Towards guidelines for development of energy conscious software

by

Edward Carlstedt-Duke
Erik Elfström

LIU-IDA/LITH-EX-A--09/012--SE

2009-03-02
Master’s Thesis
Towards guidelines for development of energy conscious software

Edward Carlstedt-Duke
Erik Elfström
Reg Nr: LIU-IDA/LITH-EX-A--09/012--SE
Linköping 2009

Supervisor: Björn Rudin
Combitech AB

Examiner: Christoph Kessler
IDA, Linköpings universitet

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-17444
Linköping University Electronic Press
Abstract

In recent years, the drive for ever increasing energy efficiency has intensified. The main driving forces behind this development are the increased innovation and adoption of mobile battery powered devices, increasing energy costs, environmental concerns, and strive for denser systems.

This work is meant to serve as a foundation for exploration of energy conscious software. We present an overview of previous work and a background to energy concerns from a software perspective. In addition, we describe and test a few methods for decreasing energy consumption with emphasis on using software parallelism. The experiments are conducted using both a simulation environment and real hardware. Finally, a method for measuring energy consumption on a hardware platform is described.

We conclude that energy conscious software is very dependent on what hardware energy saving features, such as frequency scaling and power management, are available. If the software has a lot of unnecessary, or overcomplicated, work, the energy consumption can be lowered to some extent by optimizing the software and reducing the overhead. If the hardware provides software-controllable energy features, the energy consumption can be lowered dramatically.

For suitable workloads, using parallelism and multi-core technologies seem very promising for producing low power software. Realizing this potential requires a very flexible hardware platform. Most important is to have fine grained control over power management, and voltage and frequency scaling, preferably on a per core basis.
Acknowledgments

We would like to thank the following people for their support during the creation of this work: our supervisor Björn Rudin for guidance and support throughout the project, our examiner Christoph Kessler for his quick and informative responses to all our questions, Luca Benini and Petru Eles for letting us use the MPARM simulation environment, Alexandru Andrei for helping us with MPARM, and our opponents, Patrik Eliardsson and Ulrika Uppman, for helping us improve the quality of this work.
Contents

1 Introduction ........................................ 1
  1.1 Background ....................................... 1
  1.2 Problem ........................................ 1
  1.3 Purpose ......................................... 2
  1.4 Approach ........................................ 2
  1.5 Thesis outline .................................... 2

2 Overview ........................................ 3
  2.1 Energy ........................................... 3
    2.1.1 The cause of energy consumption in a system .... 3
    2.1.2 Energy, power and battery lifetime .............. 4
    2.1.3 Energy efficiency metrics ....................... 5
  2.2 Battery technology ................................ 6
  2.3 Energy estimation .................................. 7
  2.4 System energy consumption distribution ............. 8
  2.5 Measuring energy consumption ....................... 10
  2.6 Algorithms and energy complexity .................. 11
  2.7 Compilers ........................................ 12
  2.8 Operating systems ................................ 13
    2.8.1 Cost ......................................... 13
    2.8.2 Cost mitigation strategies ..................... 14
    2.8.3 Power management features ..................... 15
  2.9 Frequency and voltage scaling ...................... 15
  2.10 Parallelism and multi-core technology ............. 17
  2.11 Quality of service ................................ 18
  2.12 Power management ................................ 18

3 Simulations ...................................... 21
  3.1 MPARM .......................................... 21
  3.2 Image filter .................................... 23
    3.2.1 Algorithm .................................... 23
    3.2.2 Compiler effects ............................. 26
    3.2.3 Quality of service ......................... 28
    3.2.4 Parallel performance ......................... 31
3.2.5 Energy ........................................... 33
3.3 Quicksort ............................................ 35
  3.3.1 Parallel Quicksort .................................. 35
  3.3.2 Our parallel algorithm .................................. 36
  3.3.3 Memory management .................................. 39
  3.3.4 Frequency scaling .................................. 40
  3.3.5 Optimizing the algorithm ............................... 43
  3.3.6 Compiler optimizations ................................ 44
  3.3.7 Combining results .................................... 47

4 Hardware .............................................. 51
  4.1 Hardware platform ..................................... 51
    4.1.1 Testing equipment ..................................... 51
    4.1.2 Measurement setup and method .......................... 52
    4.1.3 Mitigating the noise issue .............................. 53
  4.2 Hardware measurements .................................. 56
    4.2.1 Compiler settings ..................................... 56
    4.2.2 Input data ........................................... 56
    4.2.3 Frequency scaling ..................................... 59
    4.2.4 Disabling interrupts .................................... 60
    4.2.5 Comparing MPARM to hardware ........................... 62

5 Discussion ............................................. 65
  5.1 Purpose and goals ..................................... 65
  5.2 Limitations and problems ................................ 65
  5.3 Our take on low power software issues ................. 67
    5.3.1 General optimizations ................................. 67
    5.3.2 Frequency scaling ..................................... 68
    5.3.3 Compilers ............................................. 68
    5.3.4 Operating systems ..................................... 69
    5.3.5 Parallelism ............................................ 69
    5.3.6 Hardware design ....................................... 70
    5.3.7 Activity patterns ..................................... 70
    5.3.8 Energy analysis ........................................ 71
  5.4 Conclusions .......................................... 72
  5.5 Future work .......................................... 73

Glossary .................................................. 75

Acronyms .................................................. 77

Bibliography .............................................. 79

A Image filter detailed results ............................ 87

B Quicksort detailed results ............................... 91
C Quicksort queue length
   C.1 Results ........................................ 97
   C.2 Conclusions .................................... 99

D Quicksort reference comparisons 103

E AD620 fullsize plots 107

F Hardware measurements detailed results 115
List of Figures

2.1 Itsy subsystem power dissipation .................................. 9
2.2 iPAQ subsystem power dissipation ................................. 9
2.3 Sensor node subsystem power dissipation ......................... 10
2.4 Different JPEG quality settings ................................. 19
2.5 Sleep example .................................................. 20

3.1 MPARM architecture ............................................. 22
3.2 Unsharp mask example .......................................... 25
3.3 Image partitioning example .................................... 25
3.4 Parallel image filter execution and memory management .......... 27
3.5 Image filter compiler effects .................................. 29
3.6 Gains by lowering quality of service ............................ 30
3.7 Quality of service output example ............................. 30
3.8 Image filter speedup ............................................ 31
3.9 Image filter cycle costs ....................................... 32
3.10 Image filter speedup using different images sizes ............... 32
3.11 Image filter energy consumption .................................. 33
3.12 Energy consumption by subsystem ................................ 34
3.13 The parallel Quicksort ......................................... 37
3.14 Quicksort with multiple cores .................................. 38
3.15 Quicksort with private memory .................................. 39
3.16 Quicksort with scaling of the idle frequency .................... 41
3.17 Quicksort with parallel scaling .................................. 41
3.18 Quicksort with sequential scaling ................................ 42
3.19 Quicksort frequency scaling .................................... 42
3.20 Quicksort with too much scaling ................................ 43
3.21 Quicksort with algorithm optimizations .......................... 46
3.22 Quicksort with different compiler settings ....................... 46
3.23 Quicksort compiler settings with one core in MPARM .......... 47
3.24 Quicksort using all methods .................................... 48
3.25 Quicksort energy distribution ................................... 48
3.26 Quicksort using all methods and compensating for RAMs ....... 49

4.1 Hardware measurement schematic ................................ 52
4.2 Hardware setup .................................................. 53
4.3 AD620 compared to diff-functionality ............................ 54
4.4 AD620 compared to filtered diff-functionality ................... 55
4.5 AD620 compared to diff-functionality for different ranges ...... 55
4.6 Quicksort results using different compiler settings on hardware 56
4.7 Image filter results using different compiler settings on hardware 57
4.8 Total energy consumption for different input data ............... 58
4.9 Sharpening energy consumption for different input data .......... 59
4.10 Energy when scaling hardware frequency ........................ 60
4.11 Power when scaling hardware frequency ........................ 61
4.12 Time when scaling hardware frequency ............................ 61
4.13 Effects of leaving interrupts enabled ............................. 62
4.14 Comparison of Quicksort compiler effects .......................... 63
4.15 Comparison of image filter compiler effects ......................... 63
4.16 Comparison of image filter tasks ................................ 64

C.1 Active-idle relation for a queue length of one ...................... 98
C.2 Energy consumption for different queue lengths per part ............ 98
C.3 Execution time and energy consumption for different queue lengths 99
C.4 Comparison between queue, buffer and reference .................... 100
C.5 Speedup for using a queue ....................................... 100

D.1 Quicksort reference comparison with private memory ............... 104
D.2 Quicksort reference comparison with scaling of the idle frequency 104
D.3 Quicksort reference comparison with parallel scaling ............ 105
D.4 Quicksort reference comparison with sequential scaling .......... 105
D.5 Quicksort reference comparison frequency scaling ................... 106
D.6 Quicksort reference comparison with algorithm optimizations .... 106

E.1 AD620 compared to diff-functionality over 1 ms .................. 108
E.2 AD620 compared to filtered diff-functionality over 1 ms ........ 109
E.3 AD620 compared to diff-functionality over 0.2 ms ............... 110
E.4 AD620 compared to filtered diff-functionality over 0.2 ms ...... 111
E.5 AD620 compared to diff-functionality over 0.4 s ................ 112
E.6 AD620 compared to filtered diff-functionality over 0.4 s ....... 113

List of Tables

3.1 Gaussian window vector ........................................ 24

A.1 Image filter compiler effects ................................... 88
A.2 Gains by lowering quality of service ............................ 88
A.3 Image filter speedup using different images sizes ............... 88
A.4 Image filter cycle costs ........................................ 89
A.5 Image filter energy consumption ................................ 89
A.6 Image filter execution time ...................................... 89
A.7 Image filter power dissipation .................................. 89
A.8 Energy consumption by subsystem ................................ 90

B.1 Quicksort with different number of tasks ....................... 92
B.2 Quicksort when using private memory ......................... 93
B.3 Quicksort when scaling the frequency during idle ............ 93
B.4 Quicksort when scaling the frequency during the parallel part 93
B.5 Quicksort when scaling the frequency during the sequential part 94
B.6 Quicksort when scaling the frequency during all parts .......... 94
B.7 Quicksort when scaling the frequency on eight cores .......... 94
B.8 Quicksort when using other algorithms for small datasets 95
B.9 Quicksort when using different compiler optimizations in MPARM 95
B.10 Quicksort with different compiler settings in MPARM single-core 95
B.11 Quicksort when using the optimal settings 96
B.12 Quicksort energy distribution 96
B.13 Quicksort when using the optimal settings and less RAM 96

F.1 Quicksort results using different compiler settings on hardware 115
F.2 Image filter results using different compiler settings on hardware 116
F.3 Energy consumption for different input data 117
F.4 Effects of leaving interrupts enabled 118
F.5 Detailed effects of leaving interrupts enabled 118
F.6 Comparison of Quicksort compiler effects 118
F.7 Comparison of image filter compiler effects 119
F.8 Comparison of image filter tasks 119

List of Algorithms

3.1 The unsharp mask algorithm 25
3.2 The parallel unsharp mask algorithm 26
3.3 The parallel unsharp mask algorithm using private memory 28
3.4 The Quicksort algorithm 36
3.5 The Quicksort multi-core message buffer store 37
3.6 The Quicksort multi-core message buffer fetch 38
3.7 The Bubblesort algorithm 44
3.8 The Shakersort algorithm 45
3.9 The Selectionsort algorithm 45
Chapter 1

Introduction

In this chapter we will introduce this thesis. We will state the purpose of this thesis, the problem it intends to solve, the method used to create it, and finally the organization of the thesis.

1.1 Background

In recent years, the drive for ever increasing energy efficiency has intensified. The main driving forces behind this development are the increased innovation and adoption of mobile battery powered devices, increasing energy costs, environmental concerns, and strive for denser systems.

The main focus of these efforts has been on increasing the efficiency of the hardware but it has become clear that the largest gains can be made when energy efficiency is prioritized during the entire system design. Combitech therefore wishes to acquire a deeper knowledge of the issues involved in designing and implementing energy efficient software.

1.2 Problem

Embedded systems are becoming a bigger part of our lives. Many of these systems are required to become more mobile by reducing size and weight while at the same time facing increasing battery lifetime requirements. Optimizing energy consumption is therefore a very hot topic today.

There are many known strategies for how to develop hardware to consume as little energy as possible while still meeting all the performance and functionality requirements. There are, however, not that many strategies for the development of power conscious software in the public domain. Most of the material available for developing low-power software is focused on compilers and operating systems. As compiler and operating system development falls outside of immediate interests of Combitech, they would like a low power software knowledge base that is more
suited to their needs. Combitech wishes to find low power methods for application level software development and assess their impact on energy consumption.

This effort should eventually result in a collection of strategies, patterns, and guidelines for low power software development.

1.3 Purpose

The purpose of this thesis is to serve as a first step towards addressing the problem identified by Combitech. The scope of this initial step is limited to acquiring an overview of previous work and to study the energy saving potential of parallelism and multi-core systems. Therefore this thesis will present a compilation of known methods for minimizing energy consumption of an embedded system through software techniques.

In addition, a low cost method for measuring energy consumption in a system will be presented. This will enable future work to concentrate on low power techniques instead of measurement issues.

1.4 Approach

Since this thesis has two authors, the workload had to be divided. Each author had its own test program. Erik Elfström handled implementation and simulations of the image filter, found in Section 3.2. Edward Carlstedt-Duke was responsible for implementation and simulations of Quicksort, found in Section 3.3. Beside these two parts, almost everything was written, reviewed and improved by both authors.

1.5 Thesis outline

The second chapter in this thesis presents an overview of the fundamentals of low power software and prior work. The third chapter presents the simulation experiments, primarily concerned with parallelism. Chapter four contains our experimental work on actual hardware and some comparisons with the simulation environment. In chapter five we discuss the results, state our conclusions and present possible future work.
Chapter 2

Overview

A substantial amount of the project time was allocated to survey the current state of the software energy optimization domain. In this chapter we present an overview of some of the more important subjects and their potential benefits. We will not delve too deeply into each subject, instead we aim foremost to make the reader aware of these topics and some prior work in the field.

2.1 Energy

In this section we will present some of the fundamentals regarding energy concerns in low power systems.

2.1.1 The cause of energy consumption in a system

In any given system there is a vast variety of powered devices that consume energy. In this thesis we are mostly concerned with components that are heavily influenced by software, i.e. those that are central to program execution. Typically this means CMOS devices such as processor cores, various memory circuits, I/O interfaces and communication buses. The power consumption of this type of devices can be divided into two parts, dynamic and static, as seen in Equation 2.1 [1].

\[ P = P_{stat} + P_{dyn} \]  

(2.1)

\[ P_{stat} = I_{stat} \cdot V_{dd} \]  

(2.2)

The static dissipation, defined in Equation 2.2 [1], is associated with keeping a device in a powered state and is fixed regardless of the activity of the device. For a CMOS device, the largest contribution to \( I_{stat} \) is the leakage current of the transistors. Leakage increases with temperature and the trend is that leakage increases with every process generation due to lower threshold voltage and other effects.
The dynamic dissipation, modelled according to Equation 2.3 [1], is a direct effect of voltage transitions caused by the activity of the device. It is strongly linked to the activity factor \( \alpha \), denoting how often a voltage transition occurs, and the capacitive load, \( C_{load} \), that has to be driven during the transition. In the absence of any power management features in the hardware, the activity factor is the only way that software can affect the energy consumption of a system. By using a device less frequently and thus reducing its activity factor we can reduce the energy consumption. This is possible to achieve in two ways, by optimizing the software with a more efficient implementation or algorithm and reduce the overall activity factor in the system or by moving activity from an expensive unit to a less expensive one. This could for example be to move activity from an expensive off-chip memory interface to an on-chip memory that cost significantly less energy per activation.

The cost of activity in different devices can usually be inferred from data sheets but as a general rule, things become more expensive the further away the activity is from the processor core. Performing arithmetic operation on register values is cheaper than communicating with another on-chip device. This is in turn cheaper than going off-chip via power hungry interconnects. The energy cost of sending a word of data to remote systems by a wireless interface may be orders of magnitude more expensive than performing one arithmetic operation [2].

Other than the activity factor, there is very little that software can do to reduce energy consumption without at least some assistance from the hardware platform. By simply introducing hardware controlled reactive schemes to the system the software has much greater opportunities for optimization. It can now try to shape its activity-patterns so that the benefit of the hardware features is maximized. Energy consumption is thus a system-wide concern and therefore it is very important to consider early in the design and collaborate across the software and hardware boundary.

### 2.1.2 Energy, power and battery lifetime

Depending on the operating conditions of the system, the overall optimization goal might differ substantially. In the low power domain there are at least three different criteria to optimize for: energy, power and battery lifetime.

For energy-constrained systems only the total energy consumed by a task is of interest. In what manner it is consumed is of little importance, be it bursty, smooth or otherwise. This makes it a bit easier to optimize for since it affords some level of flexibility in choosing when and how to consume energy. It might also be simpler in the way that it might be sufficient to perform a static analysis of energy costs in the system and optimize towards this without any runtime feedback.

Power constrained systems, typically compact and possibly tethered to an abundant energy source, are concerned with limiting the power dissipation. This implies that total energy consumed is not as important as spreading system hot spots over time and space to ensure that the system remains within its thermal
operating parameters. Although power and thermal models can be derived from energy models it is still likely that some form of thermal feedback is required to account for changes in the temperature of the system environment.

Battery powered systems have a complex optimization task since battery capacity depends heavily on the behaviour of the load. Therefore the system needs to minimize total energy consumed and assure that it is consumed in a manner that provides a good operating point for the battery technology. Battery technology issues are discussed further in Section 2.2.

All these concerns have overlapping interests but they are not the same. To simplify our experimental work, we have only considered pure energy optimizations without regard for power limitations or battery technology.

### 2.1.3 Energy efficiency metrics

The most elementary tool for evaluating and comparing the utilization efficiency of any resource is a qualitative metric. Although a metric can serve several purposes we mainly need a tool to use as optimization criteria for a specific design and to compare the merits of different designs. In the low power domain there are currently several such metrics in use. The ones presented here originate from the CMOS circuit development domain where average power ($P$) and operation delay ($D$) are two of the most important constraints. The metrics take the form of a product of $P$ and $D$ [3], the general form is shown in Equation 2.4.

$$P^m D^n, \quad m, n \in \mathbb{Z}^*$$

(2.4)

The nonnegative integers $m$ and $n$ in the metric are used to weight power and delay. Since average power and delay are valid criteria for determining resource consumption and performance in a complete embedded system, this type of metric can be applied to define the efficiency of a complete system under software influence. These metrics are also used in their inverse form, $1/(P^m D^n)$. A typical example is $\text{Performance/Watt}$ which is equivalent to $1/(PD)$.

**Power-delay product ($P^1D^1$):** This is a commonly used metric. Since a unit of power multiplied by a unit of time produce a unit of energy, this metric captures the total energy used for an operation. It has some problems however, as noted by Sengupta and Saleh [3], the optimum of the function tends to the point of zero performance for CMOS circuits dominated by dynamic power consumption and with scalable voltage.

**Energy-delay product ($P^1D^2$):** This metric is often used when the power-delay product cannot capture the performance requirement. This can eliminate the problem of a zero performance optimum when a reduction in delay gives a quadratic increase in average power, as explained by Horowitz et al. [4].

**Energy-delay$^2$ product ($P^1D^3$):** A metric advocated by Martin et al. [5] as a means to fairly compare circuits under the influence of voltage scaling. How-
ever, Hsu et al. [6] argues that, in the case of high performance computers, this metric puts too much emphasis on speed.

We could not find a consensus on the usefulness of these metrics for software optimization and comparison. It seems that this area is currently not well understood. The software influence on energy consumption might be too complex to capture with this type of metric. The metrics all put different emphasis on power and delay based on assumptions of the characteristics of the hardware. This could possibly be problematic when trying to evaluate software behaviour on systems containing lots of devices with different characteristics. For example, some device might be under the influence of frequency scaling while others are not. This might compromise the applicability of each metric.

2.2 Battery technology

A battery is a non ideal energy source in the sense that its energy capacity is dependent on several external factors. Martin [7] and Rao et al. [8] present four main factors:

**Rate dependency:** This effect is manifested as lowered battery capacity for high discharge rates. The effect is noticeable when the discharge rate causes the active material depletion at an electrode to exceed the diffusion rate, thereby causing a gradient of active material in the electrolyte. This causes the concentration of available carriers at the electrode surface to drop. When the concentration becomes to low to maintain the electrochemical reaction at the electrode, the battery will be discharged even though there is a substantial amount of unused active material in the electrolyte further away from the electrode. This gives the effect of lowering the capacity of the battery.

**Recovery effects:** Recovery effects are strongly related to the same mechanisms that cause the rate dependency, since the charge lost due to the gradient effects will eventually become available again as diffusion causes the active material to even out. This means that charge lost due to a heavy load may be recovered to some extent during periods of light or no load.

**Temperature:** The chemical reactions in battery cells are affected by temperature. Lower temperature means lower activity, increased internal resistance and reduced full-charge capacity. At the other end, higher temperature decreases the internal resistance and increase the capacity of the battery cell, however the self discharge rate is also increased.

**Capacity fading:** Many battery technologies suffer from this effect. With each charge and discharge cycle some of the capacity is lost due to unwanted side reactions.

Since temperature is only controllable to a very limited extent and capacity fading is a very long term process, the most interesting aspects of battery technologies, from an energy aware point of view, is the rate dependency and recovery
2.3 Energy estimation

Several studies have been done on these effects, in some cases with contradictory results. Castillo et al. [9] have examined the effect of intermittent discharge on different battery technologies. Their tests were done with 50% on-off duty cycle at $1.1 \cdot 10^{-3}$ Hz and $5.5 \cdot 10^{-4}$ Hz using alkaline, nickel cadmium, nickel metal-hydride, and lithium-ion battery cells. They concluded that only alkaline battery-life improved with intermittent discharge, showing a 27% improvement. The other technologies had the same, or slightly worse, battery life-time with intermittent discharge. In contrast with this, Rao et al. [10] demonstrates a 28% improvement in battery life for a nickel metal-hydride battery using 50% duty cycle at 0.2 Hz.

Experimental data from Rao et al. [10] supported by analytical findings by R. Rao and Vrudhula [11] indicates that real world batteries are insensitive to current changes with frequencies higher than 1 Hz. This implies that fine-grained task scheduling, typically occurring at above 1000 Hz, need not be battery aware, it is sufficient to consider pure energy optimizations.

R. Rao and Vrudhula [11] also demonstrate that if unpredictable rest periods are common, battery optimizing techniques can perform worse than energy optimizing techniques due to the recovery effect.

2.3 Energy estimation

In order for developers to use iterative design methodologies and get early indications on software energy consumption it is beneficial to have tools that can estimate the power consumed when running a piece of code on a given system. There are several different types of such estimators:

**Low-level hardware simulation:** This type of estimation is based on detailed, and time consuming, simulation of the electrical properties of the target system. This, of course, requires very detailed knowledge of the system and the tools are generally intended for hardware design. Huang et al. published an example of such a technique [12].

**Instruction level simulation:** A technique that specifically targets the energy consumption of a processor, and to some extent memory, by attributing costs to each instruction executed by a code sequence. Costs are typically determined by measuring energy consumption on actual hardware for a long sequence of each instruction and averaging the result. Usually additional costs are also attributed to the ordering of instructions. If an addition is performed after a multiplication it might consume more energy than if performed after a subtraction. One limitation of this technique is that, since it is based on executed instructions, only the energy cost in the processor is easily captured. This seems to be a well researched area with a multitude of published work. Early examples include Tiwari et al. [13], Mehta et al. [14], Russell and Jacome [15]. More recent work includes Varma et al. [16] and Joe et al. [17].
Overview

**Architecture simulation:** Simulators based on this method typically use parameterized energy models based on electrical characteristics of hardware technology. This is then used to assign costs to the activities within an architecture model of the system. This makes it possible to include the energy costs of all major subsystems, not just the processor. This results in a more complete estimation. Vijaykrishnan et al. [18] developed an estimator that uses input transition sensitive energy models and a register transfer level description of the architecture. Brooks et al. [19] describes a similar estimator. Both estimators model only the processor, not a complete system. Loghi et al. [20] have used the MPARM simulator to model a more complete system-on-chip. This is the same simulator used in this thesis.

**High level function estimation:** This technique intends to model the energy consumption of higher level software constructs at the source code or function level. This typically leads to much faster simulation speeds at the cost of accuracy. There are several published attempts in this direction. Tan et al. [21], describing two techniques, one based on complexity analysis and one based on profiling. Qu et al. [22] used a method that is based on pre-characterized library calls combined with instruction level estimation to produce a cost estimation for language level functions. A method to estimate energy consumption of C language constructs is presented by Laurent et al. [23], although they still rely on simulation and profiling to predict some parameters in their models. Muttreja et al. [24], demonstrated an automated way to build energy macro models of software constructs.

All these methods share some attributes and typically the higher level estimations are based on techniques and observations of the lower level methods.

Currently there seem to be very few, if any, commercial tools targeted at software energy estimation [25]. Some of the academic research platforms mentioned here are available but there are potentially licensing, documentation, and support issues associated with their use.

### 2.4 System energy consumption distribution

The embedded market contains a wide range of hardware platforms and systems. Therefore there is no subsystem energy consumption distribution that is typical for all systems. Consequently it is important to know the characteristics of the system one is developing for. To illustrate the differences between systems we will show three examples from published papers, two handheld computers and a sensor network node.

As can be seen in Figures 2.1, 2.2, and 2.3 the sum of power dissipated in memory and processor subsystems can vary from less than 15% as in Figure 2.2, to totally dominating at almost 100%, as is the case in Figure 2.3. This illustrates clearly that, in some systems, the benefit of optimizing the energy consumption directly caused by software, in processor and memory subsystems, would not be
2.4 System energy consumption distribution

Figure 2.1 Power dissipation distribution in subsystems of the Itsy handheld computer [26].

Figure 2.2 Power dissipation distribution in subsystems of the iPAQ handheld computer [27].
Another thing to note from the example in Figure 2.1, where the power dissipation in each subsystem varies greatly for different tasks, is that the usage scenario plays an important role. Even if the role of energy efficient software is marginalized in one usage scenario it can still bring a lot of value to other scenarios.

2.5 Measuring energy consumption

To measure the energy consumption of a system, some kind of measuring points are needed. One practical solution is to have a connector on the trace between the core and its power supply. When this connector is closed, everything works as it normally would, but if we open the connector and plug in an ampere-meter, the current drawn by the processor can be monitored.

An alternate way if an ampere-meter is not available is to use a volt-meter in parallel with a small resistor. The current through the resistor will create a drop in voltage which will register on the volt-meter. The current can be calculated with the, well known, formula found in Equation 2.5 known as Ohm’s law [29].

\[ U = R \cdot I \Rightarrow I = \frac{U}{R} \]  

(2.5)

One important thing with the volt-meter solution is to use a small enough resistor so that the voltage over the processor does not fall too low, but still have the resistor large enough for the volt-meter to produce a useful result. If the volt-meter has a noise-level of 20 mV and the processor draws around 10 mA using a 2 Ω resistor is not such a good idea since the variations will be lost in the noise. On the other hand, if the supply voltage is 1.8 V and current 200 mA then using a 2 Ω
2.6 Algorithms and energy complexity

A more advanced alternative to a volt-meter is to use an oscilloscope. This enables the presentation of the current graphically which simplifies tracking how the energy consumption changes over time. By using an oscilloscope it is much easier to analyze specific parts of the software. The tricky part with oscilloscopes is measuring voltages where the reference is not ground. An ordinary oscilloscope with single-ended probes connected to a grounded outlet has the same ground in the probe as the one in the outlet. This means that if the ground-connector is connected to the reference on the circuit being measured then that point will be grounded and thereby pulled to zero. This is not a good thing when trying to measure a power supply since the reference-point on the corresponding resistor will become ground. Thereby, no current will reach the chip which was originally supplied by that power supply.

One simple solution to this problem is investing in differential probes. They are able to measure the difference by electrically isolating the measure-points from the oscilloscope. An alternative is to invest in an isolating transformer. If the oscilloscope has two or more inputs then it is possible to use the built in math functionality to find the difference. A big problem with this solution is that the precision decreases. The reason is that the offset is very large compared to the difference meaning that most of the bits are used to describe the offset and only a few are left for the variation. If, for example, the offset is 2 V then with 9 bits spread out from 0 V to 2 V would result in the smallest bit having a value of 4 mV. With a voltage drop of 40 mV, this resolution means that the smallest difference measurable is 10%.

Another alternative is to build a circuit that extracts the difference and outputs it relative to ground. This can be done with a couple of amplifiers and some resistors, as described by Molin [30].

Depending on the goal with the measurement, different equipment requirements apply. For instance, when trying to find the average energy consumption, a relatively simple multi-meter will suffice. When profiling software, some kind of sampling equipment with a high sampling rate and low noise-level is needed to gather the data. It has to be able to store many values or, preferably, sample at high speed directly to a computer for easier analysis.

2.6 Algorithms and energy complexity

Early estimations and comparisons of algorithmic complexity are important in order to determine what algorithm is most suited for a given application. Extensive work has been done in this regard from a performance point of view by means of computational complexity analysis. Recently efforts have been made to find ways to apply similar analysis to the energy requirements of algorithms. Jain et al. propose what they call an Augmented Turing Machine model [2]. Their approach is to augment a Turing Machine model with energy cost for state transitions. Zotos
et al. [31] present a fairly simple model that attributes cost to processor operations, instruction memory operations, and data memory operations. This work seems to be based on earlier findings [32].

Although no clear picture is presented on the subject of energy complexity analysis, based on the direction of the efforts present we can still get some indication on how to judge algorithms in terms of energy. The common concept presented is to apply cost functions to other complexity metrics such as the number of communication events, I/O and memory events, computations, and switching activity in data paths due to input vectors. Traditional algorithm analysis is sufficient to determine the number of computations, communication events, memory accesses, and I/O operations. Data path switching is harder to judge, especially if input vectors are not known in advance. However, Jain et al. [33] suggests that input induced switching activity in data paths has only a minor impact on energy consumption (see Section 4.2.2 for our findings on this issue). Finding an approximate unit cost of each type of operation should be possible but is of course platform specific. First order approximations should be possible for an initial rough energy complexity comparison of algorithms. Also, it is clear that the energy complexity and computational complexity is very closely tied, so choosing a fast algorithm is a good starting point.

2.7 Compilers

Traditionally compilers have been very effective in reducing the need for programmers to do low level optimizations. However, when it comes to the issue of energy optimizations most compilers provide no specific optimization strategy to minimize the energy consumption of the software. The commonly provided optimization strategies are for code size or performance, usually with different levels of aggressiveness.

The GNU Compiler Collection (GCC): The compiler provides -O1, -O2, and -O3 as optimization strategies for speed and -Os for code size [34].

The ARM compiler (armcc): This compiler provides -Ospace and -Otime to choose main optimization criteria and -O1, -O2 and -O3 for selecting aggressiveness [35].

The Microsoft ARM compiler: The compiler provides two basic optimization strategies, /O1 for minimizing size and /O2 to maximize speed. Additionally there is also the /Ox full optimization criteria with the /Os and /Ot directives for favouring small or fast code [36].

Evaluating the energy efficiency of available compilers for embedded systems is too broad a subject for inclusion in this thesis. Instead we focus on the role of the compiler in energy constrained software development and what effects it has on energy consumption.

It is clear that execution time optimization plays an important role in compiler energy optimizations [37]. Valluri and John [38] performed a study indicating that,
in the absence of true energy optimizing directives, there exists a strong correlation between execution time optimal compiler directives and energy optimal compiler directives even though they use an out-of-order, 4-issue, super-scalar architecture. Intuitively, the extra execution resources presented to the compiler should create opportunities for reductions in execution time by means of parallel execution at the cost of additional energy. In their results some of the aggressive instruction scheduling optimizations did show an increase in total energy and a reduction in execution time, but the effects were small, around 3% and 1% respectively. This suggests that the optimization strategies do not introduce a significant amount of extra work even when presented with a super-scalar architecture.

These results clearly show that performance is important for energy efficiency and also that energy optimizations and performance optimizations share a common subset of techniques. Still, energy optimizations and performance optimizations are not the same. There are many true energy optimizing compiler techniques but few seem to have made their way out of the academic research compilers. Examples of techniques are memory layout and access pattern optimizations \cite{39, 40}, register pipelining \cite{41}, low power instruction scheduling \cite{42, 43}, and link time code compaction and optimizations \cite{44}. Since low power optimization, in all its forms, is a hot topic in research and industry today, we expect that energy optimization strategies will make it into many compilers for the embedded market eventually.

### 2.8 Operating systems

As the complexity of embedded systems grow, the benefits of using operating systems increases \cite{45}. Operating systems offer clear advantages that help reduce the development time and mitigate several risks in system development. In this section we present some interesting work regarding the effects of operating systems on energy constrained embedded system.

#### 2.8.1 Cost

The benefits afforded by operating systems do come at a cost, as is the case with almost every abstraction technique. Studies have been made regarding the overhead of common embedded operating systems, both in terms of performance and energy consumption. Acquaviva et al. \cite{46} investigated several different factors in their work. They report the cost of several kernel functions and the effect on energy consumption by changing thread switching frequency. Baynes et al. \cite{47} have shown that, when executing lots of computationally light tasks concurrently, the execution of OS routines can account for as much as 95% of the energy consumed by the system and that 30–50% is to be expected in most cases. There are also other overheads such as increased size of executables.

These studies highlight the importance of understanding the trade-off between ease of development and overhead when choosing how and when to deploy an operating system on energy constrained embedded platforms. This is especially important to consider early in the design since whether to develop for a platform
with or without an operating system will have a very significant impact on the development process.

2.8.2 Cost mitigation strategies

The most effective mitigation strategy is to know the cost and overhead of operating system services and use them sparingly. One such area of services is the management of processes and interprocess communication (IPC) [48]. It is thus an interesting target for optimization. Fei et al. [49] presents several interesting techniques for reducing overhead in systems with several concurrent processes. Although they designed and implemented an automated tool to apply the source code transformations, there is nothing that prohibits manual optimizations if time and resources are available.

**Process merging:** By merging processes, this technique tries to minimize the number of concurrent processes in the system. This should reduce the need for the operating system to perform scheduling and other tasks that might incur overhead. Two processes that have a producer and consumer relationship are likely candidates for this optimization.

**Message vectorization and buffering:** Vectorization and buffering of interprocess messages can reduce the overhead associated with interprocess communication. One example where this might be beneficial is if a process A supplies samples, one at a time, to process B where the samples are used a thousand at a time. If process A were to buffer all 1000 samples before sending them, there would be only one communication event instead of 1000, possibly reducing overhead significantly.

**Computation migration:** Since sending data between processors can be expensive it is possible to achieve greater efficiency by carefully distributing computational responsibilities among the processes, such as to minimize the frequency and volume of interprocess communication. If we look at the example used in message vectorization and add that the computation performed by B results in one scalar value, we can see that moving the computation from B to A might increase the efficiency. If the computation is performed in A, only a single value needs to be transmitted to B instead of 1000.

**IPC mechanism selection:** Different types of IPC mechanisms have different costs in terms of latency and energy consumption. These costs typically depend on the size of the message as well as other factors. Better energy efficiency might thus be had by selecting the best mechanism for each communication channel in a system.

The key point is that operating system services should be used with care and efficiency. Reducing the number of processes and system calls can result in significant gains.
2.8.3 Power management features

Most operating systems provide some mechanism for system power management. The most common model is to offer different system power states, usually different forms of active, idle, and sleep modes. Due to the wide range of hardware platforms in the embedded space, support for specific platforms may be lacking or limited. Also in many cases a standard interface to system specific power saving features is not present. This can be especially troublesome in the case of dynamic voltage and frequency scaling. Therefore the importance of choosing an operating system with broad and verified support for the target platform should not be underestimated.

**LINUX**: Linux offers support for standard power management, such as Advanced Power Management (APM) and Advanced Configuration and Power Interface (ACPI), but these are of limited interest in the embedded space since many platforms lack support and ACPI more or less requires the x86 BIOS specification. CPUFreq is the kernel module for managing frequency scaling. It lacks support for many embedded processor targets but there is limited ARM support. CPUFreq does not currently support voltage scaling.

**WINDOWS MOBILE / CE**: This operating system has a power management interface for shutdown based management and limited support for dynamic management. Device drivers are responsible for implementing the different power states.

2.9 Frequency and voltage scaling

One common solution for minimizing energy consumption in a system is to lower the core clock frequency when the system is idling or under a lighter load. Since the supply voltage required is relative to the frequency, see Equation 2.7, the core voltage can be lowered with the frequency. There are many different methods for frequency and voltage scaling, ranging from running the entire program on a constant, lower, frequency so that it just meets its deadline, to methods constantly measuring load, predicting the future, and through that decide which frequency to use.

If the hardware provides support for shutting down individual components, the problem becomes a different one. It might be better to run the processor and other components at a higher speed, thus finishing faster, and then power down the components until the next scheduled cycle.

The main goal of frequency scaling is to reduce the amount of time spent in a fully awake idle-mode. This time consumes energy while producing nothing. Since energy is proportional to the square of the core voltage and only linear to time and frequency, according to Sengupta and Saleh [3] and seen in Equation 2.6, it is more effective to lower the core voltage than to finish execution fast and spend time in idle-mode. Especially considering that idle-mode still consumes energy.

\[
E = \alpha \cdot C \cdot V_{dd}^2 \cdot f \cdot t
\]  

(2.6)
\( \alpha \) is the chip activity factor, \( C \) is the total chip capacitance.

\[
f = \xi \cdot \frac{(V_{dd} - V_{th})^2}{V_{dd}} \tag{2.7}
\]

\( \xi \) is a constant, and \( V_{th} \) is the system threshold voltage and \( V_{dd} \gg V_{th} \).

There are two main ways of performing frequency scaling, online and offline [50]. Online scaling includes Dynamic Voltage Scaling (DVS) and other methods that decide which frequency and voltage to use while the program is executing. For offline scaling the voltage and frequency are decided when the code is developed. Since offline scaling is calculated once, and on much more powerful hardware, it is possible to use a more advanced algorithm, which can lead to a better result. However, offline scaling is not able to adapt to varying execution-times and, therefore, has to assume worst case execution time for all parts. This can result in a lot of slack at runtime. Online scaling is better suited to handle this slack since it will notice it and try to correct for it. Since online scaling is calculated at runtime a good algorithm is needed which uses the available hardware to its limit. Online scaling also requires extra resources for calculations where offline only apply the settings decided beforehand.

There are many different methods for frequency scaling. The simplest ones just find the lowest frequency, which enables the system to meet its deadlines, and then uses this statically. When using dynamic scaling, the methods are much more varied. Some predict the future while others scale based on the past. A comparison between different DVS algorithms has been done by Govil et al. [51]. According to their test results it is better to use simple algorithms, based on rational smoothing, than smart predicting.

Three common DVS algorithms are [52]:

**PAST**: Look at the load during the previous time period and assume that the following one will be the same. Because of this not being the case for all programs it is important to verify how the current program’s work-cycle acts.

**AVERAGE**: Try to keep the frequency at the global average needed for the program’s deadline. However, if work is lagging the frequency should be increase enough so that at least all that was missed last period is finished. This way, the longest any work can be delayed is one period.

**PEAK**: Expect the load to come in narrow peaks. If the current workload is lower than the previous period, expect the next one to be very low. If the current workload is higher than the previous period, expect the next one to be as high as the current.

The problem with voltage and frequency scaling is that it requires support from the hardware. Some of the processors available only have a few different frequencies available. It can also be hindered on application-level if the operating system does not provide support for scaling.
2.10 Parallelism and multi-core technology

Although we assume a basic understanding of software parallelism, a quick introduction is in order. Software parallelism refers to the possibility to perform computations concurrently. This can be used at several levels in a software system, from low level Instruction Level Parallelism (ILP) to higher level task and data parallelism.

There are a wide variety of hardware architectures that targets software parallelism. Each architecture is typically most efficient at exploiting one, or a few, types of parallelism. For example superscalar and VLIW cores can exploit ILP [53], while vector processors and SIMD architectures favour data level parallel algorithms [54].

Why is parallelism interesting from an energy optimization point of view? The thought behind using parallelism in a system to reduce energy consumption is based on using multiple, less power hungry, cores instead of one powerful. This can be achieved using either less complex cores with better power–performance ratio but lower absolute performance or using the same type of cores but at lower frequency and voltage. As mentioned in Section 2.9, the dynamic power dissipation is relative to the square of the supply voltage and minimum supply voltage scales approximately linear to frequency giving cores at lower frequency and voltage a better power–performance ratio. To take advantage of this the application must be possible to parallelize with enough efficiency so that the gains are not nullified.

Li and Martínez [55] performed a thorough investigation on the merits of parallel execution on Chip Multi Processors (CMPs) as a power saving technique. They present analytical findings from modelling dynamic and static power of a CMP and considering both the parallel efficiency, or speedup, of the application, and dynamic voltage and frequency scaling of the cores. Their theoretical models show that the gains from additional cores level off quickly and dwindle, even for applications with perfect scaling. The main limiting factors identified are minimum supply voltage requirements and leakage. They complement this with simulations indicating that, for the applications and platform they use, four to eight cores is typically optimal from a power optimization point of view.

Another parallel execution technique that is often claimed to be energy efficient is Simultaneous Multi Threading (SMT) [56]. The main principle is to have cores able to execute instructions from more than one program thread each cycle and thus achieve more efficient utilization of the core. SMT and CMP can utilize the same type of parallelism and therefore it is interesting to know which one is more energy efficient. Studies indicate that this is very dependent on hardware platform and application [57, 58, 59]. CPU bound applications seem to favour CMP solutions while memory bound applications can be executed more efficiently on SMT platforms.

The method of communication and synchronization of cores is also an important consideration. Poletti et al. have studied the performance and energy implications of using coherent shared memory communication versus a message passing paradigm [60]. They conclude that there is no clear winner and what solution is best at a given problem is highly sensitive to the computation–communication
ratio, available bus bandwidth, and algorithm cache friendliness.

Multi-core and multi-processor technology can be used for energy optimization by other means than pure use of parallelism. Different processor architectures can have very wide range of efficiency depending on the computational task. By including cores that excel at different types of tasks it is possible to execute tasks on a core that favours that particular type of computation [61]. Kumar et al. [62] have studied the energy saving potential of single ISA, heterogeneous multi-processors. The conclude that by having only a small number of cores (on the order of two or three) with different power characteristics, significant savings can be achieved.

There seem to be little doubt that parallelism and multi-core technology coupled with dynamic voltage and frequency scaling offer great potential. The main drawbacks are the increased complexity of the system and the potentially very significant effort needed to parallelize a software system.

### 2.11 Quality of service

One big challenge with minimizing energy consumption is to keep meeting the requirements for quality and speed. By lowering the frequency less work can be done per time unit. When powering-down components, latency is added to the access time when they are used. It is very important to find a good balance between energy and quality of service.

Flinn and Satyanarayanan [63] showed that, by using a more aggressive JPEG compression, it is possible to reduce energy consumption. However, the change in energy is very small compared to the impact on quality. In Figure 2.4 the different quality-settings from the article can be compared. There are, however, a couple of quality trade-offs that have a larger effect on energy consumption. For instance by filtering a map so that only larger roads were visible they were able to decrease the energy consumption with up to 55%.

Another example is wireless networking. WLAN adapters can be responsible for a large part of the total system energy consumption in embedded systems. Agarwal et al. [27] has created an interesting solution to this problem. Their solution is based on using Bluetooth for signalling and WLAN for data. Since Bluetooth has much lower power dissipation than WLAN, disabling the WLAN between transfers lowers the energy consumption quite a bit. This resulted in power savings of 23% to 48%.

### 2.12 Power management

Power management (PM) is more of a grouping of the methods previously mentioned than something new. Its goal is to use the right combination of methods to produce the best result on the specific system. Categories not mentioned earlier include sleep-mode and selective power-down.

Sleep-mode is an effective way to reduce the energy consumption of a component when it is not being used. This is being done by turning off as much as
Figure 2.4  JPEG images with different quality settings. The quality can, in this case, be varied between 0 and 100, and is non-linear. The quality settings are not standardized and the ones used here are specific for IJG JPEG software.
possible of this unused component, thereby reducing its energy consumption. Beside a lower energy consumption, if a resource currently in sleep-mode receives an access request it takes some time to turn it back on. This produces latency and this latency hinders the execution. It also requires quite a lot of energy to turn the resource on again and therefore it is very important not to put components in sleep-mode at the wrong time. There are quite a few algorithms for how and when to put something in sleep-mode, for instance the one presented by Hwang and Wu [64].

Selective power-down is pretty similar to sleep-mode. The difference is that power-down turns off the resource completely where sleep-mode still has some registers and other parts up and running. To completely power-down a component reduces its energy consumption to almost nothing. The problem is that if the resource is needed it takes even longer time and uses more energy to turn it back on and it might need to be reconfigured.

Figure 2.5 illustrates a situation where the component is being accessed shortly after it has entered sleep-mode. Since the procedures for entering and exiting sleep-mode cost energy, the system in the figure will have consumed more energy by entering sleep-mode than it would have been if staying in idle-mode. There is also a delay added to the system which could have been avoided.

Different methods for when to power-down have been compared by Lu et al. [65]. The results from this comparison show that a simple algorithm can often perform better than a more advanced alternative. One example of these algorithms is the timeout algorithm.

The timeout algorithm is based on the assumption that if a device has not been used for a certain period of time it will not be used for a long time. This means that if a device has been inactive for a certain time \( \tau \) then it is time to shut-down that device. The timeout method is widely used since it is so simple and still quite effective. The challenge here is to find the right \( \tau \) for the specific system.
Chapter 3

Simulations

For the multi-core tests we used a simulator called MPARM. We would have preferred to find a development board with a multi-core processor but, being unable to find one that matched our requirements, we had to make do with a simulator. We both had previous experience with MPARM and having the opportunity to use it, we did. In this chapter we describe the simulated tests and present their results.

3.1 MPARM

MPARM was developed to help during the design stage for multi-core embedded systems [66]. MPARM is a cycle accurate simulator for embedded systems capable of presenting expected energy consumption. We use MPARM with the SWARM processor model. This model emulates a fully functional ARM7 core. MPARM claims to support 1 to 31 cores but our version only allowed 1 to 20. Each core has one private RAM-memory. There are also a selectable number of shared RAM-memories. For synchronization the simulator provides hardware semaphores which act more like locks than semaphores. The simulator also provides the possibility to scale the frequency from 1 to $\frac{1}{255}$ of the original 200 MHz. For energy profiling it is possible to divide the program into tasks and get a report about the energy consumption for each task. MPARM was configured to use 8 kB instruction cache and 8 kB data cache. Both caches were set to be 4-way set associative.

The architecture in MPARM, seen in Figure 3.1, is more of a cluster that communicates via message passing in a shared memory than the multi-core systems found in personal computers. Each core runs its own program from its own private memory. When using many cores the performance of the simulated system often became limited by the single central bus connecting all cores and memories.
Figure 3.1  The architecture of MPARM as used for the simulations.
3.2 Image filter

The unsharp mask filter was chosen as a computational kernel because it is easy to decompose into parallel subtasks and is also somewhat computationally heavy.

3.2.1 Algorithm

The unsharp mask function is defined as in Equation 3.1. In this equation \( m \) and \( n \) are pixel coordinates, \( S(m, n) \) is the sharpened result, \( I(m, n) \) is the original image, \( H(m, n) \) is the output of a high pass filtering of the original image, and \( \lambda \) is the strength of the sharpening effect. \( S(m, n) \) will also need to be saturated in order to guarantee that it stays within the numeric range of the image representation.

\[
S(m, n) = I(m, n) + \lambda H(m, n) \tag{3.1}
\]

\[
H(m, n) = I(m, n) - G(m, n) \tag{3.2}
\]

In our implementation we chose to define \( H(m, n) \) as in Equation 3.2 where \( G(m, n) \) is the output of a Gaussian blur filter and \( g(x, y, \sigma) \) is a Gaussian function. The filter causes a blurring effect by assigning each pixel the weighted average of all pixels within a certain radius of the pixel position, in this case the radius is infinite. The Gaussian function is used to produce the weights in this calculation. This averaging operation is defined in Equations 3.3 and 3.4, \( x \) and \( y \) are pixel coordinate offsets, and \( \sigma \) controls the shape of the Gaussian function, i.e. the width of the bell shaped surface. Larger \( \sigma \) gives a wider bell shape.

\[
G(m, n) = \sum_{x=-\infty}^{\infty} \sum_{y=-\infty}^{\infty} g(x, y, \sigma) I(m + x, n + y) \tag{3.3}
\]

\[
g(x, y, \sigma) = e^{-\frac{x^2 + y^2}{2\sigma^2}}, \quad x, y \in \mathbb{Z}, \quad \sigma > 0 \tag{3.4}
\]

Since the Gaussian filter is linearly separable we can perform the filtering in two steps, one in each dimension, using the one dimensional Gaussian function \( g(r, \sigma) \) and an intermediate result image \( I'(m, n) \). See Equations 3.5, 3.6 and 3.7. In the one dimensional Gaussian function \( r \) is the pixel coordinate offset and \( \sigma \) still controls the width of the Gaussian function which is now a bell shaped curve.

\[
g(r, \sigma) = e^{-\frac{r^2}{2\sigma^2}}, \quad r \in \mathbb{Z}, \quad \sigma > 0 \tag{3.5}
\]

\[
I'(m, n) = \frac{\sum_{r=-\infty}^{\infty} g(r, \sigma) I(m + r, n)}{\sum_{r=-\infty}^{\infty} g(r, \sigma)} \tag{3.6}
\]
\[ G(m, n) = \frac{\sum_{r=-\infty}^{\infty} g(r, \sigma)I'(m, n + r)}{\sum_{r=-\infty}^{\infty} g(r, \sigma)} \] (3.7)

Due to the lack of hardware floating point support on the MPARM platform, all computations are done in fixed point representation to avoid costly software floating point emulation. We used a representation with 24 bits for the integer part and 8 bits for the fractional part.

Since the implementation cannot perform an averaging operation over an infinite number of pixels an approximation had to be used. We, somewhat arbitrarily, settled on using a radius of eight because it gave a reasonable computational complexity and produced a nice result. To make the Gaussian function match this radius we needed it to take an insignificant value \( \epsilon \) when \( r = 8 \). We defined \( \epsilon \) as \( \frac{1}{3} \cdot \frac{1}{256} \). Since the smallest nonzero value that can be represented in the 24.8 fixed point representation is \( \frac{1}{256} \), \( \epsilon \) will be rounded to zero. \( \sigma \) could now be calculated using Equation 3.8, derived by setting \( g(r, \sigma) = \epsilon \) and solving for \( \sigma \).

This gave us \( \sigma = 2.194 \ldots \approx 2.2 \) and the eight element vector approximation of \( g(r, \sigma) \) shown in Table 3.1.

\[
\sigma = \sqrt{-\frac{r^2}{2\ln \epsilon}}, \quad r \in \mathbb{Z}, \quad 0 < \epsilon < 1
\] (3.8)

The algorithm for the sequential implementation is very straightforward as can be seen in Algorithm 3.1. Example output of the filter is depicted in Figure 3.2.

For the parallel experiments several versions of the algorithm was tested. All have the same general principle. First the image is read from file to shared memory. Then the rows of the image are partitioned fairly across the cores so that the size of the working set for each core differ with no more than one row, see Figure 3.3 for an example. An initial synchronization is performed to guarantee that the image is available in the shared memory before proceeding. The cores then perform

<table>
<thead>
<tr>
<th>( r )</th>
<th>( g(r, 2.2) )</th>
<th>Fixed-point 24.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0000</td>
<td>\frac{1}{256}</td>
</tr>
<tr>
<td>1</td>
<td>0.9014</td>
<td>\frac{1}{256}</td>
</tr>
<tr>
<td>2</td>
<td>0.6602</td>
<td>\frac{1}{256}</td>
</tr>
<tr>
<td>3</td>
<td>0.3929</td>
<td>\frac{1}{256}</td>
</tr>
<tr>
<td>4</td>
<td>0.1900</td>
<td>\frac{1}{256}</td>
</tr>
<tr>
<td>5</td>
<td>0.0746</td>
<td>\frac{1}{256}</td>
</tr>
<tr>
<td>6</td>
<td>0.0238</td>
<td>\frac{1}{256}</td>
</tr>
<tr>
<td>7</td>
<td>0.0062</td>
<td>\frac{1}{256}</td>
</tr>
</tbody>
</table>

Table 3.1 Gaussian window vector.
3.2 Image filter

Figure 3.2  Input, output and intermediate results of the unsharp mask filter.

<table>
<thead>
<tr>
<th></th>
<th>Original image</th>
<th>Sharpened image</th>
<th>Unsharp mask</th>
<th>High frequency components</th>
</tr>
</thead>
</table>

![Original image](image1.png) ![Sharpened image](image2.png) ![Unsharp mask](image3.png) ![High frequency components](image4.png)

The Gaussian blur filter in x dimension, synchronize to guarantee availability of intermediate results, perform the y dimension blur, and the final sharpening step. The last synchronization is then done before outputting the image to file. All synchronizations, three in total, are done using barriers. Algorithm 3.2 describes the basic principle.

On the MPARM platform shared memory is uncacheable. This leads to very poor performance of Algorithm 3.2 since all image data is in shared memory. To remedy this, an alternative version was developed. In Algorithm 3.3, each core copies its working set from shared memory to private memory, performs the computations, and copies needed data back to shared memory so that it becomes available to each neighbouring core. Figure 3.4 visualizes the central steps of

![Figure 3.3](image5.png)

Figure 3.3  Example partitioning of a 12 pixels wide and 11 pixels high image.
Simulations

```plaintext
for all cores in parallel do
    if this is master core then
        $I_{shared} \leftarrow$ read image file to shared memory
    end if
    wait for all cores
    start energy measurement
    $myrows \leftarrow$ fair row partitioning
    $I'_{shared} \leftarrow$ gauss-x($I_{shared}$, $myrows$)
    wait for all cores
    $G_{shared} \leftarrow$ gauss-y($I'_{shared}$, $myrows$)
    $S_{shared} \leftarrow$ sharpen($I_{shared}$, $G_{shared}$, $\lambda$, $myrows$)
    wait for all cores
    stop energy measurement
    if this is master core then
        write $S_{shared}$ to file
    end if
end for
```

Algorithm 3.2 The parallel unsharp mask algorithm.

Because of its higher performance, Algorithm 3.3 was used for all multi-core tests and Algorithm 3.2 was not used further.

The image filter was divided into four subtasks to simplify the accounting of energy and time consumed during the simulations and measurements.

**gauss-x:** This task is the Gaussian blur along the x-axis or width of the image. This part is identical for both the sequential and parallel implementation.

**gauss-y:** This task is the Gaussian blur along the y-axis or height of the image. This part is also identical for both the sequential and parallel implementation.

**sharpen:** Here, the filtering of the original image using the unsharp mask is performed. This task differs in the respect that the target buffer is in shared memory for the parallel implementation.

**overhead:** All additional work needed due to the parallelization of the algorithm is grouped into this task. This includes calculating the size and location of the sub-images for each core, copying of data from private to shared memories etc. It is dominated by the copying operations (visualized in Figure 3.4a, 3.4c, and 3.4d).

### 3.2.2 Compiler effects

Compiler optimizations have a large impact on the performance of an executable and, as explained in Section 2.7, compilers can also affect the energy consumption. Therefore we performed a series of tests to determine the best compiler setting for the image filter code. The application was compiled and tested with five different
3.2 Image filter

Figure 3.4 Visualization of the parallel image filter algorithm as executed by the second core in a total of three. The image data is colour coded according to the partitioning seen in Figure 3.3.
Simulations

for all cores in parallel do
    if this is master core then
        \( I_{\text{shared}} \leftarrow \text{read} \) image file to shared memory
    end if
    wait for all cores
    start energy measurement
    \( \text{myrows} \leftarrow \text{fair row partitioning} \)
    \( I_{\text{private}} \leftarrow \text{copy} \ \text{myrows} \) and rows covered by gauss kernel, from \( I_{\text{shared}} \)
    \( I_{\text{private}}' \leftarrow \text{gauss-x}(I_{\text{private}}, \text{myrows}) \)
    \( I_{\text{shared}}' \leftarrow \text{copy} \) rows covered by neighbours gauss kernel, from \( I_{\text{private}}' \)
    wait for all cores
    \( I_{\text{private}}' \leftarrow \text{copy} \) rows covered by gauss kernel, from \( I_{\text{shared}}' \)
    \( G_{\text{private}} \leftarrow \text{gauss-y}(I_{\text{private}}', \text{myrows}) \)
    \( S_{\text{shared}}' \leftarrow \text{sharpen}(I_{\text{private}}', G_{\text{private}}, \lambda, \text{myrows}) \)
    wait for all cores
    stop energy measurement
    if this is master core then
        write \( S_{\text{shared}} \) to file
    end if
end for

Algorithm 3.3 Parallel unsharp mask algorithm using private memory.

compiler settings: O0, O1, O2, O3, and Os. O0 performs no optimizations, O1 to O3 performs successively more aggressive speed optimizations, and Os tries to minimize the size of the executable. The executables were run on a single core.

Results

In Figure 3.5 we can see a strong relationship between execution time and energy consumption. However, it is also clear that the power dissipation does vary with optimization setting. In this test the power consumption varies with almost 10%.

Since the O1 setting was best in terms of both performance and energy, it was used for all further tests.

3.2.3 Quality of service

Lowering the quality of service is a well known way of increasing performance and lower energy consumption. Image manipulation is well suited for different quality of service optimizations since the quality of the result can be subjective and the correctness is not a Boolean function.

We implemented a quick quality of service optimization by approximating the unsharp mask with a lower resolution version. Since most time is spent in the gauss kernel, shrinking the input to this operation greatly reduces the amount of computation needed. Also, when the input is scaled, the radius of the gauss kernel needs to be scaled by the same amount to achieve the same blurring effect. Thus
3.2 Image filter

we get additional gains from lowered computational complexity for each pixel.

The reduction of work can be approximated using Equation 3.9 where $n_{ops}$ is the number of operations per pixel in the input image and $\alpha$ is the ratio between input resolution and the internal resolution. When $\alpha = 1$, i.e. no loss in quality, each pixel requires 259 operations, $\alpha = \frac{1}{2}$ gives 60 operations and $\alpha = \frac{1}{4}$ requires 34.75 operations per pixel. This translates into work reductions of 77\% and 87\% for $\alpha = \frac{1}{2}$ and $\alpha = \frac{1}{4}$ respectively. As Equation 3.9 is derived by counting the number of operations in the source code, disregarding indexing and address calculations and treating all operators as equally costly, the expression is only a rough estimation of the computational cost per pixel.

$$n_{ops} = 224\alpha^3 + 4\alpha^2 + 31, \quad 0 < \alpha \leq 1$$

(3.9)

Results

As seen in Figure 3.6 the gains are quite substantial. By halving the resolution of the blurred image the energy consumption is reduced by more than 75\% and using one fourth of the original resolution results in a reduction of approximately 90\%. The reductions in execution time are slightly larger than the 77\% and 87\% estimates given by Equation 3.9.

The reduction in quality of the final result is depicted in Figure 3.7. The standard and $\frac{1}{2}$ resolution results are very hard to tell apart. The $\frac{1}{4}$ resolution has clear block artefacts due to the reduced resolution but could possibly still be useful in some situations. We measured the difference by computing the length of the RGB vector that separated each pixel in the standard result and the corresponding pixel in the lower quality images. The $\frac{1}{2}$ resolution image has an average

![Figure 3.5](image.png)
Figure 3.6  Comparison between different quality of service settings. Results are normalized to the standard setting. The values can be found in Table A.2.

Figure 3.7  Original image and sharpened images with different quality setting.
difference of 3.43 and a maximum of 27.4. The $\frac{1}{4}$ resolution image is clearly a lot worse with an average difference of 9.12 and a maximum of 72.3.

It is clear from these results that reducing the computational load by sacrificing quality can deliver very significant gains for this type of application.

### 3.2.4 Parallel performance

In order to ensure that the parallel implementation was reasonably robust, a basic scaling analysis was performed. This was done using a square reference image with a size of 200 times 200 pixels. This image was filtered using the parallel implementation running on 2 to 16 cores and then compared to the performance of the sequential implementation running on a single core.

In addition to this, the sensitivity to input image size was tested. This was done using the same reference image scaled to three additional sizes. In total four square images were used with sides of 200, 150, 100, and 50 pixels respectively.

#### Results

As can be seen in Figure 3.8, the filter scales nicely to eight cores but falls off at 16 cores. The majority of the performance loss is caused by bus contention. Using 16 cores, the simulator reports 100% bus utilization during the memory and bus intensive parts of the filter. This is also evident in Figure 3.9, depicting the total cycle count consumed to complete each part of the algorithm using different number of cores. The sharpening and overhead parts, being the most memory and bus intensive, show a dramatic increase in cycle count for 16 cores, indicating severe bus contention. This could be mitigated to some extent by merging the first copying step with the first gauss pass and the sharpening step with the second
Figure 3.9  The total core cycles consumed by the subtasks of the image filter using different number of cores. The values can be found in Table A.4.

Figure 3.10  The speedup of the parallel image filter compared to the sequential implementation, using different images sizes. The values can be found in Table A.3.
pass. This would produce a more favourable ratio between computations and memory references. However, it would significantly alter the code structure and make the comparisons between the sequential and the parallel implementation less clear. As the performance degradation mainly affected the 16 core configuration, it could also provide an interesting data point as to the effects of contention on the application. Therefore no further optimizations were implemented.

The results of the input size sensitivity test are plotted in Figure 3.10. It is clear that there is sensitivity to input size but as the image size goes above 100 times 100 pixels, which perhaps represent more realistic input sizes, the scaling approaches the ideal.

We concluded that the algorithm, and its parallel implementation, is sufficiently scalable for our needs, that is, to serve as an example of a simple workload with good scaling.

### 3.2.5 Energy

To evaluate the potential for energy savings using a multi-core platform a series of simulations was conducted using the MPARM simulator. The main energy saving strategy was to use multiple cores at lower frequency. This should in theory give substantial gains for an application with good scaling. The tests use a 200 by 200 pixel image and executables compiled with O1 for both the parallel and sequential codes.

![Normalized energy and time comparison for different number of cores and frequencies.](image)

**Figure 3.11** Comparison of energy consumption and execution time for the image filter using different number of cores. Results are normalized with regard to the single core reference. The values can be found in Tables A.5, A.6 and A.7.
Figure 3.12 Breakdown of energy consumption for each subsystem when executing the image filter code. The values can be found in Table A.8.

Results

Figure 3.11 shows the results of this test. We can see that by scaling the cores to 100 MHz instead of the default frequency of 200 MHz, an energy consumption reduction by roughly 15-20% can be achieved depending on the number of cores used. Although this is a significant gain, we had hoped for more, especially considering that the effort needed to parallelize an algorithm is usually quite high.

The main reason for the somewhat underwhelming improvements can be found in Figure 3.12. It is clear that the improvements gained by reducing core frequency become irrelevant very quickly as the minimum operating voltage is reached. The static consumption however continues to rise with the increase in execution time. Many of these factors are, of course, platform dependent. The operational voltage range might be larger or smaller and the ratio of static and dynamic consumption might differ. The main point however is to realize that these limitations exist on all platforms to some degree. It is common for embedded processors to support a rather narrow voltage range.

Another cause of the limited improvements can be found in the way the simulator instantiates additional cores. Since each core gets its own private memory the static energy consumption of the system increases substantially for each additional core. This means that execution time becomes more and more expensive in terms of energy for each additional core, thus requiring a significant decrease in runtime in order to keep the static energy consumption at a reasonable level. Back of the envelope calculations indicate that this increases the optimal energy consumption with 5-10% compared to using the same platform but without increasing memory size.

The key point in these results are the assumption that the hardware platform can use power management features to power down cores so that the static energy
consumption in the system becomes equal or lower than a single core system. Without such possibilities there can be no gains at all. This is obvious since the voltage and frequency scaling could not reduce the dynamic power enough to offset the increased static power dissipation. The gains observed come primarily from the lowered static energy consumption as a result of increased execution speed. So if there is no way to reduce the power dissipation after the work is completed, the parallel implementation will perform much worse than the sequential. One counter point to this is that, in this test, all cores are identical. The total computational power of multi-core configuration far exceeds the single-core version. Thus the multi-core platform could replace a much more powerful single core processor with potentially even higher static power consumption.

### 3.3 Quicksort

Quicksort is a well known sorting algorithm [67] invented by Hoare [68]. It selects a pivot-element, either by random or via some algorithm. Then it divides all other elements into two groups, one with elements larger than the pivot, and one with the elements that are smaller than, or equal to, the pivot. The pivot element is then swapped into place between the two groups. The groups are then individually put through the same routine until there is only one element left in each group. The result is a sorted list of elements. The worst case number of swaps for Quicksort is $O(n^2)$. The expected number of swaps is $O(n \log n)$. The testing was done with the algorithm described in Algorithm 3.4.

Unlike the image filter, which can be evenly parallelized, Quicksort has to be run sequentially at first to divide the data correctly. On top of the initial partitioning, the different pieces do not require the same amount of work. This is due to both the data set being of different sizes and the sorting process requiring different number of swaps depending on the data.

The results from each test are presented in the following as both a diagram and a table. The diagram only shows the result with the lowest energy consumption for each core configuration while the more interested reader can find and compare more detailed results via the tables found in Appendix B.

#### 3.3.1 Parallel Quicksort

There have been lots of studies on how to make the best parallel Quicksort. When it comes to the parallel part of Quicksort it is pretty straight forward what has to be done. The differences here lie in how the communications between the cores work. When it comes to the initial partitioning it becomes a bit trickier.

Grama et al. [69] suggests dividing the initial problem into equal parts and letting each core do partitioning using the same pivot element on their part. The cores then store information about how many of their elements are to be in the left partition and how many in the right partition. This information is then used by each core to place its left and right partitions in the right place in the memory. The same strategy is used whenever there are fewer tasks than cores.
function quicksort(left, right)
{
    min ← left
    max ← right − 1
    pivot ← right

    while min < max do
        while (element\textsubscript{min} ≤ element\textsubscript{pivot}) and (min < max) do
            min ← min + 1
        end while
        while (element\textsubscript{max} > element\textsubscript{pivot}) and (min < max) do
            max ← max − 1
        end while
        swap(min, max)
    end while

    if (min ≠ right − 1) or (element\textsubscript{min} > element\textsubscript{pivot}) then
        swap(min, pivot)
        pivot ← min
    end if

    quicksort(left, pivot − 1)
    quicksort(pivot + 1, right)
}

Algorithm 3.4 This is the Quicksort algorithm as implemented for the testing.

Tsigaś and Zhang [70] proposes a somewhat different approach. The problem is divided into blocks of such a size that two of them can occupy the cache at the same time. The cores then get two of these blocks, one from the right and one from the left. Each core then finds elements larger than the pivot in the left block and swaps them with elements smaller than the pivot in the right block. If the left block now only contains elements smaller than the pivot it is marked and vice versa for the right block. This goes on until all blocks have been processed. The unmarked blocks are then corrected sequentially. The initial partitioning is then complete. The cores are divided between the parts and the partitioning begins again if multiple cores are assigned to the same partition.

3.3.2 Our parallel algorithm

For our implementation we chose the simple approach by performing the initial partitioning sequentially, see Figure 3.13. After the initial part the task is split into two. The whole process is then repeated until there are no numbers left or the task has been divided into the predefined number of partitions. The communication between the cores is based on locks and messages passed through the shared memory. To simplify the implementation a message buffer with only one slot was used. This means that if the buffer is full, the core which is dividing will have to do the work itself. If the buffer is empty the core will input the delimiters for the first half into the buffer and then continue working on the second one.
This buffer handling has been illustrated in Algorithms 3.5 and 3.6. Later in the project, a version with a queue of variable length was tested to see how the simple buffer solution impacted the energy consumption. The results from that version indicated that the queue can increase performance but also increases the energy consumption. The study can be found in Appendix C.

Since the selection of pivot element can result in different sizes of the two groups the load will most likely not be perfectly balanced between the different cores. That some parts require different numbers of swaps, and thus different amounts of time to complete, leads to a more uneven load balance. This can be compensated for by dividing the problem into smaller parts. Too many tasks risk making the energy consumed by the overhead of moving data and commands grow larger than the energy saved.

This test was performed on five different core counts; 1, 2, 4, 8, and 16.
while sorting not done do
    WAIT(buffer_full)
    fetch_from_buffer(left, right)
    SIGNAL(buffer_empty)
    quicksort(left, right)
end while

Algorithm 3.6 This is the algorithm for the multi-core Quicksort used while waiting for work to do.

Figure 3.14 Normalized results for different numbers of cores performing Quicksort. More detailed results can be found in Table B.1.

Results

As expected, and can be seen in Figure 3.14, just increasing the number of cores does not lower the energy consumption. In fact, increasing the number of cores is not even guaranteed to decrease the execution time due to the introduction of extra work.

While varying the number of tasks, it seemed as if larger values were better. Having fewer tasks than cores is, of course, a bad choice. One problem with these tests is that when using 16 cores, the bus-usage is very near 100% which means that the bus is acting as a bottleneck for the system. This results in 16 cores actually performing worse than 8 cores. The reason for the high bus utilization is because of semaphores. These semaphores are used to control access to the buffer. The cores waiting for work are running at full speed and checking for available data by constantly polling the semaphore unit.

To sum up, all results consume more energy than a single-core and using two cores is also slower than the single-core reference. Note that this is without any scaling. This means that all cores are running at full speed all the time.
3.3 Quicksort

![Figure 3.15](image)

**Figure 3.15** Normalized results for using private memory relative to same number of cores with shared memory for the Quicksort implementation. More detailed results can be found in Table B.2.

### 3.3.3 Memory management

It was obvious in early testing that the memory would be an important part of the energy consumption. When having multiple cores in MPARM, each core has its own private memory. In addition to the private memories there is one shared memory. Because of how the caches worked, one obvious thing to test was using private memory compared to using only the shared memory. The shared memory was still used for passing data and messages between the cores. Since the private memories are found on the same bus as the shared memory and the caches are write-through, switching to private memories was not expected to solve the 100% bus usage problem for 16 cores.

**Results**

In Figure 3.15 the results from using private memory are compared to using shared memory. The comparison is done between configurations with the same number of cores and not relative to the single core reference.

The use of private memory lowered the energy consumption by a maximum of 19% compared to using the same number of cores with only shared memory. The average is not quite as impressive though at 5.1%. The minimum is even worse at -6.5%. The big problem with private memory in MPARM is its size and the fact that it is connected to the same bus as the shared memory. The size makes accesses costly energy wise and the bus configuration means that the cores still have to fight for time on the bus.

The change in execution time was more impressive. The maximum change in execution-time was a lowering of 29%. The average was 17%, and the minimum
shortening of execution time was 5.3%. These changes are due to cached data being accessed a lot faster than data in the RAM memory.

Just switching to private memory did not lower the energy consumption to the reference levels though. The lowest energy consumption is a two core configuration and it is still 35% higher than that of the reference. The execution time is lowered by up to 40% compared to the reference. The fastest execution time belonged to a configuration with eight cores.

### 3.3.4 Frequency scaling

As mentioned before, a common way to lower energy consumption is to use frequency scaling. It is not necessarily optimal to use the lowest frequency. The main reason for this is the static energy consumption which increases linearly with execution time. On top of that, the voltage quickly reaches its minimum operating value meaning that, even if the frequency decreases, the voltage cannot be lowered more.

The scaling was done in four different ways. First the frequency for when the cores were in idle-mode was varied. After that the frequency of the parallel part was varied. Then the sequential part, the top-level partitioning, had its frequency lowered. Finally all three were varied at the same time to find an optimum setting for this implementation.

When scaling the frequency for idle mode the eight different dividers tested were 2, 3, 5, 10, 25, 50, 100, and 200. These produce a resulting frequency of 100 MHz, 67 MHz, 40 MHz, 20 MHz, 8 MHz, 4 MHz, 2 MHz, and 1 MHz respectively.

For the parallel part the frequency divider was varied in four steps; 1, 2, 3, and 4. The resulting frequencies are 200 MHz, 100 MHz, 67 MHz, and 50 MHz. These frequencies were also used for testing on the sequential part.

#### Results

When looking at the individual scaling results only idle scaling, seen in Figure 3.16, showed an improvement in energy consumption, especially on the configurations with many cores. It even lowered the execution time, seen in Table B.3, in some cases due to a lowering of the number of bus accesses.

By changing to an idle-frequency of 1 MHz, the bus utilization of the configuration with 16 cores decreased from almost 100% to about 25% according to MPARM’s output.

The only configuration that got lower energy consumption from scaling the frequency in the parallel parts (see Figure 3.17) was the one with two cores. For that configuration execution time increased but the energy consumption decreased. All other configurations resulted in a higher energy consumption when scaling either the parallel or sequential (see Figure 3.18) parts.

When scaling all three parts at the same time, see Figure 3.19, the results were a lot better. Execution-time increased but the energy consumption decreased by up to 67%, with an average of 38%, compared to the same number of cores but without scaling. Meanwhile the execution-time increased with 53% to 91%. The
3.3 Quicksort

Figure 3.16 Normalized results from idle frequency-scaling on Quicksort compared to same number of cores without scaling. More detailed results can be found in Table B.3.

Figure 3.17 Normalized results from parallel frequency-scaling on Quicksort compared to same number of cores without scaling. More detailed results can be found in Table B.4.
0.2
0.4
0.6
0.8
1
1.2
1.4

2 cores 4 cores 8 cores 16 cores
Without scaling With scaling
Normalized energy
Normalized time

Figure 3.18 Normalized results from sequential frequency-scaling on Quicksort compared to same number of cores without scaling. More detailed results can be found in Table B.5.

0.5
1
1.5
2
2.5

2 cores 4 cores 8 cores 16 cores
Without scaling With scaling
Normalized energy
Normalized time

Figure 3.19 Normalized results from the combined scaling compared to same number of cores without scaling on the Quicksort implementation. The idle frequency divider was set to 50. More detailed results can be found in Table B.6.
biggest changes in energy consumption came, not very surprisingly, from the 16 core configuration. With two cores, the energy consumption was decreased with a more modest maximum of 26%. As can be seen in Figure 3.20, too much scaling can be bad for both performance and energy consumption. This is due to the static energy component, very much visible in the RAM’s energy consumption.

When only using scaling, the lowest energy consumption obtained was 8.3% more than the reference, seen in Figure D.5. This was with two cores and 85% slower than the reference. Time wise, the closest was a configuration with eight cores which took 2.7% longer and used 23% more energy compared to the reference.

### 3.3.5 Optimizing the algorithm

Quicksort is good for large pools of elements but when for small pools it can be more effective to use another algorithm. To test this, the following algorithms were used:

**Bubblesort:** Bubblesort [67], see Algorithm 3.7, moves elements in one direction. This means that one element is put into position each iteration. Does $O(n^2)$ comparisons and a maximum of $O(n^2)$ swaps.

**Shakersort:** Shakersort, see Algorithm 3.8, moves elements in both directions and should be more effective than Bubblesort. Does $O(n^2)$ comparisons and a maximum of $O(n^2)$ swaps.

**Selectionsort:** Selectionsort [67], see Algorithm 3.9, finds the smallest element in the remaining pool and moves it into its correct position. It then repeats

![Figure 3.20](image)
function bubblesort(left, right)
{
    for $i = right - 1$ to $left$ do
        for $j = left$ to $i$ do
            if $element_j > element_{j+1}$ then
                swap($j, j + 1$)
            end if
        end for
    end for
}

**Algorithm 3.7** The Bubblesort algorithm.

with the remaining elements until the list is sorted. Does $O(n^2)$ comparisons and a maximum of $O(n)$ swaps.

These algorithms were used when the size of the partition reached a predefined number of elements. The numbers tested were 4, 5 and 6. These numbers and algorithms are not necessarily best but should show if this path is worth taking.

**Results**

As expected, lower energy consumption was obtained by combining Quicksort with another algorithm. The gain seems to be dependent on the number of cores with the single-core obtaining the best improvement. The normalized results can be viewed in Figure 3.21. For all but one configuration, Selectionsort produced the best result. With two cores Bubblesort managed to take the first place.

**3.3.6 Compiler optimizations**

All of the tests were done with three different optimization settings in the compiler: O1, O2, and O3. The average energy consumptions for each optimization setting were calculated. The reference design with one core was tested with O0 and Os in addition to O1, O2 and O3.

**Results**

O2 and O3 performed about the same in all the tests. In four tests O3 came out on top and in the other three O2 did. Overall O2 resulted in a slightly smaller energy-consumption than O3 by 0.4%. O1 was constantly the worst one by up to 16% compared to the lowest result. The normalized results can be seen in Figure 3.22.

It is clear in Figure 3.23 that O0 should be avoided unless very special circumstances require it.
3.3 Quicksort

function shakersort(left, right)
{
    min ← left
    max ← right
    while min < max do
        for i = min to max − 1 do
            if element_i > element_{i+1} then
                swap(i, i + 1)
            end if
        end for
        max ← max − 1
        for i = max − 1 to min do
            if element_i > element_{i+1} then
                swap(i, i + 1)
            end if
        end for
        min ← min + 1
    end while
}

Algorithm 3.8 The Shakersort algorithm.

function selectionsort(left, right)
{
    for i = left to right − 1 do
        min ← i
        for j = i + 1 to right do
            if element_j < element_{min} then
                min ← j
            end if
        end for
        swap(i, min)
    end for
}

Algorithm 3.9 The Selectionsort algorithm.
Figure 3.21  Normalized results for using other algorithm to finish of the sorting compared to the same number of cores using only Quicksort. The algorithm and partition size varies among the best results. More detailed results for each algorithm and partition size can be found in Table B.8.

Figure 3.22  Normalized time and energy for Quicksort compiled with different compiler settings with O1 as reference. More detailed results can be found in Table B.9.
3.3 Quicksort

Figure 3.23 Normalized time and energy for Quicksort with different compiler settings using one core in MPARM with O0 as reference. More detailed results can be found in Table B.10.

3.3.7 Combining results

The last test was to see how combining all the different methods changed the energy consumption.

Results

The multi-core configurations are still unable to reach the same energy consumption as the reference design, see Figure 3.24. Using two cores came close but with a much longer execution time. When looking at the individual parts, see Figure 3.25, it is obvious that the RAM-memories are part of the problem. The optimal configurations for 8 and 16 cores use shared memory. Because of this it should be possible to decrease the size of the private memory or even remove it. Sadly MPARM does not allow this. There are switches for it but they were not implemented in our version.

Since the goal was to lower energy consumption by using multiple weaker cores instead of one powerful these tests do not entirely show the truth. All cores had exactly the same configuration.

Because the RAM-memories were such large contributors in the multi-core systems and MPARM did not allow these to change a rough estimation was calculated. The energy-consumption of each configuration’s RAM-memories was replaced by the reference value. The result can be seen in Figure 3.26. After this recalculation the 8 core configuration was able to beat the reference energy wise with a tiny margin. Time wise it was about 20% slower. All the other configurations failed in both regards.
Figure 3.24  Normalized time and energy for Quicksort using the optimal settings from each test compared to the best single-core. More detailed results can be found in Table B.11.

Figure 3.25  Normalized energy distribution for Quicksort compared to reference total. These values can be found in Table B.12.
Figure 3.26  Normalized time and energy for Quicksort using the optimal settings from each test compared to the best single-core result and compensating for the RAMs added by MPARM. These values can be found in Table B.13.
Chapter 4

Hardware

To run the tests on hardware a development board with one single-core processor was chosen. This board was used to run a few tests that the simulator was unable to perform as well as compare results with the simulator.

In this chapter we present these tests and their results. Additionally, the method for testing is described.

4.1 Hardware platform

Since we were unable to find a reasonably priced board that had multiple cores and enabled power measurements, a board from Olimex, SAM9-L9260 [71], with a single ARM9-processor was chosen. The hardware platform is based around an Atmel AT91SAM9260 processor [72]. This processor has an ARM926EJ core, a six-layer bus matrix, Ethernet and a few other features. The development board came with arm-linux pre-installed. This drastically reduced the time it took to convert the software used for testing.

It was easy to measure energy and power consumption on this board since both power supplies were routed through individual jumpers. All that needed to be done was to connect a resistor and measure the current in each line. The board has two power supplies: 1.8 V and 3.3 V. The 1.8 V supply is used only by the ARM core while the 3.3 V supply is used by the I/O on the ARM core and to power other components, like RAMs, Flash, Ethernet physical interface (PHY), etc. There is a third power supply located outside the development board: the 5 V. This power supply is used to power the board and thereby used to obtain the 1.8 V and 3.3 V. For a complete schematic see the board manual [71].

4.1.1 Testing equipment

To measure the current from each power supply we used the following equipment:

**Tektronix TDS 3034** [73]: To view the voltage-drop over the resistor as well as the trigger-signal, trace-signal and voltage-level of the measured line a
TDS3034 from Tektronix was used. This is a 300MHz oscilloscope with four channels. Our version also had a communication-module enabling remote-access via Ethernet. The scope we used had not been calibrated for quite some time and therefore had offsets between the channels. The noise-level introduced by the A/Ds in the oscilloscope was quite high at 40 mV peak-to-peak, which made measurements less accurate. This was measured by connecting the probe to the ground reference point on the oscilloscope itself. We did not have the equipment required to measure the noise at the measurement points on the development board but the limitation seemed to be the oscilloscope in this case.

**Analog Devices AD620 [74]:** To work around the high noise-level of the oscilloscope it was decided to use an AD620. This component is also known as an instrumentation amplifier. Its functionality is to take the difference between the two inputs and amplify it as specified by the external gain-resistor. By using this component, the difference could be isolated and amplified enough to not be affected by the noise-level of the oscilloscope. When using a gain resistor of 560 $\Omega$ the resulting amplification is about 89 times. This means that a 10 mV difference results in an output signal of 890 mV, more than enough to clear the noise-margin. The output from the amplifier corresponded very well to the program being executed which made us confident that this solution performed as it was supposed to.

**Velleman PS 613:** The power supply for the test-bench was a PS613 from Velleman. This power supply was set to provide $\pm 12$ V to the AD620 and 5 V to the board.

### 4.1.2 Measurement setup and method

Of the oscilloscope’s four inputs, two were used for triggering. This left two channels. With three power supplies to be measured it was not possible to measure them all at once. Therefore each power supply was measured alone and the last
channel was used to capture the voltage of the supply currently being measured. Each test was run multiple times for each power supply to try to remove any anomalies. A flashing LED on the board had a noticeable impact on two of the power supplies. Because of this all tests were run during the period in which the LED was off. The principle used to measure is illustrated in Figure 4.1. In the figure, the power supply being measured is the 1.8 V. Figure 4.2 shows the development board being measured.

4.1.3 Mitigating the noise issue

Since the oscilloscope used in our testing had four channels it was possible to use the built-in diff-functionality to calculate the drop in voltage over the measurement resistor. It became clear after a couple of tests that the precision of this method of measurement was not enough. A few different solutions were found but because we aimed for a low cost solution, investing in an AD620 seemed like a good choice.

The oscilloscope was configured to have as large offset on each channel as needed to get more details on the signal. The AD620 was used with a gain-resistor of 560 Ω resulting in an amplification of about 89 times. The resistor, over which the voltage-drop was measured, consisted of six 1 Ω resistors connected in parallel resulting in a resistance of \( \frac{1}{6} \) Ω. The reason we decided to go with this specific resistance is that it was the largest one we could use while still having a stable execution. Larger resistances tended to result in weird errors and corruptions. When comparing the results, the output from the AD620 was divided by
its amplification-factor to simplify the comparison. Since the oscilloscope had not been calibrated for quite some time there was a slight offset between its channels. This was compensated for by measuring the same signal with both channels and finding the centre point between the channels. The offset to this point was then used to correct all other measurements.

After a comparison between the diff-functionality and the AD620 it was clear that by adding a single component, the measurements could change from mostly noise to usable data. Example results from both methods can be seen in Figure 4.3.

Since the diff-functionality produces such a noisy result, we decided to compare with the running average of the diff as well. By averaging 200 samples the result displayed in Figure 4.4 was obtained. This is more useful than the original noisy measurement but still lacks a lot of details compared to the AD620 output. One could try to find a more advanced filtering method but since the result from the AD620 was that much clearer we decided to use that for our tests.

When looking at different time ranges, see Figure 4.5, the same results present themselves. The unfiltered diff-signal is very noisy and difficult to draw any conclusions from. The filtered diff-signal is better for visual analysis but the AD620 still seems superior. One problem with the AD620 is that, because of its limited bandwidth, it fails to register short, large spikes like the ones found in Figure 4.5a. The reason for the spikes is probably ringing in the power regulator caused by a sudden increase in current. Looking at the lowest value of -0.39 V, the current that matches this drop in voltage would be 2.4 A towards the power supply which seems a little odd.
Figure 4.4  Comparison between using an AD620 and the running average of the integrated diff-functionality. Total time is 1 ms. A larger version can be found in Figure E.2.

Figure 4.5  Comparison between using an AD620 and the diff, and an AD620 and the filtered diff for different time ranges. Larger versions can be found in Figures E.3, E.4, E.5 and E.6.
4.2 Hardware measurements

4.2.1 Compiler settings

A series of compiler optimization tests, corresponding to those done on MPARM, was also performed on the L9260 hardware. Both the Quicksort and the image filter code were used. Each application was compiled into five executables with different optimization settings: O0, O1, O2, O3, and Os. For this test the arm-linux-gcc compiler, version 3.3.2, was used. The executables were then run three times each on the L9260 board and the results were averaged among the three runs.

Results

The results where normalized to the O0 executable for each application. In Figures 4.6 and 4.7 we can see that energy and time track each other very closely. If we look at the power figures we see that the difference in power dissipation between optimization levels is very small, only the O0 option for the Quicksort tests differs significantly.

It is also clear that deployment of unoptimized executables in an energy constrained environment should be avoided at all costs. An optimized executable can have as much as four times higher performance and energy efficiency than an unoptimized version.

4.2.2 Input data

As input data affects the number of switching events in functional units and on data buses, it is interesting to investigate how this affects the energy consumption
4.2 Hardware measurements

of the system. This test was performed with the image filter program. Seven test images were constructed to cause different amounts of switching in the data paths. All images are 200 by 200 pixels.

**Black:** This is a completely black image, i.e. RGB values are (0,0,0) for all pixels.

**White:** This is a completely white image, i.e. RGB values are (255,255,255) for all pixels.

**Stripe x:** This is a black and white image with one pixel wide stripes in the x dimension, i.e. RGB is (0,0,0) for odd rows and (255,255,255) for even rows.

**Stripe y:** This is a black and white image with one pixel wide stripes in the y dimension, i.e. RGB is (0,0,0) for odd columns and (255,255,255) for even columns.

**Checkered:** This is an image with black and white pixels arranged in a checkered pattern.

**Alternating channel:** This is a two colour image with pixels arranged in a checkered pattern. The colours are (0,255,0) and (255,0,255), causing each channel to alternate between 255 and 0 with each new pixel.

**Alternating bit:** This is a two colour image with pixels arranged in a checkered pattern. The colours are (85,85,85) and (170,170,170), causing each bit to toggle between '1' and '0' with each new pixel.

The black and the white images are used to provide a base line. In all of the other images half the bits contain the symbol '1' and the other half the symbol '0',
only the pattern is different between images. In other words the total Hamming weight of the images are the same but the Hamming distance travelled when reading all pixels differ. Each input image also produces different intermediate results. This should induce different switching patterns in the data paths of the system and thus affect the power consumption to some degree.

One potential error source exists in the last step of the algorithm, the sharpening. This step performs saturation of the final colour values and therefore the execution of some parts of code is dependent on the input data. This is however only a very small part of the total code.

Results

The results, depicted in Figure 4.8, show that the total energy consumption of the filtering operation is dependent on the input data. A difference of up to roughly 2% can be seen. The ordering of the images also seems plausible. The black image is the least energy consuming to process, which is understandable since it should be less energy consuming to drive a lot of 0’s than 1’s through the data paths of the system. One would think then that the white image would be the most energy consuming since it contains only logical 1’s, however as seen in the graph this is not the case. The higher energy consumption of some of the other images is due to the increased switching they cause in the data paths. The alternating bit image is particularly bad since it will cause bit lines to switch to the inverse of their neighbours. This will cause the capacitive load seen by the bit line driver to increase and thus consume more energy.

Figure 4.9 depicts the effects of different input data on the last stage of the filter. As seen in the figure, this stage is more sensitive to changes in input data,
4.2 Hardware measurements

![Energy consumption for the sharpening step using different input data. Energy is normalized with respect to the black image. Observe that the y-axis does not start at zero. More detailed results can be found in Table F.3.]

Figure 4.9

It is clear that input data can affect the power consumption even when the operations performed on the different data sets are identical. The differences are small however and trying to exploit them for energy gains by manual optimizations is probably not worthwhile.

4.2.3 Frequency scaling

Since the hardware used for testing did not have support for voltage scaling, only the frequency could be scaled. Due to the voltage being constant scaling cannot reduce energy consumption for the task. It is still interesting to do this test since it will tell how the different components are affected by frequency.

The frequency for the memory and other devices on the board is fed through the processor which means that if the processor’s frequency is lowered then that affects the other devices too. Scaling is controlled by three bits and allows seven...
settings ranging from $\frac{1}{2^0} = 1$ to $\frac{1}{2^6} = \frac{1}{64}$ of the original 180 MHz clock.

Results

As expected, and seen in Figure 4.10, lowering the frequency increases the energy consumption. This is due to static consumption, not only in the core but other devices like RAMs too. What is interesting in the figure is that the memory does not seem to be affected by frequency but rather only by the number of accesses apart from the static component.

Looking at power dissipation, seen in Figure 4.11, it is obvious that static power dissipation plays quite a big part in the total power dissipation. If this was not the case then the power dissipation of the core would follow the linear reference. Once again we can see that change in frequency only has a limited effect on the RAM’s power dissipation.

Execution time is acting just like it is supposed to. As can be seen in Figure 4.12, dividing the frequency with $n$ results in the execution time being multiplied with $n$.

4.2.4 Disabling interrupts

Debian Linux was preinstalled on the testing platform. This enabled a quick start since we did not have to worry about boot-sequences, how to communicate with the board, handle files and so on. The downside is that the operating system might introduce substantial overhead to the tests. Potential trouble areas include memory allocation, scheduling, and handling of peripheral devices. To examine some of the effects we decided to run a series of test with, and without, interrupts enabled. This should prevent the preemption of the test program and minimize
4.2 Hardware measurements

<table>
<thead>
<tr>
<th>Frequency (MHz)</th>
<th>Voltage (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>180 MHz</td>
<td>1.8 V</td>
</tr>
<tr>
<td>90 MHz</td>
<td>3.3 V</td>
</tr>
<tr>
<td>45 MHz</td>
<td>Total</td>
</tr>
<tr>
<td>23 MHz</td>
<td>Linear</td>
</tr>
<tr>
<td>11 MHz</td>
<td>1/2</td>
</tr>
<tr>
<td>5.6 MHz</td>
<td>1/4</td>
</tr>
<tr>
<td>2.8 MHz</td>
<td>1/8</td>
</tr>
<tr>
<td>1.8 V</td>
<td>1/16</td>
</tr>
<tr>
<td>3.3 V</td>
<td>1/32</td>
</tr>
<tr>
<td>Total</td>
<td>1/64</td>
</tr>
</tbody>
</table>

**Figure 4.11** The normalized power dissipation of the hardware test system when scaling the frequency.

**Figure 4.12** The normalized execution time of the hardware test system when scaling the frequency.
the effects of the operating system.

The image filter program was modified to take a command line argument for disabling interrupts. When the argument was specified, the interrupts was disabled just before signalling the start of the filtering task and enabled just after signalling the end of the task. Disabling interrupts from within the test application was done by memory mapping the /dev/mem file and writing the appropriate values into the addresses associated with the configuration registers of the Advanced Interrupt Controller [72].

The application was then compiled into five executables, each with a different optimization setting. Each of these executables was then run three times with interrupts enabled and three times with interrupts disabled.

**Results**

The impact of leaving the interrupts enabled during a test run is very small as can be seen in Figure 4.13. This seems to indicate that the overhead introduced by interrupt driven operating system events is very small for this type of application.

### 4.2.5 Comparing MPARM to hardware

To check if the simulator and hardware results have similar characteristics we plotted the results from each platform that were comparable in the same graph.

**Results**

In Figure 4.14 and 4.15 we can see that the optimization effects differ slightly between the simulator and the hardware platform. Still, on both platforms the
4.2 Hardware measurements

Figure 4.14  Comparison of simulated and hardware results for the Quicksort application with different compiler optimizations. Results are normalized to the O0 results on each platform. These values can be found in Table F.6.

Figure 4.15  Comparison of simulated and hardware results for the image filter with different compiler optimizations. Results are normalized to the O0 results on each platform. The values can be found in Table F.7.
close relation between energy and time is evident. It is also clear that the power dissipation of the hardware platform is less sensitive to optimization strategies than the simulator.

In Figure 4.16 we can see how the subtasks of the image filter behave on both simulator and hardware. The results match up reasonably well for the two platforms, there is perhaps a slight tendency for the power dissipation of the simulator to vary a bit more than that of the hardware.
Chapter 5

Discussion

In this chapter we will discuss some of the limitations of our work, the conclusions drawn from both the study of previous work and our experiments, and lastly indicate possible directions for future work.

5.1 Purpose and goals

We feel that the main contribution of this thesis is to provide a solid foundation for future studies of energy constrained software systems at Combitech. The thesis provides a comprehensive overview of the subject and a pre-study of parallelism for low energy consumption. We also present a low cost and very usable measurement methodology using existing lab equipment. This enables future work to focus more on software techniques and experiments instead of dealing with measurement issues.

The greatest concerns we have with the results are uncertainties regarding their real world applicability. Since the parallelism tests were only run on a simulated platform it is unclear if the results reflect the problems and possibilities faced on real platforms. Therefore we believe that further studies of the subject are needed to be able to draw any definitive conclusions about the opportunities afforded by exploiting parallelism and multiple cores.

5.2 Limitations and problems

The initial purpose of this thesis was to evaluate known techniques to help constructing low power software and describe how to best implement them on embedded systems. This study would then serve as the basis for creating a few basic guidelines intended to aid in the construction of low power software architectures and designs. A key point of this initial purpose was to test and evaluate low power techniques on a real target system. Unfortunately the intended target system proved unsuitable to test most techniques in a meaningful way. This was due to both software and hardware issues. Another limiting factor was the
lack of an established method for fine grained power and energy measurements using available lab equipment. This led to the conclusion that there would not be enough time to find or construct a new target system, develop a low-cost testing methodology and perform all necessary studies. This necessitated a shift in focus.

The decision was made to focus mainly on evaluating one possible low-power technique and to establish a suitable low cost testing methodology using existing equipment. The choice of low power technique to evaluate fell on how parallelism and emerging multi-core processors could be used in low power applications. It is a very interesting and hot topic in the embedded development community. A simulator environment (MPARM) had already been identified as a potential test platform. Using a simulator has a lot of advantages but also disadvantages. Therefore there was a desire to use a real multi-core or multi-processor platform as well. Unfortunately the availability of low power, low cost, multi-core or multi-processor development platforms is a bit scarce. The current offerings seem to focus mainly on extreme DSP capabilities and the prices of the development platforms were prohibitively expensive for this type of project. ARM has released a multi-core processor for embedded systems. However there are not that many systems available. ARM sells their own development board with the multi-core processor but in addition to the high price tag, this board does not provide a simple way of measuring energy consumption. Due to these limiting factors, we decided to only use the simulator for multi-core testing and leave the testing on physical platforms as possible future work.

A limitation introduced by the use of MPARM was the complexity of our programs used for testing. The server used for the tests achieved between 12,000 and 32,000 simulated instructions per second. When simulating 16 ARM cores, this results in a simulated execution speed of 750 to 2,000 instructions per second. This caused even the smallest test runs to take a lot of time.

In the beginning two tests were chosen to give a hint of what effects different methods had on energy consumption. It was then meant for a larger test of a more complete system to be run. Because of the simulator running at such a low simulation speed, advanced tests were not feasible. The total time used to run the simulations is more than two weeks resulting in a total simulated runtime of approximately 50 seconds. The server was based on a dual 1 GHz Pentium III configuration which is not a very modern configuration. The performance of a more up to date computer should be at least four times higher. Even so, the simulation speed would still be a limiting factor.

MPARM does give a good estimate of energy consumption but for our tests, where we have to run the same thing over and over with different settings, it really limited the complexity of our tests. On top of this, the data outputted from MPARM is not very simple for a computer to interpret. Solving the output issue was not that hard but the solution had to be adjusted a few times when more special cases appeared. Another problem with the large text files in our version of MPARM was that partial sums for tasks were calculated incorrectly. Again not that hard to compensate for but it took some time before we realized why our results did not make any sense.

There were a few limitations with the hardware tests as well. The oscilloscope
used to monitor the different power supplies was capable of storing 10,000 samples per channel. There was no way of streaming the sample data to a computer to enable a greater data set. This made longer tests less accurate. The high noise level did not help either. Of the nine bits each sample consisted of, the noise level consumed the lowest three. Using an AD620 partially solved the noise issues. The signal to noise ratio increased but at the cost of some bandwidth. It was also not possible to verify that the AD620 produced a correct output with the equipment available.

5.3 Our take on low power software issues

In this section we will discuss our view on low power software issues.

5.3.1 General optimizations

Efficiency lies at the heart of the energy optimization issue. An efficient solution is also often a fast solution. This is evident in our results in Chapters 3 and 4, where execution time and energy consumption are strongly related. This seems to indicate that optimizing for speed is a good start for optimizing energy efficiency. While faster configurations sometimes had slightly higher power dissipation it was not enough to result in larger total energy consumption. By having efficient high performance software we can either complete the work fast and sleep, so called race to idle strategy, or afford to use performance degrading energy saving techniques such as frequency and voltage scaling [75, 76]. What is most efficient is largely system and application specific.

Efficiency and lower energy consumption could also be obtained by optimizations of the algorithms or relaxed requirements on the quality of service. Making sure that the requirements do not demand unnecessary levels of accuracy and fidelity of the system could lead to substantial reductions in workload and energy consumption. The results in Subsection 3.2.3 demonstrate that this can have a tremendous impact on time and energy consumption of a workload.

Input data transformations to reduce switching activity are a less promising area of optimizations. The results in Subsection 4.2.2 demonstrates that changing the input data did have an effect on the energy consumption, though the differences were very small. In addition, it might be very hard to control what input data is received while maintaining the same functionality. It seems that switching reduction is more suitable for automated tools such as compilers than as an optimization strategy for the developer [77, 78].

When working with a feature rich hardware platform, the wealth of power saving possibilities can be a bit overwhelming. It is not necessarily required to modify the entire software stack to maximize the use of each of these. Simply introducing them where it can be done with low effort might be enough as a first step. Powering down all completely unused peripherals is a typical example.

It can be very hard to judge what improvements are possible beforehand from each optimization. Getting an overview of the effort required to implement a power saving feature or optimize a certain part for energy efficiency might also be
hard. To get the most out of the time and energy put into the optimizations and modification of the software, it is important to have a model of the system. This model can help to make sure that the optimizations are focused on the tasks and sub systems where they have most impact. It is a good idea to start with higher level optimizations first, otherwise prior low level work may be in vain when higher level changes are introduced.

5.3.2 Frequency scaling

It is important to make a distinction between frequency scaling with voltage scaling and frequency scaling alone. Scaling only the frequency of a device can be energy efficient when there is no work to be done, as demonstrated by the results from the Quicksort tests in Subsection 3.3.4. If there is useful work to do, scaling the frequency will not reduce the energy needed to complete that work. If there is significant static power dissipation in the system, the likely outcome is an increase in energy consumption. This is evident in the frequency scaling results presented in Subsection 4.2.3.

When voltage scaling enters into the equation it becomes possible to reduce energy consumption even while performing useful operations. Although the runtime is increased and thus the energy consumed by static dissipation, the reduction in dynamic power dissipation of the scaled device could be large enough to overcome this and lead to reduced energy consumption for the system.

Frequency scaling might bring unwanted side effects if the scaling is done using a clock source shared with other devices. A typical example is peripheral devices that derive their clock from the main system clock. If these devices are shared with other system components or communicate with remote systems, the devices will probably not function properly after the frequency change. This could possibly be solved by reconfiguring the device or the remote system at the cost of additional overhead and complexity. In the worst case scenario the device becomes unusable when operating at reduced frequency. We ran into issues like these during the hardware scaling experiments. The USART derived its bit rate from the same clock as the processor and thus we lowered the bit rate when lowered the processor frequency. This was an unexpected consequence and solving the issue by reconfiguring the USART would have introduced some extra overhead. This was only a minor problem for us but a good understanding of the clock dependencies in the system at an early stage of development might make it possible to avoid such issues.

5.3.3 Compilers

The compiler can have a large impact on energy consumption. The previous work in the field of energy efficient compilers mentioned in Section 2.7 shows impressive gains from a variety of techniques. Unfortunately this does not seem to have propagated to the compilers used in the embedded industry. It is however something to keep an eye on in the coming years. A good energy optimizing compiler could provide significant gains with very little extra effort for the developer.
The compiler used in this thesis (GCC) has no energy optimizing directives, the energy savings reported in our results stem from optimizations that reduce the execution time. From our tests in Subsections 3.2.2, 3.3.6, and 4.2.1, we find that using no optimizations, or O0, generally produces very inefficient executables. The results from using the other levels of optimization are a little more varied. Each setting produced the lowest energy consumption in at least one test. When looking at the lowest total energy consumption from all the results it was pretty much tied between O2 and O3. When compiling for the development board, O0 was slightly better relative its simulated results but still fell far behind. The potential gains in both energy and performance make it well worth the relatively modest effort of running a few simple benchmarks of the code with different compiler settings.

5.3.4 Operating systems

The choice of when and what operating system to use is important for many aspects of the system design and will of course impact the energy consumption as well. However the extent of this impact is harder to judge. For applications where the scheduling and other OS functions are used heavily, the overhead could be quite severe [47]. On the other hand, as our tests in Subsection 4.2.4 showed, on an otherwise lightly loaded Linux system, the execution of a computationally intensive application seem to be relatively unaffected by at least the scheduling component of the operating system. There are, of course, other possible overheads not associated with scheduling, such as memory management.

A perhaps more important consideration is if the operating system somehow interferes with the use of power saving hardware features. Depending on several factors, such as memory protection and driver availability, the use of specific hardware energy features might be impossible or may require development of a special purpose driver or operating system API. For our tests on the hardware platform in Chapter 4, we had to circumvent the memory protection in order to gain access to hardware components such as interrupt controller, output pins, frequency control registers, etc. In our case this was very simple due to the /dev/mem file mapping the entire physical address space but other platforms may not have such a simple workaround. So while an operating system is usually a benefit to ease and speed of development, it can be detrimental to software that needs to tightly integrate with the hardware platform. Ensuring that the operating system of choice has proper level of support for the target platform, and to what extent it is possible to improve, is crucial.

5.3.5 Parallelism

Parallelism offers some exciting possibilities for energy saving. The use of frequency scaling and parallel algorithms presents an interesting way to save power. As demonstrated in our simulation results for the image filter in Subsection 3.2.5, it is clearly possible to reduce the total energy consumption of the processor cores and the entire system, given a suitable workload. However the tests also indicate that the specific architecture of the multi-core or multi-processor system is very
important, and might limit the energy gains. It is crucial to consider the static power consumption in the system and how this affects energy and time trade-offs. This is consistent with the analytical and simulation findings by Li and Martínez [55].

Another very promising aspect of multi-core technology is that it can be used to create a very dynamic hardware platform. If the power supply and clock of each core can be controlled independently then it could be possible to have both a very low power operating point and also a very computationally powerful operating point. This would be harder to achieve in a single core system, especially if sporadic interrupts or other requests makes it uneconomical or impossible to completely power off the main processor.

In the last example it might not be necessary to have an advanced multi-core system-on-chip. Simply adding a very low power micro-controller or processor might be even more effective. This low power unit could then perform basic house-keeping and request monitoring while the computational power of the main processor is not needed, prolonging the sleep periods of the power hungry processor and saving energy at the cost of more complex software and hardware. This is the direction taken by McIntire et al. [79] and Worth et al. [28] when constructing low power sensor network nodes.

5.3.6 Hardware design

Choosing a hardware platform for an energy constrained embedded system is a very difficult and complex task. Besides the actual functionality requirements there are issues with cost, manufacturing, reliability, standard compliance, etc. Because of this complexity it is possible that energy issues will take a back seat during the design. This could lead to unfortunate dependencies in the system, unusable power saving features due to external circuit requirements, and other issues that could hamper later efforts to optimize the energy consumption of the system.

It is useful to be able to treat each component as isolated from the system in such a way that its power management does not affect the rest of the system. Published examples of low power system design get successful results by decoupling functionality so that a graceful reduction in functionality and power dissipation can be performed [28]. Therefore, an analysis of requirements and effects that the power saving features incorporated in different components place on the surrounding system is important in order to avoid unnecessary limitations in the platform. Investigating what functionality and services the hardware can provide when different power saving techniques are in effect can also be beneficial. In some cases it might be possible to offer a much higher level of service in certain power modes by making only small modifications to the system.

5.3.7 Activity patterns

Controlling the timing of resource usage is an important factor in energy conscious software. The goal is to produce software with favourable activity patterns. There
are two main strategies to improve the activity patterns of the system. One is to consider each recurring activity in isolation and try to make them occur less frequently, either by cutting down the total work or by processing it in bigger batches. This is favourable because there are usually overheads involved with the invocation of energy conserving features. If activation patterns can be changed from frequent short activations to longer and less frequent activations, savings on overhead could be achieved and possibly also enabling more aggressive forms of power management.

The other strategy is to try to group different activities together so that they happen right after each other instead of occurring randomly over some interval. This is especially important on a per device basis. If all activity in a specific device can be grouped into well defined phases, it becomes much easier to do energy management for that device. For instance if all accesses to a hard disk drive are performed at a certain part of the program, it might be possible to save energy by spinning down the disk in between accesses. The same applies to processors and other system components.

One example of this is a study by Anand et al. [80] where the authors create an energy aware content cache on a hard disk drive. The cache optimizes the access patterns to the drive and the original content source, a wireless network interface, by moving accesses from one unit to the other depending on what is most favourable given the current power saving state of the two units. Activity pattern optimization has also been proposed as a compiler technique [40].

Activity patterns have also been a focus of recent Linux kernel development. Examples include the removal of kernel ticks during idle periods. Ticks are regular occurring timer interrupts used by the kernel to, among other things, define scheduling time slices. When the system is idle there is really no need to schedule work and ticks are detrimental since they constrain the maximum idle period of the CPU to the period of the tick. Other improvements include deferrable timers, timers that only fires if there is also other work that has to be done, and rounding event times to nearest second in order to processes more events at the same time. See LessWatts.org [81] and the paper on tickless idle by Siddha et al. [82] for a brief introduction to these techniques.

Taking advantage of the possibilities for activity pattern improvement can be tricky, especially in systems with many different concurrent tasks. One technique that may make this a bit easier is to enable buffering at strategic places in the system. If hardware or software components can buffer sufficiently large amounts of data or requests, it will become easier to choose when to process the input and to do it in larger quantities each time. This will probably introduce additional latency which has to be taken into consideration.

### 5.3.8 Energy analysis

Probably the most important factor in designing or modifying a system for low power is to know, or anticipate, where energy will be consumed in the system, and what activity causes this consumption. In Section 2.3 we present some techniques to estimate the power consumption of actual software. Unfortunately most of the
techniques are quite complex and might not be very suitable during initial system
design. One alternative is to use much simpler models during the initial design
exploration [83]. A very simple and straightforward method is to use a simple
spreadsheet [83, 84], enter a power estimation for each system component and
how often the component is active on average and simply sum up the average
power. Depending on hardware availability, the power data could be directly
measured on the intended component or roughly estimated from data sheets of
candidate components. The activity of each component can be roughly estimated
by the designer based on the intended behaviour of the system and its software.
This will of course only produce a very crude estimate but it can still serve as
an indication of how different implementation decisions could affect the energy
consumption of the system. A more advanced method proposed by Benini et al.
[85] is to model each component as a state machine with specific power dissipation
for each state and time costs for transitions between states. Each transition is
associated with some stimuli to the component and thus it is required to model
the abstract behaviour of each component to some extent. An abstract behaviour
model of the software could then drive this power state machine by providing
stimuli. This allows for more detailed analysis of energy consequences of selecting
different software algorithms and architectures.

We believe that early estimations of system energy consumption are important
in order to focus the optimization effort towards the area where it will be most
beneficial. Even if complex modelling is not feasible due to resource constraints,
designers should at least identify most problematic areas of the system from an
energy consumption point of view.

5.4 Conclusions

Energy conscious software is very dependent on what hardware energy features,
such as frequency scaling and power management, are available. The exception to
this is software with a lot of overhead. For this kind of software it is possible to
save energy, or rather waste less energy, by switching to another design pattern or
just optimizing it. If the hardware provides software-controllable energy features,
the energy consumption can be lowered dramatically.

It seems that one of the main concerns is to provide as many different operating
points as possible, where each point represents a specific trade-off in terms of
features, performance and energy. This is an important consideration for both
the hardware and software design since both need to be designed in a way that
does not require features or devices to be available when they are not needed. In
other words both hardware and software should be able to provide different levels
of service at different energy cost and try to maximize this dynamic range. This
freedom should make it possible for each application and task in the system to
handle their energy consumption in an intelligent manner. The greatest challenge
in this is probably to make the trade-off between system complexity and flexibility.

Using parallelism and multi-core technologies seems very promising for pro-
ducing low power software. Our experiments demonstrate that it is possible to get
significant reductions in energy consumption if the workload can be parallelized efficiently. Realizing this potential requires a very flexible hardware platform. Most important is to have fine grained control over power management, and voltage and frequency scaling, preferably on a per core basis.

A very important part of developing low power software is being able to reliably measure the energy consumption. In this thesis we have demonstrated that it is possible to get a measurement technique that offers high resolution and repeatable results, using only standard lab equipment. Such a technique can offer great insights into the energy characteristics of software.

5.5 Future work

We believe that future work should focus on applying the information gathered in this thesis on real systems. The programs used to test the methods were very simple. A more extensive test on a larger system is needed. This should bring results on what works in practice and the difficulties faced when dealing with more complex software. This should also give insight into implementation issues and how to actually construct the mechanisms needed for the energy saving techniques. It is also unclear to what extent power saving features are integrated and supported on real world hardware.

Another avenue for further studies is to take the multi-core and multi-processor energy saving techniques and test them on different architectures, preferably real hardware targeted at the embedded market. Heterogeneous and asymmetrical multi-core solutions seem especially interesting. We believe that this is a very promising area for applications which require both high performance and low energy consumption.

To make the exploration of new energy saving software techniques easier, it might also be useful to put together a hardware and software platform so that new ideas could be tested in a more controlled environment.
Glossary

AD620  Instrumentation amplifier manufactured by Analog Devices. 52, 53, 67

CPUFreq  Linux kernel subsystem for scaling clock frequency on the fly. 15

Hamming distance  The number of symbol substitutions needed to make two strings of equal length identical. 58

Hamming weight  The number of symbols in a string that differ from the zero symbol of the alphabet. In the binary case it is the number of '1's in a string of '1's and '0's. 58

L9260  A development board from Olimex. Used in this thesis for hardware tests. The full name is SAM9-L9260. 51, 56

MPARM  A cycle-accurate multi-processor system-on-chip architectural simulator. 8, 21, 66

x86  Instruction set invented by Intel. The most common architecture used in personal computers today. 15
Acronyms

ACPI  Advanced Configuration and Power Interface. 15
API   Application Programming Interface. 69
APM  Advanced Power Management. 15
ARM  Advanced RISC Machine. 12, 15, 51, 66
BIOS Basic Input Output System. 15
CMP  Chip Multi-Processor. 17
CPU  Central Processing Unit. 17, 71
DVS  Dynamic Voltage Scaling. 16
IJG  Independent JPEG Group. 19
ILP  Instruction Level Parallelism. 17
IPC  Instructions Per Cycle. 14
ISA  Instruction Set Architecture. 18
JPEG Joint Photographic Experts Group. 18
OS   Operating System. 13, 69
PM   Power Management. 18
SIMD Single Instruction, Multiple Data. 17
SMT Simultaneous MultiThreading. 17
SWARM SoftWare ARM. 21
USART Universal Synchronous Asynchronous Receiver Transmitter. 68
VLIW Very Long Instruction Word. 17
Bibliography


Appendix A

Image filter detailed results
<table>
<thead>
<tr>
<th>MPARM</th>
<th>Normalized energy</th>
<th>Normalized time</th>
<th>Normalized power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 core</td>
<td>O0 1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>O1 0.20</td>
<td>0.22</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td>O2 0.22</td>
<td>0.24</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>O3 0.22</td>
<td>0.24</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>Os 0.22</td>
<td>0.24</td>
<td>0.93</td>
</tr>
<tr>
<td>2 cores</td>
<td>O0 1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>O1 0.19</td>
<td>0.22</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>O2 0.21</td>
<td>0.23</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>O3 0.21</td>
<td>0.23</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>Os 0.22</td>
<td>0.24</td>
<td>0.91</td>
</tr>
<tr>
<td>4 cores</td>
<td>O0 1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>O1 0.19</td>
<td>0.22</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>O2 0.21</td>
<td>0.23</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>O3 0.21</td>
<td>0.23</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>Os 0.22</td>
<td>0.24</td>
<td>0.91</td>
</tr>
<tr>
<td>8 cores</td>
<td>O0 1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>O1 0.20</td>
<td>0.22</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>O2 0.22</td>
<td>0.24</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>O3 0.22</td>
<td>0.24</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>Os 0.22</td>
<td>0.24</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table A.1 Comparison of energy, time, and power for the image filter using different optimization settings. Results are normalized to the O0 setting.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Normalized energy</th>
<th>Normalized time</th>
<th>Normalized power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 core</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1/2 resolution</td>
<td>0.22</td>
<td>0.20</td>
<td>1.07</td>
</tr>
<tr>
<td>1/4 resolution</td>
<td>0.10</td>
<td>0.09</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Table A.2 Comparison between different quality of service settings. Results are normalized to the standard setting.

<table>
<thead>
<tr>
<th>Image size</th>
<th>Linear</th>
<th>50x50</th>
<th>100x100</th>
<th>150x150</th>
<th>200x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 core</td>
<td>1.00</td>
<td>2.00</td>
<td>4.00</td>
<td>8.00</td>
<td>16.00</td>
</tr>
<tr>
<td>2 cores</td>
<td>0.97</td>
<td>1.91</td>
<td>3.45</td>
<td>6.06</td>
<td>11.08</td>
</tr>
<tr>
<td>4 cores</td>
<td>0.97</td>
<td>1.93</td>
<td>3.73</td>
<td>6.94</td>
<td>12.19</td>
</tr>
<tr>
<td>8 cores</td>
<td>0.97</td>
<td>1.94</td>
<td>3.73</td>
<td>7.26</td>
<td>12.82</td>
</tr>
<tr>
<td>16 cores</td>
<td>0.97</td>
<td>1.94</td>
<td>3.82</td>
<td>7.49</td>
<td>12.82</td>
</tr>
</tbody>
</table>

Table A.3 The speedup of the parallel image filter compared to the sequential implementation, using different images sizes.
<table>
<thead>
<tr>
<th></th>
<th>gauss-x</th>
<th>gauss-y</th>
<th>sharpening</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 core ref</strong></td>
<td>27.60</td>
<td>31.95</td>
<td>3.28</td>
<td>0.00</td>
</tr>
<tr>
<td>2 cores</td>
<td>27.67</td>
<td>32.01</td>
<td>3.69</td>
<td>1.30</td>
</tr>
<tr>
<td>4 cores</td>
<td>27.67</td>
<td>32.02</td>
<td>4.32</td>
<td>1.82</td>
</tr>
<tr>
<td>8 cores</td>
<td>27.68</td>
<td>32.07</td>
<td>4.44</td>
<td>2.90</td>
</tr>
<tr>
<td><strong>16 cores</strong></td>
<td>27.69</td>
<td>32.18</td>
<td>8.92</td>
<td>9.61</td>
</tr>
</tbody>
</table>

**Table A.4** The total core cycles consumed by the subtasks of image filter using different number of cores. Results are in millions of cycles.

<table>
<thead>
<tr>
<th></th>
<th>200 MHz</th>
<th>100 MHz</th>
<th>67 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 core ref</strong></td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 cores</td>
<td>0.97</td>
<td>0.87</td>
<td>0.95</td>
</tr>
<tr>
<td>4 cores</td>
<td>0.96</td>
<td>0.83</td>
<td>0.89</td>
</tr>
<tr>
<td>8 cores</td>
<td>0.95</td>
<td>0.81</td>
<td>0.85</td>
</tr>
<tr>
<td><strong>16 cores</strong></td>
<td>1.01</td>
<td>0.83</td>
<td>0.87</td>
</tr>
</tbody>
</table>

**Table A.5** Comparison of energy consumption for the image filter using different number of cores. Results are normalized with regard to the single core reference.

<table>
<thead>
<tr>
<th></th>
<th>200 MHz</th>
<th>100 MHz</th>
<th>67 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 core ref</strong></td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 cores</td>
<td>0.51</td>
<td>1.01</td>
<td>1.51</td>
</tr>
<tr>
<td>4 cores</td>
<td>0.26</td>
<td>0.51</td>
<td>0.77</td>
</tr>
<tr>
<td>8 cores</td>
<td>0.13</td>
<td>0.26</td>
<td>0.39</td>
</tr>
<tr>
<td><strong>16 cores</strong></td>
<td>0.08</td>
<td>0.14</td>
<td>0.21</td>
</tr>
</tbody>
</table>

**Table A.6** Comparison of execution time for the image filter using different number of cores. Results are normalized with regard to the single core reference.

<table>
<thead>
<tr>
<th></th>
<th>200 MHz</th>
<th>100 MHz</th>
<th>67 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 core ref</strong></td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 cores</td>
<td>1.89</td>
<td>0.86</td>
<td>0.63</td>
</tr>
<tr>
<td>4 cores</td>
<td>3.67</td>
<td>1.62</td>
<td>1.16</td>
</tr>
<tr>
<td>8 cores</td>
<td>7.15</td>
<td>3.11</td>
<td>2.20</td>
</tr>
<tr>
<td><strong>16 cores</strong></td>
<td>12.95</td>
<td>5.92</td>
<td>4.22</td>
</tr>
</tbody>
</table>

**Table A.7** Comparison of power dissipation for the image filter using different number of cores. Results are normalized with regard to the single core reference.
Table A.8  Breakdown of energy consumption for each subsystem when executing the image filter code. Results are in $\mu$J.

<table>
<thead>
<tr>
<th></th>
<th>1 core ref</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200 MHz</td>
<td>200 MHz</td>
<td>100 MHz</td>
<td>67 MHz</td>
<td>50 MHz</td>
<td>25 MHz</td>
</tr>
<tr>
<td>Core</td>
<td>721</td>
<td>815</td>
<td>273</td>
<td>240</td>
<td>240</td>
<td>239</td>
</tr>
<tr>
<td>Dcache</td>
<td>285</td>
<td>289</td>
<td>289</td>
<td>289</td>
<td>289</td>
<td>289</td>
</tr>
<tr>
<td>Icache</td>
<td>1138</td>
<td>1210</td>
<td>1218</td>
<td>1220</td>
<td>1223</td>
<td>1226</td>
</tr>
<tr>
<td>RAM</td>
<td>249</td>
<td>80</td>
<td>112</td>
<td>143</td>
<td>174</td>
<td>297</td>
</tr>
</tbody>
</table>
Appendix B

Quicksort detailed results
<table>
<thead>
<tr>
<th>Tasks</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>1.73</td>
<td>1.73</td>
<td>1.34</td>
<td>1.27</td>
<td>1.21</td>
<td>1.21</td>
<td><strong>1.14</strong></td>
<td>1.14</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>2.01</td>
<td>2.01</td>
<td>1.60</td>
<td>1.53</td>
<td>1.47</td>
<td>1.47</td>
<td><strong>1.40</strong></td>
<td>1.40</td>
<td>1.44</td>
</tr>
<tr>
<td>4 cores</td>
<td>Time</td>
<td>1.74</td>
<td>1.21</td>
<td>0.89</td>
<td>1.02</td>
<td>0.78</td>
<td>0.78</td>
<td>0.78</td>
<td><strong>0.69</strong></td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>3.66</td>
<td>2.61</td>
<td>1.98</td>
<td>2.23</td>
<td>1.76</td>
<td>1.92</td>
<td>1.77</td>
<td>1.66</td>
<td>1.78</td>
</tr>
<tr>
<td>8 cores</td>
<td>Time</td>
<td>1.75</td>
<td>1.22</td>
<td>0.90</td>
<td>0.76</td>
<td>0.67</td>
<td><strong>0.61</strong></td>
<td>0.65</td>
<td>0.66</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>6.73</td>
<td>4.74</td>
<td>3.55</td>
<td>3.04</td>
<td>2.71</td>
<td><strong>2.48</strong></td>
<td>2.63</td>
<td>2.68</td>
<td>2.61</td>
</tr>
<tr>
<td>16 cores</td>
<td>Time</td>
<td>1.80</td>
<td>1.26</td>
<td>0.93</td>
<td>0.79</td>
<td>0.69</td>
<td>0.63</td>
<td><strong>0.59</strong></td>
<td>0.64</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>12.02</td>
<td>8.59</td>
<td>6.52</td>
<td>5.61</td>
<td>4.94</td>
<td>4.53</td>
<td><strong>4.25</strong></td>
<td>4.61</td>
<td>4.85</td>
</tr>
</tbody>
</table>

Table B.1  Normalized energy and time for the Quicksort algorithm when varying how many tasks the problem is divided into. Tests with less tasks than cores are not included.
<table>
<thead>
<tr>
<th>2 cores</th>
<th>Time</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.68</td>
<td>0.96</td>
</tr>
<tr>
<td>4 cores</td>
<td>0.82</td>
<td>1.05</td>
</tr>
<tr>
<td>8 cores</td>
<td>0.86</td>
<td>1.03</td>
</tr>
<tr>
<td>16 cores</td>
<td>0.94</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Table B.2  Normalized time and energy for the Quicksort algorithm with private memory. The values are normalized with regard to each corresponding shared memory counterpart.

<table>
<thead>
<tr>
<th>Idle divider</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.06</td>
<td>1.01</td>
<td>1.03</td>
<td>1.06</td>
<td>1.11</td>
</tr>
<tr>
<td>Energy</td>
<td>0.94</td>
<td>0.92</td>
<td>0.91</td>
<td>0.92</td>
<td>0.89</td>
<td>0.89</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>4 cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>0.98</td>
<td>0.98</td>
<td>0.98</td>
<td>1.17</td>
<td>1.12</td>
<td>1.15</td>
<td>1.22</td>
<td>1.35</td>
</tr>
<tr>
<td>Energy</td>
<td>0.84</td>
<td>0.81</td>
<td>0.79</td>
<td>0.80</td>
<td>0.77</td>
<td>0.76</td>
<td>0.77</td>
<td>0.77</td>
</tr>
<tr>
<td>8 cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>0.98</td>
<td>0.98</td>
<td>1.08</td>
<td>1.12</td>
<td><strong>1.01</strong></td>
<td>1.18</td>
<td>1.21</td>
<td>1.54</td>
</tr>
<tr>
<td>Energy</td>
<td>0.69</td>
<td>0.64</td>
<td>0.61</td>
<td>0.57</td>
<td><strong>0.52</strong></td>
<td>0.53</td>
<td>0.52</td>
<td>0.54</td>
</tr>
<tr>
<td>16 cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>1.11</td>
<td><strong>1.03</strong></td>
<td>1.23</td>
<td>1.45</td>
<td>1.83</td>
</tr>
<tr>
<td>Energy</td>
<td>0.60</td>
<td>0.52</td>
<td>0.44</td>
<td>0.41</td>
<td><strong>0.35</strong></td>
<td>0.35</td>
<td>0.36</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Table B.3  Normalized time and energy for the Quicksort algorithm when using different frequency dividers for idle relative to same number of cores but full speed.

<table>
<thead>
<tr>
<th>Parallel divider</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 cores</td>
<td>Time</td>
<td>1.53</td>
<td>2.05</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td><strong>0.95</strong></td>
<td>1.07</td>
</tr>
<tr>
<td>4 cores</td>
<td>Time</td>
<td>1.75</td>
<td>2.32</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td><strong>1.23</strong></td>
<td>1.48</td>
</tr>
<tr>
<td>8 cores</td>
<td>Time</td>
<td>1.47</td>
<td>1.93</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td><strong>1.17</strong></td>
<td>1.43</td>
</tr>
<tr>
<td>16 cores</td>
<td>Time</td>
<td>1.48</td>
<td>2.12</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td><strong>1.32</strong></td>
<td>1.85</td>
</tr>
</tbody>
</table>

Table B.4  Normalized time and energy for the Quicksort algorithm when using different frequency dividers only for the parallel part relative to same number of cores but full speed.
Table B.5 Normalized time and energy for the Quicksort algorithm when using different frequency dividers only for the sequential part relative to same number of cores but full speed.

<table>
<thead>
<tr>
<th>Parallel divider</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential divider</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2 cores</td>
<td>1.49</td>
<td><strong>1.62</strong></td>
<td>2.08</td>
</tr>
<tr>
<td>Energy</td>
<td>0.76</td>
<td><strong>0.75</strong></td>
<td>0.83</td>
</tr>
<tr>
<td>4 cores</td>
<td>1.81</td>
<td><strong>1.91</strong></td>
<td>2.38</td>
</tr>
<tr>
<td>Energy</td>
<td>0.67</td>
<td><strong>0.66</strong></td>
<td>0.72</td>
</tr>
<tr>
<td>8 cores</td>
<td>1.55</td>
<td><strong>1.66</strong></td>
<td>2.09</td>
</tr>
<tr>
<td>Energy</td>
<td>0.45</td>
<td><strong>0.45</strong></td>
<td>0.50</td>
</tr>
<tr>
<td>16 cores</td>
<td>1.59</td>
<td><strong>1.59</strong></td>
<td>2.04</td>
</tr>
<tr>
<td>Energy</td>
<td>0.33</td>
<td><strong>0.36</strong></td>
<td>0.37</td>
</tr>
</tbody>
</table>

Table B.6 Normalized time and energy for the Quicksort algorithm when using different frequency dividers for the different parts relative to same number of cores but full speed. The frequency divider for idle is set to 50.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>200 MHz</th>
<th>100 MHz</th>
<th>67 MHz</th>
<th>50 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>1.00</td>
<td>0.12</td>
<td>0.10</td>
<td>0.11</td>
</tr>
<tr>
<td>Icache</td>
<td>1.00</td>
<td>0.46</td>
<td>0.48</td>
<td>0.49</td>
</tr>
<tr>
<td>Dcache</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>RAM</td>
<td>1.00</td>
<td>1.45</td>
<td>2.06</td>
<td>2.61</td>
</tr>
<tr>
<td>Total energy</td>
<td>1.00</td>
<td>0.45</td>
<td>0.53</td>
<td>0.62</td>
</tr>
<tr>
<td>Time</td>
<td>1.00</td>
<td>1.66</td>
<td>2.50</td>
<td>3.25</td>
</tr>
</tbody>
</table>

Table B.7 Normalized time and energy values for the Quicksort algorithm when using different frequencies with eight cores. The values show that a lower frequency is not guaranteed to result in lower energy consumption.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Bubblesort</th>
<th>Shakersort</th>
<th>Selectionsort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size of dataset</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>1 core</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>0.93</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>Energy</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
</tr>
<tr>
<td><strong>2 cores</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td><strong>0.90</strong></td>
<td>0.91</td>
<td>0.98</td>
</tr>
<tr>
<td>Energy</td>
<td><strong>0.90</strong></td>
<td>0.92</td>
<td>0.98</td>
</tr>
<tr>
<td><strong>8 cores</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>0.96</td>
<td>0.96</td>
<td>1.15</td>
</tr>
<tr>
<td>Energy</td>
<td>0.96</td>
<td>0.96</td>
<td>1.12</td>
</tr>
<tr>
<td><strong>16 cores</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>1.09</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>Energy</td>
<td>1.09</td>
<td>0.98</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table B.8 Normalized time and energy for the Quicksort algorithm when using different algorithms when the dataset is smaller than the specified number.

<table>
<thead>
<tr>
<th>Test</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.11</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.05</td>
<td>1.00</td>
<td>1.02</td>
</tr>
<tr>
<td>3</td>
<td>1.16</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>1.13</td>
<td>1.01</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>1.16</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>1.15</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>7</td>
<td>1.09</td>
<td>1.00</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table B.9 Normalized time and energy for the Quicksort algorithm when using different compiler optimizations. Values are normalized with regard to the lowest result for each test.

<table>
<thead>
<tr>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1.00</td>
<td>0.44</td>
<td>0.40</td>
<td>0.40</td>
</tr>
<tr>
<td>Energy</td>
<td>1.00</td>
<td>0.44</td>
<td>0.39</td>
<td>0.39</td>
</tr>
</tbody>
</table>

Table B.10 Normalized time and energy for the Quicksort algorithm when using different algorithms when the dataset is smaller than the specified number.
Quicksort detailed results

<table>
<thead>
<tr>
<th></th>
<th>1 core</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
<th>16 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core</td>
<td>1.00</td>
<td>0.56</td>
<td>0.45</td>
<td>0.56</td>
<td>0.49</td>
</tr>
<tr>
<td>ICache</td>
<td>1.00</td>
<td>1.03</td>
<td>1.21</td>
<td>1.03</td>
<td>1.29</td>
</tr>
<tr>
<td>DCache</td>
<td>1.00</td>
<td>0.65</td>
<td>1.34</td>
<td>0.58</td>
<td>1.27</td>
</tr>
<tr>
<td>RAM</td>
<td>1.00</td>
<td>2.87</td>
<td>2.28</td>
<td>3.05</td>
<td>2.53</td>
</tr>
<tr>
<td>Total energy</td>
<td>1.00</td>
<td>1.17</td>
<td>1.18</td>
<td>1.15</td>
<td>1.23</td>
</tr>
<tr>
<td>Time</td>
<td>1.00</td>
<td>2.02</td>
<td>1.56</td>
<td>1.35</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Table B.11 Normalized time and energy values for the Quicksort algorithm when using the optimal settings from each test in relation to the best single-core result. 
*Sh* signifies shared memory while *Pr* signifies private memory.

<table>
<thead>
<tr>
<th></th>
<th>Reference</th>
<th>With optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>0.26</td>
<td>0.23</td>
</tr>
<tr>
<td>DCache</td>
<td>0.10</td>
<td>0.08</td>
</tr>
<tr>
<td>Icache</td>
<td>0.45</td>
<td>0.43</td>
</tr>
<tr>
<td>RAM</td>
<td>0.12</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Table B.12 Normalized energy distribution for Quicksort compared to reference total.

<table>
<thead>
<tr>
<th></th>
<th>Reference</th>
<th>With optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy</td>
<td>1.12</td>
<td>1.00</td>
</tr>
<tr>
<td>Time</td>
<td>1.14</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table B.13 Normalized time and energy for the Quicksort algorithm when using the optimal settings from each test and compensating for the RAMs added by MPARM in relation to the best single-core result.
Appendix C

Quicksort queue length

In this appendix we discuss the effects different queue lengths have on energy consumption and execution time.

To test different queue lengths we simply replaced the buffer from the previous implementation with a queue of a configurable length. A few other changes had to be made for the queue to be utilized properly. The pivot was changed from selecting a random element to using the median of five random elements. Another thing that got changed was the strategy for how to divide the problem. Originally the problem was divided into a certain number of parts, now the problem is divided if the number of elements in the current subproblem is larger than a certain number. The largest number of partitions the data can be divided in is 100 and, since each division results in continued work on at least one of the two partitions, the largest number of partitions that can be placed in the queue is 50. This modified implementation was then tested on eight cores with queue lengths of 1 through 7 as well as 10 and 50.

Increasing the queue length is not expected to have a large impact on energy consumption since the energy consumption from cores being idle is responsible for about 8% of the total energy consumption, seen in Figure C.1. There is a large performance increase possible though. The idle time, also seen in Figure C.1, is responsible for 46% of the total CPU time.

C.1 Results

The results confirmed what we had suspected. As can be seen in Figure C.2, the only task that benefit from a longer queue is the idle time. This is the sum of all time slots where a core is waiting for data to sort. Since the same data has to be sorted no matter the length of the queue, the total active time is pretty much the same. It varies because of overhead being introduced when placing tasks in the queue.

When looking at the energy consumption, seen in Figure C.3, it is obvious how little the idle periods use. The total energy consumption remains about the same no matter how the queue length varies. The execution time, however, varies up to
Figure C.1  The relation between the active and idle parts of the execution of Quicksort. The queue length is one and it is running on eight cores.

Figure C.2  Normalized energy consumption for the active and idle parts of Quicksort compared to using a queue of length one.
C.2 Conclusions

Our experiments showed that a queue length of one limits the performance with almost 10%. Compared to our original buffer implementation the improvement was 25%, which is a very significant increase in performance. A part of this increase is probably a result of a better selection of pivot and partitioning strategy. However, since our goal was to lower the energy consumption rather than increasing performance, using a long queue does not give a noticeable advantage. It would be more effective to optimize the active part since it is responsible for more than 90% of the energy consumption.

With a more effective algorithm for selecting the pivot element it would be about 10%, also seen in Figure C.3. This is one of those times where performance and energy consumption are not tightly coupled.

In Figure C.4 we see a comparison between the implementation with a queue, the one core reference and the implementation with a buffer. The execution time of the implementation with a queue of length five is a clear winner when it comes to performance. This becomes even clearer when looking at the speedup in Figure C.5. The speedup is calculated relative the reference using shared memory since the multi-core implementations are more or less forced to use shared memory. The speedup of the queue implementation of length five is 4.1 which seems reasonable since Eriksson et al. [86] managed to achieve a speedup of about five with a queue and four times the elements. When comparing to the reference using private memory the speedup reaches only 2.1. This is, again, due to how the caches work in MPARM. Therefore, comparing with the shared memory reference seems more fair.

![Normalized execution time and energy consumption for Quicksort using different queue lengths.](image)

**Figure C.3** Normalized execution time and energy consumption for Quicksort using different queue lengths.
Figure C.4  Normalized energy and time from the reference, the queue of length one and five, and the buffer. All values are relative the reference with shared memory.

Figure C.5  Speedup relative the reference using shared memory.
possible to get even better performance as the queue would be better utilized.
Appendix D

Quicksort reference comparisons
Figure D.1  Normalized results for using private memory for the Quicksort implementation relative to the one core reference.

Figure D.2  Normalized results from idle frequency-scaling on Quicksort compared to the one core reference.
Figure D.3  Normalized results from parallel frequency-scaling on Quicksort compared to the one core reference.

Figure D.4  Normalized results from sequential frequency-scaling on Quicksort compared to the one core reference.
Figure D.5 Normalized results from the combined scaling on Quicksort compared to the one core reference. The idle frequency divider was set to 50.

Figure D.6 Normalized results for using other algorithm with Quicksort to finish of the sorting compared to the one core reference.
Appendix E

AD620 fullsize plots
Figure E.1  Comparison between using an AD620 and the integrated diff-functionality. Total time is 1 ms.
Figure E.2  Comparison between using an AD620 and the running average of the integrated diff-functionality. Total time is 1 ms.
Figure E.3 Comparison between using an AD620 and the running average of the integrated diff-functionality. Total time is 0.2 ms.
Figure E.4  Comparison between using an AD620 and the running average of the integrated diff-functionality. Total time is 0.2 ms.
Figure E.5  Comparison between using an AD620 and the running average of the integrated diff-functionality. Total time is 0.4 s.
**Figure E.6** Comparison between using an AD620 and the running average of the integrated diff-functionality. Total time is 0.4 s.
Appendix F

Hardware measurements
detailed results

<table>
<thead>
<tr>
<th></th>
<th>Normalized energy</th>
<th>Normalized time</th>
<th>Normalized power</th>
</tr>
</thead>
<tbody>
<tr>
<td>L9260</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>O1</td>
<td>0.46</td>
<td>0.48</td>
<td>0.95</td>
</tr>
<tr>
<td>O2</td>
<td>0.44</td>
<td>0.46</td>
<td>0.97</td>
</tr>
<tr>
<td>O3</td>
<td>0.44</td>
<td>0.46</td>
<td>0.96</td>
</tr>
<tr>
<td>Os</td>
<td>0.43</td>
<td>0.45</td>
<td>0.96</td>
</tr>
</tbody>
</table>

Table F.1  Quicksort results using different compiler settings on hardware.
<table>
<thead>
<tr>
<th></th>
<th>Normalized energy</th>
<th>Normalized time</th>
<th>Normalized power</th>
</tr>
</thead>
<tbody>
<tr>
<td>L9260</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>O0</td>
<td>0.29</td>
<td>0.29</td>
<td>1.00</td>
</tr>
<tr>
<td>O1</td>
<td>0.25</td>
<td>0.24</td>
<td>1.01</td>
</tr>
<tr>
<td>O2</td>
<td>0.24</td>
<td>0.24</td>
<td>1.01</td>
</tr>
<tr>
<td>O3</td>
<td>0.27</td>
<td>0.26</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table F.2  Image filter results using different compiler settings on hardware.
<table>
<thead>
<tr>
<th></th>
<th>Black</th>
<th>White</th>
<th>Stripes x</th>
<th>Stripes y</th>
<th>Checkered</th>
<th>Alternating channel</th>
<th>Alternating bit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>total</strong></td>
<td>1.8V</td>
<td>1.000</td>
<td>1.019</td>
<td>1.015</td>
<td>1.021</td>
<td>1.022</td>
<td>1.019</td>
</tr>
<tr>
<td></td>
<td>3.3V</td>
<td>1.000</td>
<td>1.004</td>
<td>1.004</td>
<td>1.009</td>
<td>1.008</td>
<td>1.008</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1.000</td>
<td>1.010</td>
<td>1.008</td>
<td>1.014</td>
<td>1.013</td>
<td>1.012</td>
<td>1.012</td>
</tr>
<tr>
<td><strong>gauss-x</strong></td>
<td>1.8V</td>
<td>1.000</td>
<td>1.023</td>
<td>1.017</td>
<td>1.019</td>
<td>1.019</td>
<td>1.018</td>
</tr>
<tr>
<td></td>
<td>3.3V</td>
<td>1.000</td>
<td>1.003</td>
<td>1.002</td>
<td>1.006</td>
<td>1.007</td>
<td>1.005</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1.000</td>
<td>1.010</td>
<td>1.008</td>
<td>1.011</td>
<td>1.012</td>
<td>1.010</td>
<td>1.010</td>
</tr>
<tr>
<td><strong>gauss-y</strong></td>
<td>1.8V</td>
<td>1.000</td>
<td>1.015</td>
<td>1.011</td>
<td>1.021</td>
<td>1.023</td>
<td>1.018</td>
</tr>
<tr>
<td></td>
<td>3.3V</td>
<td>1.000</td>
<td>1.003</td>
<td>1.004</td>
<td>1.010</td>
<td>1.007</td>
<td>1.006</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1.000</td>
<td>1.007</td>
<td>1.006</td>
<td>1.014</td>
<td>1.013</td>
<td>1.010</td>
<td>1.010</td>
</tr>
<tr>
<td><strong>sharpen</strong></td>
<td>1.8V</td>
<td>1.000</td>
<td>1.034</td>
<td>1.035</td>
<td>1.043</td>
<td>1.042</td>
<td>1.047</td>
</tr>
<tr>
<td></td>
<td>3.3V</td>
<td>1.000</td>
<td>1.028</td>
<td>1.017</td>
<td>1.034</td>
<td>1.028</td>
<td>1.062</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1.000</td>
<td>1.030</td>
<td>1.023</td>
<td>1.037</td>
<td>1.032</td>
<td>1.057</td>
<td>1.057</td>
</tr>
</tbody>
</table>

Table F.3  Energy consumption for different input data. Energy is normalized with respect to the black image.
Average increase in power, time, and energy when leaving interrupts enabled during measurement.

<table>
<thead>
<tr>
<th></th>
<th>Energy</th>
<th>Time</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.25</td>
<td>0.60</td>
<td>0.64</td>
</tr>
</tbody>
</table>

Table F.4

<table>
<thead>
<tr>
<th></th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized energy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.8V</td>
<td>1.007</td>
<td>0.998</td>
<td>1.005</td>
<td>0.998</td>
<td>1.004</td>
</tr>
<tr>
<td>3.3V</td>
<td>1.029</td>
<td>1.008</td>
<td>1.027</td>
<td>1.018</td>
<td>1.011</td>
</tr>
<tr>
<td>Total</td>
<td>1.020</td>
<td>1.004</td>
<td>1.019</td>
<td>1.010</td>
<td>1.009</td>
</tr>
<tr>
<td>Normalized time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.8V</td>
<td>1.009</td>
<td>1.000</td>
<td>1.010</td>
<td>1.000</td>
<td>1.010</td>
</tr>
<tr>
<td>3.3V</td>
<td>1.010</td>
<td>1.000</td>
<td>1.011</td>
<td>1.000</td>
<td>1.010</td>
</tr>
<tr>
<td>Avg</td>
<td>1.009</td>
<td>1.000</td>
<td>1.010</td>
<td>1.000</td>
<td>1.010</td>
</tr>
<tr>
<td>Normalized power</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.8V</td>
<td>0.999</td>
<td>0.998</td>
<td>0.995</td>
<td>0.998</td>
<td>0.994</td>
</tr>
<tr>
<td>3.3V</td>
<td>1.019</td>
<td>1.008</td>
<td>1.016</td>
<td>1.018</td>
<td>1.001</td>
</tr>
<tr>
<td>Total</td>
<td>1.011</td>
<td>1.004</td>
<td>1.008</td>
<td>1.010</td>
<td>0.999</td>
</tr>
</tbody>
</table>

Table F.5

<table>
<thead>
<tr>
<th></th>
<th>Normalized energy</th>
<th>Normalized time</th>
<th>Normalized power</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPARM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 core</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>O1</td>
<td>0.45</td>
<td>0.48</td>
<td>0.94</td>
</tr>
<tr>
<td>O2</td>
<td>0.44</td>
<td>0.46</td>
<td>0.95</td>
</tr>
<tr>
<td>O3</td>
<td>0.44</td>
<td>0.46</td>
<td>0.95</td>
</tr>
<tr>
<td>Os</td>
<td>0.43</td>
<td>0.45</td>
<td>0.95</td>
</tr>
<tr>
<td>L9260</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>O1</td>
<td>0.44</td>
<td>0.44</td>
<td>1.01</td>
</tr>
<tr>
<td>O2</td>
<td>0.40</td>
<td>0.39</td>
<td>1.02</td>
</tr>
<tr>
<td>O3</td>
<td>0.40</td>
<td>0.39</td>
<td>1.02</td>
</tr>
<tr>
<td>Os</td>
<td>0.40</td>
<td>0.39</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table F.6

Comparison of simulated and hardware results for the Quicksort application with different compiler optimizations. Results are normalized to the O0 results on each platform.
<table>
<thead>
<tr>
<th></th>
<th>Normalized energy</th>
<th>Normalized time</th>
<th>Normalized power</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPARM</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 core</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O0</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>O1</td>
<td>0.20</td>
<td>0.22</td>
<td>0.90</td>
</tr>
<tr>
<td>O2</td>
<td>0.22</td>
<td>0.24</td>
<td>0.92</td>
</tr>
<tr>
<td>O3</td>
<td>0.22</td>
<td>0.24</td>
<td>0.92</td>
</tr>
<tr>
<td>O5</td>
<td>0.22</td>
<td>0.24</td>
<td>0.93</td>
</tr>
<tr>
<td><strong>L9260</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>O1</td>
<td>0.29</td>
<td>0.29</td>
<td>1.00</td>
</tr>
<tr>
<td>O2</td>
<td>0.25</td>
<td>0.24</td>
<td>1.01</td>
</tr>
<tr>
<td>O3</td>
<td>0.24</td>
<td>0.24</td>
<td>1.01</td>
</tr>
<tr>
<td>O5</td>
<td>0.27</td>
<td>0.26</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table F.7  Comparison of simulated and hardware results for the image filter with different compiler optimizations. Results are normalized to the O0 results on each platform.

<table>
<thead>
<tr>
<th></th>
<th>gauss-x</th>
<th>gauss-y</th>
<th>sharpen</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Normalized energy</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPARM 1 core</td>
<td>1.00</td>
<td>1.10</td>
<td>0.12</td>
</tr>
<tr>
<td>L9260</td>
<td>1.00</td>
<td>1.15</td>
<td>0.11</td>
</tr>
<tr>
<td><strong>Normalized time</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPARM 1 core</td>
<td>1.00</td>
<td>1.16</td>
<td>0.12</td>
</tr>
<tr>
<td>L9260</td>
<td>1.00</td>
<td>1.15</td>
<td>0.11</td>
</tr>
<tr>
<td><strong>Normalized power</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPARM 1 core</td>
<td>1.00</td>
<td>0.95</td>
<td>1.04</td>
</tr>
<tr>
<td>L9260</td>
<td>1.00</td>
<td>1.00</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Table F.8  Comparison of simulated and hardware results for the image filter. Results are normalized to the gauss-x task on each platform.
Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© Edward Carlstedt-Duke
Erik Elfström