science,

Distributed Systems and Operating Systems,

University of Erlangen-Nürnberg

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

Begin: September 22nd, 2002

Submission: April 22nd, 2003

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 Prufeungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäss übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den Unterschrift

Abstract

One major challenge in modern operating systems is the need to reduce the overall energy consumption. Especially with the increased use of embedded or portable devices improving energy efficiency is becoming a critical issue. Memory in particular plays an important role in the context of power consumption. This is due to the fact that even small devices (e.g. PDAs or mobile phones) are fitted with more and increasingly cheaper memory.

This thesis discusses a novel approach to save energy by exploiting memory systems with built-in power management features. These features consist of the possibility to put individual chips in several different power modes. In particular I am going to explore how the page placement should work to make the best use of the power management features. To show to what extent energy savings are possible I modified an existing Linux kernel so that it is aware of the different power modes in the memory system.

A decrease of memory power consumption is only possible if the memory isn't fully utilized all of the time. This is true for most of the standard applications which are executed by an average user. For example I have used the application 'Mpg321' for testing purposes. Examining the maximum resident set size it has shown that it only uses about 6 MB of memory. But the testing machine which I used had 1GB of main memory installed. This means that I used only 0.6% of the available memory. With this information it is clear that you could save a huge amount of energy, if you could suspend the remaining unused memory.

To suspend part of the memory it is necessary to know at every point in time when a particular memory area is used. Having this information it is possible to transition the remaining unused memory into a low power state. This can only be done if the memory by itself is separated into smaller areas which can operate independently from each other. This is true for modern memory, which is delivered as a single chip, with each single chip being divided into different memory banks. These memory banks are the single independent units which fullfill the required criteria necessary for reducing power consumption.

To exploit the power management features a Linux kernel has been modified to make use of it. In particular I have modified the memory management system to be aware of such power saving memory chips. With the control over the memory management it is possible to cluster the pages for particular applications into a minimum of memory banks. To determine the working set of an application I manage a bank usage table which holds track of the different page references for each application. With this information it is possible to cluster the pages of a single application into a minimum of memory banks. However this approach works only well if you have only one application running at a time. But if you use several applications at a time the clustering of pages can not be maintained because the page allocation scheme is a non deterministic process, making it impossible to predict when a particular application will allocate new pages. So the clustering of pages might get lost, especially if multiple applications work on the same memory bank at a time. To bypass this limitation a novel approach is introduced. If the page request for a particular memory bank fails the memory management tries to migrate or compress a page from this memory bank. Migration is the process of copying the contents of a page to another page in a different memory bank and adjusting the corresponding virtual memory mapping. Compression is the process of compressing the page contents and storing the compressed data in a cache memory. Both mechanisms ensures that the page clustering doesn't get lost over time.

With this clustering of application pages it is possible to suspend the remaining unused memory banks. This deactivation is done every time when a new process gets scheduled. During the scheduling the memory management also determines if a former inactive memory bank needs to be activated by looking at the bank usage table.

Although this approach has many benefits in terms of energy efficiency there are also some drawbacks which have to be considered. The first to mention is the longer execution time of applications due to the overhead for energy reducing measures. Another issue are the energy costs of the page migration / compression. Depending on the CPU used this can lead to a higher energy consumption than before.

The implementation has shown that it is possible to reduce the energy consumption by up to 30%. In contrast the execution times of the applications have increased by a average of 2% - 10%.

Table of Contents

1 Introduction **1**
1.1 Proposed Solution for Reducing Memory Power Consumption 1
1.2 Operational Areas 2
1.3 Structure of the Thesis 2
2 Background and Related Work **3**
2.1 Rambus RDRAM 3
2.2 Related Work 4
3 Approach **7**
3.1 Memory Architecture and Low-Power Operating Modes 7
3.2 Maximum Working Set of Applications 7
3.3 Bank Usage Accounting 7
3.4 Page Allocation Strategy 7
3.5 Bank (De-)Activation 10
3.6 Performance Issues versus Energy Saving 10
4 Implementation **13**
4.1 Understanding the Memory Management of the Linux Kernel 13
4.2 Simulating a Multibank Memory System 13
4.3 Bank Usage Accounting 14
4.4 Implementation of the Page Allocation Strategy 14
4.5 Bank (De-)Activation 15
4.6 New System Calls 15
5 Experimental Results **17**
5.1 How to Compare the Results 17
5.2 Testenvironment and Testsuite 19
5.3 Measurements 20
5.4 Subtle Changes of the Implementation 25
6 Discussion **27**
6.1 Overhead versus Energy Saving 27
6.2 Correlation between Working Set Size and Normalized Energy Savings 27
6.3 Correlation between Memory Size and Normalized Energy Ratio 28
6.4 Energy Efficiency Factors 28
6.5 TLB Cache Effects of Page Migration and Compression 31
6.6 Energy Savings in a Multitasking Environment 31
7 Future Work **33**
7.1 Deactivation of Kernel Zones 33
7.2 Optimizing the Scheduler 33
7.3 Compiler-Directed Array Interleaving 33
7.4 Combination with Hardware Directed Power Management 33
8 Summary **35**
Bibliography **36**

1 Introduction

One major challenge in modern operating systems is to reduce the overall energy consumption. To achieve this, several strategies are possible. Most of this strategies which have examined energy reducing measures didn't focus on the memory system as a source for increasing the energy efficiency. But one main reason to do this is because memory is a component which uses a significant amount of energy and modern computer systems are fitted with more and more memory which the average user doesn't utilize completely all the time, making it possible to save energy by turning off the unused memory of the system [13][14]. E.g. the memory installed in my test machine had a power consumption of about 3W at peak rates. Compared to the CPU power consumption (e.g. Intel Pentium 4 M with 1.40Ghz has a power consumption of about 26W [28]) this is not to be underestimated.

To achieve this, the memory chips by themself must be capable of some sort of power management. With the introduction of the most recent generation of memory chips [1][2], power management features have become available and new algorithms have to be investigated to make use of these features.

The new memory chips have at least three operating modes:

1. *active* - chip is currently being accessed
2. *standby* - no access at all
3. *powerdown* - the chip can't be accessed (but doesn't loose its contents)

The goal of this thesis was to explore how these power management features could be utilized in an operating system to reduce the overall energy consumption in a computer system.

1.1 Proposed Solution for Reducing Memory Power Consumption

To suspend some of the memory it is necessary that the memory is subdivided into individual areas which can operate independently from each other. This is true for all newer memory chips which are often referenced to as multibank memory. These memory chips come as a single device which are composed of multiple memory banks [15]. For example a 128MB memory chip might consist of 32 4MB memory banks. Each memory bank is an independent unit which can be put in any of the former mentioned power states.

A further criteria for suspending part of the memory is that the working set of an application is much smaller than the size of the installed memory. The analysis of the working sets of the used test applications have shown that this condition holds for many applications. E.g. the sizes of the workings sets ranged from 0.35% to 12.62% of the size of the main memory. This information shows that indeed only a little amount of the available memory is used at time.

To exploit the power management features some parts of the memory management system have been redesigned. Especially the memory management has to be aware of the individual memory chips. With this is mind it is possible to cluster the used pages of an application into a minimum number of memory banks. Now without the used pages being scattered over the whole range of the physical memory it is possible to suspend the unused memory chips into a low power mode. However this approach works only well if you have only one application running at a time. But the clustering might get lost if multiple applications work on the same memory bank at a time.

Page Migration / Page Compression

If the page allocation for a certain memory banks fails, the page allocator tries to reclaim a page from this memory bank. Simplified spoken this is done by scanning through the pages of the memory bank and trying to either migrate a page to another memory bank or trying to compress the page and store it in a so called „compression cache“. If a page has been reclaimed successfully it can be given to the process which demanded a page. This has the effect that there is no need to activate a new memory bank.

What It Costs

It shouldn't be concealed that there are some drawbacks introduced with this new technology. Copying or migrating pages and suspending or waking up memory banks lead to longer execution times for applications. This is due to an increased overhead for the memory management itself as well as latency times for transitioning memory chips into different power states. Another issue are the energy costs of the page migration / compression. Depending on the CPU used this can lead to higher energy consumption than before, especially if the CPU consumes much more energy than the memory.

1.2 Operational Areas

The main operational area of the proposed mechanism may lie in the field of mobile computing. Think of notebooks or PDAs which have plenty of RAM but which is not fully used most of the time by the average user. As notebooks are mostly used for doing simple tasks, like word processing or similar [26], using the proposed mechanism the lifetime of the battery may greatly increase.

1.3 Structure of the Thesis

The thesis is structured as follows:

- *Section 2* – Here I give some background information about RDRAM which is capable of power management, as well as some information about related work which also dealt with reducing memory power consumption.
- *Section 3* – In this section I describe the approach by which I tried to increase the energy efficiency of the memory system.
- *Section 4* – This section covers details about the implementation.
- *Section 5* – Here I present the experimental results which have been taken.
- *Section 6* – In this section I discuss the pros and cons of energy aware memory management.
- *Section 7* – Here I give some information about future work which can be done to improve energy efficiency further.
- *Section 8* – This is the summary of thesis.

2 Background and Related Work

Over the last decades computers have gained more and more power and flexibility. But parallel to this the overall power consumption also has increased. Most areas where energy could be saven are widely exploited. And most of the results have already been put into practice (harddisk spindown [7], CPU frequency scaling [8], ...). One other up to now only theoretically discussed (or only simulated) way of lowering power consumption is the use of energy aware memory chips. Although the memory system is not the main power consumer, it is using constantly energy. An example: The testing machine which I used to retrieve the experimental results was a Pentium 4 with 2Ghz and 1GB of main memory. The peak power rate of the memory has been about 3W whereas the CPU consumed about 40W. On the first sight this seems to be negligible, but in the field of mobile computing this will become an important factor, as mobile devices are outfitted with more and more memory to meet new requirements of current and future applications. Furthermore in mobile computing you will use certainly low power processors (e.g. Transmeta Crusoe [27]) to extend the lifetime of the battery.

This chapter will give a brief summarization of current memory technology as well as some related work dealing with energy saving measurements.

2.1 Rambus RDRAM

In a large memory system energy can only be saved if part of it which is currently not in use can be switched off. This is only possible in multi-bank memory systems where the memory modules operate independently from each other. With such a memory system, it is possible to selectively suspend memory modules into powerdown mode. This in turn enables you to save a significant amount of energy (depending on the memory chip used). On the other hand you have to take longer memory access latency times into account, as it takes some resynchronization time to switch from powerdown mode to active mode.

One example for multi-bank memory is Direct Rambus DRAM (RDRAM) [1] which offers not only high bandwidths but also the capability of reducing the energy consumption by selectively turning unused RAM into sleep mode. Rambus DRAM offers four different operating modes [2]:

- *Active* mode
- *Standby* mode
- *Nap* mode
- *Powerdown* mode

Each operating mode has a different power consumption and delay time for executing a memory transaction. If RDRAM is in *active* mode then it is able to immediately service a memory request (e.g. memory read or write), but power consumption is also at its highest. All other states consume less power but take longer time to satisfy a request. shows the power mode state diagram for RDRAM.

If the RDRAM is placed into *nap* mode then energy consumption drops to about 10% of the power used in *standby* mode. The lowest power mode for RDRAM is *powerdown* mode. In this mode RDRAM uses less than 1% of the power consumed in *active* mode. Both modes cost additional resynchronization time to change their state to *active* mode. summarizes power savings and resynchronization times for the Rambus DRAM power states.

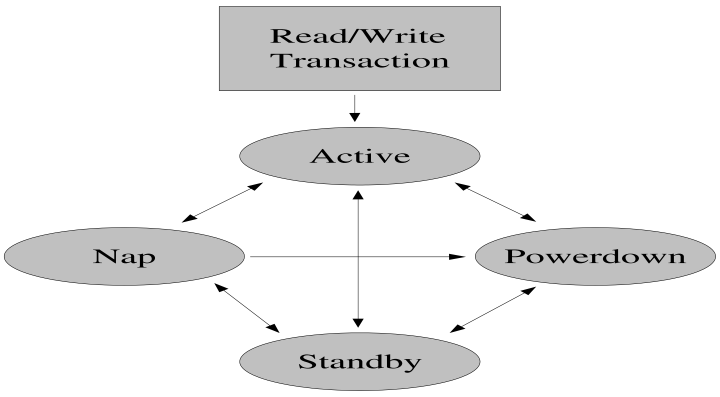


Illustration 1 Power Transition States in Direct RDRAM

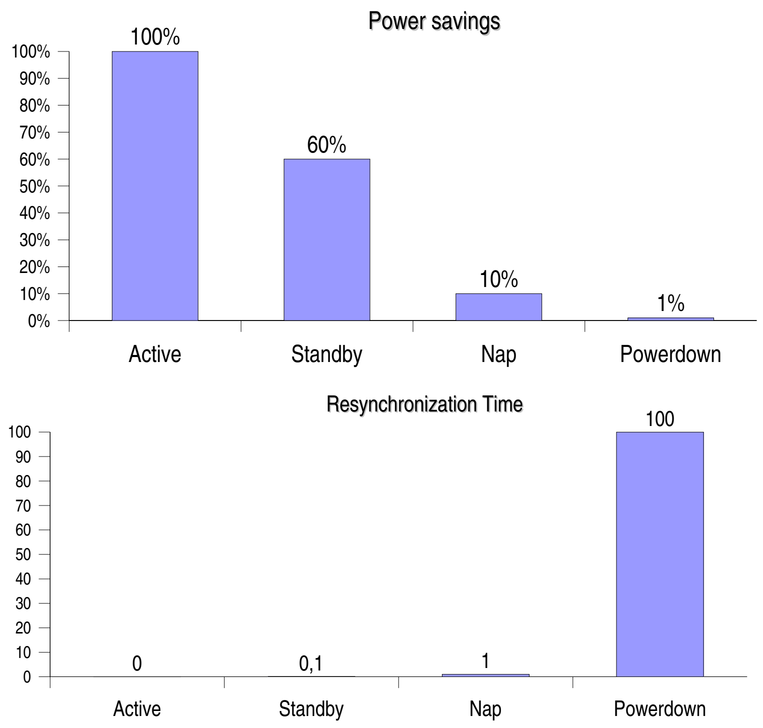


Illustration 2 Power savings and resynchronization times

2.2 Related Work

There are some studies which have examined the impact of the memory system on power consumption. But most of them are only theoretical or experimental, because they only simulate the results of their approaches rather than testing them in real life scenarios.

2.2.1 Compiler Directed Array Interleaving

One way of reducing energy usage in a multi bank memory system is to use a specialized compiler trimmed for the use of multiple memory banks. The effectiveness of power mode control schemes depend critically on the memory access patterns and data allocation strategies in these memories [3]. In particular, poor data allocation strategies will lead to a large energy losses. To make use of the energy saving features of multi bank memory systems, a source level approach can be chosen. This implies, that the compiler tries to optimize data space (array layout). This can be achieved by array interleaving, which clusters the data elements of multiple arrays accessed simultaneously into a single common data space. This has the effect that fewer memory modules need to be active at a given time. Of course this mechanism will only work well with applications that are „array based“.

The disadvantage of this approach is, that it needs sophisticated program analysis support and that it is difficult to adjust to a dynamic runtime environment.

2.2.2 Power Aware Page Allocation

A direct way to reduce power consumption in a computer system is to utilize the power management features offered by modern memory chips. A thesis written by Lebeck at all. [6] which is very close to this diploma thesis explored the interaction of page placement with static and dynamic hardware policies for utilizing energy saving features of memory chips.

Unlike my thesis the study concentrated only on the hardware itself without considering the advantages and disadvantages in a real life environment. Thus, all the results presented are only experimental. But it showed how large the benefits are.

The thesis dealt with two basic questions [6]:

1. How can the various power modes of modern memory chips be utilized (using different hardware power management policies)?
2. What is the effect of code and data placement in such power-aware memory chips?

In this context, the concept of hardware power management policy refers to the memory controller (realized in hardware). In this study two types of policies were investigated: static and dynamic.

Static policies correspond to placing all memory chips in a single power state. Hence if an access occurs, the specific DRAM chip must first transition into the active state. When the request is satisfied and there are no outstanding requests the chip is returned back to the static power state.

With dynamic policies the single memory chips are not evenly placed in a single power state. Here the time between accesses to a chip is used as a metric for transitioning to lower power states. If a chip hasn't been accessed for a certain period of time transition to the next lower power state occurs. This approach allows the individual chips to reside in different power states, based on their access patterns.

To further improve the energy efficiency the page allocation policies have to be aware of the new memory chips. This is done by allocating physical pages in a manner which fully exploits the hardware.

The main results from the study were [6]:

- Cooperative hardware and software for power aware page allocation can improve main memory energy efficiency, measured in terms of energy * delay, by 6% to 55%

- Nap mode is the most energy efficient static policy for their test applications
- Power aware page allocation without dynamic hardware support can improve energy efficiency by up to 30%, depending on application characteristics
- Dynamic hardware schemes do not improve energy efficiency for random page allocation

3 Approach

This diploma thesis describes a novel approach in the area of DRAM based energy management. It combines several mechanisms to achieve a higher energy saving ratio. The whole approach is based on the operating system (OS), meaning that applications don't have to be modified to make use of the new energy saving features. The major component is the virtual memory system where all of the management is done. This is also the place where the operating system gathers information about each individual process, mainly physical page references and usage accounting.

The novelty of this approach is, that it uses automatic page migration as well as page compression to maintain the clustering of physical pages for each process.

3.1 Memory Architecture and Low-Power Operating Modes

This thesis focuses on an architecture where the memory system is composed of multiple memory modules which can operate independently from each other [15]. When a memory bank is not actively in use it can be placed into a low-power operating mode. The model assumes that there are four power modes: active, standby, nap and powerdown. Each of these power modes has different access times and energy costs.

3.2 Maximum Working Set of Applications

As described in the introduction suspending part of the memory only makes sense if it isn't fully utilized all the time. A indicator for the occupancy of the memory is the maximum working set [16] of an application, i.e. the maximum amount of memory an application uses. The working set includes all code and data pages as well as all pages from shared memory. E.g. one of the testing applications used was 'Mpg321' which had a maximum working set of about 6MB. This means that on a machine with 512MB of main memory you use only a fraction of the available memory, namely 1.2%. This leads to the presumption that the energy savings which might be possible by using the power management features of memory chips show a direct correlation to the working set size of an application.

3.3 Bank Usage Accounting

To know exactly at every point in time which pages are used by which processes (= working set) the operating system must keep track of all memory allocations. This is done by setting up a reference table for each process, holding all the precise information. In particular, the system accounts the references for each memory bank per process. This is achieved by tracking all system calls from the application as well as tracking all page faults handled by the operating system. Hence the term 'page references' is nothing more than the number of pages which are currently owned by a particular process. With this information the allocation algorithm is able to determine where pages will be allocated next. This has the advantage that there is no need for a page table walk to get the current working set of an application.

3.4 Page Allocation Strategy

To turn off some of the available memory banks, the application running should use as few memory banks as possible. This can only be done if the physical pages used by an application are clustered into a minimum of memory banks. With the help of the bank usage table the page allocator can determine where to allocate the next page. Only if a page allocation fails a new memory bank is activated.

But this alone will not have a large impact on energy consumption, because memory banks may be split between different applications operating on the same memory modules. To wipe out the effect of fragmentation a second approach is used: If the page allocation fails (because the referenced memory banks of an application don't have any free pages left), the page allocator tries to reclaim a page. The process of reclaiming a page covers following steps:

- Scan through used pages of a memory bank and try to find a suitable page. A suitable page for migration is a page which is not owned by the process which requests a page from this memory bank. Furthermore the process owning page should have less memory bank references than the process reclaiming the page. A suitable page for compression is a page which belongs to any process and which hasn't been accessed lately.
- If such a page is found, try to get a free page in another memory bank referenced by the process which owns the found page.
- If a free page is available, copy the page to the new place. This step is called page migration. A page is only migrated if it doesn't belong to the process trying to allocate a new page.
- If there is no free page left or the page belongs to the allocating process, try to compress the page. This step is called page compression.

The above steps prevent that the memory for a process get fragmented and ensures that it is always kept in a minimum of memory banks.

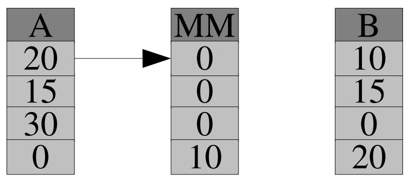
3.4.1 Page-Migration

This chapter explains in detail the process of migrating a page. To explain the underlying algorithm we assume that we have two processes A and B. Process A wants to allocate a page but the request fails, because all pages of its referenced memory banks are in use. The page allocator then searches through the used pages of the referenced memory banks of process A. For each found page (assuming it belongs to process B) it looks through the bank usage table of process B and tries to find a free page in another memory bank which is referenced by it. This ensures that the clustering of memory pages is maintained for process B. If a free page has been found the content of the page is copied to the new place.

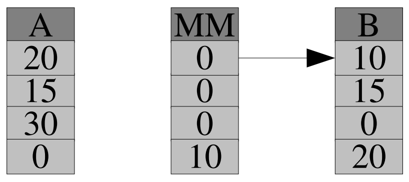
shows an example for a page allocation with page migration. In this example it is assumed that the memory is split up into four memory banks with 30 pages per bank.

As you can see without page migration a previously inactive memory bank would have to be activated for process A. Furthermore the clustering for process B doesn't get lost through the use of page migration.

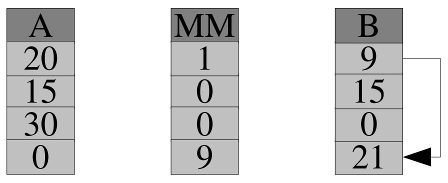
Example for a page migration process. The boxes below the letterings 'A' and 'B' show the current bank usage for the processes, e.g. the number 20 below top left box indicates that process A has allocated 20 pages from memory bank 0. The boxes below the lettering 'MM' show the free pages for the different memory banks.



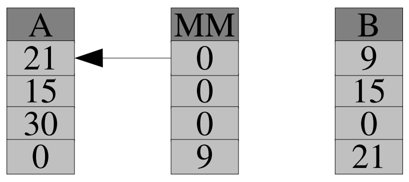
Process A wants to allocate a page from memory Bank 0, which has no pages left.



The page allocator scans through the used pages of memory bank 0 and finds a page which is used by process B.



It looks through the other referenced memory banks of process B and finds a free page in memory bank 3. The page is migrated from memory bank 0 to memory bank 3.



The now free page of memory bank 0 is given to process A.

Illustration 3 Page allocation example (A is process A, B is process B, MM is the memory management)

3.4.2 Page Compression

Page migration already offers the possibility to maintain a certain clustering of physical pages on a per process basis. But sometimes there might be a case where a page can't be migrated. This for example is true if both processes A and B (using the example from above) operate on the same memory banks. So there has to be another way to avoid the activation of an additional memory bank.

The alternative approach chosen here is to compress the page if page migration isn't possible. The strategy for compressing a page is the same as used for page migration: scan through all used pages of the referenced memory banks and try to compress a page which hasn't been accessed lately. Page compression only makes sense if the overhead for compressing a page isn't too large compared to standard swap mechanisms [5][19].

Following is a little background about compression and why it works with physical pages.

In-Memory Data Representations

Most compression algorithms only work well for certain kinds of data (mostly text or images), but would not be of great use if utilized for compressing code or data pages. This is because in-memory data is at most arbitrary – different applications operate on different kinds of data in different ways. So it cannot be assumed that there are repetitions of strings, which is the base condition for the Ziv-Lempel [9][10] compression. So we have to search for other regularities that can be used for compression. This can be done by viewing the data in memory as records and data structures. This is based on the assumption that applications usually work with arrays, whose fields are mostly one or two words. It can further be assumed that the records are usually word-aligned and that the data in those words are often numbers or pointers. Pointers by themself are nothing more than integer indices into the array of memory itself. „Integer values are usually numerically small (so that only their low-order bytes have significant information content), or else similar to other integers very nearby in memory. Likewise, pointers are likely to point to other objects nearby in memory, or be similar to other nearby pointers that is, they may point to another area of memory, but other pointers nearby may point to the same area." [5]

These are regularities which apply for most of the in-memory data, because they are mostly well-clustered. One reason is that common memory allocators tend to allocate within a small portion of memory. Other data which show regularities are, for example, virtual function table pointers in C++ objects, booleans, etc.

In-Memory Data Compression/Decompression

Compression is done by scanning the input data word by word and looking for entries that are numerically similar. This is done by testing if the high-order bits of the words are identical (partial matching). If there are repetitions, not the whole word will be written, only the low-order bits which are not identical. In the case where a word doesn't match a previous (partially or whole), the word will be written to the output. In all cases a so called tag succeeds the output, by which the applied transformation is identified.

To decompress the data, the compressed output is read through sequentially and the tags are examined. For each encountered tag the appropriate action is taken (e.g. write a word from the dictionary to the output).

3.5 Bank (De-)Activation

There are two cases when a memory bank transitions from one power state to another. The first state transition point is the scheduler. Every time when the operating system switches from one process to another the memory banks are activated or deactivated according to the bank usage table of the activated process. E.g if we have two processes A and B and these two processes share some memory banks only the remaining, unshared memory banks are deactivated.

The second state transition point is the memory management. Everytime the memory management can't satisfy a page allocation request for the used memory banks of a process a remaining unused memory bank gets activated. In turn a memory bank can be deactivated if a process drops all references to it, e.g. if a process frees all the memory previously allocated for a given memory bank.

3.6 Performance Issues versus Energy Saving

As seen above, page migration and page compression can lead to a well clustered memory layout. But on the other side this will have an impact on the performance especially if the compression or migration algorithm is called too often. Mainly this will lead to longer runtimes compared to an unmodified operating system. In the worst case there might be even no savings at all but even an increase of energy consumption as a result of a higher CPU power consumption due to the overhead in page allocation and the additional costs of copying or compressing a page. So to be efficient the energy savings of the memory system must be greater than the energy additionally consumed by the CPU.

Another part of the operating system also affected are the context switch times. The term context switch time refers to the time needed to switch from one process to another. As said before each time the operating system schedules a new process the memory banks are activated or deactivated accordingly. Depending on the memory chips used this can take quite a lot of time.

4 Implementation

To test the influences of the proposed approach the whole mechanism has been implemented into a Linux kernel. I have chosen to use Linux [12], as it is widely used and has a lot of applications suitable for testing. Furthermore the kernel is open source and well documented.

4.1 Understanding the Memory Management of the Linux Kernel

Before we can start to go deeper into the implementation details I give a brief outline of the Linux memory management.

In the Linux kernel each process has its own virtual address space (or virtual memory). This means that the processes are protected from each other. If a process claims memory from the kernel via the various system calls (like *mmap* or *sbrk*), the kernel tries to find a free memory region large enough to satisfy the request. If such a region has been found physical pages are mapped to that region. If a process releases a claimed memory region then all physical pages belonging to this region are freed [17][18].

The physical memory is divided into three so called „zones“. Each zone has certain properties, which are suited for special allocation purposes. Following is a short description of each zone.

- *DMA* zone – The memory in this zone is suitable for DMA-transactions
- *Normal* zone – Pages in this zone can be used for anything
- *Highmem* zone – Pages in this zone are not directly accessible by the kernel because they are not mapped into kernel memory

Each virtual address space is managed by a structure `mm_struct`. This structure holds all necessary information needed to maintain the address space. In particular it has a reference to the page directory, as well as references to all the allocated memory regions. Because Linux is capable of multithreading a virtual address space may be referenced by multiple processes at a time.

4.2 Simulating a Multibank Memory System

Up to now there a no RAM chips available which allow a direct control of their power states. To bypass this issue a multibank memory system has been simulated. This was done by extending the already available zone mechanism used by Linux so that it is no longer limited to three zones. To achieve this, some parts of the basic kernel memory system have been rewritten.

- The memory setup process has been modified, so that it can setup an arbitrary number of zones (which may be passed as a kernel parameter).
- The page allocator has been modified to operate on any number of zones, but is still compatible with the standard allocator.
- To simulate the transition times of the different power states a simple delay loop has been used.

In the following the term 'zone' corresponds to a single memory bank as this is the semantic used by the Linux kernel.

Another thing which has to be taken into account is that all the device drivers reside in kernel space. So it is not possible to suspend those memory zones which are used by these device drivers. For example, if an interrupt occurs, it is very bad if the memory zone which contains the interrupt service routine is suspended. Then you have to wait until the memory zone gets woken up. This would lead to very bad interrupt latencies. To avoid these drawbacks, the available memory is split up into kernel zones and user zones. Kernel zones are permanently active and cannot be suspended in contrast to user zones which are activated and suspended as needed. Furthermore user applications can't allocate memory which belongs to the kernel and vice versa.

To get some results which come close to what can be expected three different power states have been simulated: powerdown, nap and active. These match the power states of the RAMBUS DRAM memory chip presented in chapter 2.1. Standby time had to be omitted because it was impossible to retrieve the actual memory references of an application with which it would have been possible to distinguish between active and standby mode.

Because the installed memory on the used test machine didn't have any power management features the determination of the energy usage have been made by measuring the durations of the different power states. This has been done separately for each process. Each per process account is refreshed everytime the process gets scheduled. With this a very precise prediction of the consumed energy for each memory bank can be made.

4.3 Bank Usage Accounting

To implement the proposed solution correctly the kernel has to keep track of all referenced pages per process. Because Linux doesn't supply such a mechanism the kernel has been modified. This was done by providing a structure which counts all the necessary information for a zone. shows this data structure.

```
typedef struct zonestat_struct
{
        int             nreferences;                    /* number of references in this zone */
        zone_t          *zone;                          /* reference to zone */
        zone_time_t     mode_times[ZONE_NUM_MODES];     /* the times spend in each powermode */
        int             nallocated;                     /* the number of allocated pages */
        int             nfreed;                                 /* the number of freed pages */
        int             nmigrations;                    /* the number of migrations */
        int             ncompressions;                  /* the number of compressions */
        int             ndecompressions;                /* the number of decompressions */
}zonestat_t;
```

Illustration 4 Structure holding all accounting informations for a certain zone

Each address space (*mm_struct*) has an array of such data structures. The size of the array complies with the number of zones being simulated. Because Linux supports multithreading each process also has its own array of these data structures.

Given this structure it is possible to be very precise about the actual memory consumption of an application.

4.4 Implementation of the Page Allocation Strategy

The page allocation scheme is explained in detail in the following chapter. Some major changes had to be made to accommodate the kernel to the proposed modifications.

4.4.1 Page Scanning

If the page allocation fails because there are no free pages left for the requested memory banks, the kernel searches for a page which can be either migrated or compressed. To perform this operation the kernel keeps a list of used pages for each memory zone. Each page by itself stores a pointer to the memory management structure by which it is currently referenced. In addition, the page holds information about the virtual memory address to which it is mapped. With this information it is easy to determine the actual page table entry and to check if the page can be migrated or compressed.

The number of pages which will be scanned each run can be set through a system call. This ensures that the kernel doesn't spend too much time with page scanning.

4.4.2 Page Migration

A page will be migrated if the process to which it belongs has fewer references to the memory zone than the process requesting the page. In addition the process must have other memory banks in use to which the page can be migrated. If all these conditions hold, the page allocator tries to get a free page from the other referenced memory banks using a very simple and fast query mechanism. If there is a free page than the content of the page will be copied to the new place and the corresponding page table entry will be updated. To complete this task the page table entry of the migrated page has to be updated.

4.4.3 Page Compression

If a page could not be migrated, the kernel tries to compress it. The page will only be compressed if a certain period of time has been elapsed since the last access. This ensures that not currently accessed page gets swapped out. To store a compressed page I have introduced a so called „compression cache“ which is of fixed size and resides in kernel memory. The compression algorithm used is the LZO [11] data compression algorithm developed by Markus Oberhumer. There are several other compression algorithms available like Wkdm [5] or Wk4x4 [5] which are suitable for page compression, but I have concentrated on LZO only to show that is possible to save energy by using page compression. If a page has been compressed the corresponding page table entry is set to a value which indicates the position in the compression cache.

4.5 Bank (De-)Activation

When a new process is created all memory zones which are not referenced are suspended to powerdown mode. As soon as a process allocates a page from a former unreferenced memory bank it becomes activated. If a process releases all references to a certain memory bank it is transitioned to the next lower power mode.

Another point in the system where memory banks are (de-)activated is the scheduler. The policy for deactivation is as follows: An unreferenced memory bank is first placed into nap mode. If the memory bank remains unreferenced during the next scheduler quantum it is placed into powerdown mode. As soon as a memory bank gets referenced it is transitioned to active mode regardless of the former power mode.

4.6 New System Calls

There a two new system calls which come with the implementation. The first is used to configure the memory system.

- `tune_mm (int func, void *value)`

With this system call it is possible to setup specific parameters like the threshold for compression or changing the search algorithm.

The second new system call extends the wait4 system call by an additional parameter.

- `wait4_x (pid_t pid, unsigned int *stat_addr, int options, struct rusage *ru, struct mmusage *mu)`

With this api call it is possible to retrieve statistics about the zone usage. Illustration 5 shows the layout of the data structure `mmusage`. This structure contains information about the pages which were totally allocated and freed, as well as the number of compressions or migrations. All this information is provided for each zone.

```
struct mmzoneusage
{
        struct timeval times[3];            /* Accounted times for each power mode */
        long           nallocated;          /* Number of allocations */
        long           nfreed;                      /* Number of releases */
        long           nmigrations;         /* Number of migrations */
        long           ncompressions;       /* Number of compressions */
        long           ndecompressions;     /* Number of decompressions */
};

struct mmusage
{
        long                   nzones;              /* Number of zones */
        struct mmzoneusage     zoneusage[0];  /* Usage for each zone */
};
```

Illustration 5 Data structure used in extended wait4 system call

5 Experimental Results

This chapter treats with the results which were taken from the implementation and compares them with an unmodified kernel. There a mainly two categories which have been used to show the efficiency of the implementation:

- Energy consumption
- Runtime

The comparison of the energy usage involves two steps:

- computing a normalized energy ratio, and
- computing the real power consumption taking the normalized energy ratio into account

The two steps were necessary because there was no power aware memory system available with which measurements could have been taken.

5.1 How to Compare the Results

In the following I will briefly explain the steps that were made to compare the results. Especially the calculation of the energy and runtime ratios is explained.

5.1.1 Normalized Energy Ratio

This is the ratio gained by taking the different power mode times into account and computing a quotient between this value and the runtime of the unmodified kernel. This gives us the energy savings for the memory system compared to the unmodified system.

To compute the value for comparison the times for each zone and for each power state have been measured and an average power state time has been calculated. For example, to compute the average active time the sum of active times of all memory zones is taken and divided through the number of memory zones:

active-time = (sum of active-times of all zones) / number of zones

This calculation is done for every power state and a pseudo energy value is computed via the following formula ('*m*' indicates modified system):

energy(m) = active-time * 1 + nap-time * 0.1 + powerdown-time * 0.01

The power saving factors (namely 1, 0.1, 0.01) have been taken from the RAMBUS specification (see chapter 2.1 for details).

In contrast the unmodified system has only one power state namely active state. Therefore the energy calculation for the unmodified system (indicated by '*u*') reduces to:

energy(u) = active-time * 1

Having computed both values (for the modified and unmodified system) a energy ratio can be computed by dividing both values ('*m*' = modified system, '*u*' = unmodified system):

normalized energy ratio = energy(m) / energy(u)

This value indicates the energy savings of the memory system compared to an unmodified system. This model is not 100% exact as it lacks the accounting of the standby time of the memory system. This is due to the fact that it is impossible to capture the actual memory accesses at runtime (this could only be done if we simulate the whole system). Therefore I omit the standby time and assume that if the memory system is not in any power saving mode it is in active mode the whole time. So the measured results are slightly higher than what could be expected if using a power aware memory system.

5.1.2 Real Energy Ratio

This is the ratio between the actual measured power consumption of the unmodified and the modified kernel where both memory and CPU energy have been measured. The former normalized energy ratio only takes into account the energy savings of the memory system. But to make a reliable statement about the actual energy savings the power consumption of both CPU and memory has to be taken into account.

To compute the resulting energy consumption for the unmodified system following formula has been used:

energy(u) = cpu power consumption(u) + memory power consumption(u)

The energy consumption of the modified system has to be computed differently, because we want to take into account the energy savings of the memory system:

energy(m) = cpu power consumption(m) + (normalized energy ratio * memory power consumption(u))

Note that the power consumption for the memory is not actually taken from the modified system but computed via normalized energy ratio and memory power consumption of the unmodified system.

5.1.3 Runtime Ratio

A third parameter which makes a statement about the efficiency of the proposed solution is the influence it has on the performance of the whole system. This can be captured by measuring the runtimes of the applications both on the modified and unmodified system. By comparing both values we can get an idea on the impact of power aware memory management on the performance.

5.1.4 Comparing Normalized Energy vs. Real Energy

Before the actual results are shown it has to be considered that the normalized energy (which refers to the memory system only) is significantly lower than the actual energy consumed by the test installation. It would be unfair to only compare the energies consumed by the memory and to omit the power consumption of the CPU. Especially in the proposed mechanism the CPU has to do more work as before (memory management overhead).

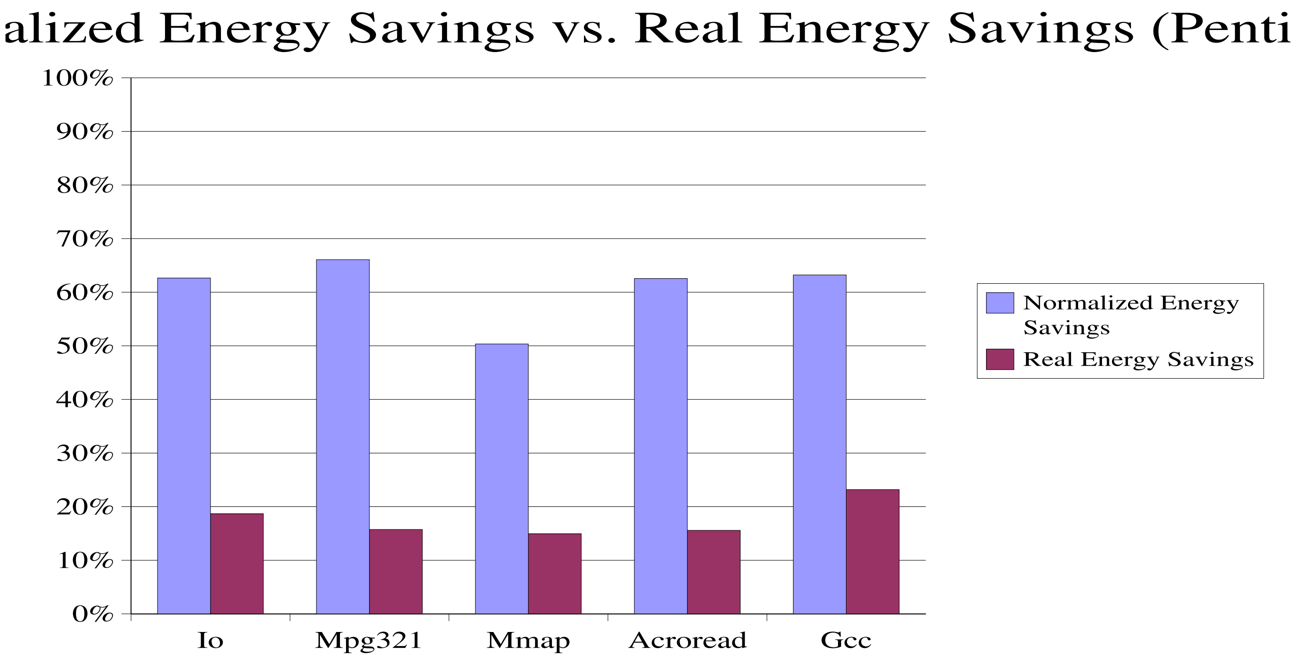


Illustration 6 Comparison of normalized energy and real energy

Illustration 6 shows the results for a kernel with an overall number of 64 zones and 16 kernel zones. The energy savings for the memory system are at least 50% (indicated by the normalized energy ratio). In contrast the real energy consumption is normally higher because here the CPU power consumption has also to be taken into account. Another factor which you have to keep in mind also is that the CPU normally consumes much more power than the memory system (in this case the CPU consumes at peak rates about 3 times more power than the memory system). E.g. 'Mpg321' uses the CPU very intense so the energy saving is lower compared to 'Mmap' though the normalized energy ratio of 'Mpg321' is lower than that of 'Mmap'.

5.2 Testenvironment and Testsuite

The modified Linux kernel has been tested on an Intel Pentium 4 [21] Processor with 2Ghz on an ASUS P4B266-SE [22] motherboard with Intel 845D Chipset and 1024Mb of DDR RAM running at 266Mhz. To test the implementation a wide range of applications as well as some synthetic benchmarks have been used.

The testsuite covers following subtopics:

- I/O performance
- Cpu performance
- Memory performance

- A mixture of the previous

Testsuite description:

- Io – Stressing the filesystem with tar where I archive a folder with about 1000 subfolders and 14000 files which has a total size of 129MB.
- Mpg321 – Stressing the CPU with a conversion of a mp3 file to a pcm file. The file size is about 4MB.
- Mmap – Stressing the memory system with reading and writing a huge memory mapped array which is 128MB in size.
- Acroread – Converting multiple pdf files to postscript. There are a overall of 4 pdf files which vary in size from 1.8MB to 4.5MB.
- Gcc – Compiling the Linux kernel.

All tests were executed ten times and the average results were taken. This guarantees fairness in the comparison.

5.3 Measurements

This chapter will discuss the results which were retrieved during the testing phase. First we want to focus on how the system behaves if we use different memory configurations, i.e. using different numbers of memory banks and using different numbers of kernel zones. Mainly we want to investigate how the energy savings and runtimes differ for the different configurations. Another fact we want to concentrate on is how the energy savings differ for different power management policies.

5.3.1 Using Different Kernel Zones

First I investigate how the energy savings differ if we use different configurations for the number of kernel zones. In the illustrations the first number in brackets indicate the overall number of memory banks whereas the second number indicates the number of kernel zones.

Energy Savings

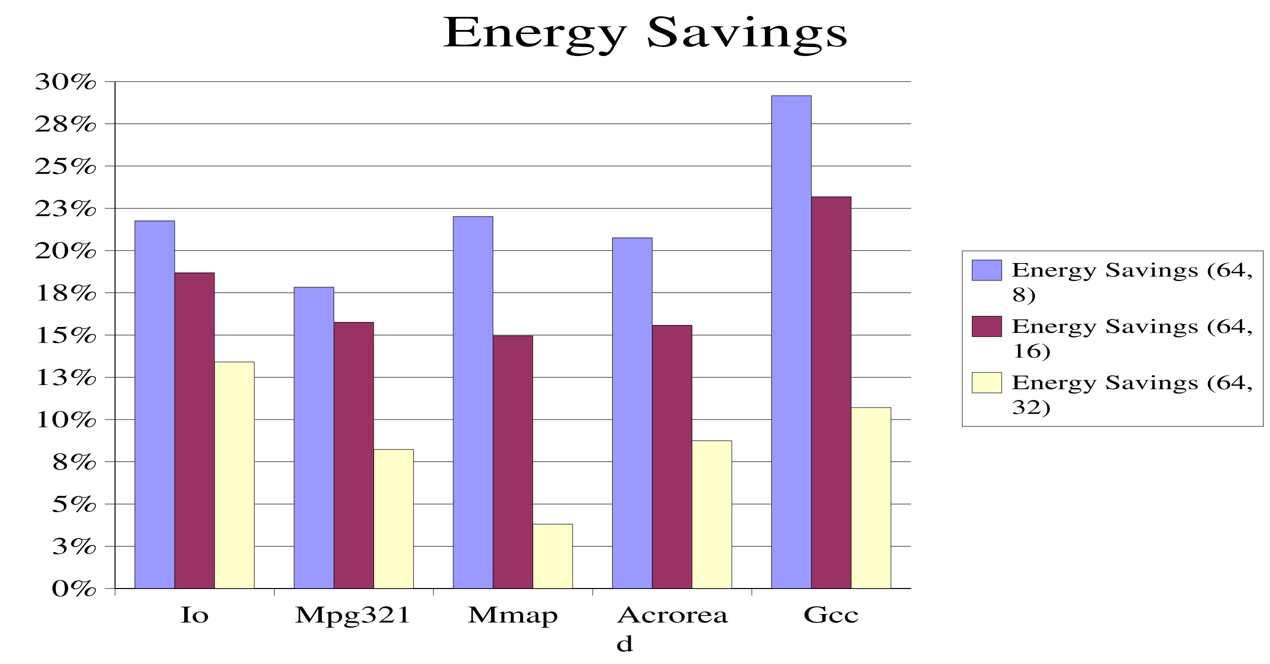


Illustration 7 Energy savings for different kernel zones

Illustration 7 shows how energy savings differ for different configurations of the kernel zones. As you can see the more kernel zones there are, the less the energy savings are. This in fact is an implication of the implementation. As said in chapter 4.2 the kernel zones are permanently active meaning that they consume always power and cannot be suspended.

Runtime Comparison

In you can see how the runtime depends on the configuration of the kernel zones. Especially for I/O bound applications the configuration has the greatest influence. This is an effect of the file system cache mechanism used by the Linux kernel. The less memory the kernel has at its disposal the less (meta-)data can be cached meaning that the harddisk has to be accessed more often. This in turn leads to longer runtimes. The other applications which are not bound by the I/O subsystem don't suffer that much from different configurations. The runtime delays for I/O bound applications might be less if we have a system where only part of the memory is used as filesystem caches.

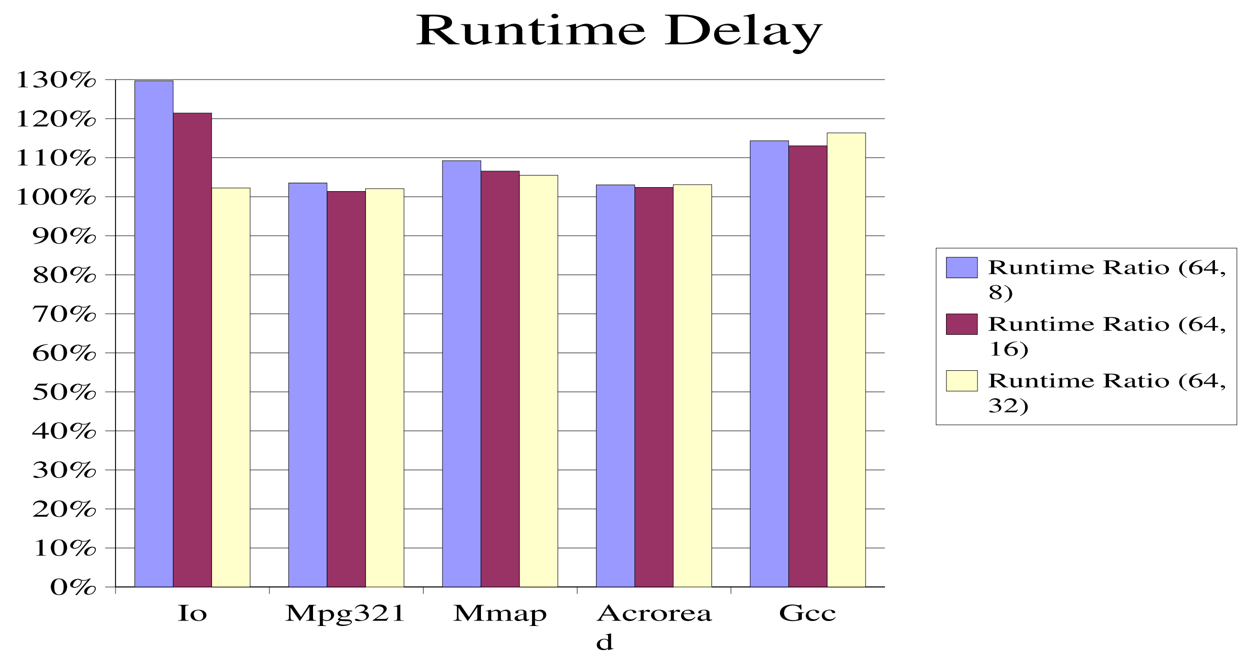


Illustration 8 Runtime delay for different kernel zones

5.3.2 Using Different Memory Banks

To compare how the systems reacts to different numbers of memory zones the number of kernel zones has been set to a fixed size (¼ of the overall number of zones).

Energy Savings

shows the energy savings for different configurations of memory zones. As you can see energy savings are lowest, if there are only few memory banks. The more memory banks there are the better the savings. This can be interpreted as a result of the granularity by which the memory is split up, meaning that the more memory banks there are the better the operating system can control the clustering of individual pages for the applications. Another reason is that with more memory banks installed the activation of one memory bank doesn't carry so much weight. But the increase of the energy savings are not linear which is mainly from the fact that there is a break-even point where no more energy savings are possible.

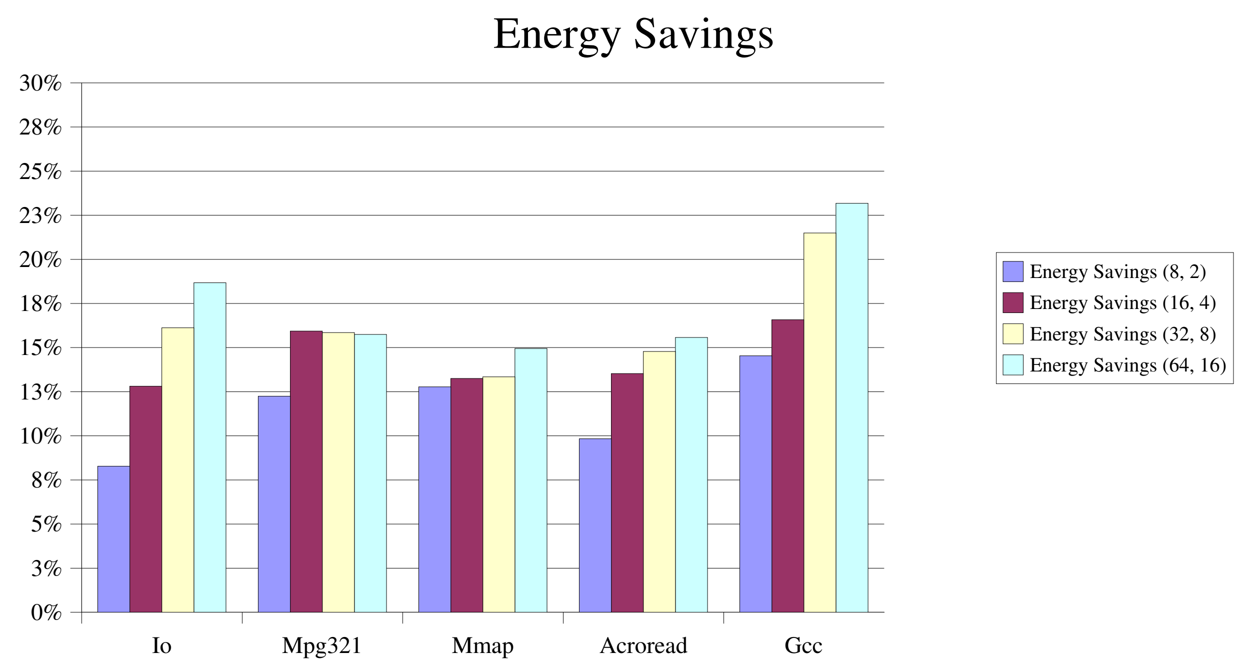


Illustration 9 Energy comparison for a different number of memory banks

Runtime Comparison

If you take a look at you can see that for all configurations the modified memory management has a great influence on the runtime. Especially in cases where the page allocation is frequently used (Io, Gcc) the runtimes increase significantly. The changes between the different configurations are only of subtle nature, i.e. they don't correlate directly to the runtime.

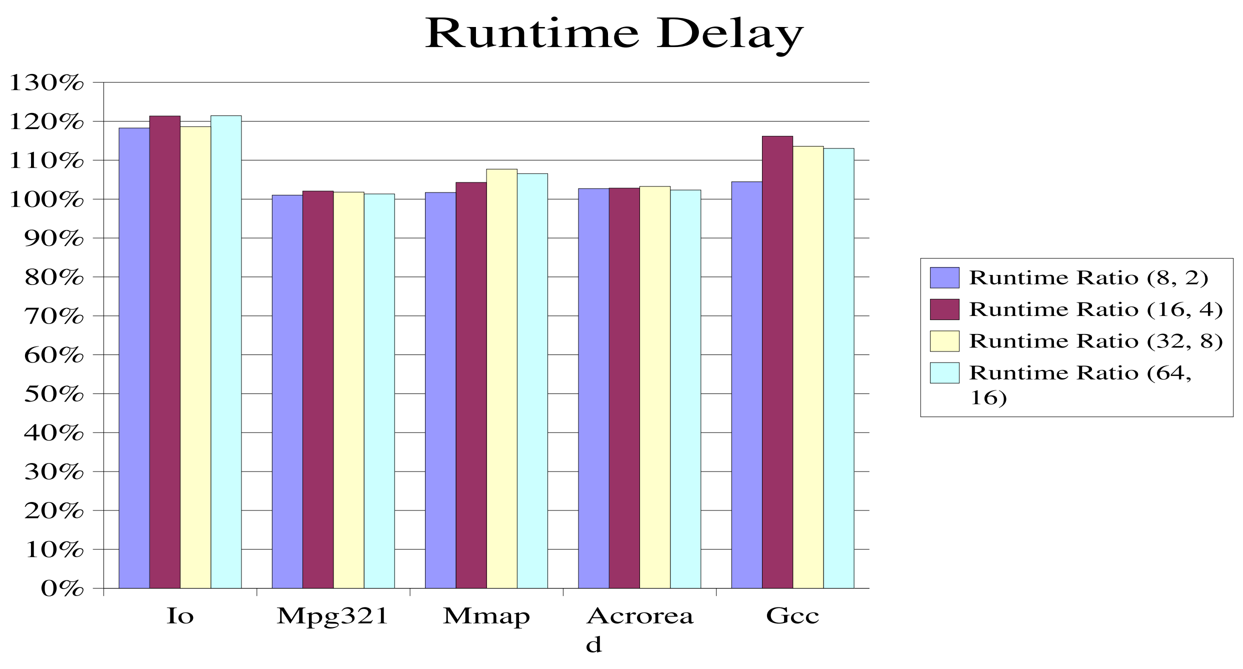


Illustration 10 Runtime comparison for a different number of memory banks

5.3.3 Using Different Power Management Policies

All the previous results were based on a three state power management policy, i.e. we used three different power states (*active*, *nap*, *powerdown*). Now we want to investigate if it is useful if we use only two power states, e.g. if we use *nap*-mode only to save energy.

Energy Savings

In you can see that a policy which utilizes only the *nap* mode has the overall lowest energy savings whereas policies which use *powerdown* or *powerdown* and *nap* mode achieve the highest energy savings. If you take a closer look you will see that the *powerdown* only policy performs best for the wide range of applications. *Powerdown* only is best because the memory banks transition from active to a low power state only during context switching or during page allocation. This means that *nap* mode is not of great use for this approach.

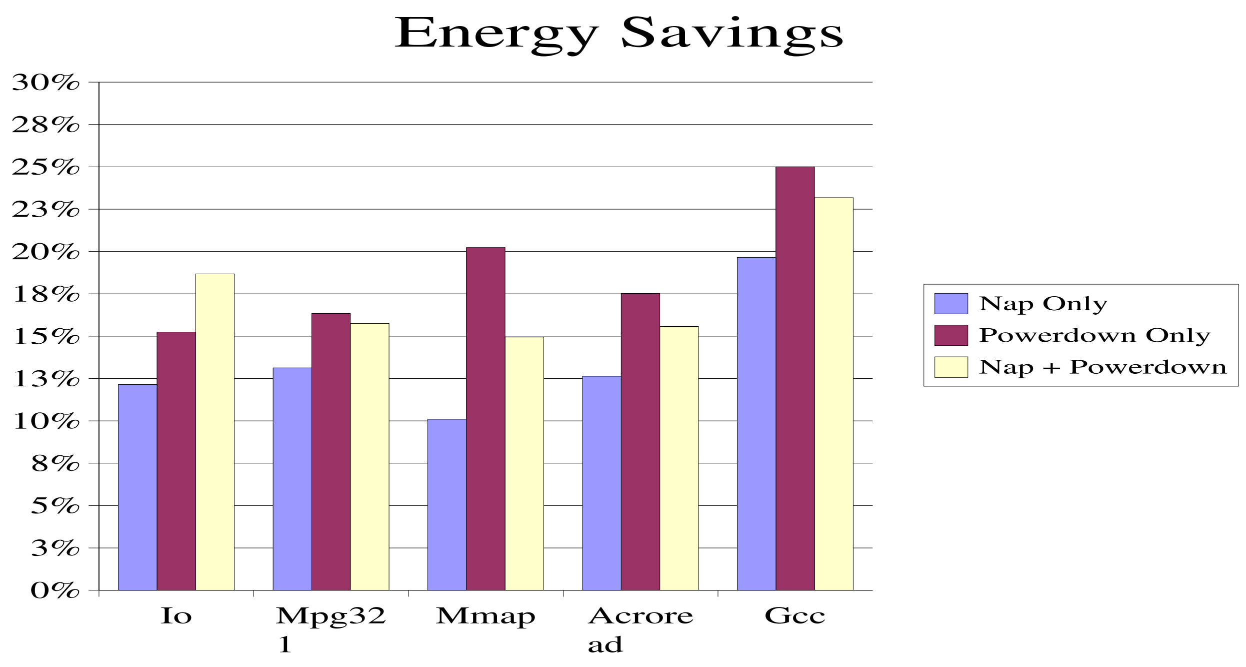


Illustration 11 Energy savings for different power management policies

The results would look different if we would incorporate some sort of hardware directed power management, where the memory controller has the ability to place the memory banks into different low power modes [6]. Then *nap* + *powerdown* mode might be more efficient.

Runtime Delay

An interesting question is if different power management policies affect the runtime. If you take a look at you will see that the changes in runtime are only of subtle nature. This is an effect of the transition times which are measured in clock cycles and are less than 100ns for the longest transition time from powerdown to active mode. Memory banks are normally only transitioned from one state to another through the context switch.

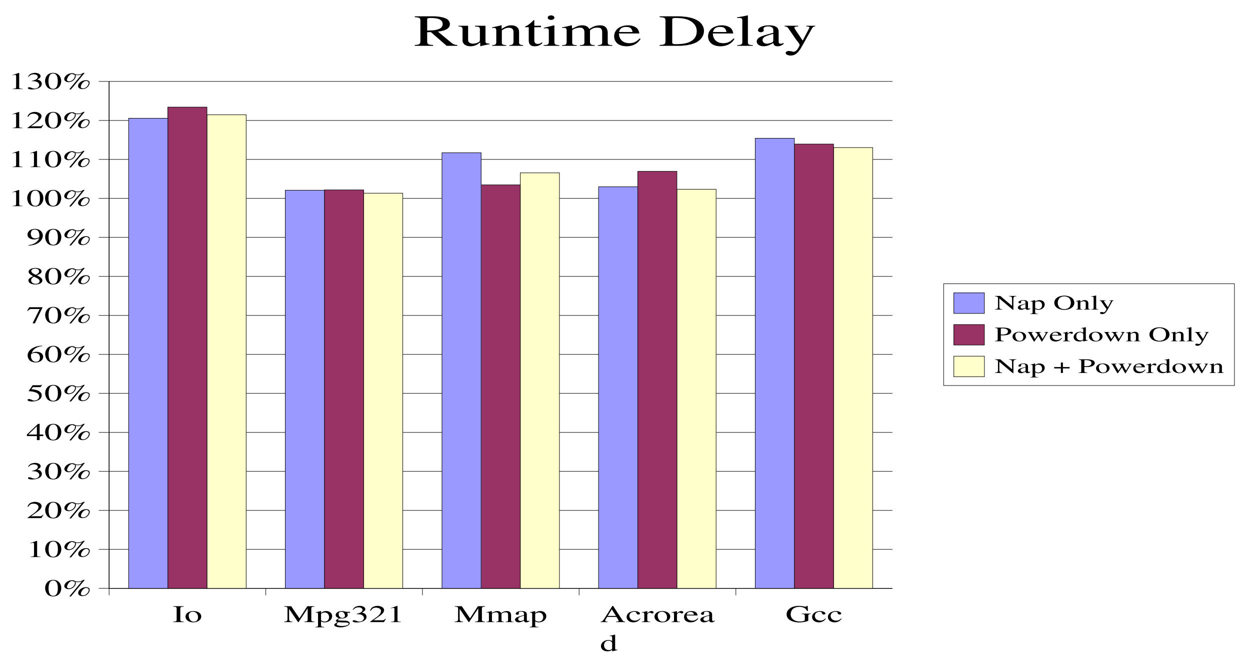


Illustration 12 Runtime comparison for different power management policies

In the terms of energy efficiency the combination of nap + powerdown mode seems to have the most advantages with respect to energy savings and runtime delays.

5.4 Subtle Changes of the Implementation

Up to now we have seen what major benefits or drawbacks there are. Now we want to investigate the more subtle changes which come with the implementation.

Context Switch Times

Another impact which comes with the implementation is the lengthening of context switch times (the time which is needed to switch from one process to another). This comes from the fact that during context switching the used memory banks of a newly activated processes need to be activated. Furthermore the kernel has to do usage accounting for the different memory banks.

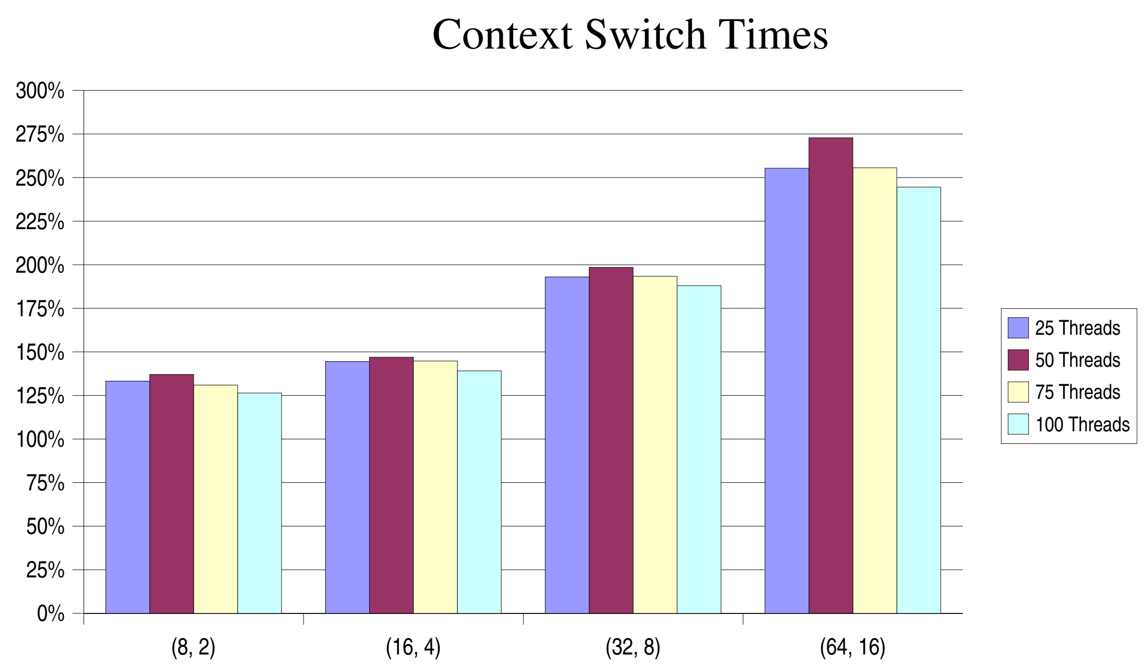


Illustration 13 Comparison of context switch times

In Illustration 13 you can see the delay effects on the context switch times for different configurations. In general it can be said, that the context switch times increase with the number of memory banks used. This is not only an effect of the transition times but also an effect of the usage accounting which has to be done for each process and memory bank.

6 Discussion

The previous chapters have shown that it is possible to save energy at least for the particular test environment which I have used. Now I want to discuss if this is only a special case or if it can be said that the approach makes sense for all environments.

6.1 Overhead versus Energy Saving

The major question is, does it makes sense to use energy aware memory management at all? On the one side we have seen that there are energy savings throughout all testcases. On the other side there is the drawback of increased runtimes. But as it has been shown that this increase only occurs in certain cases. Depending on the operational area these modifications are very useful in terms of energy efficiency. Especially in the wide field of mobile computing it would make sense to incorporate this new technology.

6.2 Correlation between Working Set Size and Normalized Energy Savings

The first interesting question is how the working set size of an application correlates to the corresponding normalized energy ratios. If you take a look at you see that there is a direct link between the working set size and the normalized energy savings. The configuration used has an overall number of 64 zones with 16 kernel zones.

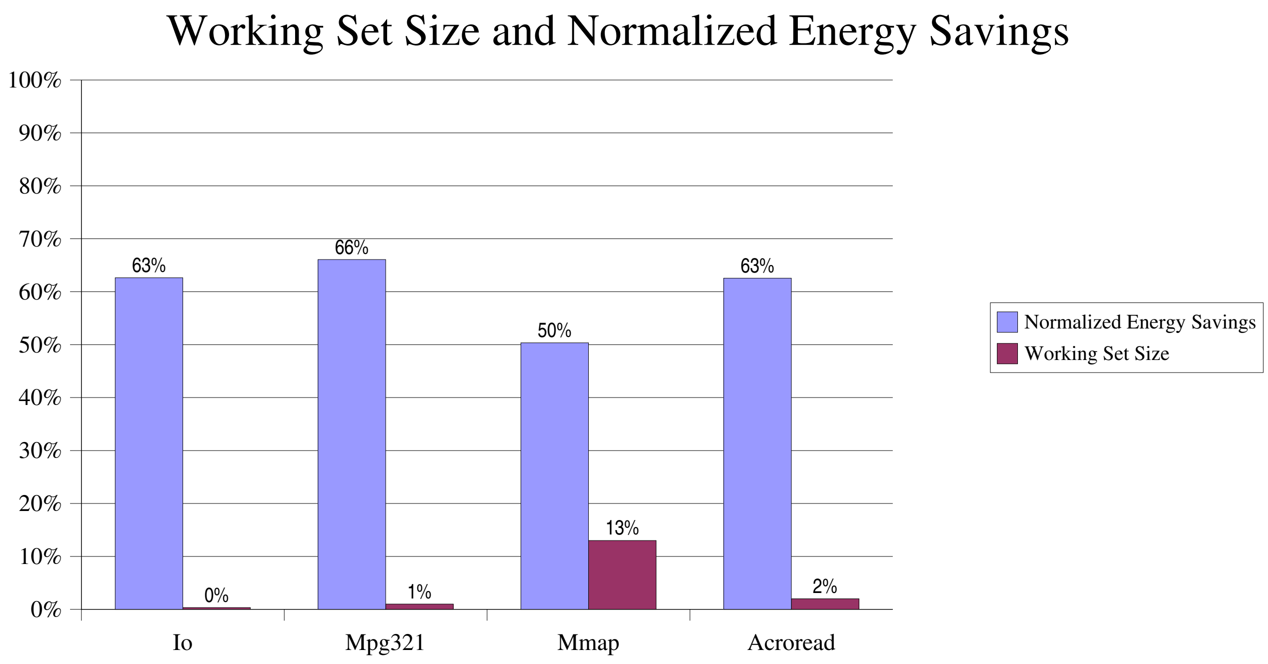


Illustration 14 Correlation between working set size and normalized energy ratio

This information substantiates the presumption which have been made in chapter 3.2, namely that applications normally use only a fraction of the available memory and therefore energy savings are possible. As shown before (Illustration 6) the results look different if you take the CPU into account. Then you don't have a direct correlation between the working set size and the achieved energy savings.

6.3 Correlation between Memory Size and Normalized Energy Ratio

As shown in the previous chapter there is a direct correlation between the working set size and the achieved energy savings. Another presumption is that the energy efficiency drops if we have less main memory installed but the working set of an application doesn't change. This is based on the assumption that the memory is more utilized than before. E.g. the test application 'Mpg321' has a working set size of about 6MB. If we have 1024MB of main memory installed this means that we use about 0.6% of the memory. If we reduce the memory size to 128MB the usage rises to about 4.7%. This effect should be visible if we do some comparisons between configurations with different memory sizes.

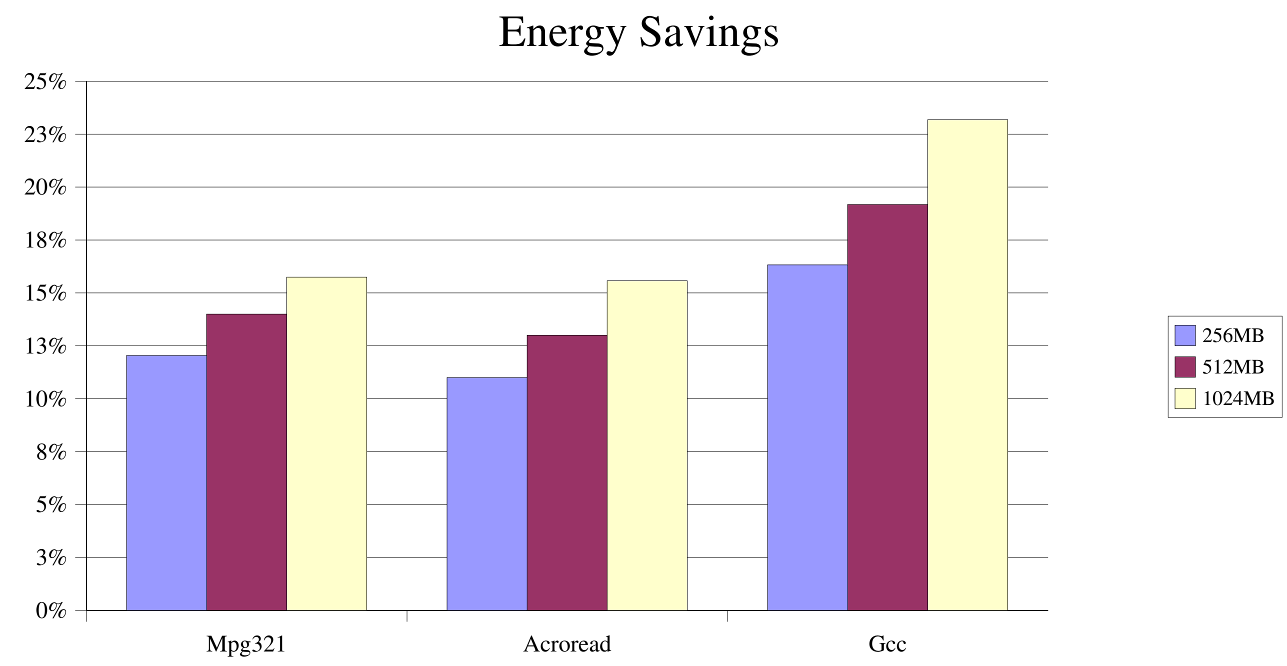


Illustration 15 Energy savings for different memory configurations

As you can seen in Illustration 15 the size of the memory has direct influence on the energy efficiency of the memory system. This is a logical deduction of the previous chapter, where I have shown that there is a direct link between the working set size and the achieved energy savings. Now we see the effects of an increasing ratio between the unchanging working set size and the overall decreasing memory size. E.g. the ratio between working set size and memory of the application 'Mpg321' increases from about 0.6% to about 2.3% if I change the memory size from 1024MB to 256MB.

6.4 Energy Efficiency Factors

Up to now we have seen that there have always been energy savings for my test environment. But can those energy savings be expected if we you use different hardware, e.g. using a CPU which is much slower than the one which I used? To make a prediction about the expected energy savings we should focus on those elements which lead to an improved energy efficiency. I can classify three different factors:

1. The power consumption ratio between memory and CPU, i.e. how much energy is

consumed by the memory compared to the CPU.

2. The operating system which is responsible for the memory management and therefore for the page placement.
3. The CPU speed which determines how fast a page can be migrated or compressed.

All these elements have a great influence on the energy efficiency. E.g. as seen before using memory chips with different configurations of the number of memory banks leads to different energy savings.

CPU power consumption vs. Memory power consumption

The factor with the greatest influence is the power consumption ratio between memory and CPU. As seen in Illustration 6 in chapter 5.1.4 there is a wide gap between the energy savings of the memory system and the overall energy savings which incorporate the CPU power consumption. This leads to the assumption that a low power CPU (e.g. Mobile Intel Pentium 4 Processor – M [20]) is more suitable than processors which are not designed for energy efficiency. In my test environment the ratio between CPU and memory was about 3 : 1 at peak power consumption. Reducing this ratio would lead to an improved energy efficiency. E.g. the ratio between CPU and memory on Intel IQ 80310 [30] evaluation board with an Intel 80200 [29] Processor and 32 MB SD-RAM memory the ratio is about 1 : 1 (this an estimate based on measuring the power consumption). To show an estimate about different energy efficiencies of different CPU to memory ratios look at . Here I have simulated a different CPU to memory ratio by halving the power consumption of the CPU.

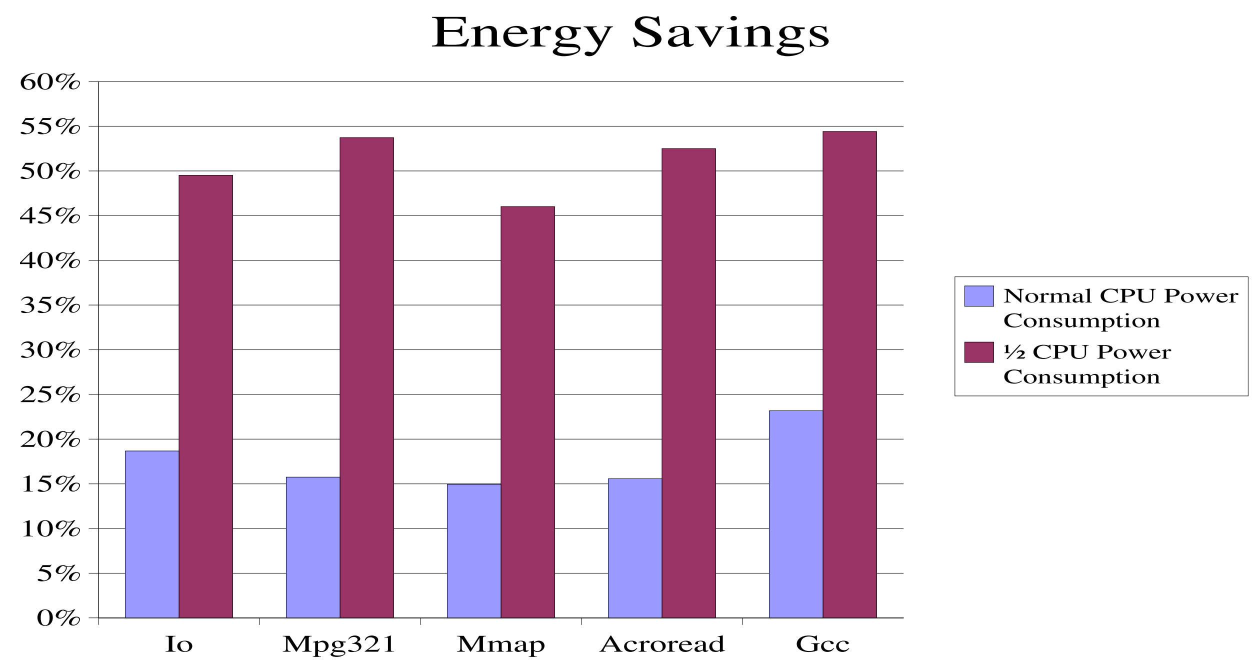


Illustration 16Comparison of different CPU power consumptions

You can see that energy savings raise by a factor about 3. This emphasizes the role of the ratio between CPU and memory.

Costs of Page Migration and Compression

Two parameters which might have also an effect on energy efficiency are the energy costs of page migration and page compression. On my test machine I have determined following absolute energy values and duration times for copying or compressing a page:

	Energy CPU [mJ]	*Energy Memory [mJ]*	*Duration [µs]*
Page Compression	1.02	0.34	304
Page Decompression	0.0096	0.0039	3
Page Copying	0.0055	0.0016	1.5

As you can see the costs for such operations are very low and are in the range of a few millijoules. This means that these factors don't have a great influence on the energy efficiency. Of course this is only true for a processor which has great processing power (such as a Pentium 4). But if you have a processor which has much less processing power (as mostly used in PDAs [23]) this could become a bottleneck if you use this operations too often. To get a feeling for the frequency of this function calls look at the following table (configuration with 64 zones and 16 kernelzones, values are averaged):

	Nr. of compressions	*Nr. of decompressions*	*Nr. of migrations*
Io	0	0	41.7
Mpg321	12	0	60.6
Mmap	0	0	33.3
Acroread	0	0	7.2
Gcc	0	0	24420

As you can see the compression functions are used very sparse in contrast to the page migration code which is used very often. The 'Gcc' case has to be treated specially because this is not only one process but a whole bunch of processes created by compiling the kernel which consists of several thousands of source files. For each subprocess the information is accounted and summed together into one account. Therefore we have this great value of migrations. To get back to the question if these functions could become a bottleneck I would say that this is not true. E.g. if you have a processor with less processing power it normally comes with less power consumption too (as it has fewer transistors). This in turn leads to an decreased ratio between CPU and memory power consumption, which has the effect that the overall energy efficiency increases.

To further reinforce this statement I give you some absolute energy values which have been measured. Then you will see how tiny the costs of page migration or compression are compared to the overall energy consumption.

	Energy [J]	*Energy costs of migration / compression [mJ]*	*Ratio [%]*
Io	24.59	0.3	0.00122
Mpg321	43.08	0.43	0.00099

	Energy [J]	***Energy costs of migration / compression [mJ]***	***Ratio [%]***
Mmap	14.6	0.24	0.00164
Acroread	20.08	0.05	0.00025
Gcc	409.63	173.38	0.04232

As you can see the total costs for page migration / compression don't have any great influence at all.

Note: Though it looks like that there is only one application running at a time it is a fact that there are normally many background daemons and at least one shell which are running at the same time (on the Linux system which I used). This is the answer to the question: Why is there any page compression or migration if we have only one application running?

6.5 TLB Cache Effects of Page Migration and Compression

A last question which needs to be discussed is how the TLB (Translation Lookaside Buffer [24][25]) cache is affected by means of page migration and page compression. Page compression used so far in this thesis is nothing more than a different implementation of a swap mechanism which uses RAM instead of a harddisk as swap device. As the analysis has shown page compression is used only rarely, so it plays no important role in terms of cache effects. On the other hand page migration is a process which is used often so there might be the assumption that the cache is affected. But if you take a closer look on how page migration is actually working then you will see that this has no effect. The TLB can only be affected if we change entries in page table of the current process. But the algorithm states out that only pages which do n't belong to the current process can be migrated. This has the effect that no page table entries of the current process are modified by this mechanism.

6.6 Energy Savings in a Multitasking Environment

Up so far I have only concentrated on a single application running at a time. An interesting fact may be how the system behaves if we have multiple applications running at a time. To test this scenario I have chosen to take the applications 'Mpg321' and 'Acroread' and let them run at the same time. The question here is how the energy savings are for such a situation: Does the system consume more or less power as before?

If you look at we see a surprising result. The normalized energy ratio as well as the real energy consumption for both applications running at the same time is the same as for the single applications.

This is a fact from the observation that the page migration is much more used than before. For the single application scenario I had an average number of 60.6 / 7.2 migrations for Mpg321 / Acroread. If both applications run at the same time the average number of migrations increased to 321.3 (only one value, because I started the applications at the same time and summed the accounting information together in the end). This means that the page placement is better than before leading to a well clustered layout of the pages. And as I said before there are normally some background daemons and at least one shell which are running all the time. So if you start a new application the page allocation algorithm won't be able to further refine the clustering of the pages leading to this interesting result. But this changes if two or more new applications are started. Then

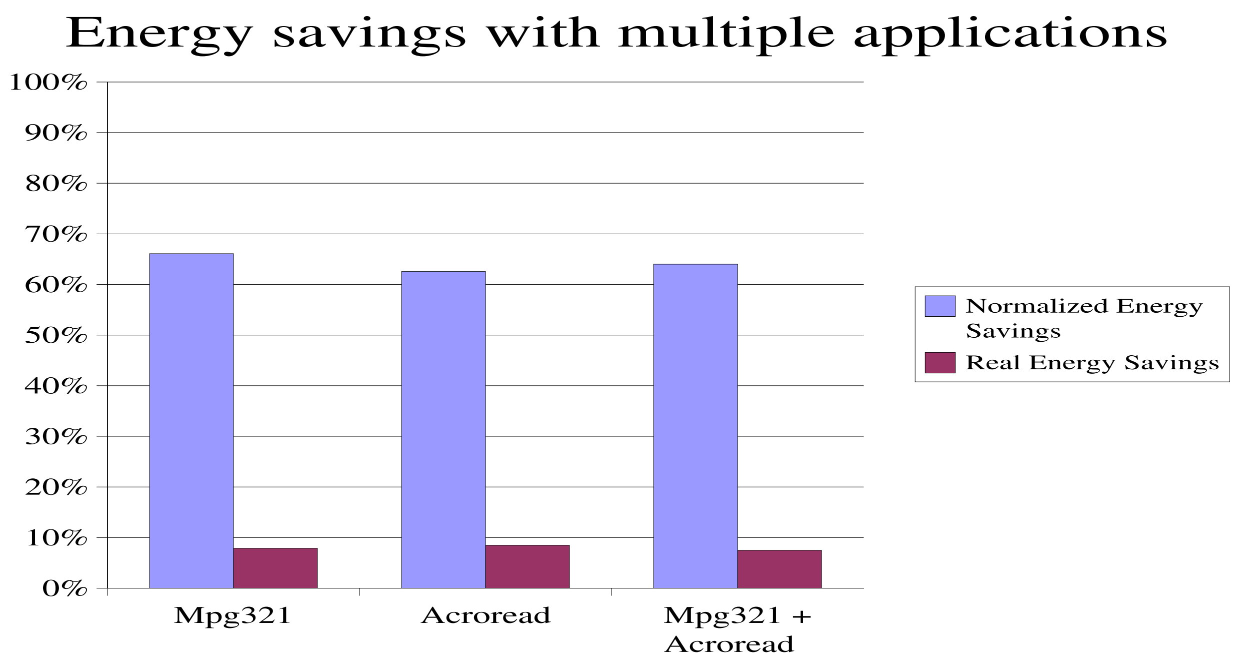


Illustration 17 Energy savings with multiple applications

the page allocation algorithm has the possibility to optimal place new pages.

7 Future Work

Although this diploma thesis presented a mechanism which can save a great amount of energy there are still some things which can be further improved. For example I concentrated on memory management only and did no modifications to the scheduler.

7.1 Deactivation of Kernel Zones

One big improvement for energy efficiency would be the deactivation of kernel zones. In the current approach a certain number of zones are associated with the kernel and are always active. This is due to the structure of the Linux kernel which has been used for the implementation.

First of all the Linux kernel is more or less a monolithic [4] kernel, which means that normally all drivers needed for the system are compiled into the kernel. Another thing to keep in mind is that the kernel uses part of the memory to cache disk accesses. Which in turn has the effect that if you limit the amount of memory usable for caching the performance of the whole system will be affected. At least the kernel needs some memory which it can use for management reasons, i.e. if an application opens a file a corresponding file structure is allocated by the kernel. For this reasons it is necessary to reserve some of the memory for the kernel.

To bypass these limitations an alternative operating system approach would likely be more efficient. In this particular case I would prefer using some sort of microkernel [4] where only an absolute minimum of management tasks are taken care of by the kernel. These are mainly process management, physical memory management and interrupt handling. All other things like device drivers or file systems would be implemented with normal userspace applications. This would have the benefit that only a minimum number of memory chips would have to stay always in active mode (namely the memory banks where the kernel is located).

7.2 Optimizing the Scheduler

One improvement with respect to context switch times could be to optimize the scheduler in a way so that it is aware of the different memory banks. This could be done by grouping those processes which share one or more memory banks. The scheduler then picks processes from these groups. Only if all the processes of this group used their scheduler quantum another group is chosen. This approach has the benefit of reduced context switch times if multiple processes share the same memory banks. On the other side this behaviour might be not desired because it changes the normal scheduler scheme which can be crucial to certain areas.

7.3 Compiler-Directed Array Interleaving

The approach described so far has been operating system specific, meaning that the applications running on the system do not have to be modified. To further improve energy efficiency it is possible to adjust applications so that they are aware of the different memory banks. This approach has been described before [3] and could be incorporated in this diploma thesis. But in terms of virtual memory this would be inapplicable, because the applications does not have direct control of which physical memory is actually used. So this approach would only make sense in systems where no virtual memory is used at all (mostly embedded systems).

7.4 Combination with Hardware Directed Power Management

Up to now the memory banks where totally controlled by the operating system. Another approach could be to allow the hardware itself to utilize the power management functions. With this it would be easy to activate only those memory banks which are actually accessed by an application. With the previous described approach memory banks get activated as soon as an application allocates pages from it regardless if the application actually accesses this memory. This means that the granularity at which memory banks transition from one power mode to another is in the range of several milliseconds (depending on the scheduler time quantum). Allowing the hardware to control the power transitions has the effect of a refined transition granularity and could lead to increased energy savings. Moreover, with this approach it would be possible to reduce the overhead for the energy aware memory management by dividing the memory in fewer zones than are actually present.

To further increase energy efficiency the page allocation algorithm could be changed in a way that does not use a first touch placement strategy but cluster pages by its access frequency. A previous study [6] showed that the energy savings could be further pushed to approximately 20% beyond the values achieved by a first-touch algorithm. In short, this method works by constructing a histogram of page accesses offline and using the results to determine initial page placements starting with the most frequently accessed page. The drawback of this method is the building of the page access histogram which can take quite a amount of time and CPU power.

8 Summary

Today memory power consumption plays an important role in many applications, especially where those applications are data-centric. At the same time it is important to improve the energy efficiency of the memory system, especially if you have only a small budget for power.

This diploma thesis showed that it is possible to achieve significant energy savings in the memory system with modern hardware. Despite earlier research on memory energy optimization which showed possible energy savings only by simulation this study has been implemented in a real life operating system (Linux).

Energy could be saved because I have observed that typical applications use only a fraction of the available memory. A further observation is that computer systems are gradually fitted with more memory, meaning that the gap between memory usage and available memory grows. With this information in mind it is possible to suspend unused memory in system and thereby saving energy.

Particulary in the field of mobile computing this new technology could lead to an improved energy efficiency.

Bibliography

[1] Rambus, RDRAM, 1999, http://www.rambus.com/

[2] Rambus, Direct Rambus Memory for Mobile PCs, 1999, http://www.rambus.com/downloads/MobileWP%20r2.4.pdf

[3] V. Delaluz, M.Kandemir, Compiler-Directed Array Interleaving for Reducing Energy in Multi-Bank Memory, in Proceedings of the 7th Asia and South Pasific Design Automation Conference and 15th International Conference on VLSI Design (VLSI Design / ASPDAC '02, Jan 2002

[4] Wikipedia, Kernel (computers), 2003, http://www.wikipedia.org/wiki/Monolithic_kernel

[5] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis, The Case for Compressed Caching in Virtual Memory Systems, University of Texas at Austin, 1999

[6] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, Carla Ellis, Power Aware Page Allocation, Technical Report CS-2000-08, Department of Computer Science, Duke University, June 2000

[7] Bjoern Beutel, Saving Energy by Coordinating Hard Disk Accesses, Thesis, Department of Computer Science, University of Erlangen, 2002

[8] Andreas Mull, Optimizing Energy Consumption by Event-Driven Frequency Scaling, Thesis, Department of Computer Science, University of Erlangen, 2002

[9] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Trans. Information Theory, 1977

[10] J. Ziv and A. Lempel, Compression of Individual Sequences Via Variable-Rate Coding, IEEE Transactions on Information Theory, 1978

[11] Markus F.X.J. Oberhumer, LZO Data Compression, 2002, http://www.oberhumer.com/opensource/lzo/

[12] Linus Torvalds, The Linux Kernel Archives, 2003, www.kernel.org

[13] John Spooner, Does Moore's Law no longer apply?, 2000, http://news.zdnet.co.uk/story/0,,t269-s2076881,00.html

[14] Martin Groeger, First products - Moore's Law, 2003, http://www.silicon-valley-story.de/sv/intel_first.html

[15] Siemens, Trends in Memory Technology, ????, http://www.htblmo-klu.ac.at/lernen/hardware/mod_041.pdf

[16] P.J. Denning, The Working Set Model for Program Behaviour, In 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1968

[17] Joe Knapka, Outline of the Linux Memory Management System, ????, http://home.earthlink.net/~jknapka/linux-mm/vmoutline.html

[18] The Linux-MM Team, Welcome to the home page of Linux-MM, 2002, http://linux-mm.org/

[19] Rodrigo S. de Castro, Compressed Caching, 2002, http://linuxcompressed.sourceforge.net/

[20] Intel, Mobile Intel Pentium 4 Processor - M, 2003, http://www.intel.com/products/notebook/processors/pentium4-m/index.htm?iid=ipp_note_proc+highlight_p4p_m&

[21] Intel, Intel Pentium 4 Processor, 2003, http://www.intel.com/home/desktop/pentium4/index.htm

[22] Asus, P4B266-SE, 2003, http://www.asus.com/products/mb/socket478/p4b266-se/overview.htm

[23] Harald Thon, Uli Ries, XScale vs. StrongARM PDA, 2002,

http://www4.tomshardware.com/mobile/20021107/asus_mypal_a600-06.html#cpu_performance_integer_floatingpoint_and_heap_management
[24] Steve Lantz, What is the TLB?, 2003, http://www.tc.cornell.edu/Services/Edu/Topics/Performance/SingleProcPerf/tlb.html
[25] Sylvia Keller, Paging / TLB, 2003, http://erde.fbe.fh-weingarten.de/keller/syso/tlb.pdf
[26] Herman de Hoop, Battery Life Benchmarks, Presentation, 1999, http://www.bapco.com/idf99hdhr2.ppt
[27] Transmeta Corporation, Transmeta Crusoe, 2003, http://www.transmeta.com
[28] Harald Thon, Rainer Pabst, 4x4: A Comparison of Four Mobile Pentium 4-M Processors, 2003, http://www4.tomshardware.com/mobile/20030212/mobile_pentium-01.html
[30] Intel, Intel IQ80310 Evaluation Platform Board Manual, Tech. Documentation, 2001
[29] Intel, Intel 80200 Processor based on Intel XScale Microarchitecture, Tech. Documentation, 2000

Energiebewusste Speicherverwaltung

Eine der grossen Herausforderungen für moderne Betriebssysteme ist die Notwendigkeit den Energieverbrauch zu reduzieren. Besonders im Hinblick auf die vermehrte Nuztung von tragbaren Geräten oder eingebetten Systemen spielt die Energieeffizienz eine grosse Rolle. Bezüglich des Energieverbrauchs ist der Speicher ein nicht mehr zu unterschätzender Faktor, vor allem deshalb, weil Speicherbausteine in den letzten Jahren immer billiger wurden und dies dazu führte, dass Computer mit immer mehr Speicher ausgestattet werden. Bislang existieren zwar einige Ansätze, um den Energievebrauch des Speichers zu senken, doch schöpfen diese Ansätze das volle Potential zur Energiesenkung nicht aus.

Diese Arbeit stellt einen neuartigen Ansatz vor, der es ermöglicht, die Energieverwaltung von neueren Speicherbausteinen auszunutzen, um den gesamten Energieverbrauch eines Computersystems zu senken. Ermöglicht wird dies durch die Aufteilung des Speichers auf mehrere sog. Bänke. Dadurch ist man in der Lage unbenutzten Speicher in einen Energiesparmodus zu versetzen. Nur der Speicher, der gerade benötigt wird, bleibt aktiviert.

Funktioneren kann das ganze natürlich nur, wenn der Speicher nicht vollständig ausgenutzt wird. Dazu muss man sich vor Augen halten, dass die meisten Anwendungen nur einen Bruchteil des zur Verfügung stehenden Speichers nutzen. Zum Beispiel habe ich durch Analyse der Working Sets herausgefunden, dass die Speicherausnutzung meiner Testapplikationen im Bereich von 0.34% - 12.62% liegt (bei einer Hauptspeichergrösse von 1024MB). Dadurch lässt sich folgern dass sich mit dieser neuen Methode ein enormes Potential zur Verbesserung der Energieeffizienz ergibt.

Die zentrale Komponente die darüber entscheidet, welcher Teil vom Speicher gerade benutzt wird, ist die Speicherverwaltung. Mit ihrer Hilfe lässt sich eine Energieverwaltung implementieren, die versucht immer einen möglichst grossen Teil des Speichers im Energiesparmodus zu halten.

Ziel dieser Arbeit war es, den Speicherallokations-Algorithmus zu modifizieren um die Speicherseiten, die eine Anwendung benötigt, immer in einer möglichst geringen Anzahl von Speicherbänken anzuhäufen. Erreicht wird dies dadurch, dass pro Prozess eine Verwaltungstabelle geführt wird, die die Speicherbanknutzung angibt, d.h. die Seitenreferenzen pro Speicherbank zählt. Die Speicherverwaltung versucht dann bei einer Seitenallokation möglichst die Speicherbänke zu verwenden, die bereits häufig referenziert werden. Ist dies nicht möglich, dann tritt ein Mechanismus in Kraft, der versucht andere Seiten dieser referenzierten Speicherbänke auszulagern. Dazu bieten sich zwei Möglichkeiten an:

1. Eine mögliche auszulagernde Seite wird in eine andere Speicherbank kopiert.
2. Eine mögliche auszulagernde Seite wird komprimiert und in einen extra dafür vorgesehenen Zwischenspeicher abgelegt.

Methode 1 wird verwendet, wenn der Prozess, zu dem die auszulagernde Seite gehört, andere Speicherbänke referenziert und in diesen Speicherbänken noch ein Platz frei ist. Ansonsten wird versucht die Seite zu komprimieren. Nur wenn beide Methoden fehlschlagen wird eine neue Speicherbank aktiviert. Diese beiden Mechanismen führen dazu, dass der Speicher für Anwendungen immer auf möglichst wenige Speicherbänke verteilt wird.

Ein entsprechender Ansatz wurde für das Betriebssystem Linux implementiert. Dabei hat sich gezeigt, dass sich durch die beschriebenen Methoden der Energieverbrauch um bis zu 30% senken lässt. Im Gegensatz dazu muss man allerdings mit längeren Laufzeiten der Programme rechnen, die im schlimmsten Fall um bis zu 35% länger sind. Die Energieeffizienz liesse sich noch weiter erhöhen, wenn man den Software-Seitigen Ansatz mit einem Hardware-Seitigem Ansatz ergänzen würde.

Zusammenfassend lässt sich sagen, dass diese Diplomarbeit eine neue Möglichkeit aufgezeigt hat wie man in modernen Computersystemen Energie sparen kann. Besonders im Bereich Mobiles Rechnen würde sich der Einsatz dieser neuen Techik lohnen.