Energy Reduction Techniques for Caches and Multiprocessors

Energy Reduction Techniques for Caches and Multiprocessors

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op 16 oktober 2009 om 12:30

door

Pepijn Jacob DE LANGEN

elektrotechisch ingenieur
geboren te Groningen, Nederland

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. K.G.W. Goossens

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. K.G.W. Goossens, promotor	Technische Universiteit Delft
Dr. B.H.H. Juurlink, copromotor	Technische Universiteit Delft
Prof. dr. N.J. Dimopoulos	University of Victoria
Prof. dr. K.G. Langendoen	Technische Universiteit Delft
Prof. dr. S.K. Nandy	Indian Institute of Science
Prof. dr. ir. H.J. Sips	Technische Universiteit Delft
Prof. dr. H.A.G. Wijshoff	Universiteit Leiden

ISBN: 978-90-72298-03-4

Keywords: Energy reduction, Caches, Multi processor scheduling

Cover design: Floris de Langen

Copyright © 2009 P.J. de Langen
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without permission of the author.
Printed in the Netherlands

The research presented in this dissertation was partially funded by the Netherlands Organization for Scientific Research (NWO, The Hague, project no. 612-064-308).

Dedicated to Nicole and Tijmen,
for all their love and support over the years.

Energy Reduction Techniques for Caches and Multiprocessors

Pepijn de Langen

Abstract

Energy consumption is a growing concern in many areas of computer architecture. Not only for the handheld embedded market, but also for desktop machines and high-end server facilities, there is a demand for ever increasing processing power while maintaining or even decreasing energy consumption. For processors embedded in battery-powered devices, consumers both demand an increasing number of features and an increase of battery lifetime. For commodity desktop and high-end server systems, the demand to reduce energy consumption is mostly fueled by cost, environmental issues, and the wish to have systems without noisy cooling systems. This dissertation studies several techniques that aim at reducing energy consumption in processors.

Part of the techniques presented in this dissertation focusses at reducing energy consumption by decreasing the amount of data transferred between a processor and external memory. Since memory is one of the known bottlenecks in computer systems, manufacturers had to employ increasingly aggressive techniques in the past decades to increase performance. The techniques proposed in this dissertation target at improving or at least maintaining performance, while reducing the amount of energy dissipated in the memory subsystem.

Another part of this dissertation focusses on reducing energy by lowering the speed of nodes in multiprocessor systems in combination with turning off some of these nodes. Multiprocessor systems have gained significant interest in the past years, mostly because power constraints have prevented further increasing clock frequencies and because instruction level parallelism has suffered from diminishing returns. Due to the way how energy is dissipated in semiconductor fabric, using multiple cores on a reduced frequency is an effective way to reduce energy consumption. Due to decreasing sizes of the components from which processors are built, it is expected that this energy model will change significantly in future years. Some of the techniques presented in this dissertation aim at reducing energy consumption in such contemporary and near-future multiprocessor systems.

Acknowledgments

This dissertation is the result of over 4 years of work in the Computer Engineering laboratory of the Technical University in Delft. This work would not have been possible without the help and support of several people.

First of all, I would like to thank my adviser and copromotor Ben Juurlink, for providing the opportunity to perform my Ph.D. research and for the guidance throughout the years. His insightful comments have significantly contributed to this work.

I am very grateful for having known Professor Stamatis Vassiliadis, both in a professional and in a personal setting. Like for most people in the Computer Engineering group, Professor Vassiliadis was a huge source of inspiration. It was a great loss to everyone when he passed away on April 7th 2007. He was a passionate researcher and a great person. He will always remain in my heart and mind as 'my professor'.

I thank Professor Kees Goossens, who suddenly had to take up the role of promotor for me and many other Ph.D. candidates.

I would like to thank my first office mates, Dan and Gabi, who have helped me significantly getting started in the first months of my Ph.D. They both were valuable sources for getting quick answers to technical questions, but also great people for having endless discussions about non-technical issues.

I want to thank all my friendly colleagues in the Computer Engineering group. I have really enjoyed working with such a wide variety of inspiring people. I am especially thankful to my good friends Carlo and Lotfi, for all the lovely dinners, the interesting discussions, and especially for their support in the last years.

I also owe my thanks to Georgi Gaydadjev, for being both a good neighbor and a good friend. I thank Bert and Lidwina, for all their technical and administrative support.

I would like to thank my parents for all the love they have brought to my life,

and for always supporting my academic endeavors. I am grateful to Gillis, Floris, and Roderik, for the fact that we are not only brothers but also very good friends. In addition, I want to thank Floris for designing the cover of this dissertation.

Last but not least, I am in great debt to Nicole and Tijmen, the two most important people in my life. This work would not even have been remotely possible if it weren't for their love, support, and understanding throughout the years. Numerous evenings, weekends, and holidays were given up or reorganized to support my Ph.D.

Pepijn de Langen Delft, the Netherlands, 2009

Contents

Abstract **i**

Acknowledgments **iii**

List of Figures **xii**

List of Tables **xiii**

List of Acronyms **xiv**

1 Introduction **1**

1.1 Motivation 1

1.2 Power and Energy 3

1.3 Challenges 4

1.4 Organization and Contributions 6

2 Reducing Cache Conflict Misses **9**

2.1 Introduction 10

2.2 Related Work 11

2.3 Detecting Conflict Misses 13

2.4 BCC and SCC Caches 14

2.5 Experimental Results 16

2.5.1 Experimental Setup 16

2.5.2 Impact of the CDT Size 17

2.5.3 Traffic 18

2.5.4 Energy Reduction 22
2.5.5 Impact on Execution Time 25
2.6 Conclusions 27

3 Memory Copies in Multi-Level Memory Systems 29
3.1 Introduction 29
3.2 Related Work 31
3.3 Memory Copies in Multi-Level Memory Systems 32
3.3.1 Copying Using Copy Engines 33
3.3.2 Limitations of On-Chip Copy Engines 35
3.3.3 Dynamic Copy Engines 37
3.3.4 Dynamic Copy Engine with Non-Temporal Fetching . 42
3.4 Experimental Results 43
3.4.1 Experimental Setup 43
3.4.2 Experiments with a Memcopy Micro-Benchmark . . . 46
3.4.3 Experiments with a TCP/IP Processing Benchmark . . 50
3.4.4 Energy Reduction 53
3.5 Conclusions 53

4 Limiting the Number of Dirty Cache Lines 55
4.1 Introduction 56
4.2 Related Work 59
4.3 Clean/Dirty Cache 61
4.4 Experimental Results 63
4.4.1 Experimental Setup 63
4.4.2 Experimental Results 65
4.5 Case Study with Cache Decay 69
4.5.1 Cache Decay 69
4.5.2 Cache Decay using the Clean/Dirty Cache 70
4.6 Conclusions 77

5 Energy Efficient Multiprocessor Scheduling using DVS 79
5.1 Introduction 80
5.2 Related Work 81
5.3 Energy Reduction using DVS 83
5.3.1 Dynamic Voltage Scaling 83
5.3.2 Voltage Scaling Requirements 84
5.3.3 Voltage Scaling in a Multiprocessor Environment . . . 87
5.4 System and Application Model 88
5.5 Energy Efficient Scheduling Algorithms 90
5.5.1 Schedule & Stretch 90
5.5.2 Leakage Aware MultiProcessor Scheduling 92
5.6 Experimental Results 95
5.6.1 Experimental Setup 95
5.6.2 Experimental Results 96
5.7 Conclusions 101

6 Energy Efficient Multiprocessor Scheduling using DVS and DPM 103
6.1 Introduction 104
6.2 Related Work 105
6.3 Preliminaries 106
6.3.1 Power Model 106
6.3.2 Effect of DVS for the Power Model 109
6.3.3 Processor Shutdown / DPM 109
6.4 Multiprocessor Scheduling with DVS and DPM 110
6.4.1 S&S+DPM and LAMPS+DPM 111
6.4.2 LIMIT-SF & LIMIT-MF 114
6.5 Experimental Results 115
6.5.1 Experimental Setup 115
6.5.2 Results for the Standard Task Graph Set 117
6.5.3 Results for MPEG-1 123
6.5.4 Results for Different Levels of Static Power 124

6.6 Conclusions 126

7 Conclusions **131**

7.1 Summary and Contributions 132

7.2 Possible Directions for Future Work 136

Bibliography **139**

List of Publications **151**

Samenvatting **153**

Curriculum Vitae **155**

List of Figures

1.1 Relative improvement in energy density of lithium ion batteries versus the number of transistors in Intel microprocessors. . 2

1.2 Total electricity use for servers in the world in 2000 and 2005, including cooling and auxiliary equipment. 3

2.1 Conflict Detection Table (CDT). 14

2.2 Relative amount of traffic produced by a 4kB BCC cache. . . . 19

2.3 Relative amount of traffic produced by a 4kB SCC cache. . . . 19

2.4 Relative amount of traffic produced by a 16kB BCC cache. . . 20

2.5 Relative amount of traffic produced by a 16kB SCC cache. . . 20

2.6 Relative amount of traffic saved by the BCC cache, when compared to a conventional cache. 21

2.7 Relative amount of traffic saved by the SCC cache, when compared to a conventional cache. 22

2.8 Energy per reference in the off-chip memory bus and CDT for conventional, BCC, and SCC caches of 1kB. 24

2.9 Energy per reference in the off-chip memory bus and CDT for conventional, BCC, and SCC caches of 4kB. 24

2.10 Miss rates for conventional, BCC, and SCC caches of 1kB. . . 26

2.11 Miss rates for conventional, BCC, and SCC caches of 4kB. . . 26

3.1 Schematic description of how a copy is performed by a copy engine 34

3.2 Schematic design of how the copy engines are implemented in several levels of the memory hierarchy 38

3.3 Pseudo-code for the dynamic copy engines. 40

3.4 Maximum speedups for copying non-resident data using an on-chip copy engine and the DCE 42
3.5 Example of how one missing block may cause all consecutive memory blocks to be copied in the same cache 44
3.6 Data traffic produced by the memcopy kernel using a 16kB L1 data cache. 47
3.7 Data traffic produced by the memcopy kernel using a 32kB L1 data cache. 48
3.8 Ratio of transferred bytes to copied bytes in the memcopy micro-benchmark. 49
3.9 Execution time reduction 51
3.10 Off-chip traffic reduction 52
3.11 Percentage of memory copies performed in L2 52

4.1 Schematic representation of the Clean/Dirty-cache 61
4.2 L2 accesses per 1000 cycles for baseline write-back and CD-caches with capacities of 36kB and 40kB. 66
4.3 IPC for write-back and CD-caches of 36kB and 40kB. 67
4.4 Dynamic energy consumed in L1 and L2 for baseline write-back and CD-caches of 36kB. 68
4.5 Dynamic energy consumed in L1 and L2 for baseline write-back and CD-caches of 40kB. 68
4.6 Average active size for normal write-back and CD-caches using cache decay. 72
4.7 L2 accesses per 1000 cycles for write-back and CD-caches using cache decay. 74
4.8 Relative energy consumption in the L1 data cache and by the additional L2 accesses when using cache decay with a period of 16000 cycles. 76
4.9 Performance of two different decay caches for the *twolf* and *vpr* benchmarks with various decay periods, relative to a write-back cache without decay. 77

5.1 Normalized energy consumption as a function of the normalized frequency for varying combinations of the dynamic and static components. 86
5.2 Normalized energy consumption as a function of the normalized frequency for varying threshold voltages. 86
5.3 Example of translating periodic tasks into a DAG. 89
5.4 Example for translating KPNs into DAGs. 90
5.5 Example graph and schedule. 91
5.6 Pseudo-code for the list scheduling algorithm. 92
5.7 Schedules produced by S&S and LAMPS. 93
5.8 Pseudo-code for the LAMPS heuristic. 94
5.9 Average power consumption of various schedules, normalized to a single fully active processor, for different benchmarks with the deadline at 1.5× the critical path length. 97
5.10 Power reduction achieved by the LAMPS scheduling heuristic over S&S. 100
5.11 Power reduction achieved by the LAMPS scheduling heuristic over S&S, when scaling the voltage in discrete steps. 101

6.1 Power consumption as a function of the normalized frequency 108
6.2 Energy consumption as a function of the normalized frequency 108
6.3 Minimum number of idle cycles required for processor shutdown to be beneficial, as a function of the normalized processor frequency. 110
6.4 Illustration of S&S and S&S+DPM. 111
6.5 Pseudo-code for the SS+DPM heuristic. 112
6.6 Pseudo-code for the LAMPS+DPM heuristic. 113
6.7 Dependence graph for processing 15 MPEG-1 frames. 116
6.8 Energy consumption relative to S&S for coarse-grain tasks. . . 118
6.9 Energy consumption relative to S&S for fine-grain tasks. . . . 119
6.10 Energy/total work as a function of the average amount of parallelism. 122

6.11 Energy consumption as a function of the relative frequency for different levels of static energy consumption. 125
6.12 Total energy consumption as a function of the relative frequency for different levels of static energy consumption. . . . 125
6.13 Energy consumption relative to S&S for fpppp and robot, using different levels of static power consumption. 127
6.14 Energy consumption relative to S&S for MPEG-1, using different levels of static power consumption. 128

List of Tables

2.1 Benchmarks and inputs from the MediaBench suite. 16

3.1 Main properties of the simulated system 45

4.1 Average percentage of dirty cache lines in a 32kB 2-way set-associative cache with a line size of 32 bytes. 56
4.2 Write miss alternatives. 58
4.3 Baseline processor configuration. 64
4.4 Energy consumption of 32kB 2-way set-associative caches with cache lines of 32 bytes. 64
4.5 Relative energy costs of caches in experimental model. 65

5.1 Six benchmarks from the Standard Task Graph set and their main characteristics. 96
5.2 Results for deadlines of 1.5× the critical path length. 98
5.3 Results for deadlines of 2× the critical path length. 98
5.4 Results for deadlines of 4× the critical path length. 99
5.5 Results for deadlines of 8× the critical path length. 99

6.1 Constants for 70nm technology. 107
6.2 Employed benchmarks from the Standard Task Graph set and their main characteristics. 116
6.3 Energy consumption relative to S&S for the MPEG-1 benchmark using various approaches. 123

List of Acronyms

Notation	Description
ABB	Adaptive Body Biasing
ABC	Allocation By Conflict
BCC	Bypass in Case of Conflict
CD-Cache	Clean/Dirty cache
CDT	Conflict Detection Table
CE	Copy Engine
CMOS	Complementary Metal-Oxide Semiconductor
CPL	Critical Path Length
DAG	Directed Acyclic Graph
DCE	Dynamic Copy Engine
DCE-NT	Dynamic Copy Engine with Non-Temporal Fetching
DMA	Direct Memory Access
DPM	Dynamic Power Management
DTSE	Data Transfer and Storage Exploration
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling
EDF	Earliest Deadline First
GOP	Group Of Pictures
IPC	Instructions Per Second

Notation	Description
KPN	Kahn Process Network
LAMPS	Leakage Aware MultiProcessor Scheduling
LAMPS+DPM	Leakage Aware MultiProcessor Scheduling with DPM
LCM	Least Common Multiple
LS-EDF	List Scheduling with Earliest Deadline First
MAT	Memory Address Table
PIM	Processing-In-Memory
SCC	Sub-block in Case of Conflict
SMT	Simultanious Multi-Threading
S&S	Schedule and Stretch
S&S+DPM	Schedule and Stretch with DPM
STTD	Shortest Time To Deadline

1 Introduction

This dissertation covers several different energy reduction techniques for contemporary and future processors. While background and motivation for each distinct technique is provided with each corresponding chapter, this introductory chapter provides the background and motivation that is common among the different chapters. Finally, the organization of this dissertation is outlined.

1.1 Motivation

Energy and power consumption are becoming increasingly important in the design of processors. Not only for processors embedded in battery powered devices, also for processors targeted at high-end server clusters energy consumption gains a growing interest. In this dissertation we propose several techniques targeted at reducing energy consumption in single- and multiprocessor systems, applicable to the embedded as well as to the high-end market.

Although computer architecture research was historically mostly targeted at enhancing performance, energy and power consumption have become increasingly important in recent years for several reasons.

The first reason is that processors are increasingly deployed in battery-powered

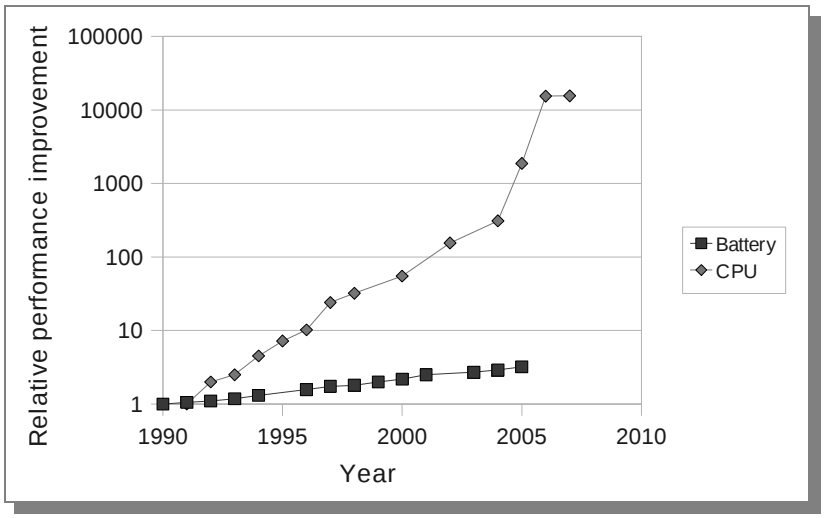

FIGURE 1.1 *Relative improvement in energy density of lithium ion batteries versus the number of transistors in Intel microprocessors [14].*

embedded systems. The physical size of batteries as well as their limited capacity demand that these systems use power as sparingly as possible. This demand is even more fueled by an increasing demand for more processing power without a comparable improvement of battery capacities. The disparity between the growth of processors and the growth of battery capacities is depicted in Figure 1.1, which shows a graph redrawn from work by Chalamala [14].

Another reason for increased attention for energy reduction relates to the increase in the total number of servers deployed around the world, and the growing importance of the electricity cost of these servers. Figure 1.2 depicts the total amount of electricity used by servers in world in 2000 and 2005, as estimated by Koomey [63]. The data in this figure is separated in electricity used for three different server classes and electricity used for cooling and auxiliary equipment. From this figure, it is clear that in 5 years the total amount of electricity used by servers around the world has more than doubled. Moreover, approximately half the electricity is used for cooling and auxiliary equipment. As a result of an increase in power consumption and a decrease in the cost of computing hardware, for many companies the electricity bills are becoming an increasingly large fraction of the total expenditure.

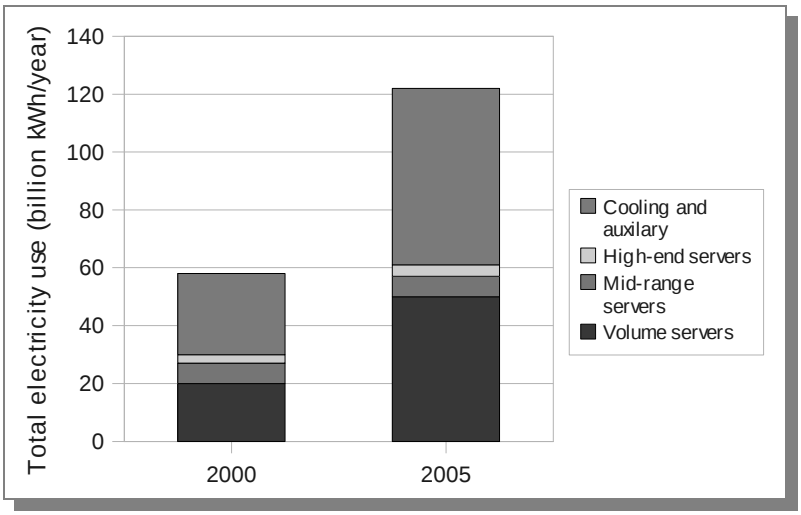

FIGURE 1.2 *Total electricity use for servers in the world in 2000 and 2005, including cooling and auxiliary equipment [63].*

1.2 Power and Energy

In many works in literature, power and energy are used interchangeably. Although there is a clear relation between these two quantities, they are only proportional if power refers to *average* power. In many cases, however, power reduction techniques aim to reduce the peak power consumption or the power density [80], in order to keep processors functional without requiring exorbitant cooling.

This dissertation presents techniques to reduce the energy or average power consumption of processors. Although there are strong relations between techniques aimed at reducing average power and techniques aimed at reducing peak power or power density, the first is the primary focus of this dissertation.

Power in CMOS circuits is generally classified in a dynamic and a static part. The dynamic part refers to the power dissipated due to switching between low and high logic levels. The static part refers to the power dissipated through leakage currents in non-ideal transistors. Dynamic power has dominated static power in the past decades, and has increased significantly due to its quadratic relation on the clock frequency. Due to increasing transistor counts and decreasing feature sizes, however, static power consumption has increased significantly and is expected to increase further in the next decades [41]. More

specifically, Borkar [8] predicted leakage current to increase by a factor of five with each technology generation, and Duarte et al. [25] predicted that static power consumption will eventually surpass the dynamic power consumption.

1.3 Challenges

In order to develop energy efficient techniques for future processors, one should know the key challenges for such processors, both from an energy and a performance perspective. To this end, two key observations were made.

The first observation is that limited memory bandwidth is and will remain a fundamental impediment to attain higher performance, rendering the memory hierarchy as one of most important considerations in the design of contemporary and future processors. To hide the long latency of off-chip memories, most processors employ several levels of caches, on-chip and/or off-chip. This is beneficial due to the inherent *temporal* and *spatial* locality of data. While memories get faster over time, processors become faster as well. In fact, processors speeds have grown at a faster rate than memory speeds have, which has led to a growing disparity between the speeds of processors and memories. Due to the limited number of processor pins and the intrinsic delay due to the physical distance between processors and off-chip memories, this disparity is even more significant for memories located off-chip. This problem is often referred to as the *memory wall* [71, 95]. To improve performance of systems with limited bandwidth and long memory latency, chip manufacturers have used increasingly larger on-chip caches, boosted the transfers speed between processors and off-chip memories, and employed aggressive prefetching techniques. While these choices provide one-time improvements, they do not fundamentally solve the problem. And while these techniques improve performance for memory-bound programs, they generally also increase the total energy consumption.

Accesses to off-chip memory are not only posing an impediment to higher performance, they also contribute significantly to the power budget [69, 86]. For example, Basu et al. [6] show that an off-chip data bus in an embedded processor consumes between 9.8% and 23.2% of the total power.

In many contemporary processors, large on-chip caches often already take up the majority of the die area. This not only drives the cost of these processors, larger caches also consume a significant amount of power.

These observations have led to the following questions:

- Especially for low-power embedded processors, memory transfers are responsible for a significant amount of energy consumption. Moreover, the memory bandwidth is a fundamental impediment to attain higher performance on memory-bound programs. How can we decrease the used memory bandwidth in processors while maintaining or possibly improving performance, without employing large caches?
- Are there situations in which caches and multi-level memories perform particularly bad? If so, can these problems be resolved efficiently?
- Since larger on-chip caches are more costly in terms of energy and manufacturing cost than smaller caches, can we attain comparable performance with processors employing significantly smaller caches?

The second observation is that industry no longer tries to improve performance by boosting the clock frequency. While this has been popular and effective in past decades, this trend cannot simply be continued for the following reasons. First, the key to increasing clock frequency has been the shrinking size of transistors. With decreasing feature size, however, it becomes increasingly hard to make reliable circuits, both due to an increase in transient errors and due to process variations. Furthermore, decreasing transistor sizes inherently lead to an increases in leakage current, which in term may lead to higher energy consumption. Secondly, power consumption in CMOS circuits grows quadratically with the clock frequency. Increasing this frequency therefore leads to fundamental cooling problems, required to guarantee proper operation. As a result, the trend of increasing clock frequencies has shifted to maintaining the same frequency but increasing the number of processor cores. Yet, this has been an energy efficient solution mostly due to the current state of technology. In current technology, dynamic power consumption (due to switching between logic levels) dominates static power consumption (due to leakage currents in non-ideal transistors). With decreasing feature sizes and with reduced clock frequencies, however, this balance may be different. When static power becomes more significant, the most energy efficient solution is no longer to use as many processors as possible on a low clock frequency. Instead, it will be more efficient to find a proper balance between the clock frequency and the number of used processor cores.

From these observations, the following questions were derived:

- While in many cases increasing the number of processor cores and decreasing the clock frequency leads to a reduction in energy, this is mostly

due to the importance of dynamic energy. In the case of increased leakage power, to what extend will increasing parallelism lead to a reduction in energy?

- In CMOS, both dynamic and static power consumption decrease with decreasing voltage and clock frequency. Lowering the clock frequency, however, also increases the time to complete computational tasks. Since energy equals power multiplied by time, the static energy consumption will actually increase when reducing the clock frequency. As a result, the total energy consumption may also start to increase below a certain speed. To what extend should the clock frequency be lowered in order to minimize energy consumption?
- When static power consumption is more significant, it becomes worthwhile to temporarily power-off processor cores. Switching cores off and on, however, requires a certain amount of time and will therefore also consume energy. Under which circumstances is switching of processor cores an interesting option?
- How do these options (number of processor cores, reduced frequency, and temporarily turning off processors) relate, and how can we find the optimal operating point that minimizes energy?

1.4 Organization and Contributions

As was already indicated in the previous section, the techniques presented in this dissertation can be classified into two areas. The first area, covered by the first three chapters, targets energy reduction in the multi-level memory system. The second area, covered by the last two chapters, proposes energy efficient scheduling techniques for near-future multiprocessor systems.

Small direct-mapped caches consume less energy than large set-associative caches. However, small direct-mapped caches incur many conflict misses. In Chapter 2, we target energy reduction by reducing the amount of (off-chip) memory traffic caused by recurring conflict misses in caches. The proposed technique is based on a structure called the *Conflict Detection Table* (CDT). The CDT is used to detect conflict misses in direct-mapped caches. Using this information, memory traffic is reduced by transferring only the requested word instead of the whole cache line from the next memory level. Two cache organizations that employ the CDT are proposed: the *Bypass in Case of Conflict* (BCC) cache and the *Subblock in Case of Conflict* (SCC). While the BCC

bypasses the cache when a conflict miss is detected, the SCC only stores the requested subblock on the same event. Experimental results show that these organizations can reduce the amount of memory traffic by up to 65% for the BCC, and up to 47% for the SCC.

Memory copies require virtually no computational power, but produce a significant amount of memory traffic. Furthermore, large memory copies can completely replace the contents of the data cache, while the newly allocated data may not be used directly after. In Chapter 3, we develop a technique to efficiently handle memory copies in multi-level memory systems, using an organization where memory copies are performed using *Copy Engines* (CEs). A technique is proposed that can dynamically decide in which level the copy should be performed in order to minimize memory traffic. By avoiding transfers back and forth between the main memory and the CPU, a significant amount of traffic and energy can be saved. Furthermore, by reducing the amount of used memory bandwidth and by performing the memory copies asynchronously in dedicated hardware, the proposed organization also improves performance. In an experimental setup with a TCP/IP processing benchmark and a 2-level cache hierarchy, the proposed technique shows to reduce memory traffic by up to 94% and improve execution time by up to 21%.

Write-back caches are usually preferred over write-through caches since they attain significantly higher performance. Furthermore, write-through caches produce significantly more write traffic, wasting precious memory bandwidth and energy. Write-back caches, on the other hand, maintain a significant number of *dirty* cache lines, which can be problematic for several reasons. Many techniques to reduce energy consumption in caches resize, reconfigure, or shut down (parts of) a cache dynamically. With these techniques, dirty cache lines have to be written back to the next memory level first. Not only can this lead to an increased number of bandwidth stalls due to bursts of write-back traffic, it can also bring additional complexity to the design of the energy reduction logic. Furthermore, with error detection clean cache lines are tolerant to transient errors, whereas dirty cache lines need error correction. In Chapter 4, we propose a cache organization called the *Clean/Dirty* (CD) cache. The CD-cache is a cache organization using two separate cache structures. The first structure is only used to store clean data. The second structure is much smaller, and is used to store all dirty data. By splitting the cache in clean and dirty data, cache energy reduction techniques can be applied efficiently to the clean data, while maintaining an acceptable amount of write traffic. The proposed organization results in a similar or higher performance than a write-back cache, while reducing the number of dirty cache lines significantly. In a case study it

is shown how the proposed organization can be applied efficiently to a energy reduction technique called *Cache Decay*. Compared to a conventional write-back cache that uses cache decay, the CD-cache with cache decay improves the energy reduction by more than twice on average.

An effective technique to reduce power consumption in modern CPUs is *Dynamic Voltage Scaling* (DVS), in which both the frequency and the supply voltage are scaled down when less performance is demanded. For multiprocessors, an effective technique is to use as many processors as possible to reduce the *makespan* of the schedule, and to use the remainder of time until the deadline (slack) to reduce the frequency and supply voltage. This technique is effective because in current technologies dynamic power consumptions dominates static power consumption. As static power consumption due to leakage currents is expected to increase dramatically in the near future, this technique may no longer suffice. Instead, it becomes more effective to balance between reducing the makespan and limiting the number of employed processors. In Chapter 5, we propose a technique called *Leakage Aware MultiProcessor Scheduling* (LAMPS). LAMPS is a scheduling heuristic that finds an optimal balance between DVS and using the correct number of processors. This is achieved by using a non-optimal but fast scheduling algorithm, which allows our heuristic to dedicate more time to finding the optimal number of processors. Results show that this heuristic improves energy consumption by up to 67%, compared to a technique that employs as many processors as possible.

In Chapter 6, we extend LAMPS by also allowing a processor to shut down temporarily. The proposed heuristic finds an optimal balance between DVS, using the correct number of processors, and shutting down these processors temporarily. Results show that this approach reduces the energy consumption by up to 14%, compared to the LAMPS heuristic presented in Chapter 5. Furthermore, two lower bounds are presented. One for the case where all processors run at the same frequency and where this frequency is fixed throughout the schedule, and one where each processor is assigned its own frequency and this frequency may change over time. Using these lower bounds, it is shown that after applying the proposed scheduling heuristic, there is little room left for improvement.

Chapter 7 concludes this dissertation by summarizing the most important contributions and directions for future research.

2

Reducing Cache Conflict Misses

Off-chip memory accesses are a major source of power consumption in embedded processors [6, 13, 93, 94]. In order to reduce the amount of traffic between the processor and the off-chip memory as well as to hide the memory latency, nearly all processors have one or more levels of cache on the same die as the processor core. However, because small caches dissipate less power and are cheaper than large caches, a small cache is preferable to a large cache in embedded processors. Furthermore, because set-associative caches consume more power than direct-mapped caches, a direct-mapped cache is preferable to a set-associative one. Small, direct-mapped caches, however, generally incur many conflict misses. To reduce the amount of traffic generated by small direct-mapped caches, in this chapter we propose and evaluate a structure called the *Conflict Detection Table* (CDT). This table can be used to determine if a memory access is expected to hit the cache. If a hit is expected and a miss occurs, then a conflict is detected and appropriate action can be taken. In addition, we propose two cache structures that use this information to apply a better caching strategy: the *Bypass in Case of Conflict* (BCC) cache and the *Sub-block in Case of Conflict* (SCC) cache. The BCC cache bypasses the cache when a conflict is detected, whereas the SCC cache fetches a sub-block of the missing cache line in such a case. Both the BCC and the SCC cache try to minimize the amount of data traffic, by only fetching the requested word instead of a whole cache line when a conflict is detected.

While bypassing the cache disallows exploiting temporal or spatial locality, the benefit is that it avoids the need to discard the previous contents of the cache line. With sub-block caching, the previous contents are discarded. In this case, however, it may be possible to exploit temporal locality if the requested sub-block is referenced again in the near future. Whether bypassing or sub-block caching is more efficient, therefore, depends on how data is referenced afterwards. Experimental results using several embedded workloads show that the BCC and SCC cache reduce the amount of traffic significantly in many cases. Furthermore, overall they incur the same number of cache misses as the direct-mapped cache. This shows that the BCC and SCC cache reduce the amount of energy consumption with a negligible reduction in performance.

Most of the material presented in this chapter has been previously published in [19].

2.1 Introduction

In order to limit the amount of off-chip memory traffic, it is essential that embedded processors make effective use of the on-chip cache. Embedded processors often exploit a small cache with limited associativity, because they are cheaper and more power efficient than large caches, and because increased associativity increases the power consumption and cycle time [84]. Small, direct-mapped caches, however, generally produce many conflict misses and, as a result, generate a significant amount of processor-memory traffic [18].

In this chapter, we present a novel technique to detect and eliminate conflict misses in caches. This technique is based on a structure called the *Conflict Detection Table* (CDT). The CDT contains the tag part of the addresses referenced by recently executed load/store instructions and is indexed by the lower-order bits of the program counter. The idea behind the CDT is that if an entry corresponding to a load/store instruction is found in the CDT and the data tag stored in this entry matches the tag of the current data address, a (spatial) hit is expected because the referenced word was loaded in the cache the previous time this instruction was executed. Furthermore, if a hit is expected but the cache access yields a miss, then a conflict is detected because the word must have been replaced by another instruction.

We propose two cache structures that employ the CDT. The first, called the *Bypass in Case of Conflict* (BCC) cache, bypasses the cache when a conflict is detected. The second, called the *Sub-block in Case of Conflict* (SCC) cache, is a sector cache that fetches only the missing sector (or sub-block) when a

conflict is detected. Both the BCC as well as the SCC cache are direct-mapped.

This chapter is organized as follows. Section 2.2 briefly discusses related work. In Section 2.3 we explain how recurring conflict misses can be detected and appropriate action can be taken. Section 2.4 presents cache organizations that use the CDT to reduce the negative effect of recurring conflict misses. The effectiveness of these caches are experimentally verified in Section 2.5. Conclusions are given in Section 2.6.

2.2 Related Work

Jouppi [47] proposed employing a small (consisting of four to eight entries), fully associative *victim cache* in order to reduce conflict misses in direct-mapped caches. Blocks evicted from the L1 cache are not immediately placed in the L2 cache but are given a second chance in the victim cache. The victim cache is fully associative, however, and fully associative caches consume more energy than direct-mapped caches [84]. Memik et al. [73] showed how the victim cache can be used to reduce energy consumption by avoiding more expensive accesses to the next memory level. They furthermore showed that this resulted in better energy-delay and energy-delay-area products than would be obtained by increasing the size or associativity.

Kin et al. [60] stated that designers should be willing to trade performance for low power, and proposed to use an unusually small *filter cache* in-between the L1 cache and the load store unit. Energy is reduced by servicing memory access from the much smaller filter cache instead of the normal L1 cache. However, due to the limited size of this filter cache, many requests are serviced from the bigger L1 cache with an increased latency, thereby causing a decrease in performance.

The Dual Data Cache proposed by González et al. [29] includes a mechanism that detects if a load instruction interferes with itself. This happens, for example, when a vector is accessed repeatedly and the vector is larger than the cache. In such a case, the vector displaces itself from the cache. This situation is even worse when the vector is accessed with a stride unequal to one and the stride and the cache size are not co-prime, because in this case not all blocks are used to cache the vector. This mechanism, however, does not detect cross-interference, i.e., it does not discover situations in which data is replaced by data referenced by a different load instruction. Juurlink proposed to apply this technique using a sector cache instead of two separate cache structures, which was called the Unified Dual Data Cache [49].

Johnson et al. [45] try not to evict a block if it is more heavily used than the arriving block that generated a miss. To do so they divide the memory into regions called *macroblocks* and employ a table called the *Memory Address Table* (MAT) that contains information about how often each macroblock is used. If the MAT indicates that the block to be replaced is more heavily used than the arriving block, the arriving block is not stored in the cache. The MAT behaves like a cache, and according to results in [45] it appears that it must be rather large in order to be effective. Furthermore, this technique is targeted at improving performance rather than reducing energy consumption.

Tam [89] proposed the *Allocation By Conflict* (ABC) replacement policy. In this organization a 1-bit counter is associated with each cache block, which is increased (decreased) each time an access to this block yields a miss (hit). A block is evicted from the cache only when two consecutive accesses produce a miss.

There are also static (compiler) approaches aimed at reducing conflict misses. The Data Transfer and Storage Exploration (DTSE) methodology [10, 11] developed at IMEC, for example, focuses on compile-time techniques. One of the steps in this methodology aims at improving locality by applying loop transformations such as loop interchange and loop splitting and merging. In another step the restricted lifetimes of (parts of) array variables are exploited to overlap them in the address space. The leads to better cache performance and also reduces the total size of the required memories. While very good results have been achieved using the DTSE methodology (speedups of up to a factor of 3 and reductions in bus load by an order of magnitude have been reported), the approach has two limitations. First, the methodology is only partially supported by tools and, hence, requires manual intervention which increases the design complexity. Second, it is unclear if it can be applied to dynamic, pointer-based applications.

Another static approach to reduce conflicts in caches is proposed by Petrov and Orailoglu [78]. Their method is aimed at application-specific customization of the data cache of embedded processors. They present an algorithm that partitions memory access instructions into groups of instructions that exhibit data reuse amongst them. Each group is mapped to a certain partition of the cache, which allows to isolate them from possibly interfering groups. Furthermore, the tag comparison can be avoided if it can be determined that a reference will invariably hit the cache.

2.3 Detecting Conflict Misses

A conflict miss occurs when a memory word is referenced twice but is in-between replaced by other data. Conflict misses occur more frequently in direct-mapped caches than in set-associative caches, because each memory word is mapped to only one cache location. Consider, for example, the following simple loop:

```
for (i=0; i<n; i++)
    a[i] = b[i]+c[i];
```

If the differences between the base addresses of the arrays `a`, `b`, and `c` are a multiple of the cache size, each `a[i]`, `b[i]`, and `c[i]` all map to the same cache line and each will replace the other in the cache so that there is no chance to exploit the spatial locality exhibited by this code. Such 'ping-pong' effects will degrade the cache performance severely. Although, in this simple example, these conflicts can be avoided by loop transformations, this may not always be possible for more complex code.

To detect conflict misses in direct-mapped caches, we propose a small structure called the CDT. The principle idea behind the CDT is that when consecutive executions of a load/store instruction access the same cache line, a cache hit is expected for all but the first access. If a hit is expected but a cache miss occurs, a different instruction must have accessed a word that is mapped to the same cache line.

As illustrated in Figure 2.1, the CDT is a cache-like structure that is indexed by the lower-order bits of the program counter. While the CDT could be designed as a set-associative cache, in this work it is assumed to be direct-mapped. Every entry contains the higher-order bits of the instruction address and the tag of the address referenced the previous time the corresponding load/store instruction was executed. Each time a load/store instruction is executed, the CDT is accessed and the higher-order bits of the instruction address are used to determine if there is an entry for the current instruction. If no entry is found, one is allocated and the data tag field is set to the tag of the current address. If the higher order bits of the instruction address match, an entry is found, and the data tag stored in this entry is compared to the tag of the current address. If these data tags also match, the requested data is expected to be already present in the cache, because it was fetched the last time this instruction was executed. From this it follows that if a cache miss is encountered, the requested data must have been replaced by a different instruction and a conflict is detected.

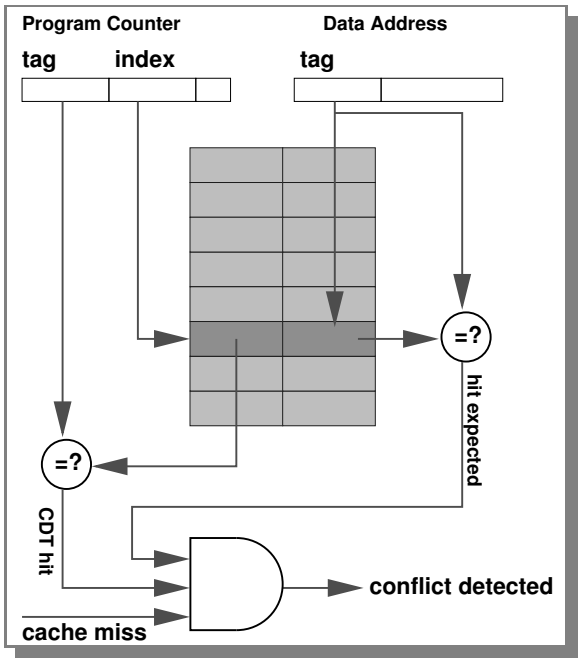

FIGURE 2.1 *Conflict Detection Table (CDT).*

A conflict can only be detected when a cache miss occurs. The information about possible conflicts, therefore, does not need to be available until after the tag comparison. This implies that the CDT will not increase the time to hit the cache. Furthermore, since the amount of required logic to implement the CDT is fairly low, it will not consume much energy. A detailed evaluation relating the energy consumed by the CDT to the energy saved by reducing memory traffic is presented in Section 2.5.

2.4 BCC and SCC Caches

When a conflict is detected, it is not known in advance which instruction will be the first to re-use this cache line. Therefore, for reducing the miss rate, it is not certain what will be most efficient: replacing the current line or bypassing the cache. The same is true for reducing the amount of traffic. If a cache line is not reused in the near future, it may be beneficial to only fetch the requested data and/or to bypass the cache. If the same cache line is accessed several times in the near future, however, this will increase the amount of traffic since

each request also requires transferring the address. The techniques presented in this chapter target memory traffic reduction by fetching only the requested word instead of the whole cache line, but only if a conflict is detected.

We propose two cache structures that employ the CDT to detect and eliminate conflict misses. Both caches are direct-mapped but have the additional possibility to bypass the cache or to store only part of the requested cache line.

- **Bypass in Case of Conflict** — The first cache organization using the CDT uses cache bypassing and is called the BCC cache. When a conflict is detected in the BCC cache, the requested word is fetched from the next memory level, but it is not stored in cache. In other words, when the CDT detects that the cache line has been replaced by another instruction, this cache will no longer store the words referenced by this instruction, as long as the instruction references the same cache line.

- **Sub-block in Case of Conflict** — The second cache organization using the CDT employs *sub-block* caching [33] and is called the SCC cache. When a conflict is detected by the CDT, only the requested sub-block is fetched instead of the whole cache line. This sub-block is then stored in the corresponding location in cache, and the rest of the sub-blocks on the same cache line are invalidated. When afterwards an invalidated sub-block is referenced, it is treated as a normal miss and the whole cache line is fetched from the next memory level.

When a miss occurs in either the BCC or the SCC cache, this data is fetched from the next memory level, similar to a conventional cache. However, when a conflict is detected in the BCC and SCC caches, only the requested word of the cache line is transferred from the next memory level, whereas in a conventional cache the whole cache line is transferred. This potentially saves a significant amount of data traffic, especially if a significant number of conflict misses occur. However, not fetching the remainder of the cache line may degrade performance if the CDT incorrectly predicts conflicts for instructions that exhibit a certain amount of spatial locality.

Both the BCC and the SCC cache avoid transferring data speculatively when a conflict is detected. The fundamental difference between the BCC and the SCC caches is that when a conflict is detected, the SCC assumes there is no benefit from spatial locality, but still tries to benefit from temporal locality by storing the requested sub-block. The BCC cache, on the other hand, tries to benefit from not replacing a previously referenced cache line.

Benchmark	Input	# of mem. refs.
adpcm-dec	clinton.adpcm	$4.59 \cdot 10^5$
adpcm-enc	clinton.pcm	$4.59 \cdot 10^5$
g721-dec	clinton.g721	$4.90 \cdot 10^7$
g721-enc	clinton.pcm	$4.78 \cdot 10^7$
gsm-dec	clinton.pcm.gsm	$8.39 \cdot 10^6$
gsm-enc	clinton.pcm	$5.19 \cdot 10^7$
jpeg-dec	testimg.jpg	$1.18 \cdot 10^6$
jpeg-enc	testimg.ppm	$6.55 \cdot 10^4$
mpeg2-dec	mei16v2.m2v	$3.28 \cdot 10^7$
pegwit-dec	my.sec & pegwit.enc	$5.31 \cdot 10^6$
pegwit-enc	my.pub & pegwit.plain	$8.52 \cdot 10^6$
epic	test_image.pgm	$7.47 \cdot 10^6$
unepic	test_image.pgm.E	$1.64 \cdot 10^6$

TABLE 2.1 *Benchmarks and inputs from the MediaBench [64] suite.*

2.5 Experimental Results

In this section, it is shown by experiments that the BCC and SCC cache reduce the amount of off-chip memory traffic without causing a significant performance degradation. It is furthermore shown how these reductions in off-chip traffic can be translated to energy improvements.

2.5.1 Experimental Setup

As benchmarks, we employed the *MediaBench* [64] benchmarking suite, which consists of a number of audio and video codecs as well as encryption and decryption routines. These benchmarks are representative of embedded multimedia applications. The *MiBench* [32] benchmarking suite, which is specifically aimed at embedded systems and also contains workloads from other application domains, was not available at the time this project was started. Moreover, MediaBench and MiBench suites have several benchmarks in common. The employed benchmarks are listed in Table 2.1.

The *sim-safe* simulator from the SimpleScalar tool set [4] was modified to generate traces containing instruction and data addresses of all executed load and store instructions. These traces were fed to our trace-driven cache simulator,

which generates several statistics, such as to the number of hits and misses and how often a replaced cache line is dirty. From these statistics, the number of transferred bytes can be computed, as well as the miss rate. We also use these statistics to compute the average energy per reference.

The simulated cache size ranges from 256 bytes to 16 kilobytes. All caches have a line size of 32 bytes, are direct-mapped, and employ the write-back policy. The sub-block size of the SCC cache is equal to the word size (32 bits). For the CDT, we have used a direct-mapped structure with 8 to 128 entries. We measured the total amount of traffic between the cache and main memory, including request (address) traffic.

Some benchmarks issue significantly more data references and therefore produce more memory traffic than others. In order to be able to compare the attained traffic savings for different benchmarks, we therefore show the relative changes between the proposed cache organizations and the conventional direct-mapped cache. For miss rates, however, relative differences do not provide proper information. If, for example, in one case the miss rate increases from 1% to 2%, and in another case it increases from 40% to 80%, the performance penalty is far more severe with the latter than with the former. Therefore, one should consider absolute differences between the miss rates of two caches rather than relative differences.

This section also quantifies the possible energy reductions by using BCC or SCC caches compared to conventional caches. While existing tools tools such as *sim-wattch* can be used to determine the energy consumption of experiments performed using SimpleScalar, we have chosen not to use these tools for several reasons. First, these tools are well suited to determine the energy consumption of existing architectures, while we employ a more custom cache architecture. Second, we are mostly interested in the savings obtained by reducing memory traffic, offset against the energy used by the CDT.

2.5.2 Impact of the CDT Size

One of the most influential design parameters for both the BCC and SCC caches is the size of the CDT. The CDT can only detect conflicts if an entry is not replaced in-between two consecutive invocations of the same load/store instruction. Furthermore, the CDT will predict subsequent conflicts as long as the entry remains available. For a very small CDT, both the BCC and the SCC will behave exactly like normal caches. For a very large CDT, on the other hand, these caches may keep predicting conflicts indefinitely, even if

these conflicts only occurred for a small period of time. Naturally, it would be preferable to keep the CDT as small as possible, in order to limit the required area and the energy consumption.

To determine the proper size of the CDT, experiments were conducted using CDT sizes ranging from 8 to 128 entries. Figures 2.2 and 2.3 depict the amount of traffic relative to a normal direct-mapped cache for the BCC and SCC caches, respectively. In both cases, a cache size of 4kB was used. Figures 2.4 and 2.5 depict results from the same experiments with caches of 16kB.

Clearly, when the CDT has too few entries, the opportunities to detect and predict conflict misses become limited, as can be seen from Figures 2.2 and 2.3 for the *adpcm-enc* and *jpeg-dec* benchmarks. A CDT with too many entries, on the other hand, can also be detrimental to traffic reduction, as can be seen for the *mpeg-dec* benchmark in Figure 2.4. In this case, the CDT incorrectly keeps predicting conflicts, leading to a significant increase in traffic in case of the BCC cache. A similar unwanted behavior can be seen for the *g721-dec* and the *g721-enc* benchmarks in Figure 2.2 and 2.3. Overall, a CDT of 32 entries shows to be a good balance, leading to no significant increase in traffic for any benchmark and to proper traffic savings for most benchmarks. Therefore, we will use a CDT of 32 entries in the remainder of this chapter.

Another notable results is that for some benchmarks, such as *adpcm-dec* and *unepic*, the savings appear to be almost independent of the CDT size. In these cases, a CDT of only 4 entries is sufficient to detect most conflicts, leading to a traffic reduction of up to 65% with *adpcm-dec* and over 20% for the *unepic* benchmark when using caches of 4kB.

As the benchmarks used in this chapter have relatively small working sets, they do not stress larger data caches significantly. In fact, from Figures 2.4 and 2.5 it can be seen that both the BCC and the SCC are only able to reduce traffic for less than half of the employed benchmarks. Because of this, and because the number of conflict misses decreases significantly with an increase in cache size, the experiments presented in this chapter are performed using relatively small caches of 1 to 16kB. The fact that these benchmarks perform relatively well on small caches can be verified by looking at the corresponding miss rates, presented in Section 2.5.5.

2.5.3 Traffic

Figure 2.6 and Figure 2.7 depict, for various benchmarks and cache sizes, the relative traffic savings upon a normal direct-mapped cache of the same size, for

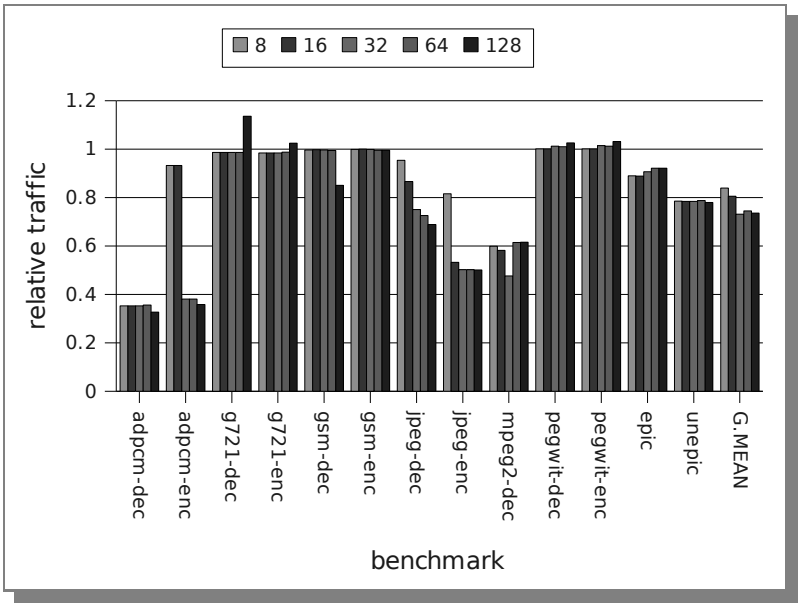

FIGURE 2.2 *Relative amount of traffic produced by a 4kB BCC cache using a CDT of 8, 16, 32, 64, and 128 entries.*

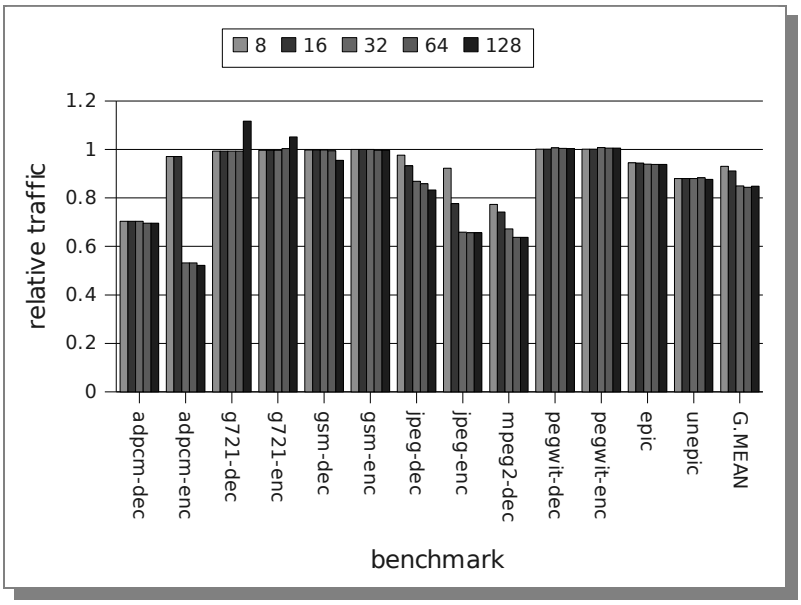

FIGURE 2.3 *Relative amount of traffic produced by a 4kB SCC cache using a CDT of 8, 16, 32, 64, and 128 entries.*

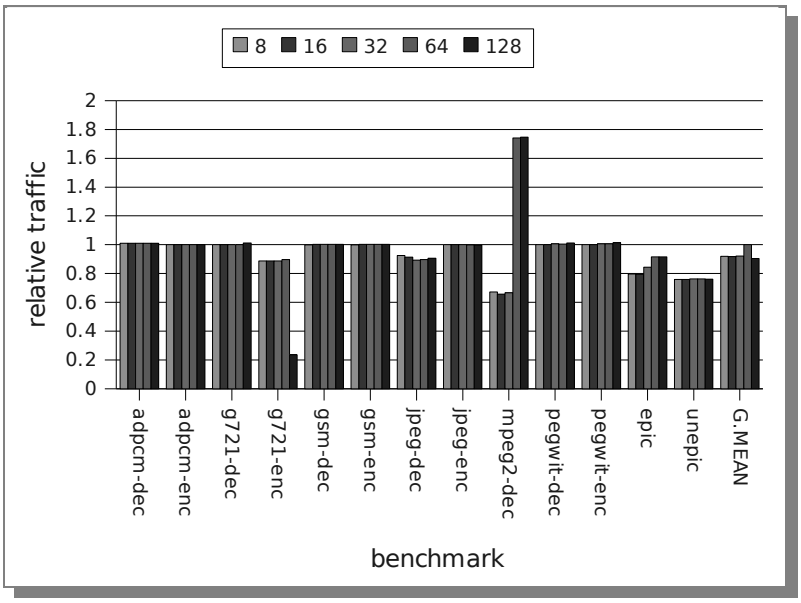

FIGURE 2.4 *Relative amount of traffic produced by a 16kB BCC cache using a CDT of 8, 16, 32, 64, and 128 entries.*

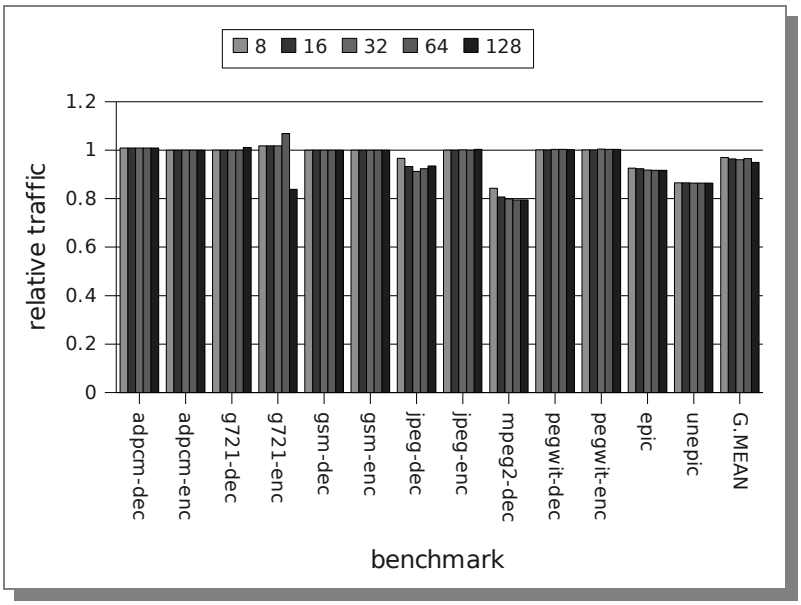

FIGURE 2.5 *Relative amount of traffic produced by a 16kB SCC cache using a CDT of 8, 16, 32, 64, and 128 entries.*

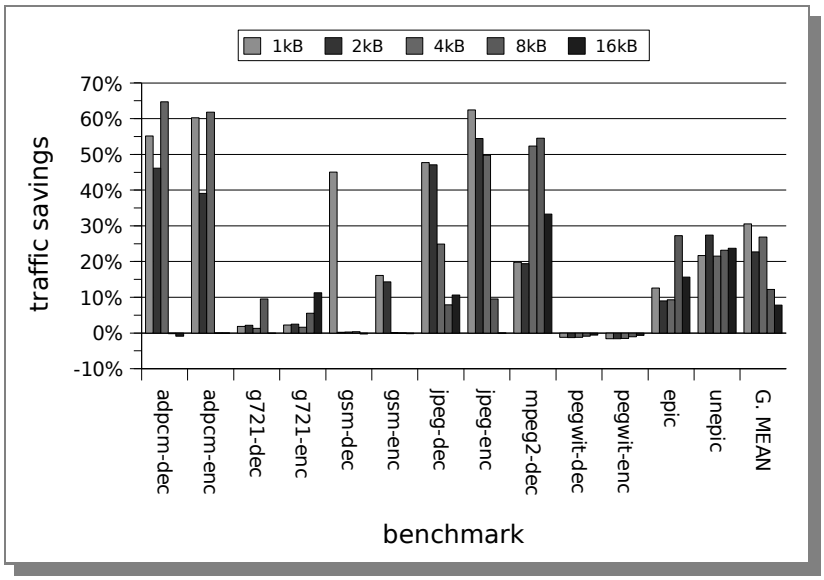

FIGURE 2.6 *Relative amount of traffic saved by the BCC cache, when compared to a conventional cache.*

respectively the BCC cache and the SCC cache. The bars labelled G.MEAN denote geometric means measured over all benchmarks. Geometric means are used instead of averages, since we are comparing relative quantities.

It can be seen that in most cases both the BCC cache and the SCC cache produce significantly less traffic than the normal direct-mapped cache. Specifically, in 55% of all benchmark/cache size combinations, the BCC cache produces at least 5% less traffic than the direct-mapped cache. The SCC cache improves upon the direct-mapped cache by at least 5% in 50% of all cases. Especially when the cache size is small (1, 2, or 4 kB), a traffic reduction of more than 50% can be achieved by the BCC as well as the SCC cache. Averaged over all benchmarks and cache sizes (using a geometric mean), the amount of traffic produced by the BCC cache is 20% smaller than the amount of traffic generated by the direct-mapped cache. For the SCC cache, the average reduction is 11%. For larger cache capacities, fewer benchmarks benefit from the proposed conflict detection technique.

The efficacy of the BCC and SCC cache depend significantly on the type and amount of locality that is exhibited by an application. In a small number of cases, the BCC and the SCC cache actually produce more traffic than the direct-mapped cache. This is particularly the case with the *pegwit-dec* and

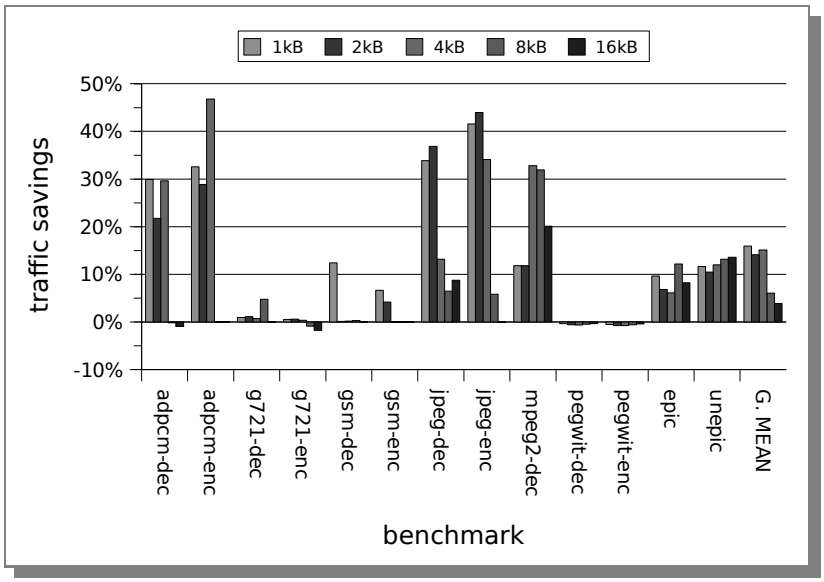

FIGURE 2.7 *Relative amount of traffic saved by the SCC cache, when compared to a conventional cache.*

pegwit-enc benchmarks. In these cases, the benchmarks suffer from decreased performance due to the fact that available spatial locality is not fully exploited. In most cases, however, both cache organizations reduce the amount of off-chip traffic considerably and, hence, the amount of energy consumed by an application. We further observe that although the BCC cache produces the least amount of traffic for most benchmarks and cache sizes, it will also increase traffic more than the SCC cache when incorrectly predicting conflicts.

2.5.4 Energy Reduction

By reducing the amount of data that is transferred between on-chip and off-chip memories, a significant amount of energy can be saved. However, this has to be offset against the energy consumed by the CDT, which is accessed on every load or store instruction. The difference in energy consumption between the conventional direct-mapped cache and the BCC and SCC can be estimated by comparing the energy saved by the traffic reduction to the energy consumed by the accesses to the CDT. In both cases, these numbers have to be multiplied by the appropriate energy cost.

Since the CDT closely resembles a cache, the energy consumption of accessing the CDT can be estimated using CACTI [90]. Using CACTI 4.1, a 128-entry direct-mapped cache with a line size of 8 bytes was modelled, using the 180 nm technology node. According to CACTI, this results in a cost of approximately 2.14pJ per access. It should be noted that we use a CDT of 32 entries in these experiments and that the CDT actually employs a line size of 4 bytes. However, the minimum line size in CACTI is 8 bytes, and the energy cost of accessing the CDT is therefore over-estimated.

To model the cost of transferring data between on- and off-chip memories, we use a lower bound by only calculating the energy involved in bit switches on the bus. The dynamic energy consumed by a single bit switch can be expressed as:

$$E = C \cdot V^2,$$

where C denotes the bus capacitance and V denotes the employed voltage. This voltage is assumed to be $1.8V$, which is the minimum voltage used in commodity memories available on the market today [38]. The capacitance of a single metal wire between on- and off-chip is estimated at $20pF$ for 180 nm technology. This is the same value as is used by Basu et al. [6], which in turn is based on data provided in [12].

We assume that, when transferring data, on average half the bits on the memory bus switch. As a result, the energy involved with these transfers can be estimated by:

$$E_{bus} = 4 \cdot \textit{traffic} \cdot C \cdot V^2,$$

where *traffic* denotes the total number of bytes transferred across the bus. For example, a transfer of one 4-byte word across the bus will consume $4 \cdot 4 \cdot 20 \cdot 10^{-12} \cdot (1.8)^2 = 1.036nJ$, whereas transferring a 32-byte cache line will cost $8.29nJ$.

The dynamic energy consumption of both the CDT and the memory bus will be lower when a smaller technology is used. However, we expect that this will benefit the CDT at least as much as the memory bus, and that the presented results are a lower bound for smaller technologies as well.

The energy consumed by the CDT is found by multiplying the number of memory accesses by 2.14pJ. This results in the graphs depicted in Figures 2.8 and 2.9, which depict the energy *per reference* consumed in the memory bus and CDT for cache sizes of 1kB and 4kB, respectively. These results include data for a conventional cache, the BCC, and the SCC.

As expected, the energy reduction of the proposed cache organizations de-

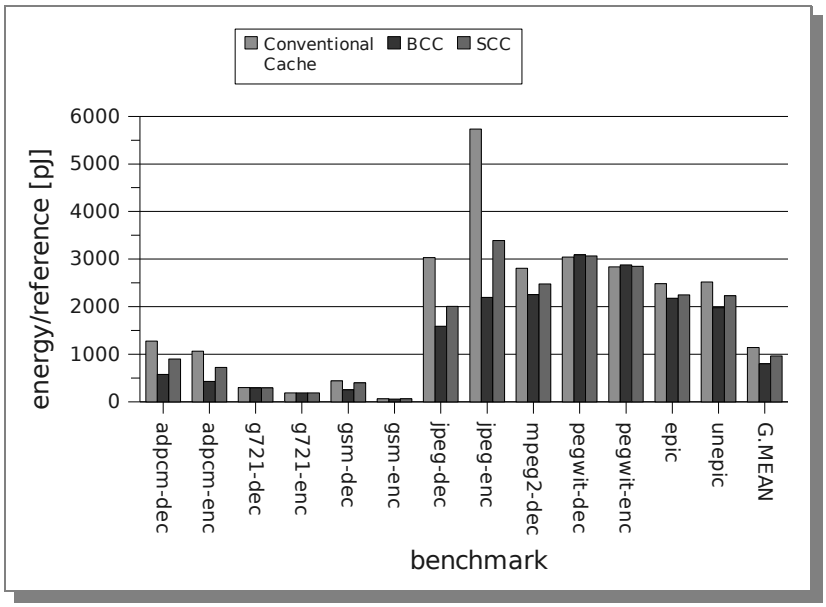

FIGURE 2.8 *Energy per reference in the off-chip memory bus and CDT for conventional, BCC, and SCC caches of 1kB.*

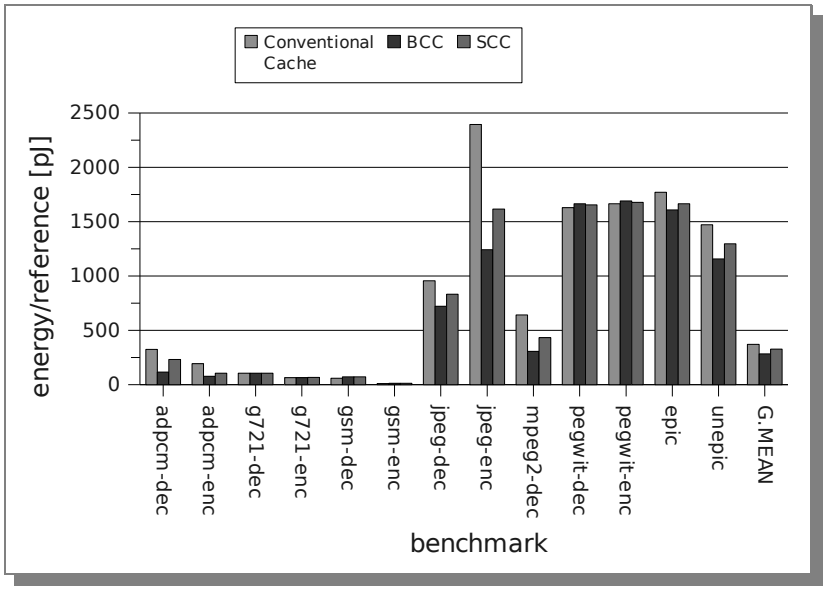

FIGURE 2.9 *Energy per reference in the off-chip memory bus and CDT for conventional, BCC, and SCC caches of 4kB.*

creases with increasing cache size. Although accessing the CDT on every memory reference adds to the total energy consumption, the reduction in traffic on the off-chip memory bus outweighs this. In fact, for the cases in Figures 2.6 and 2.7 where either the BCC or the SCC show a significant reduction in traffic, a similar energy reduction per reference can be found in Figures 2.8 and 2.9. However, where these caches in Figures 2.6 and 2.7 show an increase in traffic, the increase in energy in Figures 2.8 and 2.9 is far less significant. The reason for this difference is that while the relative amount of traffic is increased significantly, this increase is still not very significant in absolute terms. Due to a low cache miss-rate in the baseline cache, the energy consumption per reference will increase, but it will not be very significant when compared to the total number of memory references.

In general, Figures 2.8 and 2.9 show that the BCC is able to reduce energy consumption by approximately a factor of two for benchmarks like the *adpcm* and *jpeg* applications. The SCC cache also reduces energy consumption in these cases, but generally to a lesser extend. Only for the *pegwit* benchmarks there is a slight increase in energy, in which case the increase is less than 2.1%.

2.5.5 Impact on Execution Time

The results discussed above show that the BCC and the SCC cache reduce the amount of off-chip memory traffic. However, if they would increase the miss rate (and therefore execution time) significantly, no energy reduction would be achieved. To validate this, Figure 2.10 and Figure 2.11 depict the miss rates generated by the direct-mapped cache, the BCC cache, and the SCC cache, for cache capacities of 1kB and 4kB, respectively. In most cases the miss rates of the BCC cache and the SCC cache are comparable to the miss rate of the direct-mapped cache. In some cases, however, the miss rate of the BCC cache is significantly larger than the miss rate of the direct-mapped cache. In one case, for *mpeg2-dec* using a cache size of 1kB, the miss rate increases from 26% to 34% when using the BCC cache. The largest increase in miss rate for the SCC cache, from 16% to 20%, is found with the *epic* benchmark. Furthermore, in several cases the BCC cache performs better than the direct-mapped cache. On average, they perform equally.

We conclude that the BCC cache is the most efficacious cache structure. It generates significantly less off-chip traffic than a direct-mapped cache, while performing equally as well. However, for some applications the SCC cache may be preferable, especially when performance is critical.

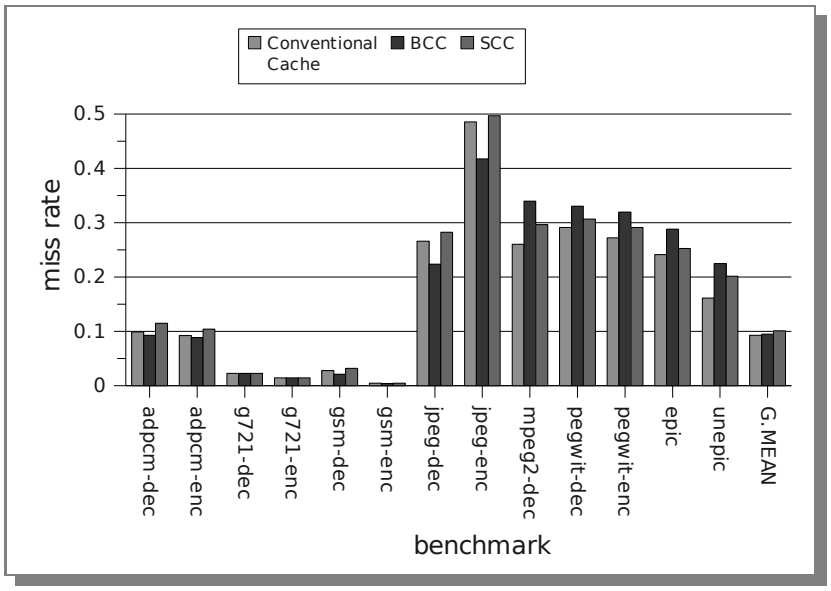

FIGURE 2.10 *Miss rates for conventional, BCC, and SCC caches of 1kB.*

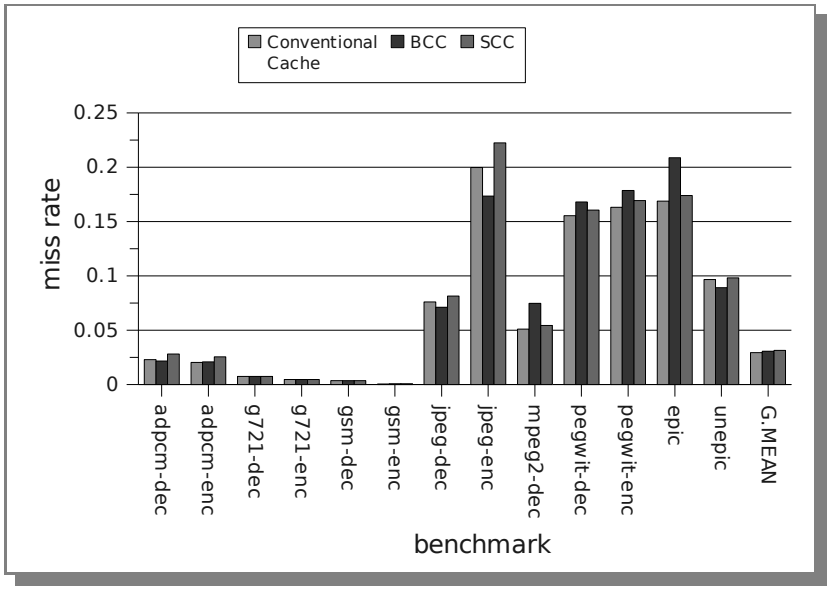

FIGURE 2.11 *Miss rates for conventional, BCC, and SCC caches of 4kB.*

2.6 Conclusions

We have proposed a technique which can detect and is often able to reduce the negative effects of recurring conflict misses. The proposed technique employs a small structure called the *Conflict Detection Table* (CDT). This conflict detection mechanism does not require much logic and as as result does not significantly add to the total energy consumption. Furthermore, since the CDT only needs to be consulted on a cache miss, it will not increase the cycle time. Consequently, it can easily be applied to on-chip caches that lack associativity. We have proposed two cache organizations that employ the CDT: the *Bypass in Case of Conflict* (BCC) cache and the *Sub-block in Case of Conflict* (SCC) cache.

The BCC cache decreases the amount of produced traffic on average by 20% with a maximum of 65%, compared to the conventional direct-mapped cache. It was also shown, however, that this cache sometimes increases the amount of traffic, which may happen when bypassing the cache for data which is accessed repeatedly. Furthermore, also the miss rate can suffer badly from inefficiently bypassing the cache. The SCC cache also decreases the amount of produced traffic considerably in most cases. On average, the reduction achieved by the SCC cache was 11% with a maximum of 47%. Only in a few cases, the BCC or SCC caches produced more traffic than a conventional direct-mapped cache. Furthermore, these increases are small. In addition, the miss rate of the SCC cache is never considerably higher than that of the conventional direct-mapped cache. While the BCC produced a higher miss rate than the SCC cache in several cases, for some benchmarks it also showed to decrease the miss rate compared to a conventional direct-mapped cache. We conclude that using the CDT to fetch sub-blocks instead of whole cache lines, significantly decreases the amount of produced traffic, and hence also the amount of energy consumed. Whether the BCC or the SCC cache is more effective, is largely determined by the application.

To further improve the effectiveness of the proposed cache organizations, the conflict detection mechanism can be extended with counters to record the number of bypass or sub-block predictions. This could be used to limit the time a certain prediction is remembered, which could avoid recurring miss-predictions.

3

Memory Copies in Multi-Level Memory Systems

Data movement operations, such as the C-style memcpy function, are often used to duplicate or communicate data. This type of function typically produces a significant amount of off-chip traffic, while it hardly requires any processing power. In this chapter, we try to reduce the energy consumption in a multi-level memory hierarchy by providing an energy efficient solution targeted at specifically this type of function. The proposed solution implements a hardware copy engine in several levels of the memory hierarchy, and provides a method to dynamically select the appropriate level to perform the copy. Reducing the required amount of communication in the off-chip memory bus not only saves energy, it can also significantly improve performance for memory-bound applications.

Most of the material presented in this chapter has been previously published in [22].

3.1 Introduction

Block copy operations, such as the C-style memcpy function, are often used to duplicate or communicate data in operating systems [9, 15], message pass-

ing systems, webservers [57], and database applications [85]. Desktop and embedded applications also often use this function, but less often and mostly for smaller blocks of data. Although a memory copy is computationally one of the least intensive functions, current software implementations still require all data to be loaded into registers and to be written back to main memory afterwards. This is wasteful, since it requires significant memory bandwidth and hardly any processing power. As a result, this can render the CPU idle, waiting for data from the congested memory system. Furthermore, current software implementations store the source data and often also the destination data in cache. Especially with large copy operations, this data will be evicted from the cache before the next use. This is often referred to as *single-usage cache pollution* [79]. Due to the increasing disparity between the speeds of processors and memories, applications are expected to spend an increasingly large fraction of their time in memory-bound functions like memcpy. For operating systems, being very memory and I/O intensive, this problem is even more severe than for user applications [75].

This chapter is targeted at reducing the amount of traffic between the processor and off-chip memory for functions such as memcpy. The goal of this traffic reduction is foremost to reduce energy consumption. However, for memory-bound applications, reducing the amount of data traffic can also provide a significant speedup.

The most common way to reduce off-chip traffic is by keeping the required data close to the CPU. This is commonly done with caches, and by adapting their designs, parameters and/or policies to specific application behavior. One of the key problems with this approach is, however, that there is no or very limited possibility for programmers to influence the dynamic behavior. Hence, the only possibility to reduce memory traffic for the memcpy function is by avoiding to allocate the cache lines corresponding to the destination addresses. This can be accomplished on some architectures by special store instructions, such as the non-temporal store instructions available in the Intel SSE instruction set extension [39].

Another approach is to employ scratchpad memories [5], in which transfers between on- and off-chip memories are programmed explicitly. The downside of this is, however, that all memory transfers have to be programmed and that it is often not known in advance which data is needed and if it will fit in the on-chip memory. *Direct memory access* (DMA), another popular technique to transfer data is not always suitable for use in memcpy due to the significant time required to initiate a transfer, as explained in [102]. Our approach is

different in that we perform block memory operations close to the actual data. More precisely, we propose to copy the data in the highest memory level where it is found.

The main contribution of this chapter is that we propose to perform memory copies in hardware copy engines and to have these copy engines implemented in several levels of the memory hierarchy. Furthermore, we show how this technique reduces the amount of traffic between the CPU and the off-chip memory system, and therefore reduces energy consumption and improves performance.

This chapter is organized as follows. Section 3.2 discusses related work. The technique to perform memory copies by using caches is explained in Section 3.3. The same section also discusses the limitations of performing the copies on-chip, and explains how these limitations can be resolved by implementing this functionality in several memory levels. In Section 3.4, the experimental results are presented and analyzed. Section 3.5 concludes the chapter and presents some directions for future research.

3.2 Related Work

Several other researchers focussed on optimizing the memcpy function. Calhoun et al. [9] showed that operating systems spend a significant amount of time copying data between buffers, and proposed to dynamically select the proper memcpy algorithm based on the size of the data as well as on predicted reuse, and to use this in a system with write combining buffers and non-temporal stores. While being able to reduce cache pollution and hence improve performance, this can only reduce the amount of off-chip traffic by not fetching the cache line associated with the destination addresses. Piquet et al. [79] proposed a general method to reduce single-usage cache pollution by bypassing the L2 cache. For memory copies, however, this method still requires the data to be transferred via registers. Duarte et al. [26] proposed to store pointers to copied data in a separate table to avoid having several copies of the same data in the cache. Future read accesses to the copy are dynamically diverted to the original source. While this method improves the performance of some kernels, the actual copy is only delayed, and hence there is no reduction in traffic. In fact, it is assumed that data is always available in the cache and the fact that it has to be written back to memory at a certain point is not considered.

In all these works, the data that is copied still has to travel from the main

memory to the CPU and back. Zhao et al. [102] recently showed the performance benefits of having data movement functions implemented in a hardware *Copy Engine* (CE). They presented a thorough overview of the possible implementation choices. Unlike our approach, however, these authors propose to implement the copy engine *either* close to the on-chip *or* close to the off-chip memory. In our approach, the copy engine is placed next to the on-chip *as well as* next to the off-chip memory. If (part of) the source is located in on-chip memory, the system will decide to perform the duplication there. If no part of the data is found on-chip, the operation is diverted to off-chip logic. In this way, the amount of off-chip traffic is minimized by only sending the corresponding addresses to the off-chip logic, instead of sending all data to the CPU and back.

Khunjush and Dimopoulos [55] found that copy operations are the main contributors to delivery latencies in message passing environments. These authors propose to use a specialized *network cache* and instructions to manage this cache. They furthermore introduce new policies to determine when to transfer messages to the data cache. Experiments show that these extensions, compared to other methods, reduce the access latency of received messages without polluting the data cache. Whereas these authors focus on reducing copy overhead in message passing environments, we target to improve performance on memory copies in general.

Our proposed idea shows some similarities with *Processing-In-Memory* (PIM) techniques, such as IRAM [76], Active Pages [74], and FlexRAM [51], since we also try to offload simple tasks to the memory system. However, in those approaches, the processing is performed in the lowest level memory. In our case, it is dynamically decided where to perform the copy, and this can happen anywhere in the memory system.

3.3 Memory Copies in Multi-Level Memory Systems

Software implementations of memcpy generally use an unrolled loop containing a series of loads followed by a series of stores. To maximally exploit data parallelism, some systems use optimized implementations of memcpy, using multi-word registers and corresponding loads and stores. However, most systems incur a significant penalty if the source or destination of a copy are not properly aligned to the granularity of memory accesses in the system. In this case, software implementations either have to use loads and stores of half word

or smaller data types, or they must load consecutive data into two registers and write back by shifting and merging their contents. Alvarez et al. [1] proposed an extension to the Altivec SIMD instruction set to support unaligned memory operations.

In this chapter, we present a novel principle for duplicating or moving data efficiently in a multi-level memory system is presented. The method is based on the observation that copied data is not always required in the near future, and that it can be harmful to performance to have several copies of the same data in small memories, especially if the data is large. First we discuss the organization of the copy engine, i.e. the hardware that performs the actual copy. Then, we explain the instruction that uses the copy engine and the potential performance improvements due to a single copy engine placed either next to the L1 cache or to the L2 cache. This is followed by a discussion on the limitations of copy engines implemented on the processor die. Finally, a dynamic version of the copy engine is discussed, where a copy engine is placed in several consecutive cache levels.

3.3.1 Copying Using Copy Engines

In this chapter we propose to use copy engines that copy a block of data of the same size as a cache line. The central part of this copy engine is a shift register twice the size of a cache line. The copy engine is placed next to the cache, as it reuses existing logic for reading and writing data from and to the cache.

The copy engine can be instructed to copy data by issuing a special *movblk* instruction. This instruction takes two operands: the starting address of the source and the starting address of the destination of the copy. While there are no restrictions on the address pointing to the source to be copied, the destination address should be aligned with a cache line (i.e.: it should point to the first byte in a cache line). The movblk instruction copies a block of data the size of a cache line and the destination of the copy is always exactly one cache line.

Figure 3.1 schematically depicts how a copy is performed in copy engine connected to the L1 data cache. In this example, the source is not aligned to a cache line boundary, and therefore the copy engine needs to fetch two consecutive cache lines and combine these into one. The copy engine first loads a cache line into the left side of the buffer. The second cache line is loaded into the right part of the buffer. The buffer is then shifted a corresponding number of places to the left, and the left part is finally written back to the cache. The time required for performing a copy is the total of the delays involved with

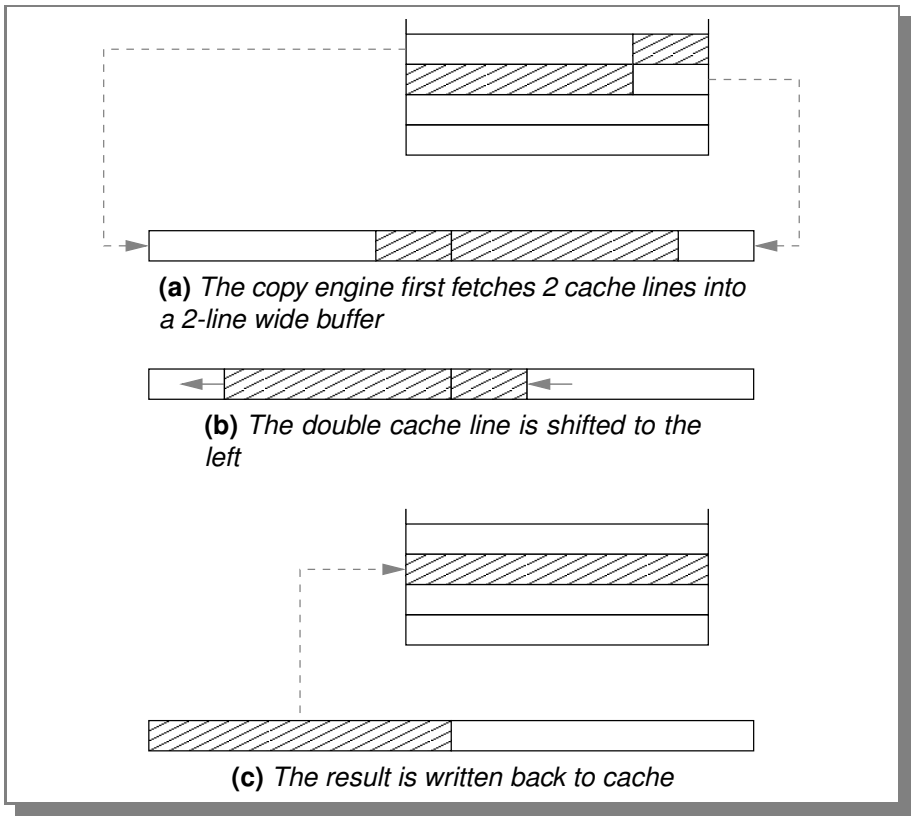

FIGURE 3.1 *Schematic description of how a copy is performed by a copy engine*

reading the source cache lines, writing the destination line, and one additional cycle for the shift buffer.

Performing copies in blocks that correspond to exactly one destination cache line has several advantages:

- By performing the copies in larger blocks, the speed of data copies can be improved significantly compared to using registers. This improvement, however, decreases for systems with larger registers. Moreover, as software implementations can be pipelined efficiently and because the memory transfers may very well be the most time consuming part, this speedup might not be significant.
- When copying in the L1 cache the destination cache line can be allo-

cated without fetching data, since the whole line is overwritten. Many caches that employ write-back also fetch the corresponding cache line on a write miss, as this avoids the need of maintaining multiple valid bits per cache line. Since our copy engine always overwrites the whole cache line, there is no need to fetch the old data on a write miss. This way, we avoid fetching the destination cache line while requiring only one valid bit per cache line.

- By reusing the existing logic for reading from and writing to the caches, our copy engines can be implemented with little additional hardware.
- By using blocks of the same size as a cache line, we make sure that the source data is located in at most two different cache lines. Using larger blocks may significantly complicate the copy process as the source data can be spread over more than two levels of the memory hierarchy. When part of the source data is found, our implementation assures that at maximum one other part needs to be fetched. Whereas larger copies can be done by performing several copies, data smaller than a cache line cannot be copied by this copy engine.

An additional advantage of this copy engine is that we do not incur a penalty when the data is unaligned. When the source and destination addresses are differently aligned, the source data has to be shifted before it can be written to the destination address. In software implementations, this is often done by loading half-word or even smaller data types and combining these before writing the result. In the copy engines proposed in this chapter, the time required for a copy is independent of the data alignments, aside from the special case where both the source and destination addresses are aligned to a cache line boundary. In this special case, the copy is performed even faster, since we only need to read from the cache once and there is no need to shift the data.

To avoid hazards in an out-of-order processor, the movblk instruction is not allowed to bypass other memory instructions and vice versa.

3.3.2 Limitations of On-Chip Copy Engines

The simplest implementation of the copy engine presented in Section 3.3.1 is to connect it to the L1 data cache. In this case, however, the copy is performed on-chip and can severely suffer from delays in the memory system if the data needs to be fetched from off-chip memory. Also, most works discussed in

Section 3.2 provide software and/or hardware solutions to improve the performance of memory copies on the processor itself. These solutions not only do not reduce the amount of data traffic, they will also not be able to provide significant performance improvements in future years, as explained below. The fundamental problem with all solutions that perform the copy on the processor is that they can only provide performance improvements as long as the time taken to perform the copy in a baseline system is significant, and cannot be hidden by the latency involved in transferring the data to the chip and writing back results. For modern processors, this is only true if the data already resides on-chip. Copying data on-chip always involves at least three steps: fetching the source data, performing the copy, and writing back the results. Although the results may not always need to be written back directly, this cannot be delayed indefinitely. Furthermore, for copies exceeding the size of the on-chip cache, writing back data is always an integral part of the copy process. The time taken to copy D bytes of data is given by:

$$t_{memcopy}(D) = t_{in}(D) + t_{copy}(D) + t_{out}(D), \tag{3.1}$$

where $t_{in}(D)$ denotes the time required to fetch D bytes of data, $t_{copy}(D)$ denotes the time required to actually copy this data, and $t_{out}(D)$ denotes the time required to write back the D bytes of copied data. It should be noted that in many modern systems, memory transfers are performed in parallel to computations. This is not taken into account in this section, as we only aim at establishing an upper bound on the time taken by memory copies. By grouping the terms related to data transfers into one term $t_{data} = t_{in} + t_{out}$, Equation (3.1) is rewritten as:

$$t_{memcopy}(D) = t_{data}(D) + t_{copy}(D).$$

Using this expression, the maximum possible improvement in performance is determined by assuming the time required to copy the data on the processor can be decreased to a negligible amount. The upper bound on the speedup for copying D bytes of data is then expressed as:

$$speedup_{upb}(D) = \frac{t_{data}(D) + t_{copy}(D)}{t_{data}(D)}. \tag{3.2}$$

By relating the time required for transferring data back and forth to the time required to perform the actual copy as:

$$\delta(D) = t_{copy}(D)/t_{data}(D), \tag{3.3}$$

Equation (3.2) is rewritten as:

$$speedup_{upb}(D) = 1 + \delta(D). \tag{3.4}$$

Most processors available today operate at a much higher speed than the interface to external memory can sustain. In fact, for many currently available processors, this difference is more than an order of magnitude. Whereas modern processors can perform multi-word operations at a speed of several gigahertz, modern memories have a theoretical peak bandwidth a few gigabytes per second. The peak memory bandwidth, however, can never be attained if the accesses cross a page-boundary. Furthermore, modern processors can easily sustain an IPC of 1 or above for only reading and writing data that is available in the cache. Therefore, it can be safely assumed that $\delta(D)$ is indeed far less than 1.

Even more importantly, since the processors increase in speed much faster than memories, the disparity between these speeds also grows each year [71]. Even though the clock frequencies of processors have not grown at the same pace in the past years, this is still true since processor manufacturers are employing increasingly more parallelism, for example by using *Simultaneous Multi-Threading* (SMT) or by putting multiple processing cores on the same chip. Since both processors and memories experience an exponential growth in speed, the difference between these two speeds also grows exponential. This implies, that even if an improvement to on-chip copying provides a significant improvement now, the technique will suffer from diminishing returns in future years.

For reasonable modern systems (i.e. $\delta \ll 1$), no solution that performs the actual copy on chip will ever provide a significant performance improvement, and any attained improvement will decrease exponentially over time. In fact, the only solution to reduce the amount of memory traffic, and the only fundamental solution to improving the speed of memory copies, is to perform the operation much closer to the data. However, performing the copies close to the main memory can hurt performance significantly if the source data is already available in the on-chip caches. Furthermore, in this case it is also necessary to flush any destination data from on-chip caches. Therefore, we propose a dynamic solution where special hardware is included in several layers of the memory system, as will be explained in the next section.

3.3.3 Dynamic Copy Engines

The copy engine presented in Section 3.3.1 can be integrated in any level of the memory hierarchy. When performing a copy of a small amount of data that was recently used, it makes sense to have a copy engine connected to the L1 data cache. When the copied data is not in the L1 cache or when the data is

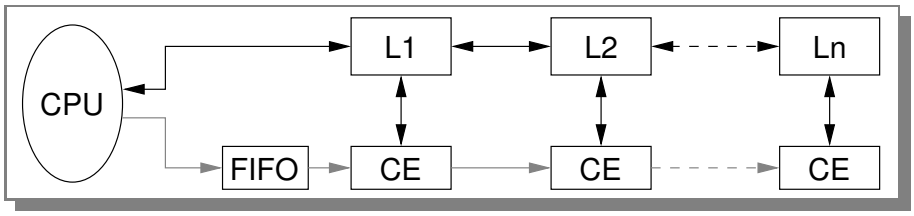

FIGURE 3.2 *Schematic design of how the copy engines are implemented in several levels of the memory hierarchy*

too large to fit in this cache, it makes more sense to perform the copy engine in a lower level cache.

When data is available in the cache close to the copy engine, the maximum attainable speedup is proportional to the difference in size of a cache line and the size of the registers. In case the data is not in the cache, however, the speedup will be severely limited by delays and stalls in the memory system, as was already discussed in the previous section.

In fact, a copy engine per se does not reduce the amount of traffic in the memory system, and as such will not provide any speedup if the copy performance is limited by memory bandwidth. In order to minimize the amount of off-chip traffic, and at the same time improve performance and reduce energy consumption, we propose the *Dynamic Copy Engine* (DCE). In the DCE, a copy engine is implemented in several consecutive levels of the memory hierarchy. This is schematically depicted in Figure 3.2. The first copy engine is employed with a FIFO buffer to allow pipelining several movblk instructions. The DCE follows the following procedure, to perform the copy in the most appropriate memory level:

- After receiving the operands, the first copy engine first checks if the source of the memory copy is present in the current memory level. In case the source is also aligned on a cache line boundary, this implies a singe cache access. Otherwise, two cache accesses are needed.

- If one or more lines with source data are found in this cache, the copy is performed by the associated copy engine. In case the source is not aligned on a cache line boundary and one of the cache lines is not present, it is fetched from the next cache or lower level memory. When the copies are performed in blocks corresponding to the cache line, the

destination cache line can be allocated without fetching the data. However, as lower level caches often have larger cache lines, this may only be possible for L1 caches.

- If no source data is found in this cache, another check is performed to find the destination cache line. If found, this line is invalidated since it will be updated in a lower memory level. The instruction is then diverted to the copy engine associated with the next level memory, where the same procedure is repeated until the last copy engine is reached. The last copy engine will always perform the copy, irrespective of whether data is available or not.

This procedure is also explained by the pseudocode depicted in Figure 3.3. In this code, the calculation of the source alignment in line 4 can be performed in parallel to fetching the first cache block in line 3. While a pipelined cache may allow a second access (line 8) to proceed while another one is still in progress, we conservatively assume that the DCE starts a possible second cache access only after the first one has completed.

Instead of moving the source data from a lower level memory to the CPU and the destination back from CPU to memory, we propose to move the *operation* down the memory hierarchy and to perform the copy close to the memory where the source is found. This way, the amount of traffic is reduced significantly by only sending addresses to the correct copy engine, instead of transferring the data from lower level memory to the CPU and back.

The amount of traffic saved by this approach depends on the address size, the size of the cache lines, and the number of transfers required by the baseline implementation. This last number depends on whether or not the baseline cache employs the fetch-on-write policy. When a write produces a miss in a cache that employs the fetch-on-write, the corresponding cache line is fetched from the next memory level. With the no-fetch-on-write policy, this data is stored without fetching the neighboring data. In the latter case, however, dirty bits should be used on a granularity equal to the smallest data type that can be stored. A more detailed explanation of cache write policies can be found in Section 4.1.

Caches that employ write-allocate generally require three transfers per copy. For caches that employ write-no-allocate, two transfers per copy is the minimum. The same is true for (write-allocate) caches that employ write-combining write buffers and support non-temporal stores. More formally, if neither the source or destination data is already in the on-chip cache, the DCE produces

```
DYNAMIC-COPY-ENGINE(src, dst)
    ▷ LINESIZE equals the cache line size in bytes
    ▷ B, B1, and B2 are cache line sized variables
 1  mask ← LINESIZE − 1
 2  S1 ← src & ¬mask
 3  B1 ← READ-FROM-CACHE(S1)
 4  α ← src & mask
 5  if α = 0
 6     then B2 ← B1
 7     else S2 ← (src +31) & ¬mask
 8          B2 ← READ-FROM-CACHE(S2)
 9  if B1 = NOT_FOUND
10     then
11          if B2 = NOT_FOUND
12            then INVALIDATE-CACHE-LINE(dst)
13                 COPY-IN-NEXT-LEVEL(src,dst)
14            else B1 ← FETCH-FROM-NEXT-CACHE(S1)
15     else
16          if B2 = NOT_FOUND
17            then B2 ← FETCH-FROM-NEXT-CACHE(S2)
18  B ← (B1 ≪ (α · 8)) | (B2 ≫ ((LINESIZE − α) · 8))
19  STORE-CACHE-LINE(B, dst)
```

FIGURE 3.3 *Pseudo-code for the dynamic copy engines.*

$2 \cdot A$ bytes of off-chip memory traffic, where A denotes the size of address in bytes. With L denoting the cache line size, a software approach produces $2 \cdot (A + L)$ bytes of traffic and even $3 \cdot (A + L)$ if the destination cache line is fetched as well. Using W to relate the address size to the cache line size as $W = L/A$ and c_{base} to denote the number of transfers required by the software implementation (2 or 3), the relative amount of traffic saved by this approach is:

$$\textit{relative_traffic}_{saved} = 1 - \frac{2 \cdot A}{c_{base} \cdot (A + L)} = 1 - \frac{2}{c_{base} \cdot (1 + W)}. \quad (3.5)$$

For a 32-bit system with a cache line size of 32 bytes ($W = 8$), this would imply a saving in data traffic of 89% or 93%, depending on whether 2 or 3 transfers are required in the baseline system.

As in Section 3.3.2, we can also derive an expression for the maximal attainable speedup for the DCE. Here, we assume the baseline system can actually perform the copy itself at the same speed as the copy engines presented in this chapter. It is furthermore assumed that the baseline system requires only two transfers to copy one cache line, as in the previous section. Using the same terminology as in Section 3.3.2, the maximum speedup of the DCE upon the baseline software implementation can be expressed as:

$$\mathit{speedup}_{upb}(D) = \frac{t_{data}(D) + t_{copy}(D)}{t_{addr} + t_{copy}(D)}, \tag{3.6}$$

where t_{addr} denotes the time required to transfer 2 addresses. Using δ from Equation (3.3) to relate the time required for transferring data to the time required for copying and by relating the time required for transferring a cache line to the time required to transfer an address as $w = t_{addr}/t_{data}$, Equation (3.6) can be rewritten as:

$$\mathit{speedup}_{upb}(D) = \frac{1+\delta}{w+\delta}. \tag{3.7}$$

Figure 3.4 depicts the maximum speedups for copying data that is not initially available in an on-chip cache. This graph is constructed using both the above expression for the maximum speedup of the DCE, and Equation (3.4) for the maximum speedup of on-chip copy engines. In this figure, δ is varied from 0 to 1, denoting respectively the case where data processing is indefinitely faster and the case where data processing is as fast as transferring data. For any reasonable system, δ is smaller than 0.5. Hence, for copying data that is not resident in on-chip caches, the speedup attainable by performing the copy on-chip is less 50%, and will significantly decrease in future years. For the DCE, on the other hand, the maximum speedup is typically more than 50%.

There are several other reasons why the DCE improves performance and reduces memory traffic. When the source data is not found in the a cache, it is also not allocated in this cache. This can reduce cache pollution if the copied block is large, and/or if the data associated with the source address is not required in the near future. Furthermore, when a data cache contains the cache line corresponding to the destination of a copy, but the copy is performed in a lower level, the cache line can be discarded even if it contains 'dirty' data. Because the data is to be overwritten, there is no need to write back the old data.

Hazard control in dynamic copy engines is done in a similar way to the description in Section 3.3.1 for the normal copy engine: memory instructions are

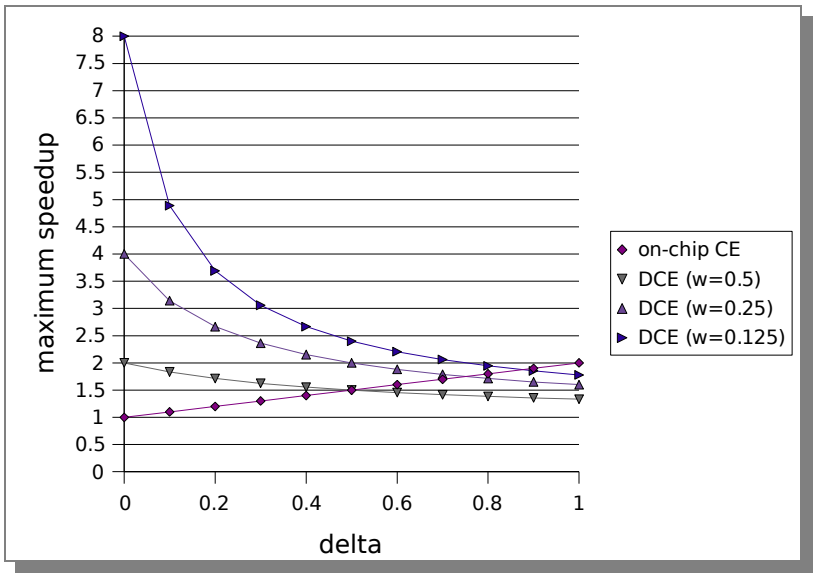

FIGURE 3.4 *Maximum speedups for copying non-resident data using an on-chip copy engine and the DCE*

not allowed to bypass the movblk instruction and vice versa. However, with the DCE, load/store instructions are allowed to execute before the movblk has finished. While a copy engine is executing, the corresponding cache is locked, however. This implies that while a copy is being performed in L2, the processor can safely read and write to and from the L1 cache in the mean time.

Caches often employ increasingly larger cache lines for lower level caches. Therefore, it may occur that the copy engines of lower level caches copy data in smaller blocks than the cache line. In these cases, the source data is still guaranteed to be located at no more than two different cache lines. The destination data, however, will no longer occupy a whole cache line. This implies that, when copies are performed in lower level caches, it may be necessary to fetch the cache line corresponding to the destination address.

3.3.4 Dynamic Copy Engine with Non-Temporal Fetching

When the source is not aligned on a cache line boundary and the copy engines needs to perform two fetches, it may happen that the first cache line is found while the second is missing. In the method described above, the missing cache line would then be fetched from the lower memory level and stored into the

cache. However, when this instruction is part of a group of several movblk instructions used to copy a larger block of data, the next movblk instructions will all find part of their source data in this cache. Since a large part of a copied block may actually not have been in the cache at the time of the first copy, this could severely reduce performance. An example of this process is depicted in Figure 3.5. Assume memcopy is instructed to copy a memory block of several cache lines in size starting at address 2. The memcopy function issues movblk instructions to copy the data in blocks starting at address 2, at address 10, at address 18, etc. As depicted in Figure 3.5a, words 2–9 are found in the first cache and are copied there. In Figure 3.5b, words 10–17 are to be copied. In this case, however, the copy engine finds 10–15 in cache, but 16 and 17 are not found. After fetching these, the copy proceeds in the first cache. Due to fetching 16 and 17, however, the first part of the next block (18–25) is now also found in cache, as is depicted in Figure 3.5d. Although the data at addresses 18–25 was originally not in cache, this block will still be copied in this cache due to fetching part of the previous block. The same problem occurs for every next block.

To make sure that one movblk instruction does not influence the decision on the next one, we propose a variation of the DCE, which we call the *Dynamic Copy Engine with Non-Temporal Fetching* (DCE-NT). In the DCE-NT, missing source data is never stored into the cache. When a copy is performed and the corresponding cache is missing part of the source data, this data is fetched from the next level but is only sent to the copy engine. This way, the number of cache lines may be copied in the current level, while other ones may be copied in a lower memory level if they were not already present in the current memory level.

3.4 Experimental Results

This section first describes the models, tools, and benchmarks used for evaluating the proposed copy engines. Thereafter, two different sections present experimental results from a memcopy micro-benchmark and a more extensive benchmark adapted from a real TCP/IP stack.

3.4.1 Experimental Setup

The experiments described in this chapter were performed using *sim-outorder* from the SimpleScalar tool set [4], which was substantially modified to in-

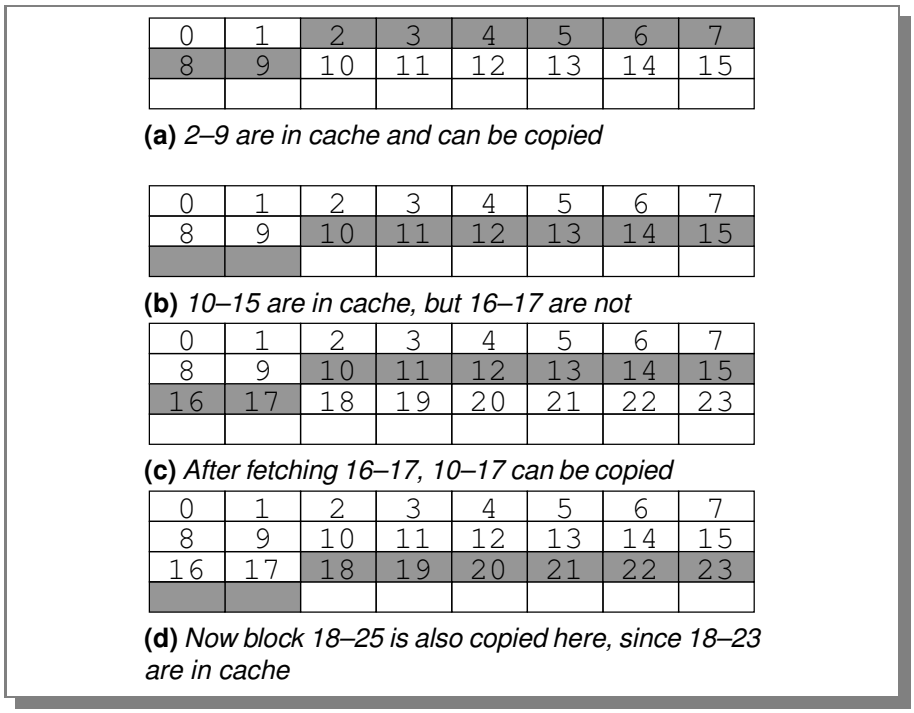

FIGURE 3.5 *Example of how one missing block may cause all consecutive memory blocks to be copied in the same cache*

clude the presented copy engines and to support the movblk instruction. We also modified the memcpy function to check for sizes and alignments, and to use the movblk instruction to copy data whenever possible. Furthermore, the linker was instructed to redirect calls to memcpy to our modified version of the memcpy function.

For these experiments, we simulate a 4-way issue out-of-order processor with a two-level cache hierarchy. The L1 data and instruction caches connect to a 1MB unified L2 cache. The L1 data cache is either 16kB or 32kB, and the L1 instruction cache is always 32kB. All caches follow the write-back and fetch-on-write policies, which means that data is only written back to the next memory level once it is replaced and that the cache will always fetch the corresponding cache line on a write miss. The L2 cache is assumed to be located off-chip. For the experiments in this chapter, increasing the L1 cache size or adding more levels of on-chip cache does not improve performance. Therefore, the results presented here are comparable to a system that for example

CPU Core	
Issue Width	4
RUU Size	128
Functional Units	2 IntALU, 1 IntMult/Div, 1 FPALU, 1 FPMult/Div
Memory Hierarchy	
Memory Ports to CPU	2
L1 Data Cache	16kB or 32kB, 4-way associative, 32B lines, 2 cycle latency
L1 Inst. Cache	32kB, 4-way associative, 32B lines, 2 cycle latency
L2 Unified Cache	1MB, 8-way set-associative, 64B lines, 15 cycle latency
Memory Latency	100 cycles
Off-Chip Penalty	10 cycles

TABLE 3.1 *Main properties of the simulated system*

has on-chip L1 and L2 caches and an L3 cache located off-chip. Table 3.1 lists the most important system parameters. We implemented an additional 10 cycle penalty for transferring addresses or data between on- and off-chip logic. This implies that both accesses to L2 and copy instructions sent to L2 experience a 10 cycle delay before they can proceed to access L2.

We use three different implementations of our proposed copy engine for this section: a system with only a copy engine in the L1 cache (CE-L1), a system based on the dynamic copy engine that can copy in both L1 and L2 (DCE), and the same system using the non-temporal version of the dynamic copy engine (DCE-NT).

In case of the dynamic copy engines, the decision where to perform the copy depends on the availability of the source data in the L1 cache, as was explained in Section 3.3. In case the source data is aligned on a cache line boundary, only one access is required to read the data. Otherwise, two accesses are needed. We conservatively assume that these two cache accesses are performed in series, although they could be performed in parallel by using two-bank interleaved caches [1]. As explained before, we furthermore assume that these accesses are not pipelined. Combined with the data from Table 3.1, the delay of the DCE is as follows:

When the source is not aligned to a cache line boundary

- If both cache lines with source data are in L1, the copy engine performs a copy in 8 cycles by issuing 2 reads (4 cycles), shifting the data (2 cycles), and writing it back to the L1 cache (2 cycles).
- If one of the source cache lines is in L1 but the other is not, the time taken by the copy engine depends mostly on the time required for fetching the missing data. Assuming that the missing data is available in L2, the copy is performed in 29 cycles, by concurrently fetching the data from L1 and L2 (10+15 cycles), shifting the double cache line (2 cycles), and writing the result to L1 (2 cycles). In addition to this, the copy is delayed until it is known that one cache line is available and the other is not. Assuming a cache miss is known after 1 cycle, this delay is 3 cycles.
- If no source data is found in L1, the copy is performed in L2. Assuming the source data is available in L2, the copy is performed in 57 cycles. After sending the copy instruction to L2 (10 cycles), the data is fetched (15+15 cycles), shifted (2 cycles), and written back (15 cycles). Since the L2 cache employs larger blocks than the L1 cache, it may be necessary to also fetch the cache line corresponding to the destination address.

When the source is aligned to a cache line boundary

- If the source data is found in L1, the copy is performed in 4 cycles (2 cycles to read from L1 and 2 cycles to write the data back).
- If the source data is not in L1 but is in L2, the copy is performed in 40 cycles (10 cycles to send the command off-chip and 30 cycles for reading and writing in L2).

3.4.2 Experiments with a Memcopy Micro-Benchmark

In this section, we present experimental results using a simple memcopy benchmark. This benchmark is intended to mimic the case where data is first produced, then copied, and where this copied data is then used for further processing. For a given size N, this benchmark performs the following steps:

- Allocate two buffers of N bytes each.
- Fill the first buffer with 1s.
- Copy the data from the first buffer to the second.
- Read out the data from the second buffer.

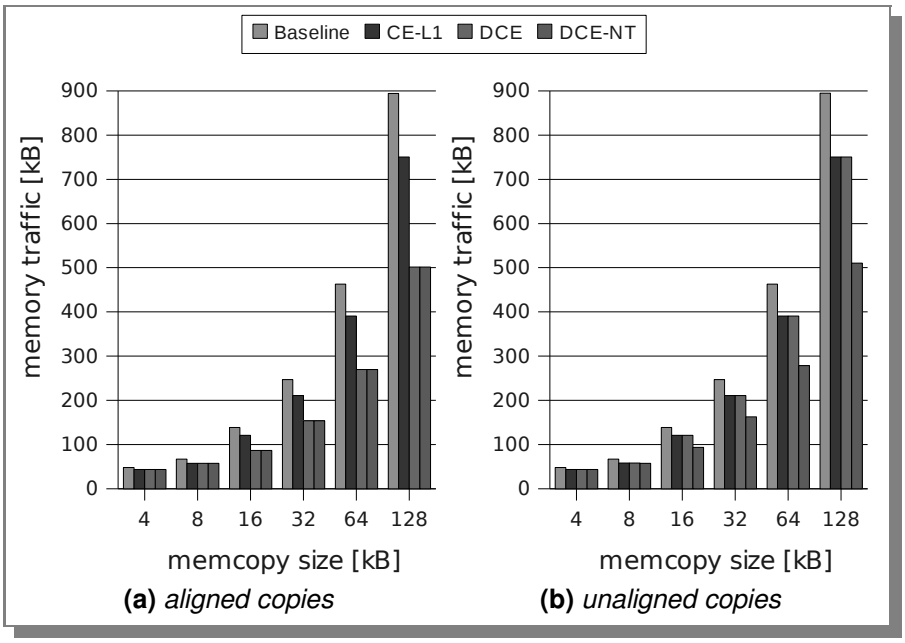

FIGURE 3.6 *Data traffic produced by the memcopy kernel using a 16kB L1 data cache.*

The experiments in this section were performed using both a 16kB and a 32kB L1 data cache. The results from these experiments are depicted in respectively Figure 3.6 and Figure 3.7. The results include data from the software implementation (Baseline), the implementation with only an L1 copy engine (CE-L1), the DCE, and the DCE-NT. Both Figures 3.6 and 3.7 include graphs from aligned as well as for unaligned copies. For aligned copies, the inter-alignment of the source and destination buffers was forced to be a multiple of the L1 cache line size. For unaligned copies, this inter-alignment was forced to be 4 bytes.

For small copies (4kB), the improvement upon the baseline case is negligible. In this case, most data traffic is generated for loading the program itself and other overhead. For slightly larger copies (8kB), there is a small improvement caused by the fact that the copy engine writes the destination data in whole cache lines, avoiding the need to load the previous contents corresponding to those addresses.

When the copies are smaller than or equal to half the size of the L1 data cache, all proposed solutions using copy engines perform equally. In these cases,

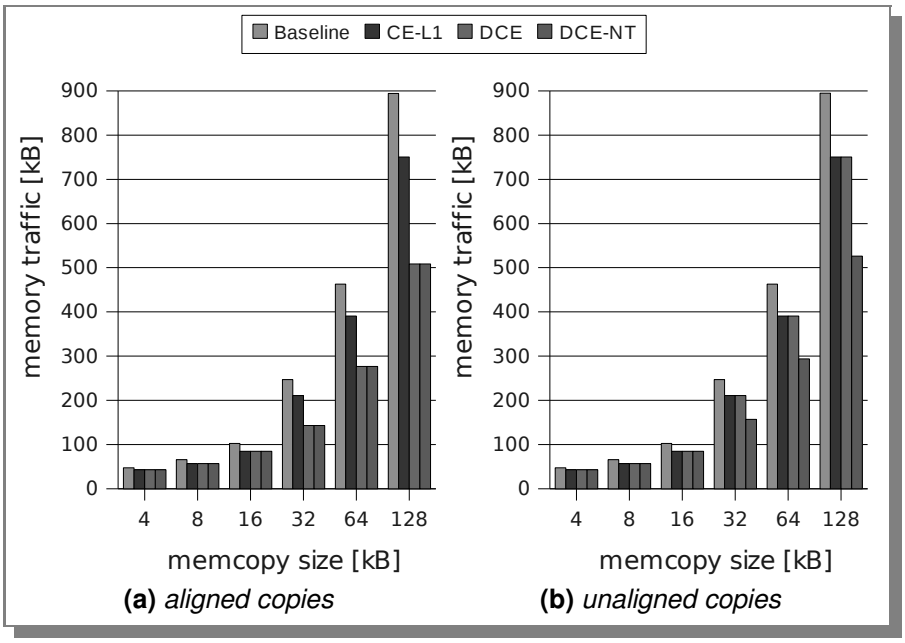

FIGURE 3.7 *Data traffic produced by the memcopy kernel using a 32kB L1 data cache.*

there seems to be no benefit from having other copies engines besides one in L1. This is in fact a normal result from our memcopy micro-benchmark, where data is first written, then copied, and the copied data is finally read again. In this case, all source data will be available from the L1 data cache, and all copies will be performed in there, irrespective of the used technique. When the buffers grow bigger than the L1 data cache, the amount of data traffic can clearly be reduced by using the copy engine in L2. In these cases, the attained savings are still not as high as described in Section 3.3.3 due to the fact that in here we also write the source buffer before and read the destination buffer after copying.

From both Figures 3.6 and 3.7, it can be seen that for aligned copies the DCE attains the same result as the DCE-NT, while for unaligned copies it performs equal to CE-L1. For unaligned copies, there is clearly no benefit from dynamic copy engines unless non-temporal fetches are used.

Another thing that can be seen when comparing Figure 3.6 to Figure 3.7, is that for larger copies, the DCE-NT actually performs better on a 16kB cache than on a 32kB cache. This is caused by the fact that in the smaller cache less

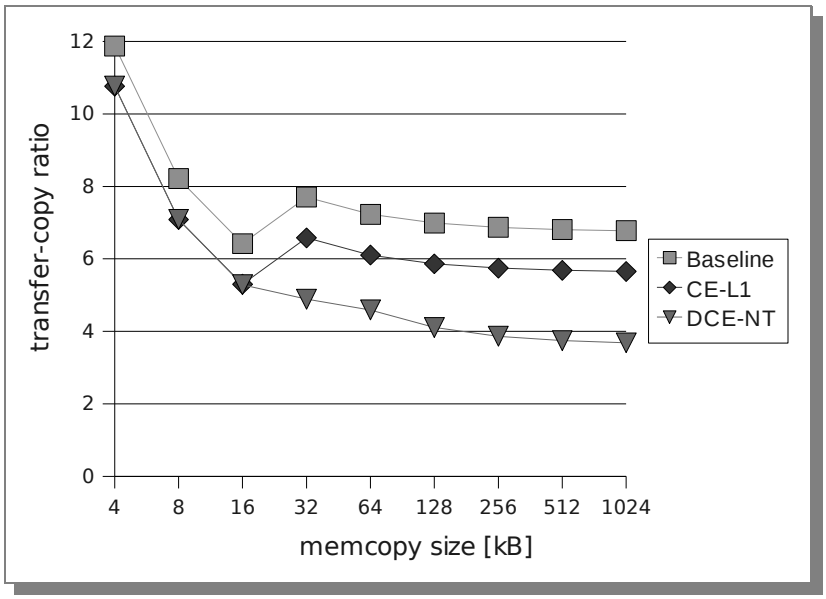

FIGURE 3.8 *Ratio of transferred bytes to copied bytes in the memcopy micro-benchmark.*

data will be available for copying in L1. As a result more data is copied in L2, reducing the total amount of memory traffic.

By dividing the amount of data traffic by the size of the copied data, one can find the number of transferred bytes for each copied byte in the memcopy micro-benchmark. Figure 3.8 depicts these ratios for unaligned copies with an L1 data cache of 32kB. The DCE is excluded from this figure as it has identical results as CE-L1.

Figure 3.8 shows a general trend of decreasing cost as the overhead becomes decreasingly significant. It also again visible that there is a steep increase when the data no longer fits in the cache. For copy sizes larger than the L1 data cache, the ratios slowly converge. For this figure, it is important to notice that, for copies exceeding the cache size, our memcopy benchmark produces an overhead of 3 bytes for every copied byte: 2 for initializing the source buffer (fetching and writing back) and 1 for verifying the corresponding byte in the destination buffer. Taking this overhead into consideration, Figure 3.8 shows that for large copies (1024kB) the baseline software routine produces more than 3 bytes of traffic for every copied byte. This is due to fetching the source, fetching the destination, and writing back the destination. The remainder is

used for sending addresses. Taking the 3 bytes overhead into account, the L1 copy engine converges to a ratio of between 2 and 3. In this case, 1 byte is saved compared to the software method by not fetching the data corresponding to the destination addresses. The DCE-NT, finally, converges to a ratio less than 1. In this case, most copies are performed by only sending the addresses to the copy engine in L2.

3.4.3 Experiments with a TCP/IP Processing Benchmark

Since memory copies are often used in operating systems, message passing parallel systems, webservers, and databases, it would be best to evaluate our system with those applications. However, in SimpleScalar such large and complex experiments would take months or years to complete. Therefore, we use a TCP/IP processing workload as was also used by Zhao et al. [102], which resembles part of the typical processing in an operation system kernel. This workload is derived from the FreeBSD stack [72]. The traces used in these experiments consist of 50,000 packets each, with fixed payload sizes of 500, 1000, and 1500 bytes.

Figures 3.9 depicts the execution time reduction upon the software implementation for the different copy engines when processing TCP/IP payloads of varying sizes. All three implementations substantially improve performance, and for each implementation the improvement increases for larger payloads. This is expected, since every call to memcpy induces overhead for checking the data size and alignments. For larger copies, this overhead is relatively less. Furthermore, for larger payloads the memcpy function constitutes a larger part of the workload.

The implementation with only an L1 copy engine (CE-L1) improves performance compared to the software implementation by between 11 and 13%. The most significant part of this speedup is due to the fact that the copy engine avoids fetching data corresponding to the destination cache line. This can be seen from Figure 3.10, which depicts the relative savings in off-chip traffic (i.e.: the total number of bytes transferred between on-chip L1 and off-chip L2) for all three implementations. The numbers in this figure include transfers for both data and instructions, and include the corresponding addresses. By not fetching the old data corresponding to the destination cache line of a copy, the CE-L1 implementation reduces the amount of traffic by around 20%. We note that the same saving is attained by the software routine on a system that has write-combining write-buffers and supports non-temporal stores. In this case, however, the system incurs a significant performance loss if the copied

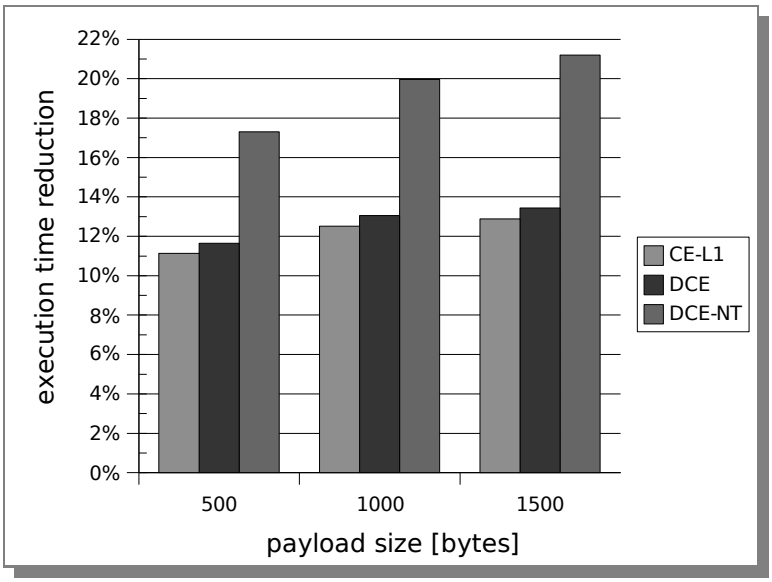

FIGURE 3.9 *Execution time reduction*

data is accessed afterwards.

Compared to the CE-L1 implementation, the dynamic copy engine (DCE) does not provide a significant additional improvement. In this case, the ability to perform the copies in an off-chip cache brings a small additional reduction in the amount of traffic when some data is not present in L1, and hence also a small improvement of the execution time. The main impediment to more significant reduction here, is that the vast majority of copies are still performed in L1. As explained before, when a larger copy is performed by a group of movblk instructions, a single movblk instruction may cause all following ones to be performed in L1 as well. Figure 3.11 depicts the percentage of all movblk instructions that is performed in the off-chip L2 cache. For all payload sizes, the normal DCE only performs 6% of the copies in L2. As most data was not located in L1 originally, this clearly limits the potential speedup and traffic savings.

When the copy engine is instructed to not store source data in the L1 cache, the percentage of copies performed in off-chip memory increases dramatically. Figure 3.11 shows that the number of copies performed in L2 is more than 86% for the smaller payloads, and reaches up to 94% for the larger payloads. Correspondingly, the reduction in execution time attained by the DCE-NT is also

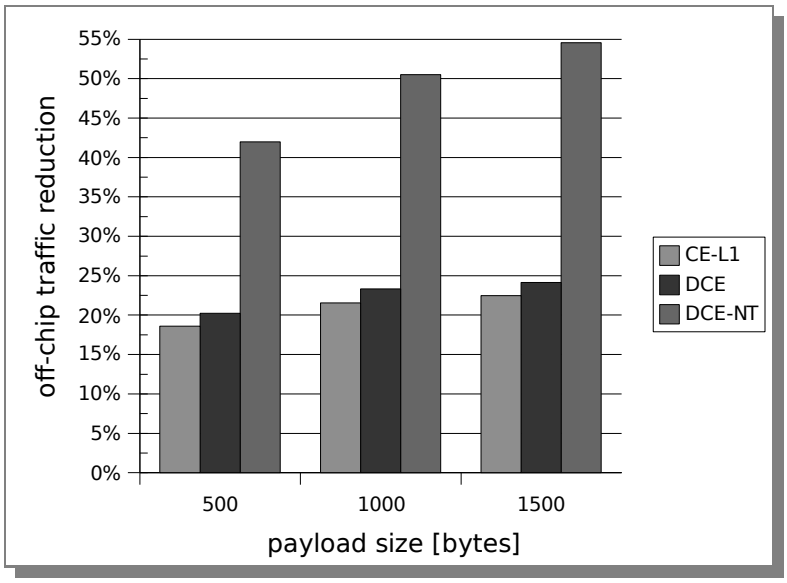

FIGURE 3.10 *Off-chip traffic reduction*

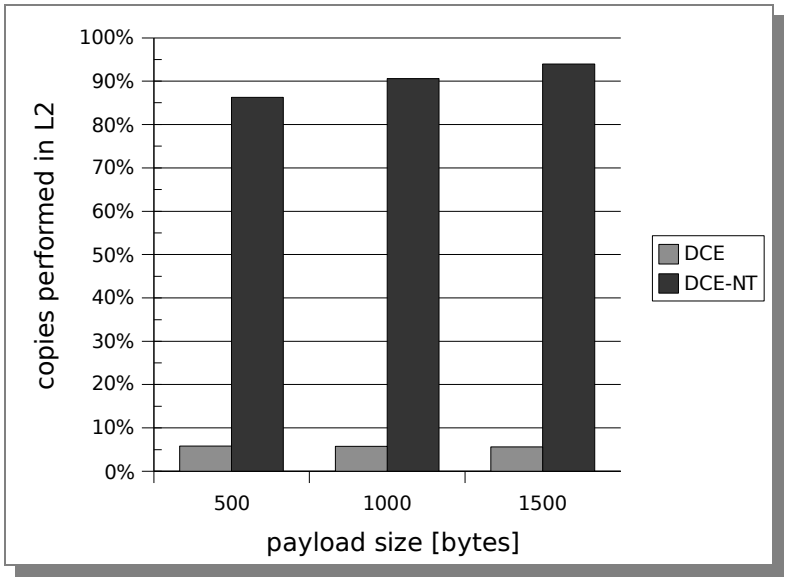

FIGURE 3.11 *Percentage of memory copies performed in L2*

much higher. For smaller payloads the DCE-NT improves performance by more than 17%, while for larger ones it improves by more than 21%. By performing the majority of copies in L2, the DCE-NT also reduces the amount of off-chip traffic significantly. Instead of fetching the source data into the processor and writing back the destination, the DCE-NT only sends 2 addresses to the off-chip logic. This way, the amount of traffic between L1 and L2 can be reduced by 42% for smaller payloads, and by more than 54% for larger ones.

3.4.4 Energy Reduction

By reducing the amount of traffic between different cache levels, a system employed with the DCE-NT will also consume less energy in the memory subsystem when copying data. As opposed to Chapter 2, however, the technique presented in this chapter does not add any new components that may increase energy consumption. In fact, the performing copies using the proposed copy engines will most likely cost less energy than performing the copies in software.

Therefore, this chapter does not contain an energy estimation like in Chapter 2. Such an estimation would be misleading in this chapter, as it would be solely based on the cost associated with memory transfers. We do state, however, that the energy saving attainable by using the DCE-NT are substantial, considering the significant savings in memory traffic. In future work, we plan to perform a more thorough evaluation of the amount of energy consumed by the copy operation itself.

3.5 Conclusions

Duplicating data is a common operation found in many applications. In some applications, like operating system kernels, it is even the most time consuming function. Unless effort is spent on reducing the number of memory transactions, the disparity between the speeds of processors and memories will prevent any further speedup. Moreover, a significant part of the total power budget is used for transactions in the memory system.

We proposed copy engines that are built alongside the cache and can perform data copies of exactly one cache line. Using this copy engine, we proposed a system called the dynamic copy engine, which implements copy engines in several levels of the memory hierarchy. By performing the copy in the highest

memory level that contains (part of) the source data, the amount of communication between different memory levels is significantly reduced. This reduction in memory traffic not only improves performance, but also significantly reduces the energy involved with transferring data between different memory levels.

Detailed experimental evaluation of dynamic copy engines in a two-level cache system showed that this approach reduces the amount of off-chip traffic by up to 94% and improves the execution time by up to 21%. Assuming that memory copies in software cost at least as much energy as performing them in hardware, these reductions in memory traffic translate to significant reductions energy. We plan to perform a more detailed evaluation of the energy consumption due to memory copies in future work.

In this chapter, we have assumed that memory copies are always performed in block sizes equal to the cache lines of the highest level cache. When multiple consecutive blocks are to be copied in a lower level cache, however, it may be possible to combine several smaller copies into larger ones. As a result, this may allow lower level caches to perform certain copies in blocks of the same size as its own cache, which would further reduce the amount of data traffic. We intend to investigate this extension to the DCE in future research.

For multiprocessor or multicore systems, reducing the load in the memory system may not only benefit the node that issues the copies, but also other nodes. Moreover, since memory copies are used extensively in shared-memory multiprocessor systems to communicate between different threads, additional benefits may be expected by using dynamic copy engines. Future research should show whether dynamic copy engines can be applied successfully in these systems as well.

4

Limiting the Number of Dirty Cache Lines

An important part of cache design is the write policy. Caches often employ write-back instead of write-through, since write-back avoids unnecessary transfers for multiple writes to the same block. For several reasons, however, it is undesirable that a significant number of cache lines will be marked "dirty". Energy-efficient cache organizations, for example, often apply techniques that resize, reconfigure, or turn off (parts of) the cache. In such cache organizations, dirty lines have to be written back before the cache is reconfigured. The delay imposed by these write-backs or the required additional logic and buffers can significantly reduce the attained energy savings. As another example, write-through caches with error detection are tolerant to transient bit errors since a valid copy of the data is contained in memory, whereas write-back caches need error correction.

In this chapter, a cache organization called the *Clean/Dirty cache* (CD-cache) is proposed that combines the properties of write-back and write-through. Our cache design avoids unnecessary transfers for recurring writes, while restricting the number of dirty lines to a hard limit. This is accomplished by adding a small *dirty cache* to the conventional L1 cache, which contains the dirty lines. Detailed experimental results show that the CD-cache reduces the number of dirty lines significantly, while achieving similar or better performance

and a lower energy consumption than a conventional write-back cache. We also present a case study where the CD-cache is used to implement an energy reduction technique called *cache decay*. Experimental results show that the CD-cache attains similar or higher performance and a lower energy consumption than a normal decay cache, while using a significantly less complex design.

Most of the material presented in this chapter has been previously published in [24].

4.1 Introduction

Most modern processors use several levels of caches to hide latency. For data written to these caches, a distinction is made between two policies: the new data is written to the cache as well as to the memory (write-through) or the new data is written only to the cache and marked 'dirty' (write-back). In the latter case, the data is only written back to the next memory level when it is evicted from the cache.

The advantage of write-back over write-through is that it avoids unnecessary transfers for recurring writes to the same block, which results in higher performance. Besides reducing performance, an increase in the amount of write back traffic also translates to an increased energy consumption, especially if the next memory level is located off-chip.

Although a write-back cache generally outperforms a write-through cache, it will contain a significant number of dirty cache lines. Table 4.1 shows for several workloads that on average 45% of the cache lines are marked dirty in a conventional write-back cache. These numbers were derived from simulations with a 32kB 2-way set-associative L1 data cache with a line size of 32 bytes and a 4-way issue out-of-order processor. More details about the employed simulation environment are given in Section 4.4.

gcc	mcf	parser	twolf	vortex	vpr
51.5%	26.3%	45.5%	51.9%	39.6%	57.1%

TABLE 4.1 *Average percentage of dirty cache lines in a 32kB 2-way set-associative cache with a line size of 32 bytes.*

For several reasons, a high number of dirty cache lines is undesirable. While delaying a write back reduces traffic by coalescing writes, it may cause problems for caches that need to flush cache lines on certain occasions. Examples are caches that need to be flushed on context switches as well as caches that need to write back cache lines to use partial shutdown, drowse modes or cache reconfiguration [36, 53]. For these caches, a large number of dirty cache lines causes problems since these cache lines have to be written back before the reconfiguration or shutdown can proceed.

Another reason against a large number of dirty caches lines comes from the area of fault tolerance. With shrinking feature sizes and increasing transistor counts, the risk of transient errors increases significantly. To allow for correct operation after such errors, digital circuits can either use *error detection* or *error correction*. In caches, error detection is sufficient for clean data, as this can be reloaded from the next memory level when an error is detected. For dirty data, however, error correction is required since there is no other up-to-date copy if an error occurs.

Besides the write-back and write-through policies, another important difference is how data is written to the cache for store instructions. While many authors (e.g.: Hennessy and Patterson [33]) make no distinction between the allocation policy and the fetch policy, Jouppi [48] makes a distinction between these policies. While the allocation policy indicates if a cache line is allocated in case of a write miss, the fetch policy refers to whether the remainder of the cache line will also be loaded in case of a miss. The final policy on how to handle write misses is therefore a combination of the *write-allocate/no-write-allocate* and the *fetch-on-write/no-fetch-on-write* policies, and on whether the data is written to the cache before or after the tag check. Table 4.2 depicts the possible combinations.

- When write-allocate is used in combination with fetch-on-write, this is simply called *fetch-on-write*. Using fetch-on-write without allocation makes no sense.
- When write-allocate is used in combination with no-fetch-on-write, this is called *write-validate*. If the store produces a miss, the remainder of the cache line is invalidated.
- When reading from a cache, the tag check and reading the data can be performed simultaneously. When writing to the cache, however, in general the tag check needs to be performed before writing the data. In this

Write-allocate?	Fetch-on-write?		Write-before-hit?
	Yes	No	
Yes	Fetch-on-write	Write-validate	No
Yes	Fetch-on-write	Write-validate	Yes
No	N/A	Write-around	No
No	N/A	Write-invalidate	Yes

TABLE 4.2 *Write miss alternatives, taken from Jouppi [48].*

case, the combination of no-write-allocate together with no-fetch-on-write is called *write-around*. When the data is written to the cache before the tag check, however, this combination is called *write-invalidate*. With write-around, storing the data is delayed until after it is known if the store produces a hit. If it misses, the data is not stored in the cache. With write-invalidate, the data is always written directly to the cache, but the cache line is invalidated in case the store misses. Since write-invalidate may overwrite other data, this policy can only be used in write-through caches.

While write-through caches can use any of these policies, write-back caches must perform the tag check before writing data to the cache. Furthermore, caches that employ fetch-on-write only need a single valid bit per cache line, whereas cache that do not employ fetch-on-write require valid bits at the same granularity as the smallest data type that can be used in store instructions. For example, to be able to store single bytes in a cache with a line size of 32 bytes, 32 valid bits would be required per cache line. Storing data in a cache with less granularity is possible, but not without first merging the data with neighboring addresses. An additional advantage of the fetch-on-write policy is that this implies that data is always transferred in cache line sized blocks, which reduces design complexity. Caches that do not employ fetch-on-write also need be able to do *partial writes* to the next memory level. To allow for a fair comparison, all caches in this chapter use the fetch-on-write policy.

In this work we explore the potential of a cache organization called the CD-cache. The CD-cache cache puts a hard limit on the number of dirty cache

lines, while maintaining or improving performance compared to a write-back cache of the same size. The proposed organization consists of a cache that is only used for loads, and a much smaller write-back cache that is used to store dirty data. Experimental results show that the proposed design attains similar or higher performance than a write-back cache, while restricting the number of dirty cache lines to a hard upper limit.

The organization of the CD-cache, using a relatively small cache structure with a significantly larger one, provides an additional advantage. Serving data from a smaller cache structure requires significantly less energy than from a larger one. Hence, the CD-cache reduces the dynamic energy consumption by directing all writes and a significant number of loads to the smaller cache structure. This is the same principle as is exploited in the energy reduction technique known as the *filter cache* [60, 61].

This chapter is organized as follows. Section 4.2 discusses related work. The design of the CD-cache cache is presented in Section 4.3. Experimental results are presented in Section 4.4, and a case study with cache decay is provided in Section 4.5. Section 4.6 summarizes this work and presents directions for future research.

4.2 Related Work

Several researchers investigated techniques that use different cache structures for different behavior. Many of these works propose to use separate caches for data that exhibits temporal and/or spatial locality [29, 46, 49, 82]. Our work is similar since we also try to separate data. However, in our case this is based on whether the data is clean or dirty.

The structure of the proposed CD-cache resembles that of the *write cache* [47], which in turn shows similarities with the *miss cache* and *victim cache* [48]. The write cache is in principle a write-buffer with the notable difference that, instead of writing updates to the next memory level as soon as possible, it only writes back data in case it needs to make room for new entries. This way, it is possible to coalesce more writes and hence decrease the amount of write traffic. The CD-cache proposed in this work differs in the following ways. First, in the CD-cache, both cache structures are mutually exclusive. As a result, stores only need to update one cache line in the CD-cache. Second, we use two cache structures in parallel, whereas the write cache is placed between the write-through data cache and the write buffer. Third, we employ the fetch-on-write policy. When a write miss occurs in the write cache, however, the cache line

is allocated but the remainder of the cache line is not fetched into the cache. As explained in Section 4.1, combining the allocate-on-write and the no-fetch-on-write policies requires valid bits for each sub-block in a cache line. This imposes additional complexity to support stores smaller than the sub-block size. The fetch-on-write policy is also preferred for caches that support error correction, since error correction is more efficient on larger blocks and the no-fetch-on-write policy would therefore impose additional complexity.

Chu et al. [16] evaluated several write buffer configurations for on-chip caches. They proposed to either flush the whole write buffer at certain intervals or on certain events, or to write back separate entries from the write buffer in the background. This is done by writing updates both to the L1 data cache and to the write buffer, and having the write buffer clear the dirty bit in the L1 data cache after writing back data. This approach, however, puts additional pressure on the data cache as it also has to be accessed by the write buffer. Also, the buffer in this approach is not used to service load instructions like in our work. Furthermore, in this approach part of the cache lines in the data cache will still be marked dirty for some time, whereas in our approach these cache lines are always clean.

Lee et al. [65] proposed a technique called *Eager Write-Back*. This write-back cache does not wait to write back data until a line is evicted. Instead, the memory bus is monitored and write-backs are issued whenever the bus is idle. This approach can significantly improve the performance of memory bound programs. However, it does not put a limit on the number of dirty cache lines.

Zhang [100] proposed to improve cache reliability by using a small fully associative buffer to replicate stored data in the L1 data cache. Depending on the required amount of replication, duplicates evicted from the write buffer might either be discarded or written back to memory. Our work differs in the aspect that we do not duplicate stored data. When data is written to the dirty cache, the corresponding cache line is copied from the clean part of the CD-cache and then invalidated. On eviction from the dirty cache, the data is only copied back to the clean cache if no conflicting requests have been issued in the mean time. The advantage of this technique is that, in our approach, data residing in the primary cache is always clean.

In this chapter, we also present a case study using the CD-cache for a cache energy reduction technique, introduced by Kaxiras et al. [37, 53, 54], called *cache decay*. Cache decay is based on a technique called the *gated-Vdd*, proposed by Powell et al. [81], which saves energy by turning of unused memory cells. In cache decay, this technique is employed to reduce energy by turning

of individual cache lines in the data cache. Our work extends upon this work by providing a method to handle dirty data efficiently.

A similar technique to reduce static dissipation is to turn cache lines to a low-power mode, instead of turning them off completely. This idea has been proposed as *Drowsy Caches* by Flautner et al. [27]. However, decreasing supply voltages and feature sizes make caches more vulnerable to transient errors. As a result, drowsy modes with an even lower supply voltage will require significant error detection/correction facilities to ensure proper operation, which in turn may offset any attained savings. The CD-cache proposed in this chapter could be used to assist in this situation also, as error correction only needs to be applied to dirty cache lines while error detection is sufficient for clean data.

4.3 Clean/Dirty Cache

The proposed design comprises two caches: a primary *clean cache* and a much smaller secondary cache called the *dirty cache*. While the clean cache is only used to store 'clean' data, the cache lines in the dirty cache are marked 'dirty' by definition. Furthermore, the contents of these caches are mutually exclusive. Figure 4.1 shows a schematic description of the CD-cache.

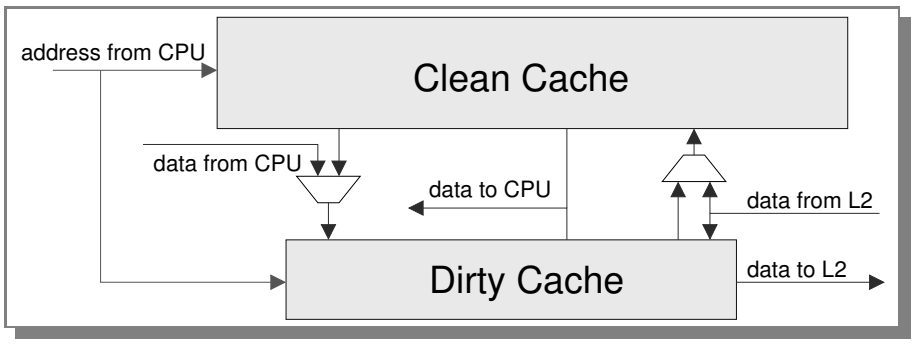

FIGURE 4.1 *Schematic representation of the Clean/Dirty-cache*

In the CD-cache, writes always go to the dirty cache and are never allocated in the clean cache. Although the data from store instructions is never written to the clean cache, it may still be necessary to access the clean cache if the corresponding cache line is allocated in there. Therefore, when a write misses in the dirty cache, a lookup is performed in the clean cache. If the data is

found in the clean cache, this data is invalidated and copied to the dirty cache, thereby keeping the two structures mutually exclusive. If the data is not found in either cache, the cache line is fetched from the next memory level and stored in the dirty cache. The dirty cache employs the write-back policy. Data that is evicted from the dirty cache is always written back to the next memory level, since it is dirty by definition. When this happens, the clean cache is probed again to see if the corresponding cache line is still available, i.e., this cache line is not used to store other data and is therefore in *invalid* state. If this is the case, the data is also written back to the clean cache. Our experimental results show that this causes a small reduction of L2 accesses.

Data that is read can reside in either the clean or the dirty cache. Therefore, when a read is issued, a lookup has to be performed in both caches. In the CD-cache, this is done in parallel. When a read misses in both caches, the data is fetched from the next memory level and is stored in the clean cache.

Using a split cache design as explained above, however, can impact both the access time and the energy consumption. Since writes always go to the dirty cache and since both structures are mutually exclusive, there is no negative impact on the *hit time* for writes. For reads, the data can reside in either cache. This implies the tags in both caches have to be checked before the data can be read out. It is therefore assumed that read hits experience a one cycle additional delay compared to a normal write back cache of the same size. This furthermore implies that for every read, the proposed cache organization spends twice as much energy on comparing tags as compared to a normal cache. However, since the dirty cache is much smaller than the clean cache, reading data from the dirty cache takes less energy than reading from the clean cache or from a normal cache. The same is true for writes, which always go to the smaller dirty cache but often incur additional tag checks. A more detailed discussion of the energies involved with comparing tags and reading out data is presented in Section 4.4.

An important property of the proposed design is that data allocated by load instructions can only be replaced by other load instructions, and that data allocated by store instructions can only be replaced by other store instructions. This implies that issuing a load will never cause a write-back event, and as such, are not delayed by these. Loads can, however, be delayed by write-back events due to previous instructions.

By limiting the size of the dirty cache, the maximum number of cache lines marked dirty is significantly reduced compared to a cache that employs a write-back policy. By keeping a small windows of recent stores in the L1 cache, a

significant number of L2 cache accesses is avoided compared to a cache that employs the write-though policy.

4.4 Experimental Results

In this section, experimental results are presented that show how the CD-cache can be used to limit the amount of dirty data to an upper limit, without compromising performance or increasing energy consumption.

4.4.1 Experimental Setup

For the experiments in this chapter, a selection of the SPEC2000 [87] integer benchmarks has been used. These benchmarks could be compiled and simulated without errors. To limit simulation time and yet capture different program characteristics, the SPEC2000 reduced input set [62] was used. All simulations ran for at least 500 million cycles and were stopped after 1 billion cycles. All experiments were performed using *sim-outorder* from the SimpleScalar tool set [4], which was extended with a detailed memory hierarchy, including write-buffers and miss-status-holding-registers (MSHRs). The *sim-outorder* simulator was configured to resemble a modern multiple-issue superscalar processor. The most important parameters for the baseline system are shown in Table 4.3. For the CD-cache, the same parameters were used, aside from the following differences. Read accesses to the CD-cache incur an additional delay of 1 cycle. Writes that hit the dirty cache do not incur this penalty, but writes that hit in the clean cache incur a penalty of 2 cycles since the cache line has to be transferred from the clean to the dirty cache.

The CD-caches used in this section are equipped with a 32kB cache structure to store the clean data, and a 4 or 8kB structure to store dirty data. Since the capacity of a cache has a significant impact on its performance, these CD-caches are compared with baseline caches of respectively 36kB and 40kB. *Sim-outorder* was modified to allow for simulations with cache sizes that are not a power of 2.

In this chapter, we calculate energy by multiplying statistics produced by *sim-outorder* with the corresponding costs. The exact energies consumed by certain events, such as L1 accesses, depend on the used implementation and technology. Table 4.4 depicts a number of results produced by the CACTI 5.3 cache modeling tool [91] for 32kB 2-way set-associative caches with 32-byte cache lines, using high-performance transistors. Although this table shows

CPU Core	
Issue Width	4
RUU Size	64
Functional Units	2 IntALU, 1 IntMult/Div, 1 FPALU, 1 FPMult/Div
Memory Hierarchy	
L1 Data Cache	36kB or 40kB, 2-way set-associative, 32B lines, 2 cycle latency
L1 Inst. Cache	32kB, 2-way set-associative, 32B lines, 2 cycle latency
L2 Unified Cache	1MB, 4-way set-associative, 128B lines, 12 cycle latency
Memory Latency	100 cycles

TABLE 4.3 *Baseline processor configuration.*

technology [nm]	45	45	65	65
number of banks	1	2	1	2
access time [ns]	0.828	0.750	1.392	1.261
dynamic energy per read [nJ]	0.061	0.048	0.107	0.084
leakage power [W]	0.040	0.031	0.046	0.035
dynamic power by tags	4.89%	5.89%	3.07%	5.88%

TABLE 4.4 *Energy consumption of 32kB 2-way set-associative caches with cache lines of 32 bytes.*

significant differences for different implementations, the relative differences between the energy consumption of various parts is fairly constant. More specifically, assuming a clock frequency of between 1 and 2GHz, the dynamic energy consumed by one cache access is approximately twice as high as the energy dissipated through leakage in a single cycle. Similarly, CACTI simulations show that approximately 5% of the energy involved with a cache access is used to compare tags, and that a cache of only 4 or 8kB requires less than half the energy of a 32kB cache.

Therefore, the following model is used to measure energy consumption in the simulated caches: An L1 cache access is assumed to consume twice as much dynamic energy as is dissipated through leakage by the whole cache in one

Operation	Cost
static leakage energy per cycle for the whole L1 cache	1
dynamic energy for accessing the L1 cache (only data)	1.9
dynamic energy per tag comparison	0.1
dynamic energy for accessing the L2 cache (including tags)	10

TABLE 4.5 *Relative energy costs of caches in experimental model.*

cycle. Of this dynamic energy, 5% is used for the tag comparison and 95% for reading out data. CACTI results show that accesses to the 1MB L2 cache require approximately 5 times as much energy as accesses to the L1 cache. This implies that an L2 access requires 10 times as much energy as is dissipated through leakage in the L1 cache. It is noted that Hu et al. [37] use the same energy relation between L2 accesses and L1 leakage in most of their work. For the CD-cache, twice as many tag comparisons are required. On the other hand, reads and writes that produce a hit in the dirty cache only need to access this smaller cache structure. Since accesses to a 4 or 8kB cache require less than half the energy of accesses to a 32kB cache, it is therefore assumed that accesses to the smaller dirty cache reduce the energy involved with reading out data by half. It should be noted that leakage energy is dissipated every cycle, even when there are no cache accesses. The details of this model are also depicted in Table 4.5.

4.4.2 Experimental Results

Figure 4.2 depicts the number of accesses to the unified L2 cache for both the write-back and the CD-caches. As expected, the CD-cache in general increases the number of accesses to L2 compared to the baseline write-back cache, since it needs to write back much more often. Although in most cases these increases are moderate, for the *twolf* and *vpr* benchmarks they are quite significant, both in absolute and relative terms. In one case, with the *mcf* benchmark in Figure 4.2b, the CD-cache cache actually produces fewer L2 accesses than a conventional write-back cache. This small improvement is due to the fact that separating clean and dirty data adds a small amount of associativity.

Although the CD-cache increases the number of L2 accesses in almost all cases, this does not necessarily imply a performance reduction. Figures 4.3a and 4.3b depict the IPC for the baseline and CD-caches of 36 and 40kB, respec-

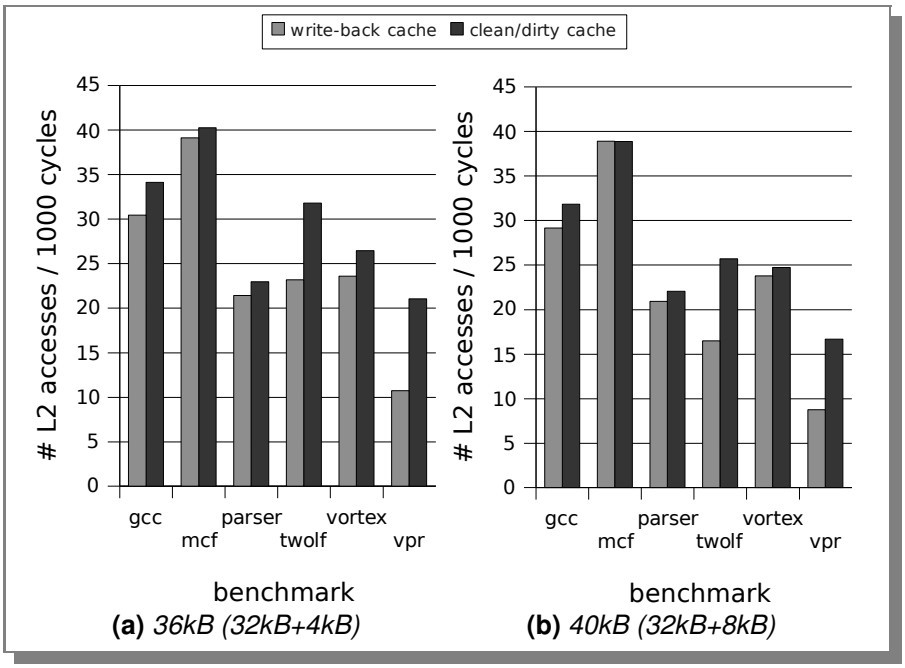

FIGURE 4.2 *L2 accesses per 1000 cycles for baseline write-back and CD-caches with capacities of 36kB and 40kB.*

tively. For both figures, the differences between the baseline and CD-caches are rather small, and in most cases are in favor of the CD-cache. Even for the *twolf* and *vpr* benchmarks, which showed significant increases in L2 accesses, there is no real reduction in performance. This is due to the following reasons. First, the simulated out-of-order processor is able to effectively hide the latency of the additional L2 accesses, especially if they do not occur concurrently with other L2 accesses. Second, due to the limited number of dirty cache lines in the CD-cache, the accesses to L2 are higher in frequency but also better spread over time. More importantly, in a normal write-back cache a read may result in both a transfer from the next memory level to retrieve the requested data and a transfer back to the next memory level when dirty data is evicted from the cache. In the CD-cache, on the other hand, reads can never result in writing back data, and as a result experience less delay.

Improving performance is, however, not the primary objective of the CD-cache. The goal is to limit the number of dirty cache lines while maintaining comparable performance. Furthermore, since the CD-cache is targeted at cache energy reduction techniques, it is important to not increase energy con-

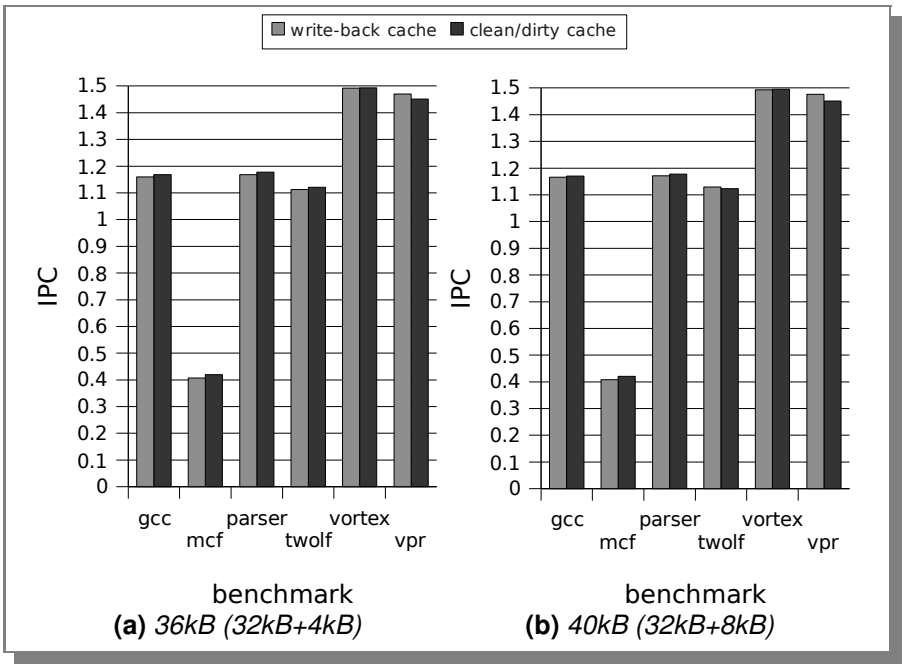

FIGURE 4.3 *IPC for write-back and CD-caches of 36kB and 40kB.*

sumption in other parts. Figures 4.4 and 4.5 depict the dynamic energy consumed in the cache hierarchy relative to the baseline write-back cache for the same experiments as before. The energy consumption in these figures is measured by using the results produced by SimpleScalar and multiplying these by the corresponding energy cost, as was described in Section 4.4.1. Figures 4.4 and 4.5 depict separate results for the energies involved with tag comparisons, reading out data, and accesses to the L2 cache.

For all benchmarks depicted in Figures 4.4 and 4.5, the CD-cache actually reduces the energy consumption. While the energy consumption in this organization is increased by performing twice as many tag comparisons and by issuing more requests to the L2 cache, there is also a significant reduction in the energy dissipated by reading and writing data from and to the L1 cache. All write operations and also some reads are performed on the much smaller dirty cache, which requires significantly less energy per access. As mentioned before, this is the same technique as is exploited in the energy reduction technique called the filter cache [60, 61].

In this section results were presented for a 32kB cache equipped with either a

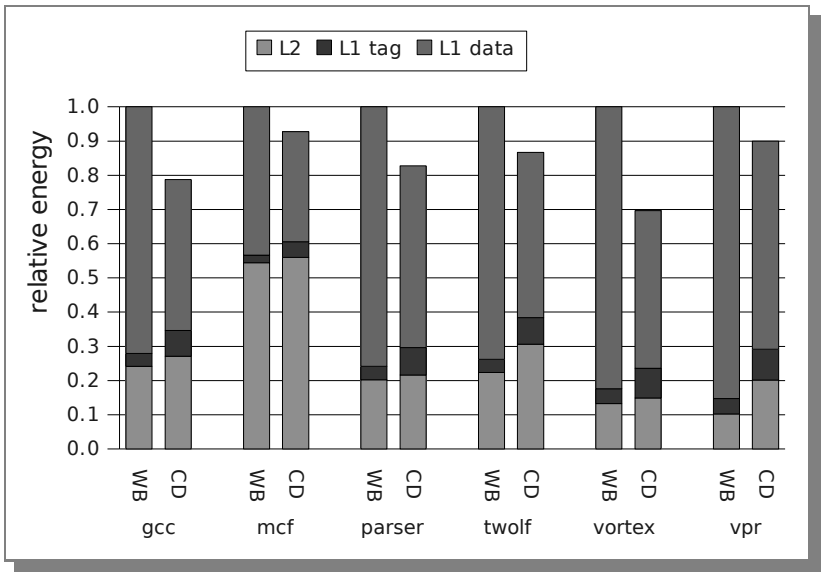

FIGURE 4.4 *Dynamic energy consumed in L1 and L2 for baseline write-back and CD-caches of 36kB.*

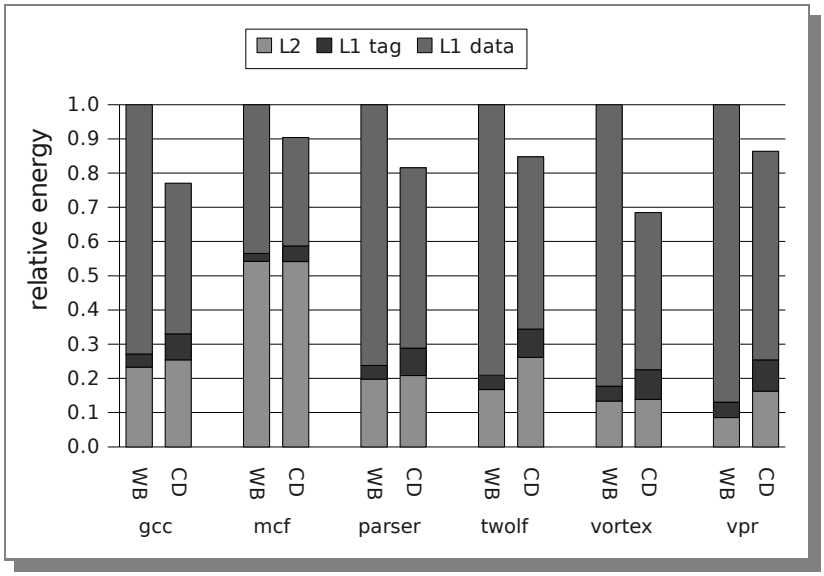

FIGURE 4.5 *Dynamic energy consumed in L1 and L2 for baseline write-back and CD-caches of 40kB.*

4kB or a 8kB dirty cache. Although a larger dirty cache can reduce the number of L2 accesses, it always increases the number of dirty cache lines. When the goal is to reduce the number of dirty cache lines, the 32kB cache equipped with a 4kB dirty cache might be preferable above a 8kB dirty cache, since the first has twice as few dirty blocks by definition. As shown in Table 4.1, approximately half the cache blocks are marked dirty in a normal write-back cache. Hence, the number of dirty blocks is reduced by $50\% \times 1280/128 = 5$ times with a 4kB dirty cache, and by $50\% \times 1280/256 = 2.5$ times when using a 8kB dirty cache. Higher improvements can be attained by using a smaller dirty cache, however, at the expense of an increase in L2 accesses.

4.5 Case Study with Cache Decay

In this section, the CD-cache is used to implement an energy saving technique called cache decay [37, 53]. After briefly explaining cache decay, experimental results are presented that show how the CD-cache can be used to assist cache decay.

4.5.1 Cache Decay

Cache decay reduces the static power consumption in caches by switching off the power supply to cache lines that have not been used for a certain number of cycles. This is implemented as follows. Each cache line is associated with a 2-bit saturating local counter. The local counters are incremented at fixed time intervals, and are reset to 0 when the corresponding cache line is used. To minimize switching activity, a hierarchical counter mechanism is used, where a single global counter is used to provide ticks to increment the local counters at specified intervals. When a local counter saturates, it generates a signal to switch off the cache line. The valid bits are never switched off and indicate whether a cache line is powered on.

Kaxiras et al. [53] have shown that, based on the decay interval, a significant number of cache lines will be switched off. By switching off cache lines, however, the cache miss rate increases. This can have a negative impact on performance. More importantly, the energy savings obtained by switching off cache lines has to be offset against the increase in accesses to the next memory level. These authors have shown that a decay cache can result in the same miss rate as a normal cache, while having fewer powered-on cache lines. It was also shown that the decay cache results in a lower miss rate than a normal

cache with the same number of active cache lines.

Dirty cache lines, however, pose a problem with cache decay as dirty lines have to be written back to memory before they can be decayed. In [37], the authors proposed to avoid bursts of write-backs on the global tick signal by cascading the global signal from one local counter to the next with a one cycle delay. This, however, assumes that write-backs can always be written from the cache to a buffer without experiencing stalls. The memory subsystem is a known bottleneck and the same bandwidth that is used for write-backs is also used for other means. This makes it hard, or at least very costly, to always guarantee sufficient bandwidth for writing back one cache line per cycle. Other solutions are to stall the decay process whenever the write-back buffer is full or to only decay dirty cache lines in case there is sufficient room in the write buffers. Such solutions, however, would require significantly more complex decay hardware, since data from cache lines cannot simply be written out on decay, but only after verifying that there is room in the write-buffer. This, in turn, can easily lead to an increase in energy consumption.

4.5.2 Cache Decay using the Clean/Dirty Cache

In this section, we show how the CD-cache can be used to efficiently implement cache decay. The main advantage of separating clean and dirty data in the CD-cache is that data in the clean cache can be decayed without a problem. This makes it possible to implement cache decay efficiently in this part of the cache, without the need to include additional write-back buffers. For simplicity, in this section we assume that dirty cache lines are not decayed in the CD-cache.

Cache decay using the CD-cache is compared to a baseline write-back cache that employs cache decay. The write-back cache is assumed to have an additional write buffer with a limited number of entries for writing back data. It is furthermore assumed that cache lines cannot be decayed if there is no room in the write buffer. In case the write buffer is full, decaying cache lines is delayed for a whole decay period, until the next global tick arrives.

The most important parameter in cache decay is the decay period. The decay period determines the intervals at which cache decay is performed. Smaller decay periods lead to a smaller number of active cache lines, but to a larger number of accesses to the next memory level.

Another important parameter for cache decay in the CD-cache is the size of the dirty cache. A larger dirty cache decreases the number of accesses to the

next memory level. However, a higher number of cache lines dedicated to store dirty data also implies an increase in the number of active cache lines, as it is assumed that dirty cache lines cannot be decayed.

To limit the complexity and simulation time, cache decay is used only in the L1 data cache. Furthermore, performing cache decay in L2 would give unrepresentative results as the employed benchmarks do not significantly stress the L2 data cache. In reality, cache decay would benefit any cache that consumes a significant amount of energy through leakage currents, especially large on-chip L2 caches as are common in many modern CPUs.

As noted before, cache decay is performed only in the L1 data cache. The static energy saved by cache decay can therefore be estimated by the average fraction of L1 that is decayed multiplied with the duration of the benchmark. This has to be offset against the increased number of accesses to the next memory level. An exact comparison between these two is very implementation and technology specific, as was also indicated by Kaxiras et al. [53]. We therefore use the same energy model as used in Section 4.4, which relates the dynamic energy involved with various operations in the caches to the leakage current of the whole L1 data cache during 1 cycle.

Figure 4.6 depicts the average active size when employing cache decay with varying periods on both a write-back and a CD-cache. The active size of a cache is defined as the percentage of cache lines that are powered on. As explained before, cache decay in a normal write-back cache can be limited due to the limited size of the write buffer. The results depicted in this figure include data for write-back caches with write buffers of 4, 8, and 16 entries. These are labelled respectively WB-4, WB-8, and WB-16. Both these caches are in total 36kB, where in the CD-cache 32kB is used for clean data and 4kB is used for dirty data.

The graphs presented in Figure 4.6 clearly show that the effect of cache decay on a conventional write-back cache can be limited if there is insufficient room to write back dirty cache lines. While for almost all benchmarks, a larger write buffer leads to a decreased active size, for most benchmarks a write buffer of more than 16 entries is required to attain the same number of decayed cache lines as the CD-cache. This is because the CD-cache can always decay cache lines in the clean part that have not been used recently. The write-back cache, on the other hand, may need to write back significantly more dirty cache lines than there is room for in the write buffer.

When, based on the decay interval, the number of active cache lines becomes really small, the size of the dirty cache starts to play a significant role. This

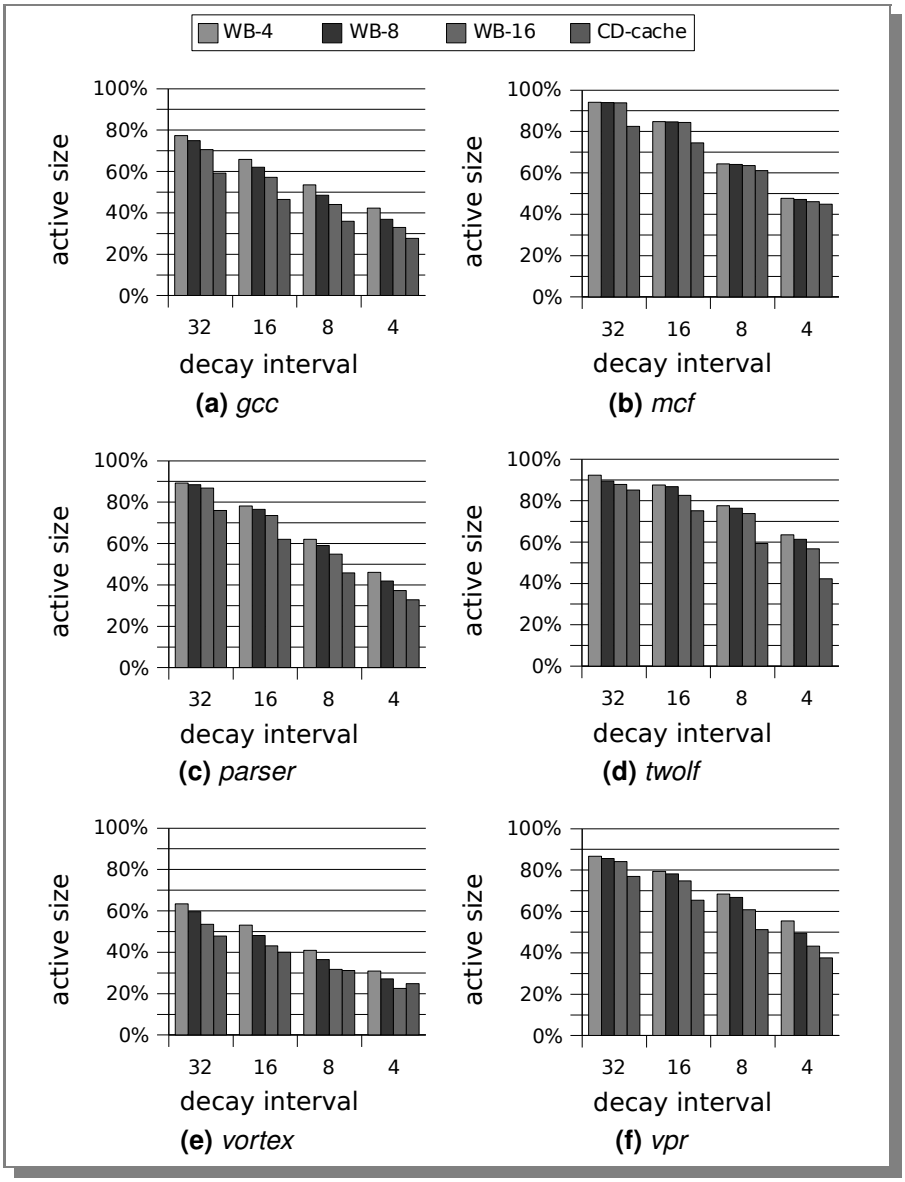

FIGURE 4.6 *Average active size for normal write-back and CD-caches using cache decay. Decay intervals are in 1000 cycles.*

happens, for example, with *vortex* with a decay interval of 4000 cycles, depicted in Figure 4.6e. In this case, the CD-cache has on average 287 active cache lines. With a clean part of 32kB (1024 cache lines) and a dirty part of 4kB (128 cache lines), this implies that only 159 clean cache lines are active on average, and that almost half of the active cache lines are dirty cache lines that cannot be decayed. When targeting such a significant reduction in active cache lines, a smaller dirty cache should be employed.

Figure 4.7 depicts the number of L2 accesses per 1000 cycles for the write-back cache and the CD-cache, using cache decay with various decay periods. This figure also includes results for the write-back and CD-caches without cache decay. These are labelled ∞, as they can be viewed as having an infinite decay period. The write-back caches in these experiments are equipped with a 16-entry write buffer.

For some benchmarks, like *mcf*, there is only a marginal increase in L2 accesses, even when the decay period is rather short. For other cases, like *vpr*, the CD-cache with a decay period of 4000 cycles increases the number of L2 accesses almost threefold. However, these more significant increases only occur when the number of L2 accesses was already low in the baseline cache (cf. Figure 4.2). For benchmarks with a higher number of L2 accesses in the baseline cache, the increase in L2 accesses is generally limited.

As expected, the number of active cache lines decreases and the amount of traffic to and from L2 increases with decreasing decay periods. With a large decay period, hardly any of the cache lines are turned off. With a very small decay period, on the other hand, a significant number of additional L2 accesses is generated. For an optimal energy reduction, the chosen decay period depends on the application and on the difference in energies due to leakage currents and L2 accesses. Hu et al. [37] describe an adaptive technique to adjust the decay period to the behavior of an application. This, however, is beyond the scope of this work.

The exact energy savings attained by cache decay depends on the relative amount of static energy consumed by the caches versus the amount of dynamic energy required for a transaction from or to the next memory level. In some cases, it might be worthwhile to have a few additional accesses to the next memory level to allow a significant part of the cache to be shut down. In other cases, it might be better to shutdown a smaller part of the cache, while making sure the number of accesses is not increased.

Figure 4.8 depicts the total energy consumption in the L1 data cache and by the additional L2 accesses for the write-back cache and the CD-cache. The

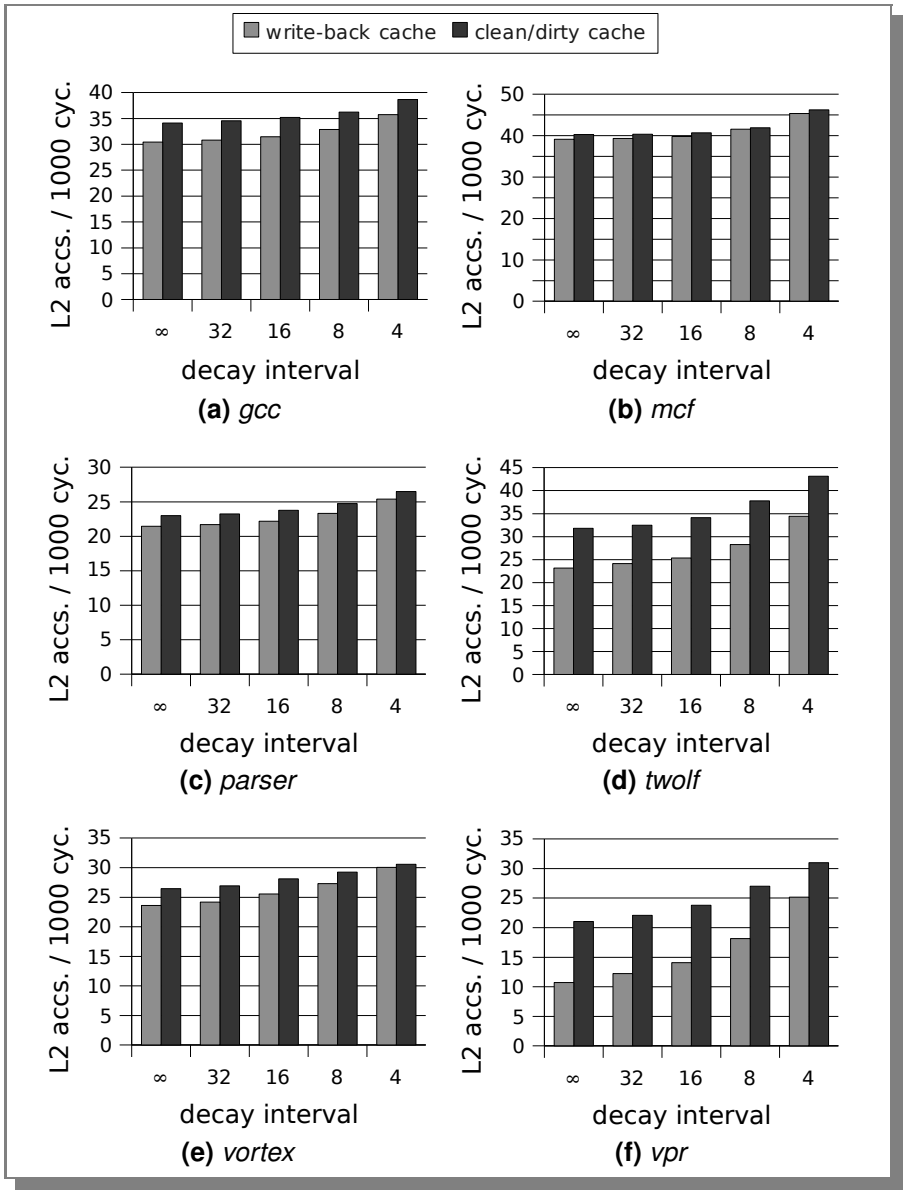

FIGURE 4.7 *L2 accesses per 1000 cycles for write-back and CD-caches using cache decay. Decay intervals are in 1000 cycles.*

energy in this figure is scaled to the total energy consumed in the L1 data cache for a baseline write-back cache without cache decay. Furthermore, the same energy model as in Section 4.4 is used, where an L1 access is assumed to consume twice as much energy as is dissipated through leakage by the whole L1 cache in one cycle, and where an L2 access is assumed to consume 5 times as much energy as an L1 access. The results depicted in Figure 4.8 indicate the amounts of energy consumed in the L1 data cache, separated by static energy consumption due to leakage currents (L1 leakage), dynamic energy for reading and writing data (L1 data), and dynamic energy for tag comparisons. Since cache decay increases the number of L2 accesses, we also include the energy consumption due to additional L2 accesses in this figure. As before, the write-back caches are equipped with a 16-entry write buffer.

The results depicted in Figure 4.8 show that the CD-cache with cache decay consumes considerably less energy than a write-back cache using cache decay. On average, the energy reduction achieved by using cache decay with the CD-cache is more than twice as high as the the energy reduction achieved by using cache decay in a normal write-back cache. The maximum is found with the *parser* benchmark, where the energy reduction is improved by a factor of 2.12. The additional energy consumed by the increased number of tag comparisons in the CD-cache is easily offset by the decrease in energy required for reading and writing data and by the reduction in active size. By separating clean and dirty data, the CD-cache can employ cache decay on the clean part much more successfully than a write-back cache, leading to a decrease in the total energy.

So far, we have assumed that L2 accesses consume 10 times as much energy as is dissipated by the L1 cache in a single cycle. Although this is a reasonable measure, this ratio may turn out to be quite different when using other cache configurations or a different technology. From Figure 4.8, however, it can be seen than even when L2 accesses would dissipate twice as much energy, the CD-cache would still consume less energy than a write-back cache.

The results presented in this section indicate that it may be possible to attain higher savings by exchanging even more active cache lines for accesses to L2. It should be noted, however, that additional L2 accesses do not only consume energy, but can also induce additional delays and thereby reduce performance. This is illustrated by Figure 4.9, which depicts how the performance of the *twolf* and *vpr* benchmarks drops for decreasing decay periods. This shows that care should be taken to not decrease the decay period too much, and make sure that the number of L2 accesses is not significantly increased for both energy and performance reasons. As was also shown in Section 4.4, the CD-

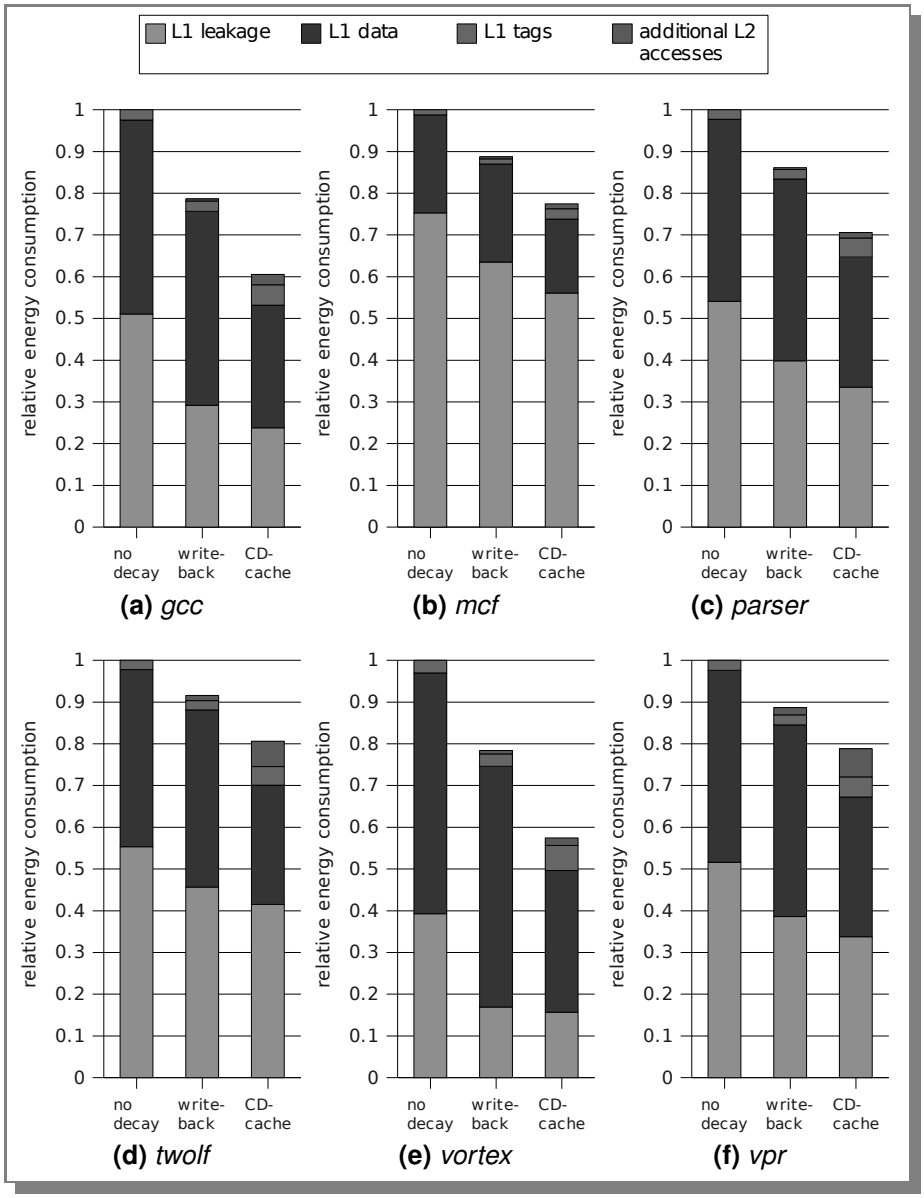

FIGURE 4.8 *Relative energy consumption in the L1 data cache and by the additional L2 accesses when using cache decay with a period of 16000 cycles.*

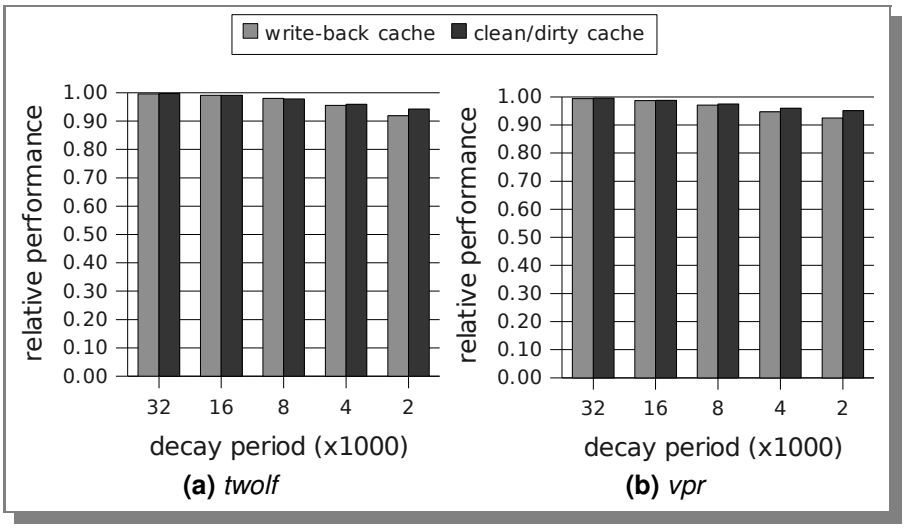

FIGURE 4.9 *Performance of two different decay caches for the* twolf *and* vpr *benchmarks with various decay periods, relative to a write-back cache without decay.*

cache often increases L2 accesses while improving performance, due to the fact that the CD-cache spreads these L2 accesses better over time than the write-back cache. This also makes the CD-cache a better candidate for cache decay than the write-back cache, since it is more capable of hiding the latencies of additional L2 accesses.

4.6 Conclusions

We have proposed the CD-cache, a novel cache organization for handling writes in on-chip caches. The most important property of the CD-cache is that it limits the number of 'dirty' cache lines without having to 'write-through'. Its performance is comparable to that of a conventional write-back cache of the same size, while the number of dirty cache lines is reduced significantly. This property can then be used to benefit energy reduction techniques that shut down or reconfigure cache lines.

We have also shown that the CD-cache, although generating more write-backs, improves performance slightly compared to a write-back cache. This is due to

the fact that the CD-cache can hold a limited number of dirty cache lines. As a result, in the CD-cache, the updates to lower level memory are spread more evenly than with a normal write-back cache.

In the proposed organization, reads and writes that produce a hit in the dirty cache are serviced by this much smaller cache structure. It was shown that this reduces the total cache energy consumption compared to a normal write-back cache.

To show how a cache with a limited number of dirty lines may benefit energy reduction techniques, a case study was included. In this case study, it was shown how the dirty cache can be used for a simple and efficient cache decay implementation. Compared to a write-back decay cache with a 16-entry write buffer, the CD-cache with cache decay improves the energy reduction by more than twice on average, partially due to a smaller active size and partially due to accessing the smaller L1 cache structure.

In the experiments presented in this chapter, the CD-cache was only used in L1. In future work, we intend to perform measurements using a CD-cache in lower levels as well.

The CD-cache is a good candidate for implementing other cache energy reduction techniques that need to reconfigure or shut down cache lines as well. Furthermore, since clean cache lines only require error detection instead of error correction, the CD-cache is expected to be a good candidate for making fault-tolerant caches as well. In future research, we intend to investigate how the CD-cache can be used for these means.

For the CD-cache described in this chapter, it was assumed that the dirty cache lines could not be decayed. However, it may be possible to decay dirty cache line in case there is room in the write buffer. Furthermore, the size of the dirty cache could be made to adapt to applications dynamically. Both these options are interesting candidates for future research. In future work, we also intend to investigate the benefits of using a dynamic decay period.

5

Energy Efficient Multiprocessor Scheduling using DVS

Multiprocessors are increasingly deployed to realize high-performance embedded systems. Because in current technologies the dynamic power consumption dominates the static power dissipation, an effective technique to reduce energy consumption is to employ as many processors as possible in order to finish the tasks as early as possible, and to use the remaining time before the deadline (the slack) to apply voltage scaling. However, since the static power consumption is expected to become more significant, this approach will no longer be efficient when leakage current is taken into account. In this chapter, we first show for which combinations of leakage current, supply voltage, and clock frequency, the static power consumption dominates the dynamic power dissipation. These results imply that, at a certain point, it is no longer advantageous from an energy perspective to employ as many processors as possible. Thereafter, a heuristic is presented to schedule the tasks on a number of processors that minimizes the total energy consumption. Experimental results obtained using a public task graph benchmark set show that our leakage-aware scheduling algorithm reduces the total energy consumption by up to 24% for tight deadlines (1.5x the critical path length) and by up to 67% for loose deadlines (8x the critical path length) compared to the approach that employs as many processors as possible to maximize the slack that can be

used for voltage scaling.

Most of the material presented in this chapter has been previously published in [20].

5.1 Introduction

In contemporary and future embedded as well as high-performance microprocessors, power consumption is one of the most important design considerations. Not only does this apply to processors embedded in battery powered devices, but also in desktop machines and high-performance dedicated systems power consumption is a fundamental problem that limits clock frequencies. Through the advent of (single chip) multiprocessors for the embedded market, such as the IBM/Sony/Toshiba Cell architecture [35] and the ARM11 MPCore [3], power consumption is becoming increasingly important for multiprocessor systems as well.

Power consumption can generally be classified in dynamic and static power consumption. The first relates to the power that is dissipated due to switching activity, while the second one is due to leakage currents. Because in current technologies the dynamic power consumption dominates the static power consumption, and because the dynamic power dissipation grows quadratically with the supply voltage, *dynamic voltage scaling* (DVS) [77] is an effective technique to reduce the power consumption. Consequently, when scheduling tasks on a multiprocessor system, it is advantageous to employ as many processors as possible in order to maximize the remaining time before the deadline. This slack can then be exploited to lower the clock frequency and supply voltage. However, as technology scales to increasingly smaller feature sizes, static power dissipation due to leakage current is expected to grow exponentially in the near future [53]. In this case, using as many processors as possible combined with voltage scaling will no longer provide an efficient solution. In other words, while in the past the static power consumption could more or less be ignored, it should not be neglected in the future.

In this chapter, a scheduling algorithm is presented that is targeted at a near future technology, where leakage current is responsible for a significant part of the total power dissipation. The algorithm presented in this chapter schedules task graphs on a number of processors that is sufficient to meet the deadline, while the total power consumption is minimized.

This chapter is organized as follows: Section 5.2 contains an overview of re-

lated work. In Section 5.3, we describe the conditions under which voltage scaling can be applied to reduce energy consumption. The employed system and application models are explained in Section 5.4. Section 5.5 describes our scheduling and voltage selection algorithm. Experimental results are provided in Section 5.6. In Section 5.7 conclusions are drawn and directions for future research are given.

5.2 Related Work

Reducing power consumption has been an important research topic in recent years and many techniques at the process, circuit design, and micro-architectural level have been proposed. One of the most promising techniques is dynamic voltage scaling (DVS), where both the clock frequency and the supply voltage are scaled down when peak performance is not needed. DVS is also referred to as dynamic voltage/frequency scaling (DVFS). Several existing processors such as the Intel XScale [40] support DVS.

Another approach to reduce energy consumption is to shut down idle parts of a system. In this chapter, we assume that processors cannot be turned off and on during execution. Turning processors off temporarily, which is often referred to as *dynamic power management* (DPM), will be considered in Chapter 6. Related works that employ DPM will be described in the corresponding section of that chapter.

To guarantee real-time performance, embedded multiprocessors and *Systems-on-Chip* (SoCs) in general are usually over-budgeted, i.e., they generally contain more processing cores, more memory, and support a higher bandwidth than needed. In these cases, DVS may be applied to reduce energy consumption. Applying DVS to multiprocessor scheduling has been investigated by a significant number of researchers. An overview is provided by Jha [44]. A technique proposed by several authors [31, 103] is to use existing scheduling techniques such as list scheduling with *earliest deadline first* (EDF), to finish the tasks as early as possible and to use the remaining slack before the deadline to lower the supply voltage. These authors, however, did not include leakage current in their power estimations. Several authors have proposed this technique using different names and, therefore, we refer to it as *Schedule and Stretch* (S&S). This approach will be explained in detail in Section 5.5.

Using a detailed power model that includes static as well as dynamic power, Jejurikar et al. [43] showed that there is an optimal operating point, called the critical speed, at which the total energy consumption is minimized. Lowering

the supply voltage below this point increases the energy consumption. They computed processor slowdown factors based on the critical speed. A similar approach was followed by [83], who employed a fixed priority instead of EDF. In contrast to our work, both these works focussed on single-processor scheduling and assumed that tasks are independent and arrive periodically with deadlines.

Zhang et al. [101] used the same real-time model as we do (weighted DAGs with deadlines). They did not use EDF scheduling but scheduled in such a way to have more slowdown opportunities. In Chapter 6, we analyze the effect of employing a different scheduling algorithm. Furthermore, they did not determine the number of processors that yields the least energy consumption. Kianzad et al. [56] presented an integrated approach, combining scheduling and DVS in a genetic algorithm. Varatkar et al. [92] proposed to execute part of the code on a lower supply voltage while minimizing communication. Some researchers proposed to improve DVS by also adjusting the threshold voltage when scaling the supply voltage [30, 70]. Others extended this to scheduling for real-time multiprocessor systems [2, 97]. None of these works, however, attempted to determine the optimal number of processors.

Xu et al. [96] proposed to minimize energy consumption by applying DVS and choosing the correct number of employed processors. Their work, however, targets embedded clusters in which the nodes provide the same type of service in a client-server model.

Most work on scheduling to reduce energy in DVS-enabled systems assumes that the voltage and frequency can be changed during execution. Depending on the granularity of these changes, this is referred to as either *inter-task* or *intra-task scheduling*. With inter-task scheduling, changes are only made between two task, whereas with intra-task scheduling, changes may occur within a task. Assuming that frequency and voltage may change in between or even within tasks, however, vastly increases the search-space and therefore makes the scheduling algorithm costly. In many cases, optimal scheduling using variable voltages has been proven to be NP-complete [2, 98, 99].

Another interesting issue, is that behavior of applications or parts of application can depend significantly on the provided input. In some cases, this makes it much harder to provide performance guarantees. A solution to this problem is provided by Gheorghita et al. [28]. They proposed a technique to classify input data into so-called *scenarios*. By detecting the correct scenario at run-time, the system can efficiently adapt to the required performance.

Our work differs in the following ways. First, in contrast to all other works ex-

cept [96], we exploit DVS as well as finding the optimal number of processors. Second, we focus on multiprocessor scheduling while others focussed mainly on single-processor scheduling. Third, we assume that applications are represented as weighted DAGs whereas many others assumed independent periodic tasks with deadlines. Finally, we use a publicly available set of task graphs, whereas most others used randomly generated graphs.

5.3 Energy Reduction using DVS

This section starts by explaining the concept of voltage scaling and deriving how static and dynamic power dissipation relate to leakage current, supply voltage, and clock frequency. From this, we then derive the extend to which voltage scaling can be used to decrease the energy consumption for a certain processor. Thereafter, it is shown how these results can be used for scheduling of parallel tasks on multiprocessor systems.

5.3.1 Dynamic Voltage Scaling

One of the most successful circuit-level power saving techniques applied by several CPU manufacturers is DVS. DVS is effective because it significantly reduces the amount of dynamically dissipated power. DVS works as follows. When there is no need for peak performance, the clock frequency in a DVS-enabled processor can be lowered. This already reduces the dynamic power consumption by reducing the amount of switching activity. Moreover, the minimum required supply voltage in a CMOS transistor is largely determined by the frequency or cycle time. By increasing the cycle time the supply voltage may be lowered, and since the dynamic power scales quadratically with the supply voltage this allows for significant dynamic power savings. The exact benefits of DVS are be explained below.

Some researchers prefer to reserve the acronym DVS to denote the technique where *only* the supply voltage can be scaled, and use the term *dynamic voltage and frequency scaling* (DVFS) to denote the technique where both the supply voltage and the frequency can be changed. In this work, the term DVS always denotes the technique that includes frequency scaling.

5.3.2 Voltage Scaling Requirements

The efficacy of DVS largely depends on the proportion between dynamic and static power consumption. When static power consumption can no longer be neglected, as is predicted to be the case in near future technologies [8, 25, 53], the extend to which voltage scaling can be used to reduce energy consumption becomes limited. In this section we explain required conditions for reducing energy by using voltage scaling and the impact on energy consumption when time is also taken into account.

The power consumption in a CMOS gate can be approximated by:

$$P = D + S = C_L \cdot V^2 \cdot f + I_q \cdot V, \tag{5.1}$$

where C_L is the load capacitance, V is the supply voltage, I_q is the leakage current, and f is the clock frequency. The first term (D) in this equation corresponds to the amount of dynamically dissipated power, caused by switching circuitry. The second term (S) models the amount of statically dissipated power, generated by leakage current.

We start with looking at what the requirements are for voltage scaling to be beneficial for the total energy consumption. For this purpose, we will first derive an expression for the normalized power dissipation.

To normalize Expression (5.1), we define that at maximum frequency f_{max} and corresponding supply voltage V_{max}, a processor will dissipate $P_{max} = D_{max} + S_{max}$ power. The normalized total, dynamic, and static power dissipation ($\mathcal{P}$, $\mathcal{D}$, and $\mathcal{S}$) can then be written as:

$$\mathcal{P} = \frac{P}{P_{max}} = \mathcal{D} + \mathcal{S} = \frac{D}{P_{max}} + \frac{S}{P_{max}}. \tag{5.2}$$

We then define δ and σ as:

$$\delta = \frac{D_{max}}{D_{max} + S_{max}}, \quad \sigma = \frac{S_{max}}{D_{max} + S_{max}}. \tag{5.3}$$

In other words, δ and σ ($0 \leq \delta, \sigma \leq 1$) denote the fraction of the total power dissipation at maximum frequency that is caused by switching activity and the fraction that is caused by leakage current.

Let $\mathcal{V} = V/V_{max}$ be the normalized voltage and $\mathcal{F} = f/f_{max}$ the normalized frequency. The expressions for the normalized dynamic and static power dissipation can then be rewritten as:

$$\mathcal{D} = \delta \cdot \frac{D}{D_{max}} = \delta \cdot \frac{C_L \cdot V^2 \cdot f}{C_L \cdot V_{max}^2 \cdot f_{max}} = \delta \cdot \mathcal{V}^2 \cdot \mathcal{F}, \tag{5.4}$$

and

$$\mathcal{S} = \sigma \cdot \frac{S}{S_{max}} = \sigma \cdot \frac{I_q \cdot V}{I_q \cdot V_{max}} = \sigma \cdot \mathcal{V}. \tag{5.5}$$

Combining these equations with Equation (5.2) results in:

$$\mathcal{P}(\mathcal{F}, \mathcal{V}) = \delta \cdot \mathcal{V}^2 \cdot \mathcal{F} + \sigma \cdot \mathcal{V}. \tag{5.6}$$

Slowing down the clock will increase the time required to finish a task. Mainly due to unpredictable behavior in the memory system, the execution time of a task does not solely depend on the clock frequency. However, since reducing the frequency will make memory accesses relatively less costly, it can be assumed that scaling down the frequency by a factor of N will increase the processing time by less than a factor of N. Using a normalized expression for time, $\mathcal{T} = 1/\mathcal{F}$, the expression for the normalized energy consumption $\mathcal{E}$ then becomes:

$$\mathcal{E}(\mathcal{F}, \mathcal{V}) = \mathcal{P}(\mathcal{F}, \mathcal{V}) \cdot \mathcal{T} = \delta \cdot \mathcal{V}^2 + \sigma \cdot \mathcal{V}/\mathcal{F}. \tag{5.7}$$

From this equation it can be seen that voltage scaling only reduces the energy consumption if for a certain $\mathcal{F} < 1$ there exists a $\mathcal{V} < 1$, so that $\mathcal{E}(\mathcal{F}, \mathcal{V}) < 1$.

The supply voltage of a processor must be sufficient to guarantee that the logic levels are always safely reached before the end of a clock cycle. This implies that a minimum supply voltage is required, depending on the clock frequency. From [59], we take the following expression approximating the relation between normalized voltage and frequency:

$$\mathcal{V} = \beta_1 + \beta_2 \cdot \mathcal{F}, \tag{5.8}$$

where $\beta_1 = V_{th}/V_{max}$ and $\beta_2 = 1 - \beta_1$, with V_{th} denoting the threshold voltage. Again, $\mathcal{F}$ represents the normalized frequency.

Combining this with Equation (5.7) yields:

$$\mathcal{E}(\mathcal{F}) = \delta \cdot (\beta_1 + \beta_2 \cdot \mathcal{F})^2 + \sigma \cdot (\beta_1/\mathcal{F} + \beta_2). \tag{5.9}$$

Figure 5.1 depicts the normalized energy consumption as a function of the normalized frequency for the cases where the static energy at maximum frequency (σ) constitutes respectively 25%, 50%, and 75% of the total energy consumption. Clearly, when 75% or more of the total energy consumption is due to the static component, there is almost no room for energy reduction through DVS. Even in the small range where the total energy is less than the energy at maximum frequency (when the relative frequency is between 0.83 and 0.96), the savings are negligible. When the dynamic component is more dominant, the

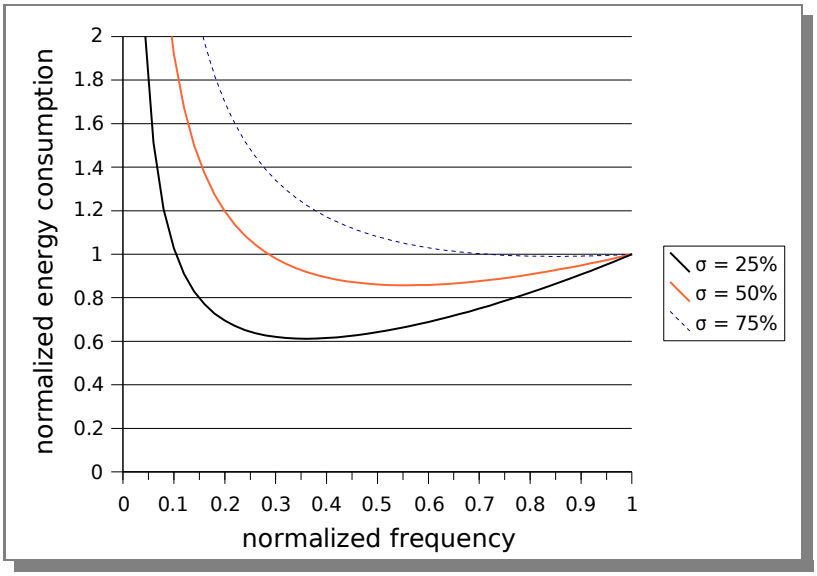

FIGURE 5.1 *Normalized energy consumption as a function of the normalized frequency for varying combinations of the dynamic and static components.*

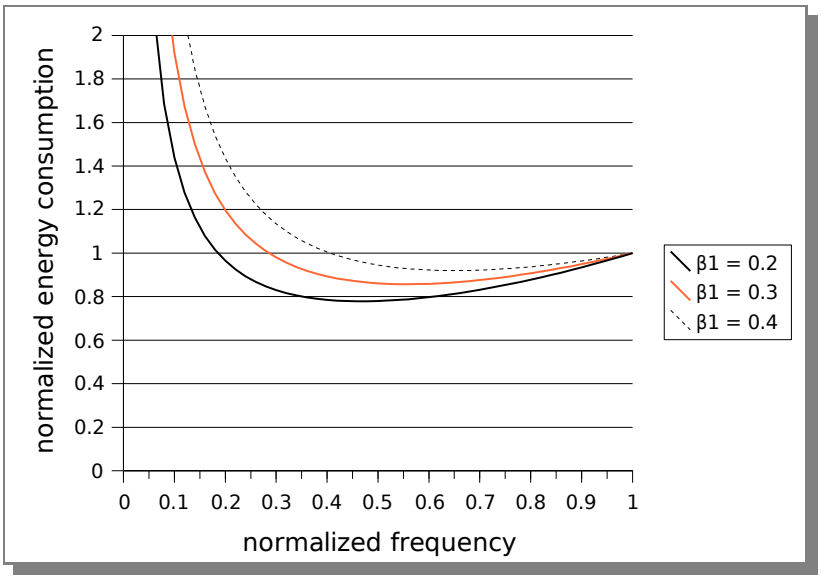

FIGURE 5.2 *Normalized energy consumption as a function of the normalized frequency for varying threshold voltages.*

range of voltages which lead to energy reduction clearly increases. However, even when the static component constitutes only 25% of the total energy consumption, excessive voltage scaling can be detrimental to the energy reduction. In this case, the optimum relative frequency is approximately 0.36.

Another important parameter in Equation (5.9) is β_1, the ratio between the threshold and supply voltage, as this parameter relates the clock frequency to the employed supply voltage. Figure 5.2 depicts the normalized energy consumption as a function of the normalized frequency for a number of different relative threshold voltages. In this figure, it is assumed that at maximum frequency, the leakage current is responsible for 50% of the total energy consumption ($\sigma = 0.5$). This figure shows that a higher threshold voltage diminishes the possibility to effectively employ voltage scaling.

Figure 5.2 also shows that, for a certain threshold voltage and amount of leakage current, there is an optimal frequency at which the total energy consumption is minimized. With a threshold voltage of 0.3 times the maximum supply voltage ($\beta_1 = 0.3$), this optimal frequency is about 0.56 times the maximum frequency. This implies that scaling the frequency to below this point will result in a higher energy consumption than when running the processor at a normalized frequency of 0.56 and turning the processor off for the remainder of time. In order to do this, however, it must be possible to shutdown the processors. This will be covered in Chapter 6.

5.3.3 Voltage Scaling in a Multiprocessor Environment

In the previous section, the circumstances under which it is useful to employ voltage scaling in a single processor were determined. For a multiprocessor system, the requirements for lowering energy consumption by voltage scaling are equivalent to the case with only one processor. For technologies with very low leakage current, where voltage scaling always decreases the energy consumption, the lowest energy solution is to run the tasks on as many processors as possible, with the lowest possible frequency. On the other hand, when the energy consumption cannot effectively be decreased by voltage scaling, the lowest-energy solution is to run the tasks on as few processors as possible.

Our approach is based on the following assumptions: First, it is assumed that all employed processors must stay on all the time. In other words, it is not possible to turn processors on or off during execution. Unused processors, however, can be turned off. In the next chapter, we will relax this assumption. Second, it is assumed that all processors run at the same clock frequency. Un-

der these assumptions, the normalized power consumption of a multiprocessor with N processors is given by:

$$\mathcal{P}_{multi} = N \cdot (\alpha \cdot \delta \cdot \mathcal{V}^2 \cdot \mathcal{F} + \sigma \cdot \mathcal{V}), \tag{5.10}$$

where α denotes the activity (i.e. the fraction of time that the processors are busy). Using Equation (5.8), Equation (5.10) can be rewritten as:

$$\mathcal{P}_{multi} = N \cdot \alpha \cdot \delta \cdot (\beta_1 + \beta_2 \cdot \mathcal{F})^2 \cdot \mathcal{F} + N \cdot \sigma \cdot (\beta_1 + \beta_2 \cdot \mathcal{F}).$$

5.4 System and Application Model

We assume a shared memory multiprocessor system running parallel applications, for which the scheduling and mapping are statically determined. In this work, an application is represented as a weighted *directed acyclic graph* (DAG) [17]. In this DAG $G = (V, E, w)$, each node $v \in V$ corresponds to a task, each edge $e \in E$ to a dependency between 2 tasks, and each weight $w(v)$ denotes the execution time of task v. It is furthermore assumed that this system is CPU bound, so that the additional power dissipation and delay caused by communication can be ignored.

Each task in this DAG must be mapped onto 1 out of N processors, in such a way that all processors finish before deadline D. The activity (α) of the resulting schedule is given by:

$$\alpha = \sum_{v \in V} \frac{w(v)}{N \cdot D}.$$

We first show how other application models can be translated to DAGs. As explained by Liberato et al. [67], real-time applications with periodic tasks can be translated to DAGs using the *frame-based scheduling* paradigm. In this case, a frame with a length equal to the *least common multiple* (LCM) of all task periods is constructed, in which each task is assigned a certain amount of processing time. Thereafter, the schedule is repeated frame after frame. An example of this is depicted in Figure 5.3. In this example, the LCM of all task periods depicted in Figure 5.3a is 30. A DAG is then constructed from all tasks within this frame, by creating dependencies from a root node to the first occurrences of each task, and dependencies between each recurring task instance. This is illustrated in Figure 5.3b.

Another common application model based on functional or pipelining parallelism is the *Kahn Process Network* (KPN) [50], where a group of processes

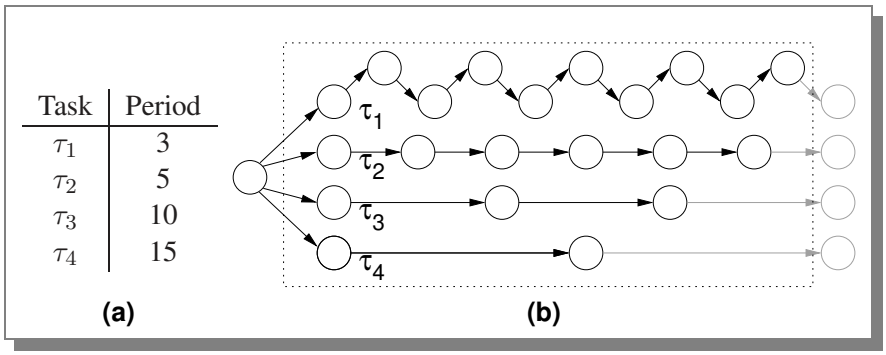

FIGURE 5.3 *Example of translating periodic tasks into a DAG.*

are connected by communication channels to form a network of processes. Each process is in principle infinite and receives data over its input channels, processes it, and sends the results over the output channels. In this model, there is not a single deadline but a certain throughput must be guaranteed. This model can be converted to DAGs by making several copies of the KPN, by translating edges in the KPN to edges between successive copies in the DAG and adding an edge from each node in the ith copy to the corresponding node in the $(i + 1)$st copy. The output nodes of the first copy are assigned an arbitrary but reasonable deadline. The deadline of the output nodes of each successive copy is set to the deadline of the corresponding node in the previous copy plus the reciprocal of the throughput. A simple example is depicted in Figure 5.4. In the KPN in Figure 5.4(a), task T_1 successively receives inputs $I_1, I_2, \ldots$, processes them, and sends the results to T_2. Task T_3 receives inputs $J_1, J_2, \ldots$ but also receives data from T_2. It combines input J_{i+1} with the ith data received from T_2 and sends the result to T_2. In the DAG in Figure 5.4(b), each node is replicated a number of times. Let T_i^j denote the jth copy of task T_i. Then T_1^j receives input I_j and T_3^j receives input J_j. There are edges from T_1^j and T_3^j to T_2^{j+1}. Because T_3 combines input J_{i+1} with the ith data received from T_2, there are also edges from T_2^j to T_3^{j+1}. To indicate that not all inputs are available at time zero, there are also edges from T_i^j to T_i^{j+1}. This could also be modeled by adding dummy input nodes whose weights are equal to the time the input becomes available.

Throughout this chapter, it is assumed that all processors run at the same frequency, and that this frequency is constant while executing an application. This brings several advantages, but also implies that the opportunity for energy reduction through voltage scaling is limited to some extend, especially

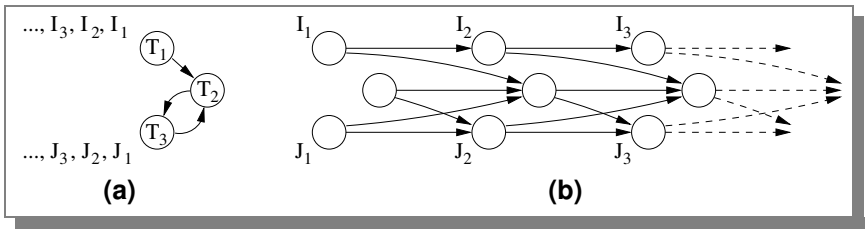

FIGURE 5.4 *Example for translating KPNs into DAGs.*

for very unbalanced task graphs. This deficit will be discussed in Chapter 6, which shows that this limitation is in many cases not significant, especially when processors are allowed to shut down temporarily. The advantages of using only one frequency and of disallowing this frequency to change, are that this allows for less complex hardware and a less complex and therefore much faster scheduling heuristic. In this chapter, it is furthermore assumed that the frequency and voltage can be scaled on a continuous range.

5.5 Energy Efficient Scheduling Algorithms

This section describes the baseline approach that is used for comparison, followed by the heuristic to schedule task graphs on the number of processors that minimizes the total energy consumption.

5.5.1 Schedule & Stretch

As mentioned in Section 5.2, several researchers [31, 103] use existing scheduling techniques to minimize the makespan of the schedule, and use the remaining slack to lower the frequency and supply voltage. The most commonly used scheduling algorithm in this case is list scheduling with earliest deadline first (LS-EDF). We refer to this approach as *Schedule and Stretch* (S&S). EDF was originally presented as the *deadline driven scheduling algorithm* [68], and some authors refer to it as *shortest time to deadline* (STTD).

In order to schedule using EDF, first deadlines are assigned to nodes by traversing the graph in a reverse topological order. The nodes with no outgoing arcs are assigned a deadline equal to the *critical path length* (CPL). The deadline of each other node is determined by the deadlines and weights of its successors. The deadline of a node is calculated by subtracting the weight of a successor

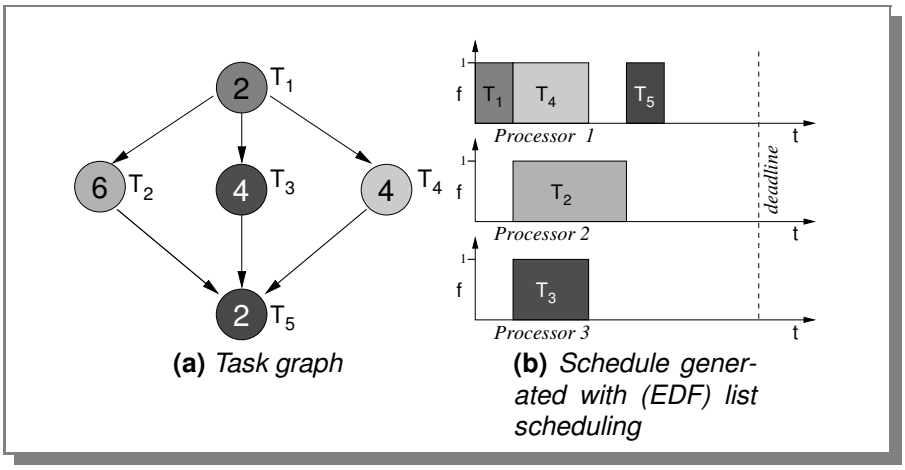

FIGURE 5.5 *Example graph and schedule.*

node from its deadline, and by taking the minimum of these values. More formally, the deadline $d(i)$ of node i is given by:

$$d(i) = \min_{j \in succ(i)} (d(j) - w(j)),$$

where $succ(i)$ denotes the set of successors of node i, and $w(j)$ the weight of node j. For the task graph depicted in Figure 5.5a, this implies that first node T_5 is assigned a deadline of 10, thereafter nodes T_2, T_3, and T_4 are assigned a deadline of $10 - 2 = 8$, and finally node T_1 is assigned a deadline of 2.

After assigning the deadlines, the tasks are scheduled onto processors using list scheduling. Figure 5.6 depicts the pseudo-code of the scheduling algorithm. The algorithm starts by initializing the ready time for each processor to zero, and by putting all nodes without predecessors in the $ready_queue$. Then, a loop is entered in which the task with the earliest deadline is taken from the $ready_queue$ and scheduled onto the first available processor (i.e., the processor whose last task finishes first). Thereafter, the indegrees of its successors are decreased by one. Successors that have no predecessors left are put in the $ready_queue$. The algorithm ends when there are no tasks left in the $ready_queue$.

Figure 5.5b depicts the Grantt Chart resulting from using the scheduling approach described above. From this figure, it can be seen that after the scheduling process, there are certain periods in which a processor is idle. This idle time is often referred to as *slack*. In the S&S algorithm, the power consumption is decreased by using the slack that remains at the end of the schedule to

```
LIST-SCHEDULE-EDF(G, N)
 1  ▷ G is a weighted directed acyclic graph
 2  S ← ∅
 3  for p ← 1 to N
 4       do proc_ready[p] ← 0
 5  for c ∈ START_NODES(G)
 6       do ENQUEUE(c, ready_queue)
 7  while ready_queue ≠ ∅
 8       do
 9          t ← FIND-EARLIEST-DEADLINE(ready_queue)
10          DEQUEUE(t, ready_queue)
11          p ← FIND-FIRST-AVAILABLE-PROCESSOR()
12          SCHEDULE(t, p, S)
13          proc_ready[p] ← proc_ready[p] + weight[t]
14          for c ∈ successors[t]
15               do
16                  indegree[c] ← indegree[c] −1
17                  if indegree[c] = 0
18                    then ENQUEUE(c, ready_queue)
19  return S
```

FIGURE 5.6 *Pseudo-code for the list scheduling algorithm.*

lower the clock frequency and supply voltage of all processors, as depicted in Figure 5.7a.

This technique is effective as long as the dynamic power consumption dominates the static power consumption. When static power consumption becomes more significant, however, this is no longer the case.

5.5.2 Leakage Aware MultiProcessor Scheduling

Due to the static power dissipation, employing the maximum number of processors will not always result in the least energy consumption. Our *Leakage Aware MultiProcessor Scheduling* (LAMPS) algorithm determines the number of processors that results in the lowest energy consumption.

Many other approaches try to optimize for inter-task or intra-task DVS concur-

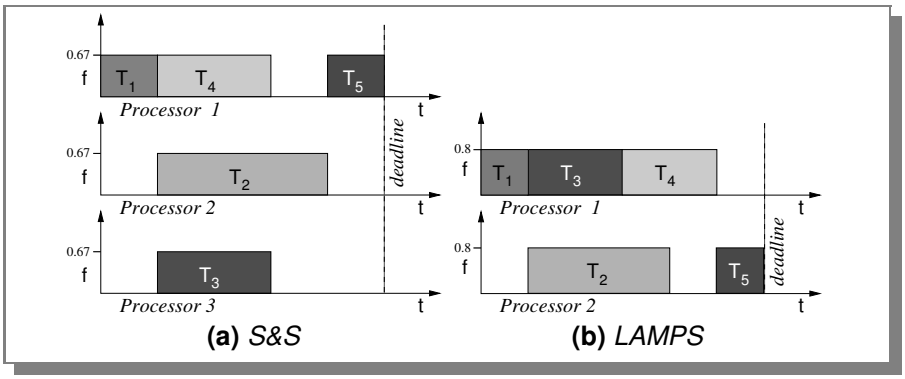

FIGURE 5.7 *Schedules produced by S&S and LAMPS.*

rently with scheduling tasks. In LAMPS, however, the scheduling is performed by a near-linear time algorithm and a single, constant frequency/voltage pair is assumed for all employed processors. By using a simple, near-linear time algorithm for scheduling and voltage selection, LAMPS can easily process large graphs and can spend additional time for optimizing on the number of employed processors.

Our LAMPS algorithm works as follows. Let the task graph be represented by a weighted DAG $G = (V, E, w)$, where V corresponds to the tasks, E to task dependencies, and $w(v)$ denotes the execution time of task v. The algorithm starts with determining the minimal number of processors required to finish the tasks before the deadline. This step is performed as follows. First, lower bound N_{lwb} is established on the number of processors needed to complete the tasks before the deadline D and an upper bound N_{upb} on the number of processors that can be employed efficiently:

$$N_{\text{lwb}} = \lceil \sum_{v \in V} w(v)/D \rceil, \quad N_{\text{upb}} = |V|.$$

Thereafter, a binary search [17] is performed on the interval $[N_{\text{lwb}}, N_{\text{upb}}]$ to determine the minimal number of processors N_{min} required to finish the task graph on time. First, it is determined if $N = (N_{\text{lwb}} + N_{\text{upb}})/2$ are sufficient to finish before the deadline. This is done using the list scheduling algorithm shown in Figure 5.6. If the makespan of the schedule produced by the list scheduler is less than or equal to the deadline, the search continues on the interval $[N_{\text{lwb}}, N]$. If not, the search continues on the interval $[N + 1, N_{\text{upb}}]$.

After having found the minimal number of processors N_{min} required, the number of processors that requires the least amount of energy is determined. This

```
LEAKAGE-AWARE-MULTIPROCESSOR-SCHEDULE(G, deadline)
    ▷ G is a weighted directed acyclic graph
    ▷ CPL is the critical path length of G.
 1  N ← FIND-MINIMUM-PROCS(N_lwb, N_upb, deadline)
 2  E_min ← ∞
 3  repeat S ← LIST-SCHEDULE-EDF(G, N)
 4         f ← makespan(S) · f_max / deadline
 5         E ← CALCULATE-ENERGY-CONSUMPTION(S, f)
 6         if E < E_min
 7           then E_min ← E
 8                S_min ← S
 9         N ← N + 1
10    until makespan(S) = CPL
11  return S_min
```

FIGURE 5.8 *Pseudo-code for the LAMPS heuristic.*

step is performed as follows. First, we determine the total energy consumption for N_{min} processors. This is done by lowering the clock frequency and supply voltage so that the task graph is completed exactly at the deadline, as in the S&S algorithm. We note that in this chapter, unless noted differently, the voltage and frequency are assumed to be scalable on a continuous range. In other words, the schedule is stretched so that it finishes exactly on time. The task graph is also scheduled on $N_{\text{min+1}}$, $N_{\text{min+2}}$, etc. processors and the energy consumption of the schedule is computed, until the makespan is equal to the CPL. At this point, increasing the number of processors will always increase the total energy consumption. The algorithm returns the schedule with the number of processors that requires the least amount of energy. Figure 5.8 depicts the pseudo-code for the LAMPS heuristic.

The reason for performing a linear search instead of a binary search in the second phase of the algorithm is that the energy consumption as a function of the number of processors can have local minima. Consequently, a binary search will not always find the optimal solution. An example of this will be given in Section 5.6.

Figure 5.7b illustrates the schedule generated by LAMPS. Instead of 3 processors, the task graph shown in Figure 5.5a is scheduled on only 2 processors but with a higher frequency. Nevertheless, because the third processor is turned

off, the schedule produced by LAMPS consumes less energy than the schedule generated by S&S.

In Section 5.3.2, it was noted that scaling the frequency to below the optimal frequency will actually increase the energy consumption. However, since the option to shut down processors temporarily is assumed to be not available, results with frequencies below the optimum still reduce the energy consumption.

The time complexity of the algorithm depends on the structure of the task graph and the time it takes to perform list scheduling. Let T_{ls} denote the time required to perform list scheduling. The time T_{LAMPS} taken by the LAMPS algorithm is given by:

$$T_{\text{LAMPS}} = \log_2(N_{\text{upb}} - N_{\text{lwb}}) \cdot T_{\text{ls}} + M \cdot T_{\text{ls}},$$

where M is the number of iterations of the second phase (number of iteration required until the makespan of the generated schedule no longer decreases). In practice, for all benchmarks finding the optimal configuration never took more than six seconds on a 3GHz Pentium 4.

5.6 Experimental Results

In this section, the results of our LAMPS scheduling approach are presented and compared to the S&S algorithm.

5.6.1 Experimental Setup

In the experiments, we assume a technology where leakage current contributes to the overall power consumption to a much larger extend than it does today. Specifically, it is assumed that half of the power consumption at maximum frequency is due to this leakage current ($\delta = 0.5, \sigma = 0.5$). Furthermore, we assume that the threshold voltage is 0.3 times the supply voltage ($\beta_1 = 0.3$, $\beta_2 = 0.7$), which, according to [59], is representative for current technology. In the next chapter, we will also consider other ratios between static and dynamic power. Note that, although we have assumed a relatively high leakage current in this processor, according to Figure 5.2, it is still theoretically possible to reduce energy consumption by voltage/frequency scaling, until the frequency becomes lower than 29% of the maximum frequency.

The experimental results have been obtained using a scheduling tool based on the pseudo-codes in Figures 5.6 and 5.8 and the power model described in

Section 5.3. Table 5.1 lists the benchmarks that have been used, as well as the number of nodes and edges, the length of the critical path, and the total weight of all the nodes (total work). The first three benchmarks have been derived from real applications, while the other three have been randomly generated. These benchmarks were taken from the *Standard Task Graph Set* [52]. Since this set does not provide deadlines, we have used deadlines of 1.5, 2, 4, and 8 times the CPL. We note that other works usually used randomly generated graphs.

name	number of nodes	number of edges	critical path	total work
fpppp	334	1196	1062	7113
robot	88	130	545	2459
sparse	96	128	122	1920
proto001	273	1688	167	4711
proto003	164	646	556	1599
proto279	1342	16762	735	13302

TABLE 5.1 *Six benchmarks from the Standard Task Graph set [52] and their main characteristics.*

5.6.2 Experimental Results

Figure 5.9 depicts the average power consumption of various schedules, normalized to the power consumption of 1 fully active processor at maximum frequency, as a function of the number of employed processors for the case that the deadline is 1.5x the length of the critical path. It can be seen that local minima can exist that are not global minima. This happens, for example, for the *sparse* benchmark at 14 processors. Therefore, a full search must be performed on the number of processors, in order to find the optimum for a certain graph and deadline.

Tables 5.2, 5.3, 5.4, and 5.5 depict the results obtained for deadlines of 1.5, 2, 4, and 8 times the critical path length, respectively. Results are presented for LAMPS as well as for S&S. For each benchmark and scheduling approach, the (optimal) number of processors N, the normalized frequency $\mathcal{F}$, and the normalized total power consumption $\mathcal{P}$ are listed.

For the S&S algorithm, since it produces schedules whose makespans are equal

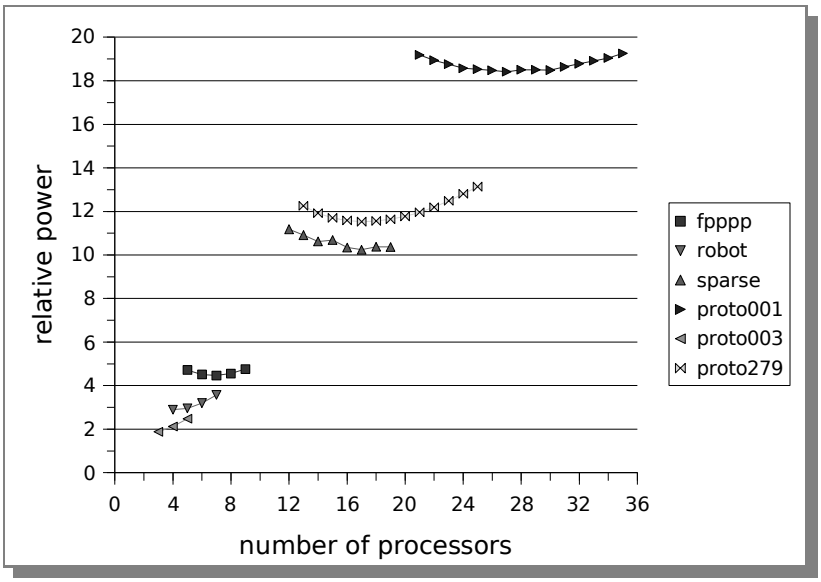

FIGURE 5.9 *Average power consumption of various schedules, normalized to a single fully active processor, for different benchmarks with the deadline at 1.5× the critical path length.*

to the CPL, the normalized clock frequency is given by the ratio of the critical path length to the deadline. Also note that for S&S, the number of processors is independent of the deadline. This algorithm employs as many processors as possible to finish the tasks as early as possible in order to maximize the amount of slack that can be used to lower the clock frequency. LAMPS, on the other hand, uses fewer processors and a slightly higher clock frequency to balance the static and dynamic power dissipation.

Figure 5.10 depicts the power reduction achieved by the LAMPS algorithm compared to S&S. Since the schedules generated by both algorithms finish at exactly the same time (at the deadline), they can be compared by power dissipation instead of energy consumption. LAMPS achieves significant energy savings relative to S&S. Furthermore, as expected, the improvement increases with the deadline. For example, if the deadline is tight (1.5x the critical path length), the relative power reduction ranges from 1% to 24%. On the other hand, if the deadline is loose (8x the CPL), the improvement ranges from approximately 55% to 67%. If the deadline is less strict, fewer processors can be used to finish the tasks on time. This allows LAMPS to improve upon S&S which always employs as many processors as can be used to reduce the

	LAMPS			S&S		
benchmark	**N**	$\mathcal{F}$	$\mathcal{P}$	**N**	$\mathcal{F}$	$\mathcal{P}$
fpppp	7	0.76	4.46	9	0.67	4.76
robot	4	0.82	2.90	7	0.67	3.57
sparse	17	0.72	10.24	19	0.67	10.37
proto001	27	0.79	18.41	36	0.67	19.37
proto003	3	0.74	1.88	5	0.67	2.48
proto279	17	0.78	11.53	25	0.67	13.14

TABLE 5.2 *Results for deadlines of 1.5× the critical path length.*

	LAMPS			S&S		
benchmark	**N**	$\mathcal{F}$	$\mathcal{P}$	**N**	$\mathcal{F}$	$\mathcal{P}$
fpppp	6	0.64	3.19	9	0.50	3.63
robot	3	0.77	2.05	7	0.50	2.75
sparse	14	0.65	7.51	19	0.50	7.84
proto001	22	0.69	13.01	36	0.50	14.68
proto003	2	0.75	1.32	5	0.50	1.93
proto279	15	0.65	8.20	25	0.50	10.04

TABLE 5.3 *Results for deadlines of 2× the critical path length.*

makespan of the generated schedule.

It can also be seen from Tables 5.2 to 5.5 and Figure 5.10 that the amount of improvement also depends on the benchmark. For example, for the *sparse* benchmark a power reduction of only 1% is achieved when the deadline is 1.5x the critical path length, while for *proto003* a power saving of 24% is attained. The reason for this behavior is that LAMPS requires 17 processors to finish *sparse* on time, while S&S requires 19, so only 2 or $2/19 = 10.5\%$ of the processors can be turned off to save power. On the other hand, for *proto003*, 2 out of 5 or 40% of the processors can be turned off, which results in a more significant power reduction. The geometric means of the savings by LAMPS upon S&S are 11%, 17%, 39%, and 61% for deadlines of respectively 1.5, 2, 4, and 8 times the length of the critical path.

	LAMPS			S&S		
benchmark	**N**	$\mathcal{F}$	$\mathcal{P}$	**N**	$\mathcal{F}$	$\mathcal{P}$
fpppp	3	0.58	1.47	9	0.25	2.33
robot	2	0.57	0.98	7	0.25	1.79
sparse	7	0.59	3.48	19	0.25	4.96
proto001	12	0.60	6.19	36	0.25	9.35
proto003	1	0.72	0.63	5	0.25	1.27
proto279	8	0.58	3.97	25	0.25	6.45

TABLE 5.4 *Results for deadlines of 4× the critical path length.*

	LAMPS			S&S		
benchmark	**N**	$\mathcal{F}$	$\mathcal{P}$	**N**	$\mathcal{F}$	$\mathcal{P}$
fpppp	2	0.42	0.75	9	0.13	1.81
robot	1	0.56	0.48	7	0.13	1.40
sparse	3	0.66	1.71	19	0.13	3.83
proto001	6	0.59	3.04	36	0.13	7.24
proto003	1	0.36	0.33	5	0.13	1.00
proto279	4	0.57	1.96	25	0.13	5.01

TABLE 5.5 *Results for deadlines of 8× the critical path length.*

Generally, processors that support DVS can only scale to a fixed number of predetermined voltage/frequency pairs. In this chapter, however, we have so far assumed that the frequency and voltage can be scaled to any value between 0 and the maximum. To show the impact of scaling in discrete steps, we have also performed the experiments with the limitation that the normalized supply voltage can only be scaled in discrete steps of 5% of the maximum frequency, similar to [43]. To perform these experiments, the frequency f calculated in line 4 of the pseudo-code depicted in Figure 5.8 has been rounded up to the next discrete level. Because S&S uses as many processors as possible and because the deadline is based on the length of the critical path, the makespan of a schedule produced by S&S and the employed frequency are solely determined by the deadline. Therefore, the increase in power consumption when using discrete scaling in S&S is independent of the structure of the benchmark. In this

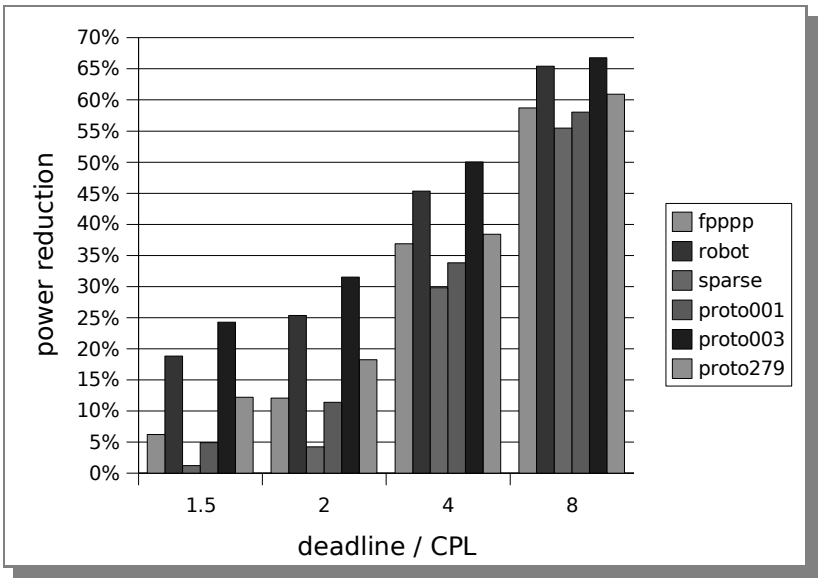

FIGURE 5.10 *Power reduction achieved by the LAMPS scheduling heuristic over S&S.*

case, this increase is completely determined by the distance to the next higher supported frequency and the fraction of time the processors are busy (α). For deadlines of 1.5, 2, 4, and 8 times the length of the critical path, the increases in power consumption for S&S are in the ranges 5.4–5.6%, 0–0%, 5.6–5.8%, and 3.3–3.4% respectively. Because one of the supported normalized frequencies is exactly 0.5, there is no loss when the deadline is set at 2 times the length of the critical path. With LAMPS on the other hand, the operating frequency will vary across the different benchmarks. As a result, the increase in power consumption will be different for different benchmarks.

Figure 5.11 depicts the improvements of LAMPS upon S&S, when scaling the voltage in discrete steps. Because S&S with a deadline of 2 times the CPL does not suffer from using discrete frequency/voltage pairs, the improvements by LAMPS upon S&S for this deadline are always lower for discrete voltages than for continuous voltages. For the other deadlines, the results depend on whether S&S or LAMPS suffers more from having to use discrete steps. If LAMPS has a higher increase in power consumption, the improvements upon S&S compared to the continuous case will be less. Examples of this are *robot* and *proto003* for deadlines of 4 times the length of the critical path. When S&S suffers more from the discretization than LAMPS, on the other hand,

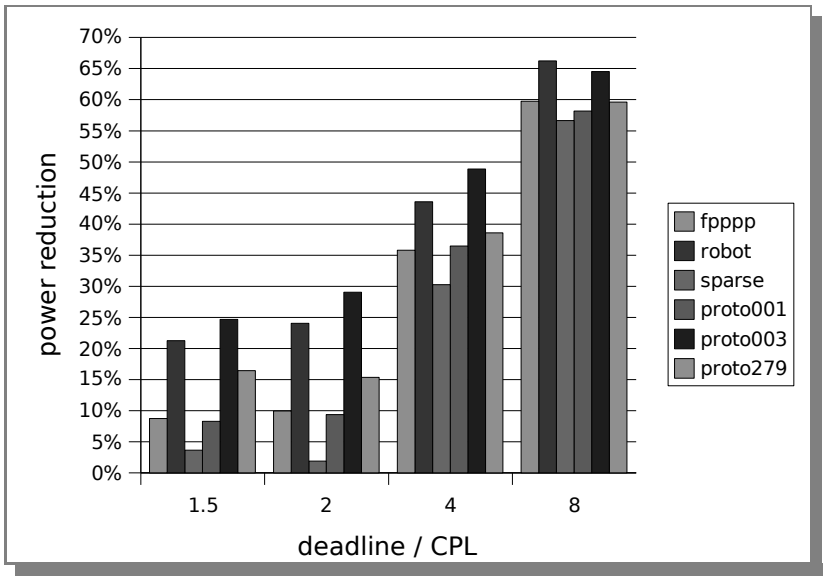

FIGURE 5.11 *Power reduction achieved by the LAMPS scheduling heuristic over S&S, when scaling the voltage in discrete steps.*

the improvements will be higher than in the continuous case. This happens, for example, with all the benchmarks when deadlines of 1.5 times the length of the critical path are used. In general, LAMPS also effectively reduces the energy consumption if voltage scaling is limited to discrete steps. It must be noted, however, that these steps should be chosen carefully, in order not to waste too much power.

5.7 Conclusions

As feature sizes keep decreasing, the contribution of leakage current to the total energy consumption is expected to increase significantly. In this chapter, it was shown that when the static power dissipation becomes more significant, employing the maximum number of processors to maximize the amount of slack that can be used to lower the supply voltage is no longer optimal from an energy perspective.

Depending on the amount of leakage current and the amount of parallelism exhibited by the application, the proposed LAMPS algorithm determines the number of processors, their clock frequency, and the corresponding supply

voltage that minimizes the total energy consumption. The experimental results show that LAMPS reduces the total amount of dissipated power/energy by up to 24% for tight deadlines and by up to 67% for loose deadlines.

When voltage scaling is limited to discrete steps, the energy consumption of a schedule produced by LAMPS will be slightly higher. However, since this is also the case with S&S, the improvements made by LAMPS will be lower in some cases, while they will be higher in other ones. In general, the improvement of LAMPS upon S&S for discrete voltage scaling is close to the improvement for scaling on a continuous range.

Ultimately, the relative amount of energy dissipated by leakage currents is expected to become much larger than the amount of energy dissipated through switching activity. At that point the lowest energy solution will be to perform as much as possible sequentially on one processor, and to employ parallelism only if the required performance demands to do so.

We have assumed that all processors operate at the same frequency. By slowing down some processors more than others, it could be possible to produce a more balanced schedule that consumes less power than the schedule generated by LAMPS. Another option to further reduce power consumption is to allow frequencies to change over time. Both these options, however, will likely make scheduling more complicated and therefore also more time consuming. The option to shut processors down temporarily will be investigated in Chapter 6, which also discusses the possible improvements that can be attained by using separate frequencies per processor and by allowing these frequencies to change over time.

6

Energy Efficient Multiprocessor Scheduling using DVS and DPM

When peak performance is unnecessary, DVS can be used to reduce the dynamic power consumption of embedded multiprocessors. In the previous chapter, it was shown that with an increase in static power consumption, it will be more effective to limit the number of processors employed (i.e., to turn some of them off), instead of using as many processors as can effectively be used to reduce the makespan of the schedule. In this chapter, the previously presented scheduling heuristics are extended with the option to shut processors down temporarily. Experimental results obtained using a public benchmark set of task graphs and real parallel applications show that our approach reduces the total energy consumption by up to 10% for tight deadlines (1.5x the critical path length) and by up to 51% for loose deadlines (8x the critical path length) compared to the heuristic that only employs DVS. Especially with coarse-grain task graphs and tight deadlines, *Leakage Aware MultiProcessor Scheduling with DPM* (LAMPS+DPM) saves more than twice as much energy as LAMPS. We also compare the energy consumed by our scheduling algorithms to two absolute lower bounds, one for the case where all processors continuously run at the same frequency, and one for the case where the processors can run at different frequencies and these frequencies may change over time. The results show that the energy reduction achieved by

our best approach is close to these theoretical limits.

Most of the material in this chapter has been previously published in [21, 23].

6.1 Introduction

As discussed in Chapter 5, the power consumption of high-performance embedded systems is a prime design consideration. In Chapter 5, a heuristic named LAMPS was presented which balances the amount of slowdown using DVS against the number of employed processors. In this chapter, we extend LAMPS as well as S&S with the option to shut processors down temporarily. The technique to shut down parts of a system temporarily when they are idle is often referred to as *dynamic power management* (DPM) [7]. The heuristics presented in this chapter determine the best trade-off between DVS, DPM, and finding the optimal number of processors. The goal of the proposed scheduling heuristics is to minimize the total energy consumption. We refer to the extend versions of S&S and LAMPS as *Schedule and Stretch with DPM* (S&S+DPM) and LAMPS+DPM, respectively.

Furthermore, we formulate two absolute lower bounds that produce schedules that consume the least amount of energy possible. The first is for the case where all processors run at the same frequency throughout the entire schedule. The schedules produced by S&S(+DPM) and LAMPS(+DPM) have this property. The second is for the case where the processors can run at different frequencies and these frequencies may change over time.

Experimental results are obtained using a public benchmark set of task graphs with precedence constraints and real parallel applications. The results show that our best approach (LAMPS+DPM) reduces the total energy consumption by up to 21% for tight deadlines (1.5x the critical path length) and by up to 52% for loose deadlines (8x the critical path length) compared to S&S. Compared to LAMPS, LAMPS+DPM decreases the total energy consumption by up to 11% respectively 14%. We also analyze how the results are affected by the average amount of parallelism, which is defined as the total amount of work divided by the critical path length. Comparing the results to the theoretical lower bounds indicates that there is little room left for improvement. For example, for fairly coarse-grain task graphs, LAMPS+DPM attains over 98% of the possible energy saving, provided the frequency is the same for all active processors and constant throughout the schedule. Compared to a lower bound where the frequency may be different for each processor and where these frequencies may change during execution, LAMPS attains on average over 88%

of the potential savings.

In order to be able to generalize the conclusions from this chapter to a wider range of power consumption scenarios, we also conduct experiments with a power model where the static power consumption is up to 4 times higher and up to 4 times lower than assumed in the rest of this chapter. Since our best approach optimizes on both the number of employed processors and the balance between DVS and DPM, it also attains good savings when the amount of static power consumption is either significantly higher or lower.

This chapter is organized as follows. Section 6.2 contains an overview of related work. The power model and the techniques used to reduce energy consumption are explained in detail in Section 6.3. In Section 6.4, the S&S and LAMPS heuristics are extended with the option to shut down processors temporarily. Experimental results for randomly generated as well as task graphs derived from real applications are provided in Section 6.5. Finally, in Section 6.6, conclusions are drawn and some directions for future research are given.

6.2 Related Work

Related works that do not employ DPM were already covered in Section 5.2 of the previous chapter.

Jejurikar et al. [43] proposed to reduce the frequency and voltage to the critical speed, and to employ DPM for the remainder of time. However, as was already mentioned in Section 5.2 these authors used a single processor and assumed independent periodic tasks. The same real-time model was assumed in [66], but DVS was not considered. Irani et al. [42] also used this model but assumed a continuous voltage range and presented a theoretical analysis of systems which can use DVS and DPM. Specifically, they presented an offline algorithm with a competitive ratio of 3 and an online algorithm with a constant competitive ratio.

Swaminathan et al. [88] proposed an offline I/O device scheduling algorithm to optimize energy savings using DPM. They proposed an NP-complete optimal solver, as well as a near-optimal polynomial time heuristic. Kim et al. [58] proposed a DPM technique to reduce energy in cluster interconnects.

Our work differs in the following ways. As noted in Section 5.2, we focus on multiprocessor scheduling and assume that applications are represented as weighted DAGs. Furthermore, in this chapter we use a detailed power model

and limit the voltage scaling to discrete steps. Besides exploiting DVS and finding the optimal number of processors, this chapter also considers shutting processor down. Finally, the results presented in this chapter are derived from experiments with thousands of graphs from a publicly available set of task graphs and a task graph derived from a real application (MPEG-1).

6.3 Preliminaries

In this section we first describe the power model, which is different from the one used in the previous chapter for reasons explained below. Then, we explain the effect of two primary ways to reduce power dissipation: dynamic voltage scaling and processor shutdown.

Throughout this chapter, the system and application models are the same as explained in Section 5.4 of the previous chapter. However, through this chapter we assume that the voltage can only be scaled to discrete level. Furthermore, this chapter employs a more detailed power model than was used in Chapter 5. This is required, because this chapter assumes that the processors of a multiprocessor system can be independently turned off or put into a deep sleep mode for short amounts of time. As turning a processor off and on again incurs a certain penalty in terms of delay and energy, this cannot be neglected in the power model. Therefore, the power model described in Section 5.3 is no longer sufficient for the techniques described in this chapter. Due to the new power model, the discrete voltage scaling, and the availability of processor shutdown, the effect of using voltage scaling is slightly different from the previous chapter.

6.3.1 Power Model

For this chapter, we use the power model described in [43], which in turn is based on the model and parameters given in [70], where it has been verified with SPICE simulations. In this model, the power consumption of a processor is given by:

$$P = P_{\text{AC}} + P_{\text{DC}} + P_{\text{on}}, \tag{6.1}$$

where P_{AC} is the dynamic power consumption (due to switching activity), P_{DC} is the static power consumption (due to leakage current), and P_{on} is the intrinsic power consumption needed to keep the processor on. Like [43], we assume P_{on}

K_1 =	0.063	K_6 =	$5.26{\cdot}10^{-12}$	V_{th1} =	0.244
K_2 =	0.153	K_7 =	-0.144	I_j =	$4.8{\cdot}10^{-10}$
K_3 =	$5.38{\cdot}10^{-7}$	V_{dd0} =	1.0	C_{eff} =	$0.43{\cdot}10^{-9}$
K_4 =	1.83	V_{bs} =	-0.7	L_d =	37.0
K_5 =	4.19	α =	1.5	L_g =	$4.0{\cdot}10^{6}$

TABLE 6.1 *Constants for 70nm technology ([43, 70]).*

is $0.1W$. The dynamic power is given by:

$$P_{\text{AC}} = a \cdot C_{\text{eff}} \cdot V_{\text{dd}}^2 \cdot f,$$

where a is the activity factor, C_{eff} is the effective switching capacitance, V_{dd} is the supply voltage, and f is the operating frequency. The static power is given by:

$$P_{\text{DC}} = V_{\text{dd}} \cdot I_{\text{subn}} + |V_{\text{bs}}| \cdot I_j,$$

where I_{subn} is the sub-threshold leakage current, V_{bs} is the voltage applied between body and source, and I_j is the reverse bias junction current. The sub-threshold leakage current is given by:

$$I_{\text{subn}} = K_3 \cdot e^{K_4 \cdot V_{\text{dd}}} \cdot e^{K_5 \cdot V_{\text{bs}}},$$

where K_3, K_4, and K_5 are constants. The relation between operating frequency, supply voltage, and threshold voltage is:

$$f = (V_{\text{dd}} - V_{\text{th}})^{\alpha} / L_d \cdot K_6,$$

where L_d represents the logic depth and K_6 and α are constants for a certain technology. Finally, the threshold voltage is given by:

$$V_{\text{th}} = V_{\text{th1}} - K_1 \cdot V_{\text{dd}} - K_2 \cdot V_{\text{bs}},$$

where V_{th1}, K_1, and K_2 are constants. We use the same 70nm technology constants as [43, 70]. These constants are listed in Table 6.1. The maximum frequency of this processor is 3.1GHz, which requires a supply voltage of 1V. Figures 6.1 and 6.2 depict the resulting power consumption and energy per cycle as a function of the normalized operating frequency.

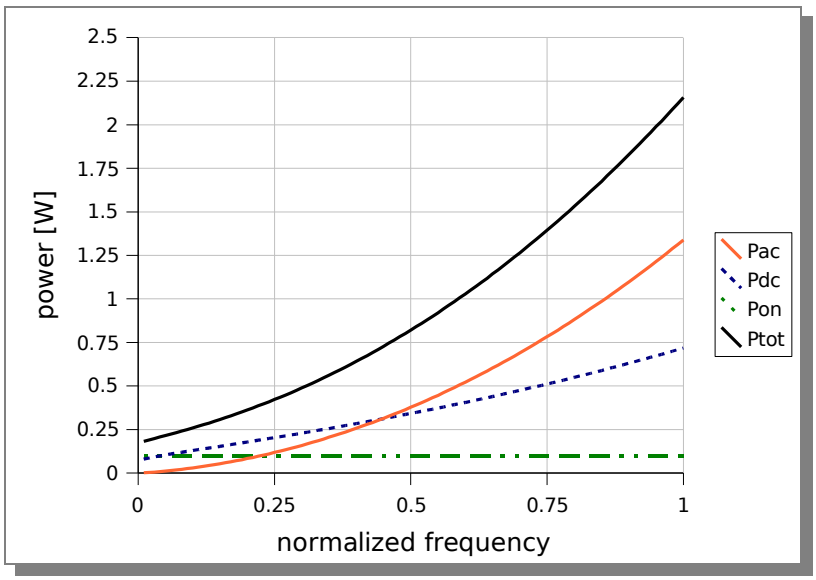

FIGURE 6.1 *Power consumption as a function of the normalized frequency*

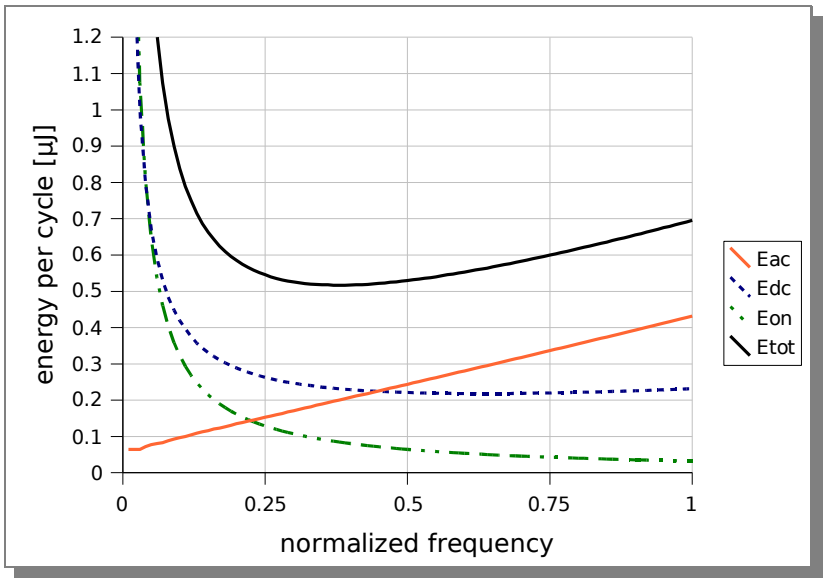

FIGURE 6.2 *Energy consumption as a function of the normalized frequency*

6.3.2 Effect of DVS for the Power Model

DVS mainly reduces the dynamic power consumption, which increases quadratically with the supply voltage. The static component, although having a exponential relation with supply voltage, does not decrease as much with decreasing supply voltage as the dynamic component, as is depicted in Figure 6.1.

As was already explained in Chapter 5, energy equals power times time, the energy consumption will therefore increase if the frequency is decreased below a certain point. Figure 6.2 depicts the energy per cycle as a function of the normalized frequency. It can be seen that the *optimal* or *critical* frequency (f_{crit}) is 0.38 times the maximum. Because of the discrete voltage levels, however, the critical frequency is reached at a supply voltage of 0.7V, corresponding to a normalized frequency of 0.41. Scaling below this frequency will reduce the power consumption but not the total energy consumption, provided that the processors can be shut down for the remaining time. When remaining slack is insufficient to employ DPM, scaling below f_{crit} will, in fact, reduce the total energy consumption.

6.3.3 Processor Shutdown / DPM

The second technique to reduce the energy consumption of a multiprocessor system is to put idle processors temporarily in a deep sleep or shutdown mode. The advantage of this technique over DVS is that it reduces all terms of the total power consumption, not only the dynamic part. When shutting down a processor, however, the contents of, e.g., caches and branch predictors are lost. When a processor is switched back on, they have to be warmed up again, which causes additional delay and consumes extra energy. We use the estimates of Jejurikar et al. [43], who estimated that a processor in sleep state consumes about $50\mu W$ of power and that shutting down and resuming a processor incurs an energy overhead of $483\mu J$. This overhead includes the supply voltage switching as well as the energy spent to warm up caches and predictors. The additional delay incurred by temporarily powering down can be hidden by waking up the processor a short time before the end of the idle period. This is possible because the task graphs are static and the schedules are determined a priori.

The technique where idle parts of a system are temporarily shut down is most often referred to as DPM. Note, however, that when DPM is used within a processor, this implies that idle parts of the processor are turned off. In this work, we assume that processors are turned off completely. In other words, we employ DPM on a multiprocessor system instead of on separate processors.

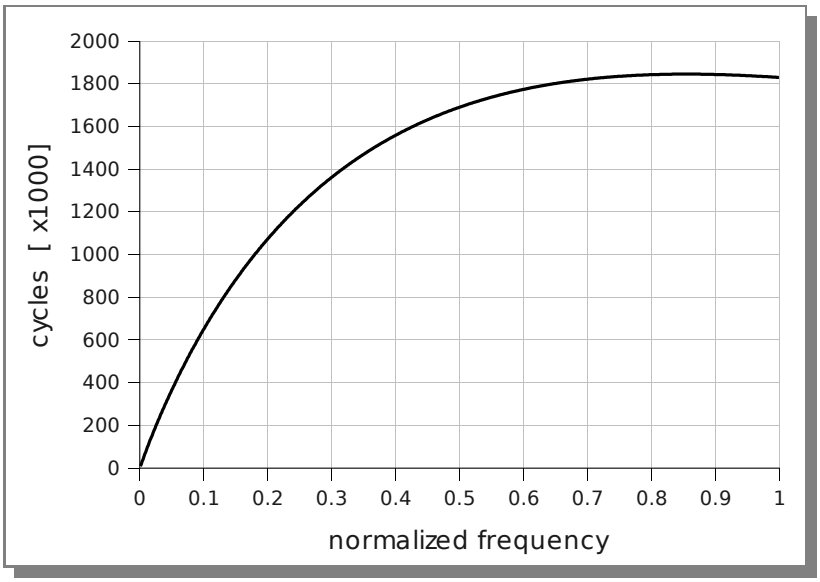

FIGURE 6.3 *Minimum number of idle cycles required for processor shutdown to be beneficial, as a function of the normalized processor frequency.*

Because of the energy overhead of shutting down and waking up a processor, processor shutdown is only beneficial if the processor is idle for a sufficiently long period. Figure 6.3 depicts this minimum number of idle cycles as a function of the normalized frequency. It shows that, in order to save energy consumption by putting a processor temporarily in shutdown mode, a significant number of idle cycles is required. When clocked at half the maximum frequency, for example, an idle period of at least 1.7 million cycles is required. Since in most cases applications with rather fine-grain tasks will have relatively short idle periods (unless the task graph is very unbalanced), such applications will in general not benefit from shutting down processors temporarily between the execution of two tasks. However, it might still be energy efficient to shut down at the end of the schedule, provided the deadline is relatively long.

6.4 Multiprocessor Scheduling with DVS and DPM

In this section we extend the S&S and LAMPS scheduling approaches with the option to shut down processors temporarily. In the schedules produced by

these approaches, all processors run at the same operating frequency and this frequency is constant throughout the whole schedule. As described in Chapter 5, both S&S and LAMPS employ LS-EDF to perform the actual scheduling. EDF does not necessarily produce the best schedule, however. To investigate if other scheduling algorithms could result in additional energy gains, we also present an ideal model in which idle processors are assumed to consume no energy. Furthermore, we also show the improvements that could be attained if the frequency could vary among processors and over time.

6.4.1 S&S+DPM and LAMPS+DPM

We extend S&S with the option to shut down processors temporarily. We refer to this heuristic as S&S+DPM. In S&S+DPM, the task graph is again first scheduled using the EDF policy. Thereafter, the optimal balance between processor slowdown (through DVS) and shutdown is determined by gradually scaling the operating frequency from the maximum frequency to the minimum frequency required to meet the deadline.

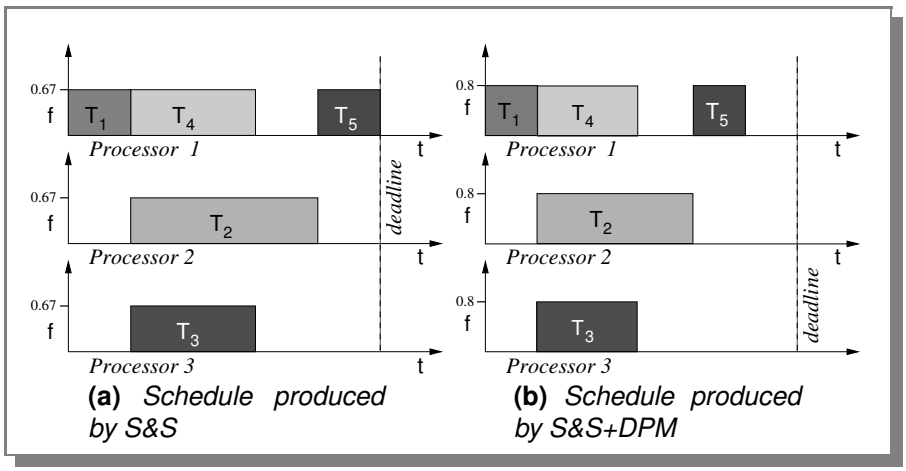

FIGURE 6.4 *Illustration of S&S and S&S+DPM.*

Figure 6.4 illustrates the different schedules produced by S&S and S&S+DPM. In both cases the example task graph from Chapter 5 (Figure 5.5a) is first scheduled using LS-EDF to minimize the makespan of the schedule, or in other words, maximize the amount of slack before the deadline. Thereafter, the slack that remains at the end of the schedule is used to lower the clock frequency and supply voltage of all processors. In the schedule depicted in Figure 6.4b, only

```
SCHEDULE-AND-STRETCH-WITH-DPM(G, deadline)
    ▷ G is a weighted directed acyclic graph.
    ▷ CPL is the critical path length of G.
 1  N ← FIND-MINIMUM-PROCS(N_lwb, N_upb, CPL)
 2  S ← LIST-SCHEDULE-EDF(G, N)
 3  E_min ← ∞
 4  f_min ← ⌈CPL · f_max / deadline⌉
 5  for f ← f_min to f_max
 6       do S' ← PROCESS-SHUTDOWN-PERIODS(S, f)
 7          E ← CALCULATE-ENERGY-CONSUMPTION(S', f)
 8          if E < E_min
 9            then E_min = E
10                 S_min = S'
11  return S_min
```

FIGURE 6.5 *Pseudo-code for the SS+DPM heuristic.*

part of the slack at the end of the schedule is exploited to lower the frequency. The remaining slack is used to shutdown processors temporarily. We note that this example is merely meant for illustration. In reality, for the given task graph it is not advantageous to employ processor shutdown.

Determining the balance between slowdown and shutdown is performed by gradually scaling the operating frequency from the maximum frequency to the minimum frequency required to meet the deadline using discrete voltage level steps of 0.05V. For each frequency, the remaining slack both inside as well as at the end of the schedule is used to shut down processors, provided the idle period is longer than the minimum idle period to result in energy savings (cf. Figure 6.3). In other words, the slack is only used to shut down a processor if it is large enough to make up for the additional energy consumption due to loss of state.

Figure 6.5 depicts the pseudo-code of the S&S+DPM heuristic. The heuristic starts on line 1 with a binary search for the minimum number of processors, as explained in Section 5.5.2. In this case, however, we do not search for the minimum number of processors required to meet the deadline, but instead search for the minimum number of processors that results in a schedule with a makespan equal to the CPL. After setting S to schedule for N processors and initializing E_{min} to infinity, the minimum required frequency f_{min}

```
LEAKAGE-AWARE-MP-SCHEDULE-WITH-DPM(G, deadline)
    ▷ G is a weighted directed acyclic graph
    ▷ CPL is the critical path length of G
 1  N ← FIND-MINIMUM-PROCS(N_lwb, N_upb, deadline)
 2  E_min ← ∞
 3  repeat S ← LIST-SCHEDULE-EDF(G, N)
 4         f_min ← ⌈makespan(S) · f_max / deadline⌉
 5         for f ← f_min to f_max
 6             do S' ← PROCESS-SHUTDOWN-PERIODS(S, f)
 7                E ← CALCULATE-ENERGY-CONSUMPTION(S', f)
 8                if E < E_min
 9                  then E_min ← E
10                       S_min ← S'
11         N ← N +1
12    until makespan(S) = CPL
13  return S_min
```

FIGURE 6.6 *Pseudo-code for the LAMPS+DPM heuristic.*

is determined in line 4. Since in this chapter we assume discrete voltage levels and frequencies, this minimum frequency is rounded up to the next supported frequency. Thereafter it enters a loop over all frequencies from f_{min} to f_{max}. Then, for each frequency the sufficiently long idle periods are used to shutdown processors, after which the energy consumption is calculated. In lines 8–10 the minimal energy consumption and the corresponding schedule are recorded. It ends by returning the schedule with the minimum energy consumption.

We also enhance the LAMPS heuristic with the option to shut down processors and refer to the resulting heuristic as LAMPS+DPM. As in LAMPS, the number of processors that minimizes the total energy consumption is determined by iterating over the number of employed processors, and by calculating the energy consumption for every found schedule. For each number of processors, the balance between DVS and processor shutdown is determined by scaling the frequency from the maximum to the minimum frequency required to meet the deadline. For each frequency, we then use the available slack to shut down processors, similar to the S&S+DPM heuristic.

The pseudo-code for the LAMPS+DPM heuristic is depicted in Figure 6.6.

In line 1 the heuristics starts by determining the minimum number of required processors to finish the task graph before the deadline. As in S&S, S&S+DPM, and LAMPS this is done with a binary search. Then, the graph is scheduled and evaluated repeatedly. Starting with the minimum number, this is done for an increasing number of processors, until the makespan of the schedule equals the CPL. The schedule is evaluated as follows. In line 4 the minimum frequency is calculated, which is rounded up to the next available frequency. As with S&S+DPM, the inner loop (lines 5–10) iterates over all frequencies that are sufficient for the schedule to meet the deadline and calculates for each frequency the resulting energy consumption. It finishes by returning the schedule that results in the lowest energy consumption.

If T_{ls} denotes the time required to perform list scheduling, the time complexity of the LAMPS+DPM heuristic is given by:

$$T_{\text{LAMPS+DPM}} = \log_2(N_{\text{upb}} - N_{\text{lwb}}) \cdot T_{\text{ls}} + M \cdot T_{\text{ls}} \cdot F,$$

where M is the number of different processor counts (number of iterations of the repeat loop) and F the number of different voltages. For all benchmarks finding the optimal configuration never took more than 20 seconds on a 3GHz Pentium 4. This is mainly due to the fact that the employed scheduling algorithm runs in near-linear time.

6.4.2 LIMIT-SF & LIMIT-MF

In the approaches described above, the schedule is always produced by EDF. It is known, however, that EDF is not always optimal for multiprocessor scheduling. Furthermore, in our approaches the frequency is always constant throughout the entire schedule. To investigate if additional energy can be saved by employing a different scheduling algorithm or by allowing different frequencies, we also define two lower bounds, one for the case with a single frequency (LIMIT-SF) and one for the case where multiple frequencies are allowed (LIMIT-MF).

LIMIT-SF has the following characteristics. First, idle processors are assumed to consume no energy at all. In other words, only active cycles are considered when calculating the energy consumption and, consequently, there is no benefit from or penalty for shutting down processors. Second, the number of processors is equal to the number of tasks. Since idle processors consume no energy, using fewer processors will not reduce the energy. Third, the frequency is scaled down to the optimal frequency if possible to meet the deadline, or

otherwise as much as possible. No schedule can consume less energy than this ideal model, provided that the frequency is the same for all active processors and is constant throughout the schedule.

The difference between LIMIT-MF and LIMIT-SF is that in LIMIT-MF all tasks are scheduled at the critical frequency. Because of this and since idle processors are assumed to consume no energy, LIMIT-MF is an absolute lower bound, even for the case where processors can run at different speeds and where the frequency may change over time. We note, however, that it may happen that the schedule produced by LIMIT-MF does not meet the deadline.

Since both LIMIT-SF and LIMIT-MF do not depend on any particular scheduling algorithm, this implies that these results cannot be improved by employing a different scheduling algorithm than EDF.

6.5 Experimental Results

In this section, we present and compare the results of the different scheduling approaches. We use the same power model as used by [70] and [43], as explained in Section 6.3. We again emphasize that a processor in sleep state consumes 50μW and that shutting down and waking up a processor dissipates 483μJ of energy.

6.5.1 Experimental Setup

For the experiments we use task graphs from the Standard Task Graph Set [52] as in Chapter 5, as well as a task graph for *MPEG-1* encoding presented by Zhu et al. [103]. The *MPEG-1* encoding task graph consists of an encoding sequence of 15 I, B, and P frames, and is depicted in Figure 6.7. The execution times of these tasks are assumed to be 36.7M, 17.8M, and 73.4M cycles for I, B, and P frames respectively. We have used the maximum execution times for the *Tennis* sequence as presented in [103], scaled to match the maximum clock frequency of 3.1GHz. The deadline was set at 0.5 seconds for a *group of pictures* (GOP) of 15 frames, to match a real-time encoding requirement of 30 frames per second.

Like in Chapter 5, we use the 3 graphs from the Standard Task Graph Set which were generated from actual applications: *fpppp*, *robot*, and *sparse*. This set also contains 2700 randomly generated graphs, grouped by the number of nodes. Each group in this set consists of 180 different graphs. Whereas in

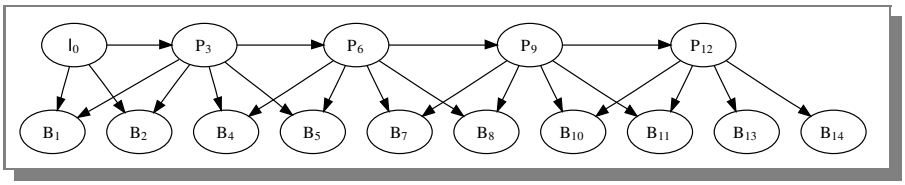

FIGURE 6.7 *Dependence graph for processing 15 MPEG-1 frames.*

name	number of nodes	number of edges	critical path	total work
fpppp	334	1196	1062	7113
robot	88	130	545	2459
sparse	96	128	122	1920
50	50	66–926	24–447	204–644
500	500	698–24497	67–1941	2563–5530
5000	5000	7132–2491411	62–17386	27009–54010

TABLE 6.2 *Employed benchmarks from the Standard Task Graph set [52] and their main characteristics.*

Chapter 5 three randomly generated graphs were used, the experiments for this chapter were performed for all graphs in this collection. We only present the results for a limited number of groups in this work, since the results for graphs of other sizes are comparable. For both the real application graphs and the groups of random graphs, the number of nodes and edges, the critical path length, and the sum of all node weights (total work) are listed in Table 6.2.

As in Chapter 5, since the Standard Task Graph Set does not provide deadlines, we use deadlines of 1.5, 2, 4, and 8 times the CPL when running at the maximum frequency of 3.1GHz. It also does not define the unit of the task weights. Instead, the weights are given as integers in the range from 1 to 300. Therefore, two different scenarios are considered. In the first scenario, corresponding to rather coarse-grain tasks, a weight of 1 in a task graph implies an execution time of $3.1 \cdot 10^6$ cycles, which is 1 millisecond when running at the maximum frequency of 3.1GHz. In the second scenario, corresponding to relatively fine-grain tasks, the same weight implies an execution time of $3.1 \cdot 10^4$ cycles, which at maximum frequency takes 10 microseconds.

6.5.2 Results for the Standard Task Graph Set

Figures 6.8 and 6.9 depict the relative energy consumption for coarse-grain and fine-grain tasks, respectively. For each scenario, we show the energy consumption for deadlines of 1.5, 2, 4, and 8 times the CPL. Each figure shows the results of the four different approaches explained in Section 6.4, as well as the theoretical limits. Throughout this section, S&S is used as the baseline against which we compare the other heuristics.

First, we compare the energy consumption of the schedules produced by the LAMPS heuristic to the energy consumption of the schedules generated by S&S. Although a similar comparison was already made in Chapter 5, it is repeated here because we use a different power model and a partially different set of benchmarks, and because in this chapter we limit voltage scaling to discrete levels. Figures 6.8 and 6.9 show that LAMPS improves upon S&S mainly for less strict deadlines. This can be expected because for tight deadlines (1.5x the CPL), LAMPS requires the same or nearly the same number of processors as S&S to meet the deadline, and therefore consumes the same or nearly the same amount of energy as S&S. In other words, if the deadline is tight, there is less opportunity to turn off processors. For loose deadlines (8x the CPL), on the other hand, LAMPS consumes significantly less energy than S&S, simply because it can employ fewer processors. In this case LAMPS reduces the total energy consumption by 43% on average compared to S&S with a maximum of 48%. For fine-grain tasks, depicted in Figure 6.9, the relative differences between S&S and LAMPS are the same as with coarse-grain tasks, since both heuristics do not shut down processors. The differences between these results and the results from Chapter 5 are mostly due to the different power model.

We now compare S&S+DPM to S&S. Because S&S employs a large number of processors, it consumes a significant amount of static power. Therefore, S&S+DPM improves upon S&S significantly, by shutting down idle processors temporarily. The gains, in this case, are considerably larger for coarse-grain tasks (23% on average for a deadline of $2\times$ the CPL) than for fine-grain tasks (5% on average for a deadline of $2\times$ the CPL), because in the latter case the slack is often not large enough to make shutdown beneficial.

S&S+DPM also improves upon LAMPS for coarse-grain tasks. For fine-grain tasks, however, LAMPS is usually more energy efficient. Generally, the average length of idle periods in a graph grows with increased granularity. When these idle periods are not long enough, DPM cannot be used effectively. In this case it is beneficial to limit the number of employed processors. When the idle periods are sufficient to make DPM beneficial, as is the case with the

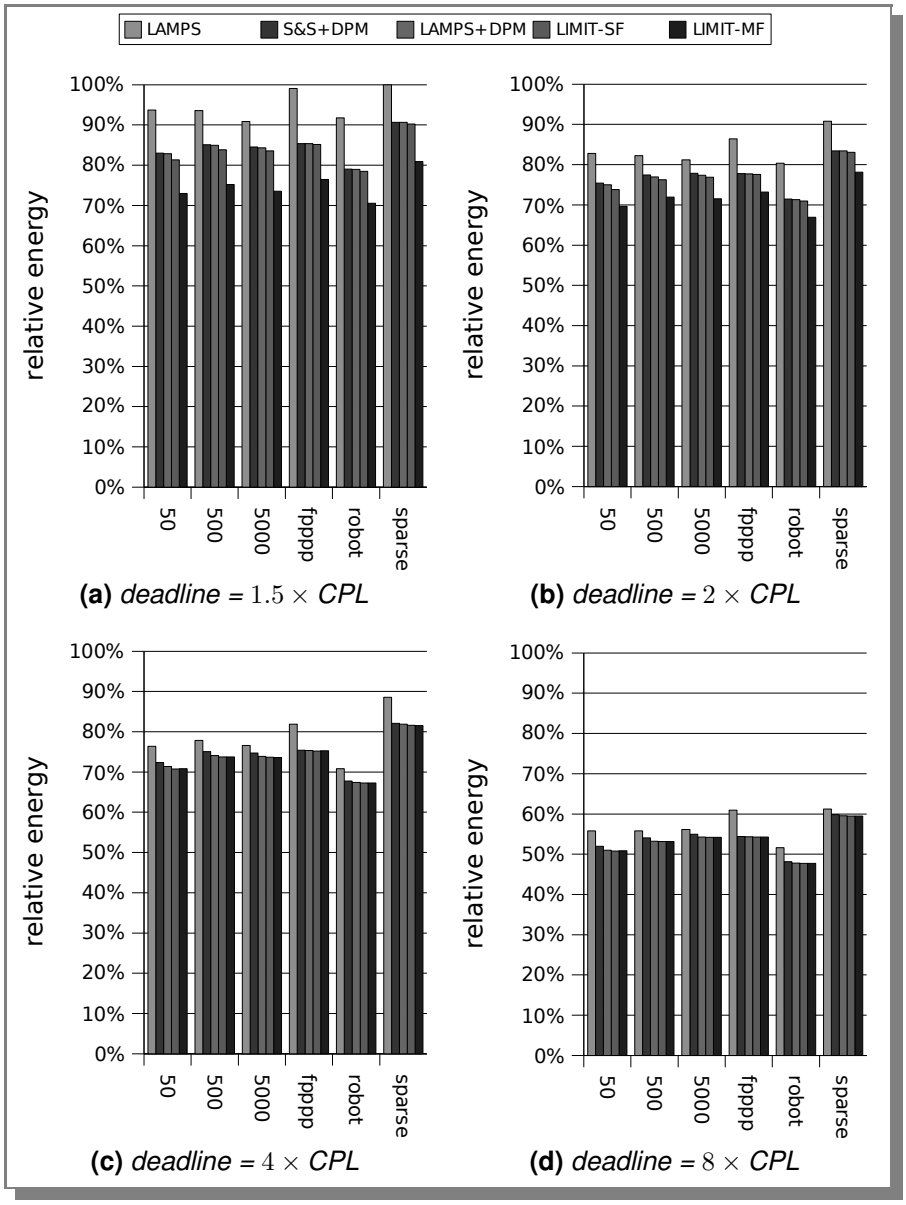

FIGURE 6.8 *Energy consumption relative to S&S for coarse-grain tasks.*

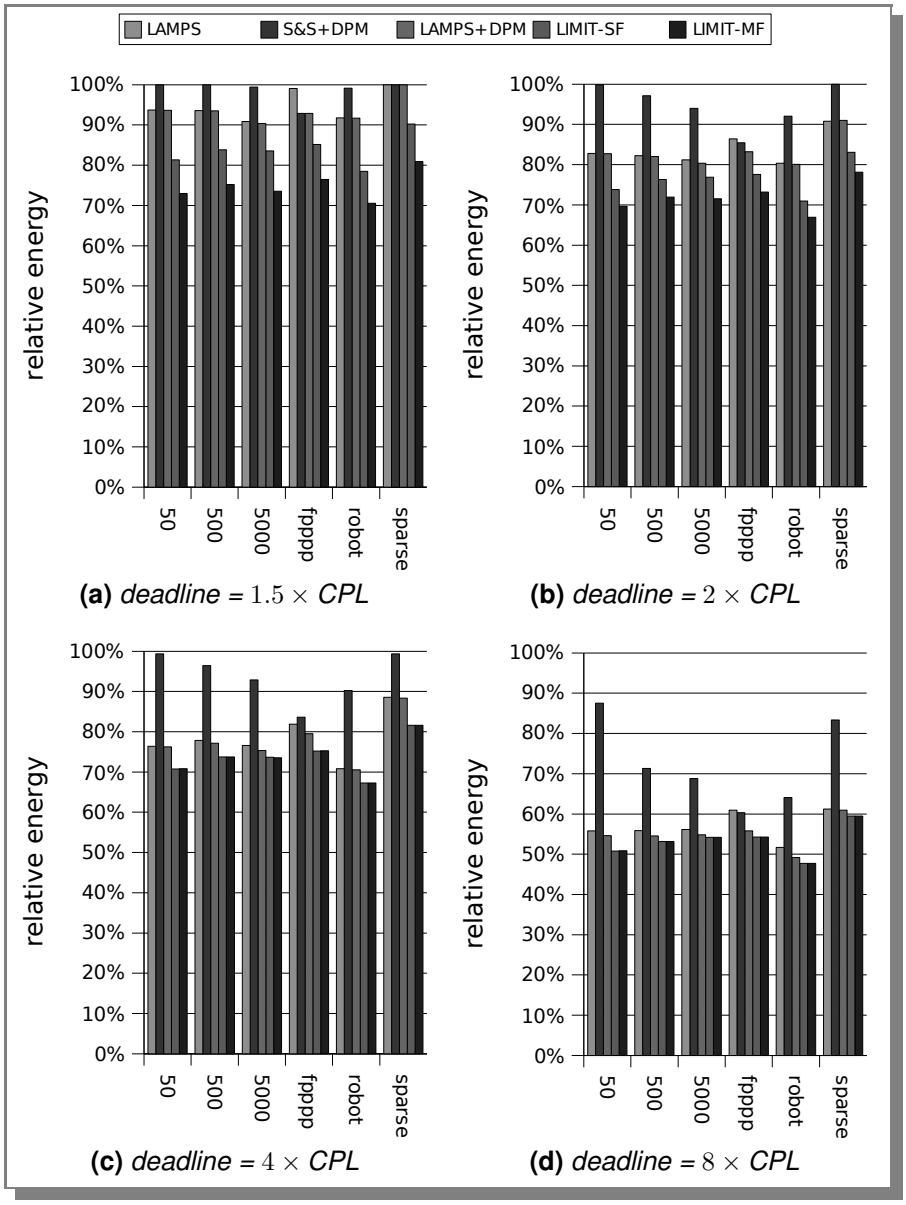

FIGURE 6.9 *Energy consumption relative to S&S for fine-grain tasks.*

coarse-grain task graphs in Figure 6.8, the DPM-enabled approach is more efficient than the approach that only uses DVS but employs the optimal number of processors.

LAMPS+DPM improves upon LAMPS mostly for coarse-grain tasks. Again, the main reason for this is that for fine-grain tasks, the slack is often not large enough to make shutting down worthwhile. With coarse-grain tasks, however, a significant amount of energy can be saved by shutting processors down temporarily. The improvement of LAMPS+DPM over LAMPS is typically less than the improvement of S&S+DPM over S&S. This is because in LAMPS the static dissipation is already reduced by using a smaller number of processors compared to S&S. For coarse-grain tasks, the maximum improvements by LAMPS+DPM upon LAMPS are 11% and 14%, for deadlines of $1.5\times$ and $8\times$ the CPL respectively.

For coarse-grain tasks, the total improvement by LAMPS+DPM upon S&S is 28% on average, with a maximum of 21% for deadlines of $1.5\times$ the CPL and a maximum of 52% for deadlines of $8\times$ the CPL. For fine-grain tasks, LAMPS+DPM improves upon S&S by 23% on average, with a maximum of 10% for deadlines of $1.5\times$ the CPL and a maximum of 50% for deadlines of $8\times$ the CPL.

LIMIT-SF in Figures 6.8 and 6.9 gives an upper limit on the energy savings using our current single-frequency model. Using LIMIT-SF as the maximum attainable reduction, it shows that LAMPS+DPM attains more than 91% of the possible energy reduction with coarse-grain tasks, for all combinations of benchmarks and deadlines. For fine-grain tasks and strict deadlines ($1.5\times$ the CPL), LAMPS+DPM achieves more than 50% of the potential savings on 54% of the benchmarks. In this case, the idle periods are not long enough to apply DPM, while LIMIT-SF assumes that no energy is consumed during these periods. In other words, LIMIT-SF may be far from the actual lower bound. With less strict deadlines ($2\times$ the CPL), LAMPS+DPM attains more than 53% of the possible savings on all benchmarks.

In Figures 6.8 and 6.9, LIMIT-MF is an indication for the possible improvements that could be attained by allowing the processors to run at a different frequency, and by allowing these frequencies to change over time. The results indicate that there is hardly any room for improvements when the deadline is relatively loose. In fact, when the deadline is $4\times$ the CPL or larger, the energy consumed by LIMIT-MF is equal to the energy consumed by LIMIT-SF. For stricter deadlines, some savings may be attained, but mostly for fine-grain tasks. In the case of fine-grain tasks with strict deadlines ($1.5\times$ the CPL),

LAMPS+DPM attains on average only 24% of the savings attained by LIMIT-MF, because the periods of inactivity are often too small to make shutting down worthwhile. In this case, allowing varying frequencies might result in some additional savings. However, when the deadline is less strict and/or the task graph is fairly coarse-grained, shutting down processors becomes worthwhile. For example, using coarse-grain tasks with a deadline of $4\times$ the CPL, the LAMPS+DPM attains for each benchmark at least 98% of the energy reduction attained by LIMIT-MF. In this case, scheduling tasks at different frequencies will not provide a significant improvement. It should be noted that LIMIT-MF is not aware of any deadline. The actual lower bound may therefore be much larger, especially for strict deadlines.

To further explain why LAMPS and LAMPS+DPM provide significant energy savings for certain task graphs, Figure 6.10 depicts the total energy divided by the total work as a function of the average amount of parallelism. Figure 6.10a depicts these results for coarse-grain tasks, while Figure 6.10b shows the results for fine-grain tasks. In both cases, a deadline of $2\times$ the CPL is used. The total energy has been divided by the total work because there is almost a linear relationship between them. The average amount of parallelism is defined as the total work divided by the CPL. A linked list, for example, has an average amount of parallelism of 1. Each dot represents one task graph, and both figures depict the results for randomly generated graphs with 1000, 2000, 2500, and 3000 nodes. In Figure 6.10a, the results for S&S+DPM are barely visible since they are almost identical to the results produced by LAMPS+DPM and are hidden below those.

From the figures it can be seen that the energy consumption per unit of work for S&S increases significantly when the average amount of parallelism becomes small. The same is visible for S&S+DPM with fine-grain tasks. This shows that especially when the average amount of parallelism is small, the energy consumption will increase when the option to shut down processors is not available or cannot be used effectively. The reason for this is that S&S will try to use as many processors as possible, but when the parallelism is low, they will be idle but continue to consume energy. For fine-grain tasks, the idle periods are often not long enough to save energy by shutting processors down temporarily, which is why S&S+DPM with fine-grain tasks consumes significantly more energy than LAMPS and LAMPS+DPM. For both LAMPS and LAMPS+DPM, a small amount of parallelism has no significant effect on the energy consumption per unit of work, as both approaches can decide to use fewer processors.

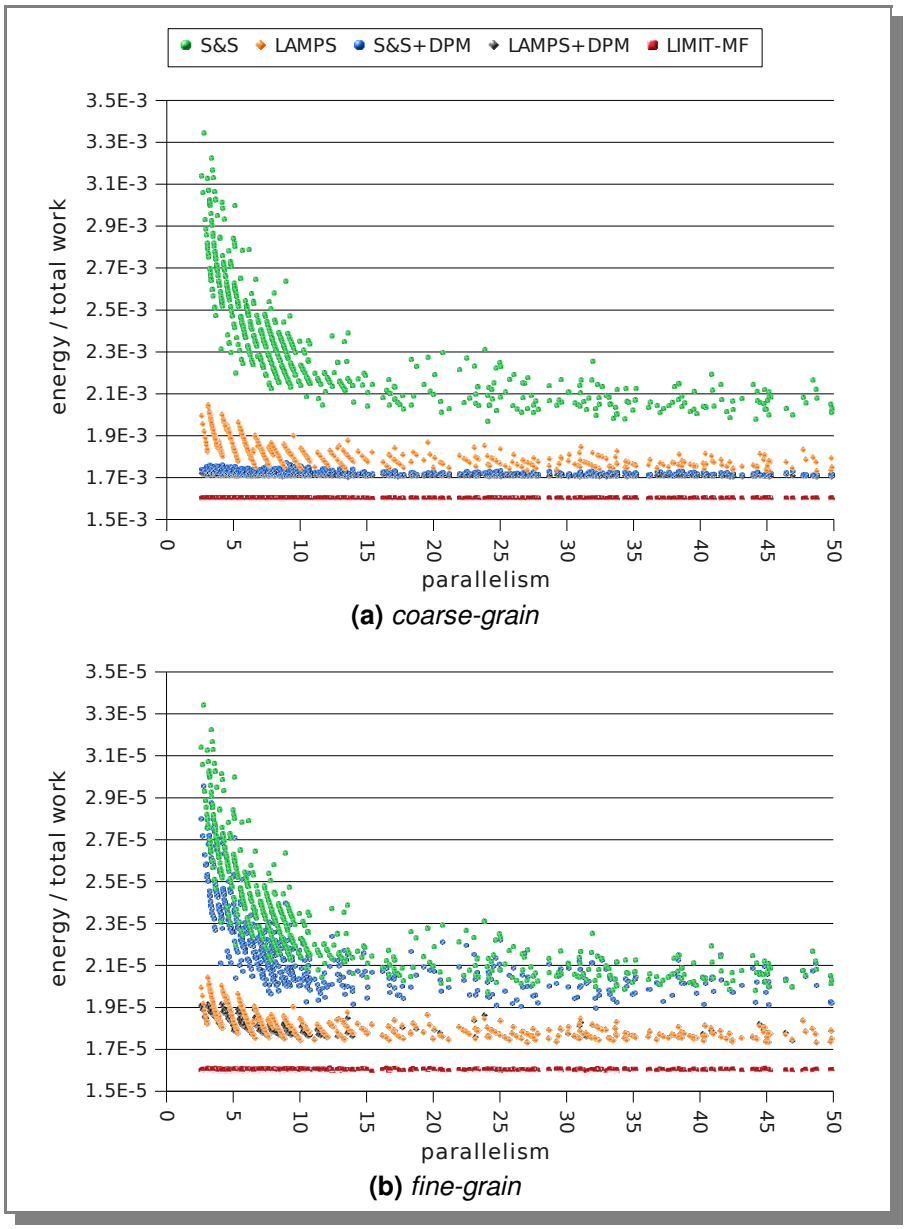

FIGURE 6.10 *Energy/total work as a function of the average amount of parallelism for coarse- and fine-grain tasks. Each dot represents one task graph.*

Benchmark	Consumed energy [J]	Number of processors
S&S	18.116	7
LAMPS	13.290	3
S&S+DPM	10.949	7
LAMPS+DPM	10.947	6
LIMIT-SF	10.940	N/A
LIMIT-MF	10.940	N/A

TABLE 6.3 *Energy consumption relative to S&S for the MPEG-1 benchmark using various approaches.*

When looking closely it can be seen that some dots are clustered to form almost a line. These clustered results, especially visible when parallelism is low, are schedules that employ the same number of processors. For solutions with the same number of processors, the energy consumption decreases as the average parallelism more closely approaches the number of employed processors. Again, this effect is most clearly visible for S&S and S&S+DPM, which employ as many processors as can effectively be used to exploit parallelism.

6.5.3 Results for MPEG-1

Figure 6.7 depicts the task graph for the *MPEG-1* benchmark. The results from experiments with this benchmark are presented in Table 6.3. Similar to the previous experiments, these numbers were obtained by scheduling the task graph using the heuristics described in Section 6.4 and by measuring the energy consumption using the models described in Section 6.3.

When S&S is used to schedule this graph, it employs as many processors as can be used to reduce the makespan of the graph, which in this case is 7 processors. LAMPS, on the other hand, determines that using 3 processors is more efficient, and is hence able to reduce the energy consumption by about 27% compared to S&S. S&S+DPM also uses the maximum number of processors, but being able to shut processors down temporarily, it reduces the energy consumption by 40% compared to S&S. LAMPS+DPM reduces the energy consumption by nearly the same amount as S&S+DPM, albeit using one processor less. This shows that the periods of slack in the schedule are long enough to offset the cost of using one additional processor. Furthermore, the results for S&S+DPM and LAMPS+DPM are extremely close to the lower limits LIMIT-

SF and LIMIT-MF. From this we conclude that it will not be possible to further reduce the energy consumption by using a different scheduling algorithm or by allowing processors to run at different and/or changing frequencies.

6.5.4 Results for Different Levels of Static Power

In the previous experiments we have user a power model which assumes that, at maximum frequency, the dynamic component is approximately twice as large as the static one. However, several researchers [8, 25] predict that this static component will increase dramatically, eventually surpassing the dynamic component. On the other hand, circuit-level techniques like *adaptive body biasing* (ABB) [70] might be used to reduce leakage current, and hence static power consumption.

In this section, results are presented where the static component of the power consumption is several times higher or lower. This is done by modifying the power model to include a scaling factor for the static component. Equation (6.1) is thus rewritten to:

$$P = P_{\text{AC}} + \delta \cdot P_{\text{DC}} + P_{\text{on}},$$

where δ denotes static power scaling factor, i.e. the amount of static power relative to the baseline model. Note that this scaling factor is simply meant to translate the power model to a situation where the static power consumption is supposed to be δ times higher, and has nothing to do with the scaling of frequencies and voltages.

Figure 6.11 depicts the dynamic energy consumption and the resulting static energy consumption for the baseline model as well as for situations with a static component that is 2 or 4 times higher or 2 or 4 times lower. The total energy per cycle for each of these situations is depicted in Figure 6.12. When the static component is 2 times higher than assumed before (Edc $\times 2$), the processor already dissipates more power statically than dynamically, even when clocked at the highest frequency. When the static component is assumed to be twice as low (Edc $/2$), the static component is smaller than the dynamic one for relative frequencies as low as 25%. In both these cases, the critical frequency (i.e. the frequency corresponding to the lowest energy per cycle) stays close the critical frequency in the baseline model. In fact, because we employ a limited number of discrete frequency/voltage pairs, the rounded critical frequency is identical for baseline model and these two deviates. When the static component is scaled up or down by a factor of 4, on the other hand, the critical frequency does change.

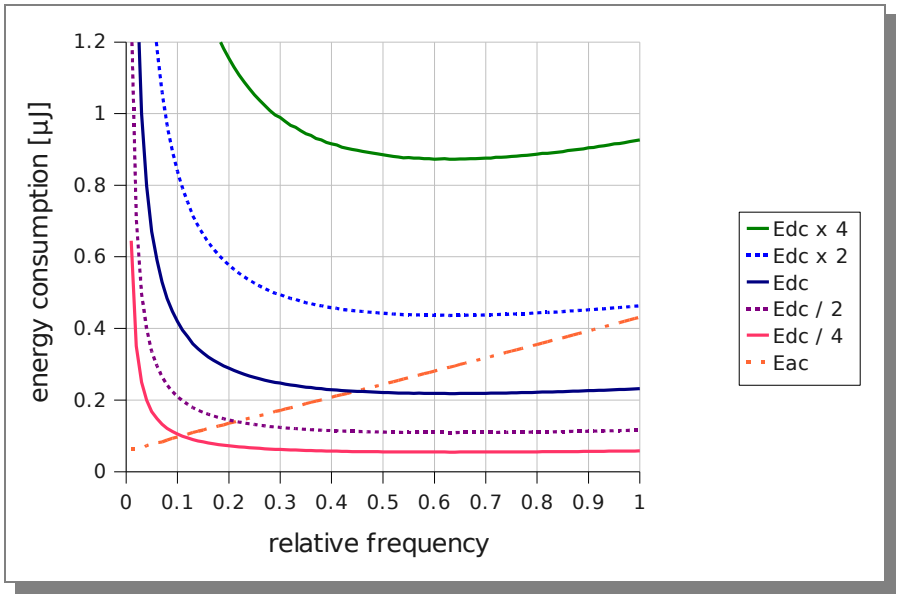

FIGURE 6.11 *Energy consumption as a function of the relative frequency for different levels of static energy consumption.*

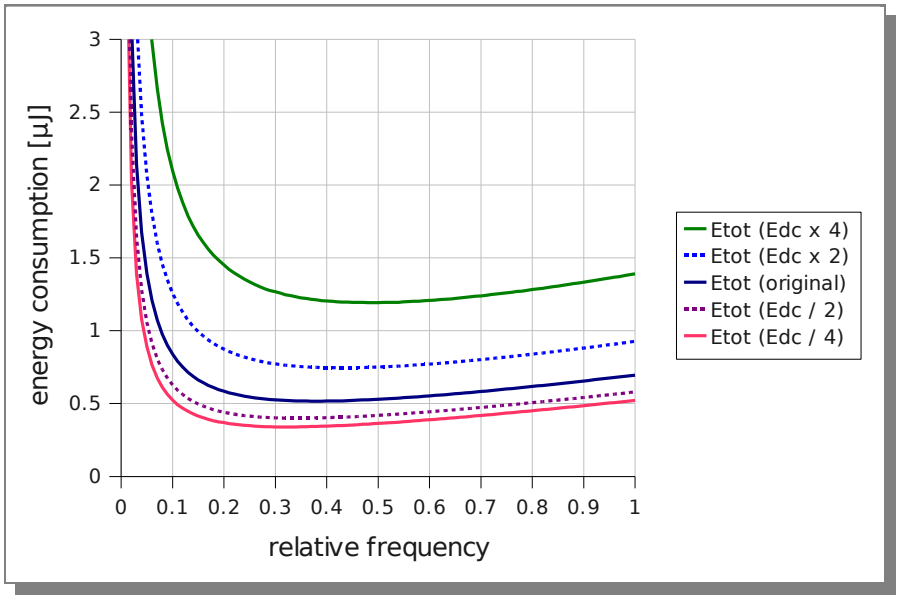

FIGURE 6.12 *Total energy consumption as a function of the relative frequency for different levels of static energy consumption.*

Figure 6.13 depicts the energy consumed by the various scheduling approaches relative to the energy dissipated by S&S when the power is 2 or 4 times higher, or when it is 2 or 4 times lower than in previous sections. For brevity, the results of only 2 benchmarks are presented. The other benchmarks behave similar. Figures 6.13a and 6.13b depict the results for the *fpppp* application scheduled for a deadline of 2 times the CPL, for coarse-grain and fine-grain task graphs respectively. Figures 6.13c and 6.13d depict the same results for *robot*. Figure 6.14 depicts similar results for the *MPEG-1* benchmark.

Clearly, when the power consumption due to leakage currents increases, the improvement upon S&S increased for all heuristics. This is expected, since both DPM as well as the technique to optimize on the number of processors benefit from a higher static energy component. However, even when the static component is scaled 4 times higher or 4 times lower, the order between the employed scheduling approaches remains the same. Again, it can be seen that in case of coarse-grain task graphs, only optimizing on the number of employed processors is not sufficient. The energy consumption in this case can be reduced much further by using DPM. With fine-grain task graphs, on the other hand, the periods of inactivity are often too small to effectively use DPM. In this case, it becomes essential to limit the number of employed processors.

In case the static power consumption is significantly lower than the dynamic one, there is clearly a difference between the energy consumption of LIMIT-MF and LIMIT-SF. When static power consumption increases, however, this difference becomes insignificant. From this one can conclude that having processors operate at different and/or changing frequencies is an mostly interesting option in case the dynamic power consumption is significantly higher than the static one. However, this result must be used with care, as both LIMIT-SF and LIMIT-MF are theoretical limits, and there is no guarantee that these limits can really be obtained. Furthermore, per-core DVS and frequently changing frequencies significantly add to the complexity of the design, due to the multiple clock domains. This can easily offset the benefits of finer grain DVS. A similar observation is made in recent work by Herbert et al. [34].

6.6 Conclusions

As feature sizes keep decreasing, the contribution of leakage current to the total energy consumption is expected to increase. Depending on the amount of slack that remains before the deadline, the amount of parallelism, and the granularity of the application, voltage scaling as well as shutting down processors can be

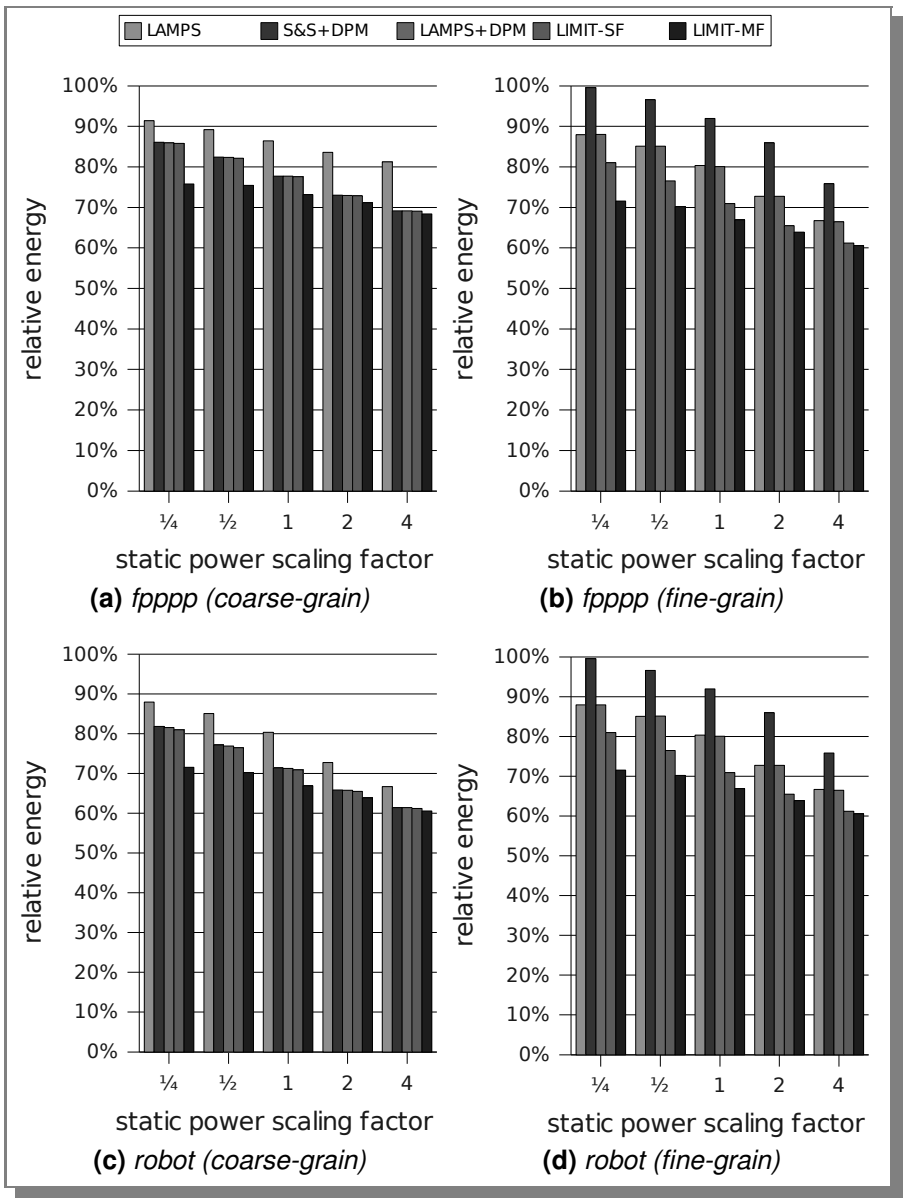

FIGURE 6.13 *Energy consumption relative to S&S for fpppp and robot, using different levels of static power consumption.*

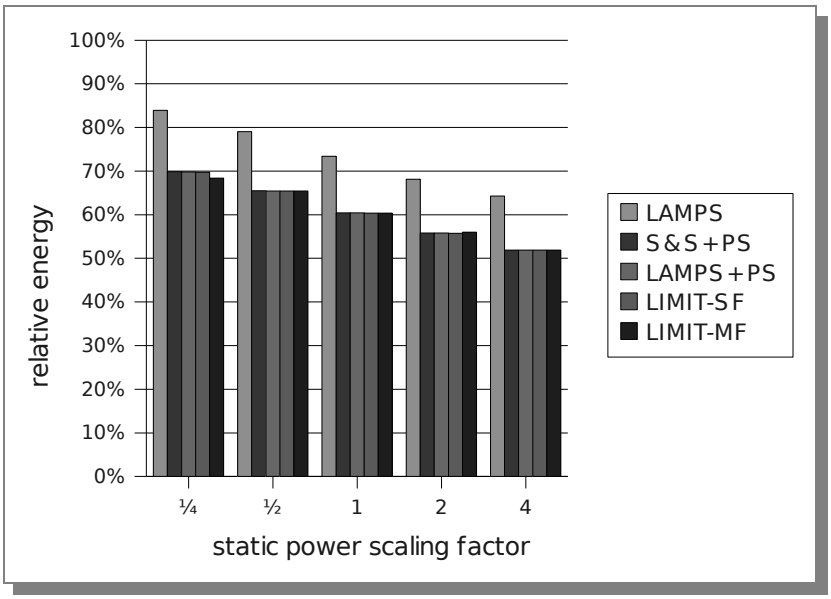

FIGURE 6.14 *Energy consumption relative to S&S for MPEG-1, using different levels of static power consumption.*

used to reduce the energy significantly. At the same time, it is important not to employ too many processors.

By employing a scheduling algorithm with near-linear time complexity, we developed a reasonably fast heuristic that determines an optimal balance between voltage scaling, turning off processors, and choosing the correct number of processors.

For coarse-grain task graphs with long idle periods, DPM was shown to be more effective than optimizing on the number of employed processors. For fine-grain task graphs, the idle periods become too small to make DPM effective. It was shown that, in those cases, optimizing on the number of employed processors is more effective.

We have shown that our best approach, LAMPS+DPM, reduces the energy consumption of a parallel *MPEG-1* implementation by approximately 40% compared to S&S. For a set of randomly generated task graphs, LAMPS+DPM reduces the total energy consumption by up to 21% for tight deadlines and up to 52% for loose ones compared to the S&S algorithm. For coarse-grain tasks and a single frequency, LAMPS+DPM attains at least 92% of the possible energy reduction, i.e., the energy reduction achieved by LIMIT-SF compared to

S&S. Since LIMIT-SF is independent of the scheduling algorithm, this implies that there is almost no room left for improvement by using other scheduling algorithms than EDF. It was furthermore shown that, even when the static component of the power consumption is 4 times higher or lower, the proposed heuristic attains savings close to the limit. This is due to the fact that the heuristic mainly applies DVS when the static power is much lower than the dynamic power, mainly applies DPM when the static power is more significant, and only increases the number of employed processors when this reduces the total energy consumption.

Even when multiple frequencies are allowed, LAMPS+DPM reduces the energy consumption close to the theoretical limit (LIMIT-MF). For loose deadlines ($4\times$ or $8\times$ the critical path length), LIMIT-MF consumes the same amount of energy as LIMIT-SF, and so LAMPS+DPM again attains at least 92% of the potential savings with coarse-grain tasks. As a result, it will be nearly impossible to reduce the overall energy consumption further by using other scheduling algorithms that produce schedules in which different processors can run at different frequencies and in which the frequency can change over time. For more fine-grain tasks and stricter deadlines, some improvements might be attained by using multiple frequencies or by using other scheduling algorithms such as the algorithm proposed by Zhu et al. [103], which maximizes the amount of useable slack, or the integrated approach described in [56]. We intend to investigate the impact of these techniques on fine-grain task graphs in more detail in future research. It should be noted, however, that since LIMIT-MF does not take the deadline into account, real scheduling approaches will probably not reach this limit. Consequently, the actual benefit from having multiple frequencies will probably be much less than suggested by this lower limit. Furthermore, whereas our heuristic runs in (near-linear) polynomial time, employing more advanced scheduling algorithms can easily increase the running time of this heuristic to a higher order polynomial or make it exponential. As stated before in Section 5.2, optimal scheduling using variable voltages has been proven to be NP-complete in many cases [2, 98, 99].

In future research, we also intend to apply the proposed heuristic to other application models, such as periodic independent tasks and dependent task graphs with cycles. Additionally, we have assumed that tasks have a fixed and a priori known execution time. Future research should focus on extending the proposed heuristic to a situation where tasks have a variable or unknown execution time. These extensions could include approaches to profile parallel applications, as well as techniques to perform scheduling and make energy reduction decisions (partially) online.

7 Conclusions

This chapter presents a summary of the objectives and achieved results of this dissertation, and list the main contributions. Finally, it presents some research directions on energy reduction techniques.

This dissertation presented several techniques to reduce energy consumption in processors for both the embedded and high-performance market. The focus was put on two aspects that have growing importance in both research and industry: the memory subsystem (which is a growing consumer of energy and cause of delay), and combining voltage scaling and power management techniques with scheduling interdependent tasks in multiprocessor platforms (which are becoming increasingly common).

For reducing energy in the memory subsystem, traffic between different memory levels (especially between on- and off-chip) was considered a significant source of energy consumption. Furthermore, due to the increasing disparity between the speed of processors and memories, designers employ increasingly aggressive techniques in an effort to provide sufficient data to processors to keep them busy. The observation was made that these efforts often increase the amount of energy consumed in the memory system, and put significant stress on the memory bandwidth. One of the objectives in this work, therefore, was to find ways to reduce the amount of traffic between two cache levels or between a cache and the main memory. Another part of this work focussed at

assisting other existing energy reduction techniques for caches.

For reducing energy in multiprocessor systems, the observation was made that decreasing feature sizes will significantly change the way how energy is dissipated in CMOS logic. Whereas in the past years the energy consumption was completely dominated by the dynamic component (i.e.: energy due to switching between high and low logic levels), in near-future technologies the static component (i.e.: energy due to leakage in non-ideal transistors) can no longer be neglected. Due to this change, multiprocessor systems should no longer aim to employ as much parallelism as possible, since it may be more advantageous to turn off some processor nodes instead. The main objective in the last two chapters of this dissertation, therefore, was to research how much energy can be saved by a scheduling approach that also takes static power consumption into account.

7.1 Summary and Contributions

This dissertation presents several techniques to reduce energy in the memory subsystem, as well as techniques to combine multiprocessor scheduling with energy reduction through slowing or shutting down processors.

Chapter 2 described a hardware technique to efficiently detect conflict misses in caches with no or limited associativity. This technique was then used to construct two cache organizations that exploit this information to predict future conflict misses. These caches were named the *Bypass in Case of Conflict* (BCC) cache and the *Sub-block in Case of Conflict* (SCC) cache. When a conflict is predicted, these two caches do not fetch the whole cache line associated with the request, but only the requested data. Whereas the BCC cache bypasses the cache and transfers the requested data only to the proper register, the SCC cache stores the requested in the cache but invalidates the remainder of the cache line. By predicting conflict misses, both these caches are able avoid cache replacements and significantly reduce memory traffic, thereby reducing energy consumption. While the reduction in memory traffic and energy was overall higher in the BCC cache than in the SCC cache, it also showed to incur more cache misses for a number of benchmarks. The difference in performance between the BCC and the SCC cache is therefore largely determined by application behavior. In this chapter, the following contributions were made:

- A technique was presented to detect conflict misses in caches with limited or no associativity. This technique was based on a component called

the *Conflict Detection Table* (CDT). Using the information supplied by the CDT, recurring conflict misses could be dealt with in an effective way.

- The BCC cache was presented, which bypasses the cache on future references to conflicting addresses.
- The SCC cache was presented, which only fetches and stores the required sub-block for conflicting addresses.
- The BCC and SCC caches were shown to decrease the amount of produced traffic by up to 65% and up to 47%, respectively. This, in turn, translated to significant energy savings. For both cache organizations, performance remained comparable to a conventional cache.

Memory copies were found to cause a significant amount of traffic in multi level memory systems. As a result, programs that make significant use of memory copies (most notable operating systems) may suffer significant performance loss and waste a significant amount of energy. In Chapter 3, a technique was presented to perform copies of memory blocks efficiently in hardware by making use of the existing cache infrastructure and introducing a specialized instruction. By implementing this hardware in multiple memory levels, a technique was then created to perform these memory copies as close to the source data as possible. Experimental results showed that this technique attains significant savings in memory traffic and the number of higher level cache accesses, without sacrificing performance. Since this is achieved using only few additional hardware components, these savings translate to a significant energy reduction. The contributions made in this chapter are as follows.

- A copy engine was presented, which remained simple by reusing existing cache infrastructure. A new instruction was proposed to issue instructions to such copy engines.
- It was shown how these copy engines can be implemented in several layers of the memory system, thereby reducing traffic between these layers.
- A technique was presented to allow the hardware to decide dynamically on the optimal level to perform a memory copy.
- Experiments with a TCP-processing benchmark showed that this technique reduced memory traffic by up to 94%, leading to a decrease in energy consumption and an increase in performance of up to 21%.

Several existing techniques reduce energy consumption in caches by reconfiguring the cache or by throttling power on parts of the cache at runtime. With write-back data caches, however, a large part of such a cache may contain dirty data. As a result, reconfiguration or shutdown is often not possible without first writing back the contents of the dirty cache lines to next memory level. As a large percentage of cache lines may contain dirty data, these energy reduction techniques may be severely limited if there is insufficient room in the write-back buffer. Chapter 4 presented a split organization called the *Clean/Dirty cache* (CD-cache), that is aimed at limiting the number of dirty cache lines. By limiting the number of dirty cache lines, techniques that perform cache reconfiguration or partial shutdown may be implemented on the major part of the cache without requiring additional buffers and control logic for writing back data. It was shown that the proposed cache organization significantly limited the number of dirty cache lines, while attaining a performance comparable to a normal write-back cache. In a case study, it was shown how this cache organization can be used to more efficiently implement energy reduction techniques like cache decay.

- A cache design named the CD-cache was presented, which stores clean and dirty data in separate structures.
- It was shown that the CD-cache bounds the number of dirty cache lines to a hard upper limit, while maintain comparable performance.
- The CD-cache was shown to consume less energy than a normal write-back cache, by directing a significant number of access to a smaller cache structure.
- Using a case study, it was shown how the CD-cache can provide a low complexity solution for dealing with dirty cache lines in energy reduction techniques such as *cache decay*. Experimental results showed that, using the CD-cache instead of a normal write-back cache for cache decay resulted in twice a much energy reduction.

In Chapter 5, it was investigated how *dynamic voltage scaling* (DVS) can be used by the scheduler to reduce energy consumption of a multiprocessor system. For this work, a near-future technology was assumed where, due to decreasing feature sizes, energy consumption due to leakage currents is expected to be much higher. It was shown that significant savings could be attained by optimizing on the number of employed processors. In order to keep the scheduling algorithm reasonably fast, the presented approach uses

the same frequency for all processors and maintains this frequency throughout the schedule. Furthermore, the task graphs were scheduled using a near-linear complexity algorithm, which allows a full search over the limited range of reasonable processor counts. This resulted in the following contributions:

- A heuristic named *Leakage Aware MultiProcessor Scheduling* (LAMPS) was presented, which combines multiprocessor scheduling with an energy reduction technique called DVS, while optimizing on the number of employed processors.
- By employing a non-optimal but fast scheduling routine and by assigning a single frequency-voltage pair to the whole task graph, LAMPS was able to compare results for many different processor counts in relatively short time.
- In a context where frequencies can be scaled on a continuous scale, this technique showed to reduce energy consumption by 24% for tight deadlines and by up to 67% for loose deadlines, compared to an approach that uses as many processors as possible.

In Chapter 6, the approach presented in Chapter 5 was extended by allowing the processors to be shut down temporarily, a technique which is referred to as *dynamic power management* (DPM). The resulting scheduling technique optimizes on the number of employed processors and the level of voltage scaling, while using possible opportunities to shut processors down temporarily. This was again accomplished by using the single frequency model and a near-linear complexity scheduling algorithm. Furthermore, the results were compared to lower bounds for both a single frequency model and a model with fine-grain DVS per processor. It was shown that the presented technique attains savings close to these limits. In this chapter, the following contributions were made:

- A scheduling heuristic named *Leakage Aware MultiProcessor Scheduling with DPM* (LAMPS+DPM) was presented, which attempts to find an optimal combination of DVS, DPM, and the number of processors that should be employed.
- A lower bound named LIMIT-SF was presented, which indicates the absolute lower limit for any scheduling approach, assuming a single frequency/voltage pair is used for all processors and tasks.
- A lower bound named LIMIT-MF was presented, which denotes the absolute lower limit for the situation where a different frequency/voltage pair may be assigned to each task or even to parts of a task.

- Using these lower bounds, it was shown that LAMPS+DPM attains up to 92% of the potential savings with coarse-grain task graphs.
- It was shown that for most cases employing multiple frequencies or using a more optimal scheduling approach would provide limited benefit.

7.2 Possible Directions for Future Work

The conflict detection mechanism for caches, proposed in Chapter 2, sometimes suffers from mispredictions. These mispredictions are mostly caused by the fact that the proposed mechanism can only detect conflicts when the are not predicted as such, but cannot detect whether predicted conflicts are still correct. In future work, the ill effects of these misprediction could be decreased by extending the *Conflict Detection Table* (CDT) with counters to record the number of times a conflict was predicted. This would make it possible to reset an entry after a certain number of predictions, thereby limiting the number of possible consecutive mispredictions. Furthermore, we intend to compare these results to similar work, such as the *victim cache*.

The *Dynamic Copy Engine* (DCE), proposed in Chapter 3, focusses on reducing memory traffic due to memory copies. These memory copies are known to occur significantly in operating system, both for internal operating system tasks as for I/O demanded by applications. Therefore, it would be interesting to perform experiments with a full-system simulator. This would allow for more detailed measurements on the attained savings, by simulating full operating systems and applications in combination with an operating system. Future work could furthermore focus on how this technique could be applied in the context of shared-memory multiprocessor systems.

In Chapter 4, the CD-cache was used to efficiently implement cache decay. The reported energy reductions in this work were limited due to the fact that cache decay was only implemented in a relatively small L1 cache. Experiments should be performed to find the possible savings for large on-chip L2 caches. Furthermore, several other low-power caches that are hindered by dirty cache lines could be evaluated with this technique. Future research may also include experiments combining the proposed cache organizations with fault-tolerant caches.

The heuristics proposed in Chapters 5 and 6 could be extended to include different scheduling approaches, as well as to allow multiple voltage/frequency pairs. However, it should be noted that this is only beneficial when relatively

fine-grain tasks graphs are used, and that it may not be possible at all to attain the lower limit. Furthermore, these scheduling approaches should preferably be performed in polynomial time to allow for optimization on the processor count. Alternatively, a heuristic may be developed that can efficiently estimate to optimal processor count. Another interesting options for future research, is to apply the proposed heuristic to other application models, such as periodic (independent) tasks. In this work, task scheduling and making decisions on how to reduce energy were done offline. Future work could focus on developing techniques to perform these tasks (partially) online, optionally allowing to deal with variable execution times. Future work may also include evaluation of these approaches on real hardware.

Bibliography

[1] M. ALVAREZ, E. SALAMI, A. RAMIREZ, AND M. VALERO, *Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video Codec Applications*, in Proceedings of the International Symposium on Performance Analysis of Systems & Software, 2007, pp. 62–71.

[2] A. ANDREI, M. SCHMITZ, P. ELES, Z. PENG, AND B. M. AL-HASHIMI, *Overhead-Conscious Voltage Selection for Dynamic and Leakage Energy Reduction of Time-Constrained Systems*, in Proceedings of the Conference on Design, Automation, and Test in Europe, 2004, pp. 518–525.

[3] ARM LTD, *ARM11 MPCore*. http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

[4] T. AUSTIN ET AL., *SimpleScalar 3.0*. http://www.simplescalar.com/.

[5] R. BANAKAR, S. STEINKE, B.-S. LEE, M. BALAKRISHNAN, AND P. MARWEDEL, *Scratchpad Memory: A Design Alternative for Cache On-Chip Memory in Embedded Systems*, in Proceedings of the International Symposium on Hardware/Software Codesign, 2002, pp. 73–78.

[6] K. BASU, A. CHOUDHARY, J. PISHARATH, AND M. KANDEMIR, *Power Protocol: Reducing Power Dissipation on Off-Chip Data Buses*, in Proceedings of the IEEE/ACM International Symposium on Microarchitecture, 2002, p. 345.

[7] L. BENINI AND G. DE MICHELI, *Dynamic Power Management: Design Techniques and CAD Tools*, Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[8] S. BORKAR, *Design Challenges of Technology Scaling*, IEEE Micro, 19 (4), 1999, pp. 23–29.

[9] M. CALHOUN, S. RIXNER, AND A. L. COX, *Optimizing Kernel Block Memory Operations*, in Proceedings of the Workshop on Memory Performance Issues, 2006.

[10] F. CATTHOOR, *Energy-Delay Efficient Data Storage and Transfer Architectures and Methodologies: Current Solutions and Remaining Problems*, Journal of VLSI Signal Processing, 21 (3), 1999, pp. 219–231.

[11] F. CATTHOOR, K. DANCKAERT, C. KULKARNI, E. BROCKMEYER, P. G. KJELDSBERG, T. VAN ACHTEREN, AND T. OMNES, *Data Access and Storage Management for Embedded Programmable Processors*, Kluwer Academic Publishers, 2002.

[12] F. CATTHOOR, E. DE GREEF, AND S. SUYTACK, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*, Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[13] F. CATTHOOR, F. FRANSSEN, S. WUYTACK, L. NACHTERGAELE, AND H. DE MAN, *Global Communication and Memory Optimizing Transformations for Low-Power Signal Processing Systems*, in Proceedings of the VLSI Signal Processing Workshop, 1994.

[14] B. CHALAMALA, *Portable Electronics and the Widening Energy Gap*, Proceedings of the IEEE, 95 (11), 2007, pp. 2106 – 2107.

[15] J. CHAPIN, A. HERROD, M. ROSENBLUM, AND A. GUPTA, *Memory system performance of UNIX on CC-NUMA multiprocessors*, in Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, 1995, pp. 1–13.

[16] P. P. CHU AND R. GOTTIPATI, *Write Buffer Design for On-Chip Cache*, in Proceedings of the IEEE International Conference on Computer Design, 1994, pp. 311–316.

[17] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, The MIT Press and McGraw-Hill Book Company, 1989.

[18] P. J. DE LANGEN AND B. H. H. JUURLINK, *Off-Chip Memory Traffic Measurements of Low-Power Embedded Systems*, in Proceedings of the Workshop on Circuits, Systems and Signal Processing (ProRISC), 2002, pp. 351–358.

[19] ——, *Reducing Traffic Generated by Conflict Misses in Caches*, in Proceedings of the ACM International Conference on Computing Frontiers, 2004, pp. 235–239.

[20] ——, *Leakage-Aware Multiprocessor Scheduling for Low Power*, in Proceedings of the International Parallel and Distributed Processing Symposium, 2006.

[21] ——, *Trade-offs Between Voltage Scaling and Processor Shutdown for Low-Energy Embedded Multiprocessors*, in Proceedings of the International Workshop on Computer Systems: Architectures, Modeling, and Simulation, 2007, pp. 75–85.

[22] ——, *Memory Copies in Multi-Level Memory Systems*, in Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2008.

[23] ——, *Leakage Aware Multiprocessor Scheduling*, Journal of VLSI Signal Processing, 57 (1), 2009.

[24] ——, *Limiting the Number of Dirty Cache Lines*, in Proceedings of the Conference on Design, Automation, and Test in Europe, 2009.

[25] D. DUARTE, N. VIJAYKRISHNAN, M. J. IRWIN, AND Y. TSAI, *Impact of Technology Scaling and Packaging on Dynamic Voltage Scaling Techniques*, in Proceedings of the IEEE International ASIC/SOC Conference, 2002.

[26] F. DUARTE AND S. WONG, *A memcpy Hardware Accelerator Solution for Non Cache-line Aligned Copies*, in Proceedings of the International Conference on Application-specific Systems, Architectures and Processors, 2007, pp. 397–402.

[27] K. FLAUTNER, N. S. KIM, S. MARTIN, D. BLAAUW, AND T. MUDGE, *Drowsy Caches: Simple Techniques for Reducing Leakage Power*, in Proceedings of the International Symposium on Computer Architecture, Washington, DC, USA, 2002, pp. 148–157.

[28] S. V. GHEORGHITA, T. BASTEN, AND H. CORPORAAL, *Intra-task scenario-aware voltage scheduling*, in Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, New York, NY, USA, 2005, pp. 177–184.

[29] A. GONZÁLEZ, C. ALIAGAS, AND M. VALERO, *A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality*, in Proceedings of the International Conference on Supercomputing, 1995.

[30] R. GONZÁLEZ, B. GORDON, AND M. HOROWITZ, *Supply and Threshold Voltage Scaling for Low Power CMOS*, IEEE Journal of Solid-State Circuits, 32 (8), 1997.

[31] F. GRUIAN AND K. KUCHCINSKI, *LEneS: Task Scheduling for Low-Energy Systems Using Variable Supply Voltage Processors*, in Proceedings of the Conference on Asia South Pacific Design Automation, 2001, pp. 449–455.

[32] M. R. GUTHAUS, J. S. RINGENBERG, D. ERNST, T. M. AUSTIN, T. MUDGE, AND R. B. BROWN, *MiBench: A Free, Commercially Representative Embedded Benchmark Suite*, in Proceedings of the International Workshop on Workload Characterization, 2001, pp. 3–14.

[33] J. L. HENNESSY AND D. A. PATTERSON, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 3rd. ed., 2003.

[34] S. HERBERT AND D. MARCULESCU, *Analysis of dynamic voltage/frequency scaling in chip-multiprocessors*, in Proceedings of the International Symposium on Low Power Electronics and Design, 2007, pp. 38–43.

[35] H. P. HOFSTEE, *Power Efficient Processor Architecture and the Cell Processor*, in Proceedings of the International Symposium on High-Performance Computer Architecture, 2005, pp. 258–262.

[36] J. HU, M. KANDEMIR, N. VIJAYKRISHNAN, AND M. J. IRWIN, *Analyzing Data Reuse for Cache Reconfiguration*, ACM Transactions on Embedded Computing Systems, 4 (4), 2005, pp. 851–876.

[37] Z. HU, S. KAXIRAS, AND M. MARTONOSI, *Let caches decay: reducing leakage energy via exploitation of cache generational behavior*, ACM Transactions on Computer Systems, 20 (2), 2002, pp. 161–190.

[38] INTEL CORPORATION, *DDR2 Specifications*. http://www.intel.com/technology/memory.

[39] ——, *Intel 64 and IA-32 Architectures Software Developer's Manual*. http://www.intel.com/products/processor/manuals.

[40] ——, *Intel XScale Technology*. http://www.intel.com/design/intelxscale/.

[41] INTERNATIONAL ROADMAP COMMITTEE, *International Technology Roadmap for Semiconductors, 2007 Edition, Executive Summary*. http://www.itrs.net/Links/2007ITRS/ExecSum2007.pdf.

[42] S. IRANI, S. SHUKLA, AND R. GUPTA, *Algorithms for Power Savings*, in Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, 2003, pp. 37–46.

[43] R. JEJURIKAR, C. PEREIRA, AND R. GUPTA, *Leakage Aware Dynamic Voltage Scaling for Real-Time Embedded Systems*, in Proceedings of the Conference on Design Automation, 2004, pp. 275–280.

[44] N. K. JHA, *Low-Power System Scheduling, Synthesis and Displays*, IEE Proceedings on Computers and Digital Techniques, 152 (3), 2005, pp. 344–352.

[45] T. L. JOHNSON AND W. W. HWU, *Run-Time Adaptive Cache Hierarchy Management via Reference Analysis*, in Proceedings of the International Symposium on Computer Architecture, 1997, pp. 315–326.

[46] T. L. JOHNSON, M. C. MERTEN, AND W. W. HWU, *Run-Time Spatial Locality Detection and Optimization*, in Proceedings of the International Symposium on Microarchitecture, 1997, pp. 57–64.

[47] N. P. JOUPPI, *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*, in Proceedings of the International Symposium on Computer Architecture, 1990, pp. 364–373.

[48] ——, *Cache Write Policies and Performance*, in Proceedings of the International Symposium on Computer Architecture, 1993, pp. 191–201.

[49] B. JUURLINK, *Unified Dual Data Caches*, in Proceedings of the Euromicro Symposium on Digital Systems Design, 2003, pp. 33–40.

[50] G. KAHN, *The Semantics of a Simple Language for Parallel Programming*, in Information Processing, 1974, pp. 471–475.

[51] Y. KANG ET AL., *FlexRAM: Toward an Advanced Intelligent Memory System*, in Proceedings of the International Conference on Computer Design, 1999, pp. 192–201.

[52] H. KASAHARA, T. TOBITA, T. MATSUZAWA, AND S. SAKAIDA, *Standard Task Graph Set*. http://www.kasahara.elec.waseda.ac.jp/schedule/.

[53] S. KAXIRAS, Z. HU, AND M. MARTONOSI, *Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power*, in Proceedings of the International Symposium on Computer Architecture, 2001, pp. 240–251.

[54] S. KAXIRAS, Z. HU, G. J. NARLIKAR, AND R. MCLELLAN, *Cache-Line Decay: A Mechanism to Reduce Cache Leakage Power*, in Proceedings of the International Workshop on Power-Aware Computer Systems-Revised Papers, 2000.

[55] F. KHUNJUSH AND N. J. DIMOPOULOS, *Comparing Direct-to-Cache Transfer Policies to TCP/IP and M-VIA During Receive Operations in MPI Environments*, in Proceedings of the International Symposium on Parallel and Distributed Processing and Applications, 2007, pp. 208–222.

[56] V. KIANZAD, S. S. BHATTACHARYYA, AND G. QU, *CASPER: An Integrated Energy-Driven Approach for Task Graph Scheduling on Distributed Embedded Systems*, in Proceedings of the IEEE International Conference on Application-Specific Systems, Architecture Processors, 2005, pp. 191–197.

[57] P. KILLELEA, *Web Performance Tuning*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[58] E. J. KIM, K. H. YUM, G. M. LINK, N. VIJAYKRISHNAN, M. KANDEMIR, M. J. IRWIN, M. YOUSIF, AND C. R. DAS, *Energy optimization techniques in cluster interconnects*, in Proceedings of the International Symposium on Low Power Electronics and Design, 2003, pp. 459–464.

[59] N. S. KIM, T. AUSTIN, D. BLAAUW, T. MUDGE, K. FLAUTNER, J. S. HU, M. J. IRWIN, M. KANDEMIR, AND V. NARAYANAN, *Leakage Current: Moore's Law Meets Static Power*, IEEE Computer, 36 (12), 2003, pp. 68–75.

[60] J. KIN, M. GUPTA, AND W. H. MANGIONE-SMITH, *The Filter Cache: An Energy Efficient Memory Structure*, in Proceedings of the ACM/IEEE International Symposium on Microarchitecture, 1997, pp. 184–193.

[61] ——, *Filtering Memory References to Increase Energy Efficiency*, IEEE Transactions on Computers, 49 (1), 2000.

[62] A. J. KLEINOSOWSKI AND D. J. LILJA, *MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research*, IEEE Computer Architecture Letters, 1 (1), 2006, p. 7.

[63] J. G. KOOMEY, *Estimating Total Power Consumption by Servers in the U.S. and the World*, tech. rep., Lawrence Berkeley National Laboratory, 2007. http://enterprise.amd.com/Downloads/svrpwrusecompletefinal.pdf.

[64] C. LEE, M. POTKONJAK, AND W. H. MANGIONE-SMITH, *Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems*, in Proceedings of the International Symposium on Microarchitecture, 1997, pp. 330–335.

[65] H.-H. S. LEE, G. S. TYSON, AND M. K. FARRENS, *Eager Writeback - A Technique for Improving Bandwidth Utilization*, in Proceedings of the ACM/IEEE International Symposium on Microarchitecture, 2000, pp. 11–21.

[66] Y. LEE, K. P. REDDY, AND C. M. KRISHNA, *Scheduling Techniques for Reducing Leakage Power in Hard Real-Time Systems*, in Proceedings of the Euromicro Conference on Real-Time Systems, 2003, pp. 105–112.

[67] F. LIBERATO, S. LAUZAC, R. MELHEM, AND D. MOSS, *Fault Tolerant Real-Time Global Scheduling on Multiprocessors*, in Proceedings of the Euromicro Conference on Real-Time Systems, 1999, pp. 252–259.

[68] C. L. LIU AND J. W. LAYLAND, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, 20 (1), 1973, pp. 46–61.

[69] D. LIU AND C. SVENSSON, *Power Consumption Estimation in CMOS VLSI Chips*, IEEE Journal of Solid-State Circuits, 29 (6), 1994, pp. 663–670.

[70] S. MARTIN, K. FLAUTNER, T. MUDGE, AND D. BLAAUW, *Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Lower Power Microprocessors under Dynamic Workloads*, in Proceedings of the International Conference on Computer-Aided Design, 2002, pp. 721–725.

[71] S. A. MCKEE, *Reflections on the Memory Wall*, in Proceedings of the ACM International Conference on Computing Frontiers, 2004, p. 162.

[72] M. K. MCKUSICK, K. BOSTIC, M. J. KARELS, AND J. S. QUARTERMAN, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, 1996.

[73] G. MEMIK, G. REINMAN, AND W. H. MANGIONE-SMITH, *Reducing Energy and Delay Using Efficient Victim Caches*, in Proceedings of the International Symposium on Low Power Electronics and Design, 2003, pp. 262–265.

[74] M. OSKIN, F. T. CHONG, AND T. SHERWOOD, *Active pages: a computation model for intelligent memory*, in Proceedings of the International Symposium on Computer Architecture, 1998, pp. 192–203.

[75] J. K. OUSTERHOUT, *Why Aren't Operating Systems Getting Faster As Fast as Hardware?*, in Proceedings of the USENIX Summer Conference, 1990, pp. 247–256.

[76] D. PATTERSON, T. ANDERSON, N. CARDWELL, R. FROMM, K. KEETON, C. KOZYRAKIS, R. THOMAS, AND K. YELICK, *A Case for Intelligent RAM*, IEEE Micro, 17 (2), 1997, pp. 34–44.

[77] T. PERING AND R. BRODERSON, *Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor System*, in Proceedings of the Power Driven Microarchitecture Workshop, attached to ISCA98, 1998.

[78] P. PETROV AND A. ORAILOGLU, *Performance and Power Effectiveness in Embedded Processors - Customizable Partitioned Caches*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 20 (11), 2001, pp. 1309–1318.

[79] T. PIQUET, O. ROCHECOUSTE, AND A. SEZNEC, *Minimizing Single-Usage Cache Pollution for Effective Cache Hierarchy Management*, Tech. Rep. PI-1826, IRISA, 2006.

[80] F. J. POLLACK, *New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)*, in Proceedings of the ACM/IEEE International Symposium on Microarchitecture, 1999, p. 2.

[81] M. POWELL, S.-H. YANG, B. FALSAFI, K. ROY, AND T. N. VIJAYKUMAR, *Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories*, in Proceedings of the International Symposium on Low Power Electronics and Design, 2000, pp. 90–95.

[82] M. PRVULOVIĆ, D. MARINOV, Z. DIMITRIJEVIĆ, AND V. MILUTINOVIĆ, *Split Temporal/Spatial Cache: A Survey and Reevaluation of Performance*. IEEE TCCA Newsletter, 1999.

[83] G. QUAN, L. NIU, X. S. HU, AND B. MOCHOCKI, *Fixed Priority Scheduling for Reducing Overall Energy on Variable Voltage Processors*, in Proceedings of the International Real-Time System Symposium, 2004, pp. 309–318.

[84] G. REINMAN AND N. P. JOUPPI, *An Integrated Cache Timing and Power Model*, Tech. Rep. CACTI 2.0, COMPAQ Western Research Lab, Palo Alto, California, 1999.

[85] M. ROSENBLUM, E. BUGNION, S. A. HERROD, E. WITCHEL, AND A. GUPTA, *The impact of architectural trends on operating system performance*, in Proceedings of the ACM Symposium on Operating Systems Principles, 1995, pp. 285–298.

[86] T. SIMUNIC, L. BENINI, AND G. D. MICHELI, *Energy-Efficient Design of Battery-Powered Embedded Systems*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 9 (1), 2001.

[87] STANDARD PERFORMANCE EVALUATION CORPORATION. http://www.spec.org/.

[88] V. SWAMINATHAN AND K. CHAKRABARTY, *Pruning-based, energy-optimal, deterministic I/O device scheduling for hard real-time systems*, Trans. on Embedded Computing Sys., 4 (1), 2005, pp. 141–167.

[89] E. TAM, *Improving Cache Performance Via Active Management*, PhD thesis, University of Michigan, Ann Arbor, 1999.

[90] D. TARJAN, S. THOZIYOOR, AND N. P. JOUPPI, *CACTI 4.0*, Tech. Rep. HPL-2006-86, HP Labs, 2006.

[91] S. THOZIYOOR, N. MURALIMANOHAR, J. H. AHN, AND N. P. JOUPPI, *CACTI 5.3*. http://www.hpl.hp.com/research/cacti/.

[92] G. VARATKAR AND R. MARCULESCU, *Communication-Aware Task Scheduling and Voltage Selection for Total Systems Energy Minimization*, in Proceedings of the International Conference on Computer-Aided Design, 2003, pp. 510–517.

[93] N. VIJAYKRISHNAN, M. J. IRWIN, H. S. KIM, AND W. YE, *Energy-driven integrated hardware-software optimizations using SimplePower*, in Proceedings of the International Symposium on Computer Architecture, 2000, pp. 95–106.

[94] N. VIJAYKRISHNAN, M. KANDEMIR, M. J. IRWIN, H. S. KIM, W. YE, AND D. DUARTE, *Evaluating Integrated Hardware-Software Optimizations Using a Unified Energy Estimation Framework*, IEEE Transactions on Computers, 52 (1), 2003, pp. 59–75.

[95] W. A. WULF AND S. A. MCKEE, *Hitting the memory wall: implications of the obvious*, SIGARCH Computer Architecture News, 23 (1), 1995, pp. 20–24.

[96] R. XU, D. ZHU, C. RUSU, R. MELHEM, AND D. MOSS, *Energy-Efficient Policies for Embedded Clusters*, in Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2005, pp. 1–10.

[97] L. YAN, J. LUO, AND N. K. JHA, *Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Heterogeneous Distributed Real-time Embedded Systems*, in Proceedings of the International Conference on Computer-Aided Design, 2003, pp. 30–37.

[98] C.-Y. YANG, J.-J. CHEN, AND T.-W. KUO, *An Approximation Algorithm for Energy-Efficient Scheduling on A Chip Multiprocessor*, in Proceedings of the Conference on Design, Automation, and Test in Europe, 2005, pp. 468–473.

[99] S. ZHANG, K. S. CHATHA, AND K. S. CHATHA, *Automated techniques for energy efficient scheduling on homogeneous and heterogeneous chip multi-processor architectures*, in Proceedings of the Conference on Asia and South Pacific Design Automation, 2008, pp. 61–66.

[100] W. ZHANG, *Replication Cache: A Small Fully Associative Cache to Improve Data Cache Reliability*, IEEE Transactions on Computers, 54 (12), 2005, pp. 1547–1555.

[101] Y. ZHANG, X. S. HU, AND D. Z. CHEN, *Task Scheduling and Voltage Selection for Energy Minimization*, in Proceedings of the Conference on Design Automation, 2002, pp. 183–188.

[102] L. ZHAO, L. N. BHUYAN, R. R. IYER, S. MAKINENI, AND D. NEWELL, *Hardware Support for Accelerating Data Movement in Server Platform*, IEEE Transactions on Computers, 56 (6), 2007, pp. 740–753.

[103] D. ZHU, R. MELHEM, AND B. R. CHILDERS, *Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multiprocessor Real-Time Systems*, IEEE Transactions on Parallel and Distributed Systems, 14 (7), 2003, pp. 686–700.

List of Publications

Journals

1. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Leakage Aware Multiprocessor Scheduling*, Journal of VLSI Signal Processing, 57 (1) (2009).

International Conference Proceedings

2. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Reducing Traffic Generated by Conflict Misses in Caches*, in Proceedings of the ACM International Conference on Computing Frontiers (2004), pp. 235-239.

3. B. H. H. JUURLINK AND **P. J. DE LANGEN**, *Dynamic Techniques to Reduce Memory Traffic in Embedded Systems*, in Proceedings of the ACM International Conference on Computing Frontiers (2004), pp. 192-201.

4. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Leakage-Aware Multiprocessor Scheduling for Low Power*, in Proceedings of the International Parallel and Distributed Processing Symposium (2006).

5. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Trade-offs Between Voltage Scaling and Processor Shutdown for Low-Energy Embedded Multiprocessors*, in Proceedings of the International Workshop on Computer Systems: Architectures, Modeling, and Simulation (2007), pp. 75–85.

6. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Memory Copies in Multi-Level Memory Systems*, in Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (2008).

7. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Limiting the Number of Dirty Cache Lines*, in Proceedings of the Conference on Design, Automation and Test in Europe (2009).

National/Local Conferences Proceedings

8. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Reducing Conflict Misses in Caches by Using Application Specific Placement Functions*, in Proceedings of the Workshop on Circuits, Systems and Signal Processing (ProRISC) (2006), pp. 300–305.

9. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Combining Voltage Scaling and Processor Shutdown to Reduce Energy Consumption in Embedded Multiprocessors*, in Proceedings of the Conference of the Advanced School for Computing and Imaging (2007), pp. 195–202.

10. **P. J. DE LANGEN**, B. H. H. JUURLINK, AND S. VASSILIADIS, *HandBench: A Benchmarking Suite for Processors Embedded in Handheld Devices*, in Proceedings of the Workshop on Circuits, Systems and Signal Processing (ProRISC) (2004).

11. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Off-Chip Memory Traffic Measurements of Low-Power Embedded Systems*, in Proceedings of the Workshop on Circuits, Systems and Signal Processing (ProRISC) (2002), pp. 351–358.

12. **P. J. DE LANGEN** AND B. H. H. JUURLINK, *Reducing Conflict Misses in Caches*, in Proceedings of the Workshop on Circuits, Systems and Signal Processing (ProRISC) (2003), pp. 505–510.

13. **P. J. DE LANGEN**, B. H. H. JUURLINK, AND S. VASSILIADIS, *Microprocessor Scheduling to Reduce Leakage Power*, in Proceedings of the Workshop on Circuits, Systems and Signal Processing (ProRISC) (2005), pp. 383–389.

Samenvatting

Energie verbruik is in toenemende mate belangrijk in verscheidene gebieden van de computerarchitectuur. Niet alleen voor de markt van draagbare 'embedded' computers, maar ook voor desktop machines en high-end server-faciliteiten is er een alsmaar toenemende vraag naar krachtigere computers met een gelijkblijvend of zelfs verminderd energieverbruik. Voor processoren in draagbare, batterijgevoede apparaten willen gebruikers meer en meer mogelijkheden en een alsmaar langere levensduur van de batterij. Voor gewone desktop machines en toegewijde server systemen komt de vraag naar lager stroomverbruik voornamelijk voort uit economische motieven, milieu overwegingen en de vraag naar systemen zonder luidruchtige koeling. Dit proefschrift onderzoekt verscheidene technieken om energieverbruik van processoren te verminderen.

Een deel van de technieken die in dit proefschrift behandeld worden richten zich om het verminderen van energieverbruik door het beperken van de hoeveelheid data verkeer tussen een processor en extern geheugen. Omdat geheugen een bekende beperkende factor is in computer systemen hebben fabrikanten in toenemende mate agressieve technieken moeten toepassen om vooruitgang te kunnen boeken. De technieken die in dit proefschrift behandeld worden zijn gericht op het verminderen van data verkeer met verbeterde of in ieder geval gelijkblijvende snelheid.

Een ander deel van dit proefschrift is gericht op het verminderen van energie verbruik door de kloksnelheid van onderdelen van een multiprocessor systeem te verlagen, in combinatie met het uitschakelen van onderdelen. Multiprocessor systemen hebben in de afgelopen jaren in toenemende mate aandacht gekregen, voornamelijk omdat beperkingen ten aanzien van het te gebruiken vermogen verhinderd hebben dat de kloksnelheid verder opgevoerd kon worden, en omdat er met parallellisme op instructie niveau steeds minder winst gehaald kon worden. Vanwege de manier hoe energie gedissipeerd word in halfgeleider componenten, is het gebruik van meerdere processoren op een lagere kloksnelheid een effectieve manier gebleken om de hoeveelheid gebruikte energie te verminderen. Echter, door het steeds kleiner worden van de componenten waar processoren van gemaakt worden, is de verwachting dat dit energie model drastisch zal veranderen in de komende jaren. Sommige technieken die in dit proefschrift gepresenteerd worden doelen op het verminderen van energieverbruik van zulke van hedendaagse en toekomstige multiprocessor systemen.

Curriculum Vitae

Pepijn de Langen was born on February 11th 1976 in Groningen, the Netherlands. He undertook secondary education at the Praedinius Gymnasium in Groningen. Thereafter, he studied at the faculty of Electrical Engineering, Delft University of Technology, where he received received a Bachelor of Science and a Master of Science degree.

In 2004, Pepijn started working as a researcher in the Computer Engineering laboratory of Delft University of Technology, where he carried out his Ph.D. research under supervision of Dr. Ben Juurlink. This research was carried out in a project funded by the Netherlands Organization for Scientific Research (NWO). This dissertation presents the results of this research.

Pepijn's research interests include advanced computer architectures, memory systems, hardware/software co-design, and techniques for reducing power consumption.