ENERGY-AWARE SCHEDULING AND DYNAMIC RECONFIGURATION IN REAL-TIME EMBEDDED SYSTEMS

By

WEIXUN WANG

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2011

© 2011 Weixun Wang

To my wife and my parents

ACKNOWLEDGMENTS

Time flies by, four years of graduate school life here at UF elapsed like a shuttle. Looking back after going through this challenging journey, all I can see are maturations, achievements and triumphs. However, I'm not alone on my way to Ph.D. but surrounded by many other people to whom I am sincerely grateful.

First of all, I really appreciate my adviser Prof. Prabhat Mishra for what he has done for me. It was him who led me to open the door of computer science. I will never forget all his kind instructions and enduring support. He not only guided me to overcome challenging problems, not also taught me how to explore new directions. More importantly, he is always considerate for me and helps me building my own career. He is the person who made this dissertation come true.

I would also like to thank my other Ph.D. committee members: Prof. Sartaj Sahni, Prof. Tao Li, Prof. Greg Stitt and Prof. Ann Gordon-Ross for their precious advises and criticisms. I appreciate Prof. Sanjay Ranka with whom I collaborated for two years. His profound knowledge in algorithms helped me a lot in my research. I also thank my lab-mates, Mingsong Chen, Kanad Basu, Xiaoke Qin, Chetan Murthy, Kartik Shrivastava, Hadi Hajimiri and Kamran Rahmani. It was my great pleasure to work with them. I really enjoyed our friendship and I hope it will last forever.

Last but not least, I sincerely thank my family for their love, encouragement and support. I wouldn't be able to achieve anything without my parents' raising since my childhood. Their all-embracing love and care – guiding me patiently, supporting my decisions, forgiving my faults – makes me grow up freely. My most special appreciation is dedicated to my wife Yan. Her love and devotion paved the road to my doctoral degree. She always held my hands and gave me courage whenever I need. I have realized how lucky I am to marry her.

This work was partially supported by grants from National Science Foundation (NSF) grant CCF-0903430 and Semiconductor Research Corporation (SRC) grant 2009-HJ-1979.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS 4

LIST OF TABLES 9

LIST OF FIGURES 10

ABSTRACT 14

CHAPTER

1 INTRODUCTION 15

1.1 Optimizations in Real-Time Embedded Systems 16
1.2 Opportunities and Challenges 18
1.2.1 Dynamic Reconfiguration Techniques 18
1.2.2 Potential Optimization Opportunities 19
1.2.3 Challenges 20
1.3 Research Contributions 21

2 MODELING OF REAL-TIME AND RECONFIGURABLE SYSTEMS 25

2.1 System Model 25
2.2 Energy Models 25
2.2.1 Cache Energy Model 25
2.2.2 Processor Energy Model 26
2.2.3 Bus Energy Model 28
2.2.4 Main Memory Energy Model 28
2.3 Thermal Model 29
2.4 Summary 29

3 DYNAMIC CACHE RECONFIGURATION FOR SOFT REAL-TIME SYSTEMS 30

3.1 Related Work 31
3.1.1 Caches in Real-Time Systems 31
3.1.2 Reconfigurable Cache Architectures 32
3.1.3 Caches Tuning Techniques 33
3.2 SACR: Scheduling-Aware Cache Reconfiguration 35
3.2.1 Overview 35
3.2.2 Phase-based Optimal Cache Selection 36
3.2.3 Statically Scheduled Systems 43
3.2.4 Dynamically Scheduled Systems 43
3.2.4.1 Conservative Approach 43
3.2.4.2 Aggressive Approach 46
3.2.5 Impact of Storing Multiple Cache Configurations 50

3.3 Design Space Exploration for Two-Level Cache Reconfiguration 52
3.3.1 Exhaustive Exploration 53
3.3.2 Same Level One Cache Tuning – SLOT 54
3.3.3 Two-Step Tuning – TST 54
3.3.4 Independent Level One Cache Tuning – ILOT 55
3.3.5 Interlaced Tuning – ILT 57
3.4 Experiments 59
3.4.1 Experiments Setup 59
3.4.2 Results: Single-level SACR 60
3.4.2.1 Energy Saving 60
3.4.2.2 Suitability of Statically Determined Configurations 65
3.4.2.3 Impact of Storing Multiple Cache Configurations 67
3.4.2.4 Analysis of Input Variations 69
3.4.2.5 Hardware Overhead 70
3.4.3 Results: Multi-level SACR 73
3.4.3.1 Optimal Cache Configuration Selection 74
3.4.3.2 Energy Saving 75
3.4.3.3 Insights behind Results 76
3.4.3.4 Exploration Efficiency 77
3.5 Summary 78

4 ENERGY-AWARE SCHEDULING WITH DYNAMIC VOLTAGE SCALING . 80

4.1 Related Work 81
4.2 PreDVS: Preemptive Dynamic Voltage Scaling 83
4.2.1 Overview 83
4.2.2 Problem Formulation 85
4.2.3 Approximation Scheme 88
4.2.3.1 Problem Transformation 88
4.2.3.2 Approximation Algorithm 95
4.2.4 Efficient PreDVS Heuristics 101
4.2.4.1 Heuristic Without Problem Transformation 101
4.2.4.2 Heuristic With Problem Transformation 104
4.3 DSR: Dynamic Slack Reclamation 105
4.3.1 Overview 105
4.3.2 Dynamic Slack Reclamation Algorithm 107
4.3.2.1 Tasks without Arrival Time Constraints 109
4.3.2.2 Tasks with Arrival Time Constraints 110
4.3.3 Algorithm 113
4.4 Experiments 116
4.4.1 PreDVS 116
4.4.1.1 Experimental Setup 116
4.4.1.2 Results 117
4.4.2 DSR 123
4.4.2.1 Experimental Setup 123

4.4.2.2 Results . . . 124
4.5 Summary . . . 130

5 SYSTEM-WIDE ENERGY OPTIMIZATION WITH DVS AND DCR . . . 132

5.1 Related Work . . . 133
5.2 System-wide Leakage-aware DVS and DCR . . . 135
5.2.1 Power Estimation Framework . . . 136
5.2.2 Two-Level Cache Tuning Heuristic . . . 137
5.2.3 Critical Speed . . . 138
5.2.3.1 Processor + L1 Cache . . . 139
5.2.3.2 Processor + L1/L2 Cache . . . 140
5.2.3.3 Processor + L1/L2 Cache + Memory . . . 141
5.2.3.4 Processor + L1/L2 Cache + Memory + Bus . . . 142
5.2.4 Real-Time Voltage Scaling and Cache Reconfiguration . . . 144
5.2.4.1 Profile Table . . . 144
5.2.4.2 Reconfiguration Selection Heuristics . . . 145
5.2.5 Procrastination . . . 146
5.3 A General Dynamic Reconfiguration Algorithm . . . 148
5.3.1 Overview . . . 148
5.3.2 Algorithm . . . 149
5.3.2.1 Extended Complete Bipartite Graph . . . 151
5.3.2.2 Minimum-Cost Path Algorithm . . . 153
5.4 Experiments . . . 157
5.4.1 System-wide Energy Optimization . . . 157
5.4.1.1 Experiments Setup . . . 157
5.4.1.2 Results . . . 159
5.4.2 General Algorithm for Dynamic Reconfiguration . . . 163
5.4.2.1 Experiments Setup . . . 163
5.4.2.2 Results . . . 164
5.5 Summary . . . 172

6 TEMPERATURE- AND ENERGY-CONSTRAINED SCHEDULING . . . 174

6.1 Related Work . . . 176
6.2 Background . . . 177
6.3 TCEC Scheduling Approach . . . 177
6.3.1 Overview . . . 177
6.3.2 Modeling with Extended Timed Automata . . . 179
6.3.3 Problem Variants . . . 184
6.4 Experiments . . . 185
6.4.1 Experiments Setup . . . 185
6.4.2 Results . . . 185
6.4.2.1 Solving TCEC Problems . . . 185
6.4.2.2 Running Time Variations . . . 185
6.5 Summary . . . 187

7 ENERGY OPTIMIZATION OF CACHE HIERARCHY IN MULTICORE SYSTEMS 188

7.1 Related Work . . . 189
7.2 Background and Motivation . . . 190
7.2.1 Architecture Model . . . 190
7.2.2 Motivation . . . 192
7.3 Dynamic Cache Reconfiguration and Partitioning . . . 193
7.3.1 Problem Formulation . . . 193
7.3.2 Static Profiling . . . 195
7.3.3 DCR + CP Algorithm . . . 196
7.3.4 Task Mapping . . . 199
7.3.5 Varying Cache Partitioning Scheme . . . 203
7.3.6 Gated-V_{dd} Shared Cache Lines . . . 206
7.4 Experiments . . . 206
7.4.1 Experimental Setup . . . 206
7.4.2 Results . . . 208
7.4.2.1 Energy Savings . . . 208
7.4.2.2 Deadline Effect . . . 209
7.4.2.3 Task Mapping Effect . . . 210
7.4.2.4 Effect of Varying Cache Partitioning . . . 211
7.4.2.5 Gated-V_{dd} Cache Lines Effect . . . 212
7.5 Summary . . . 213

8 CONCLUSIONS AND FUTURE WORK . . . 214

8.1 Conclusions . . . 214
8.2 Future Research Directions . . . 216

A LIST OF PUBLICATIONS . . . 218

REFERENCES . . . 221

BIOGRAPHICAL SKETCH . . . 233

LIST OF TABLES

Table page

2-1 Constants for 70nm technology . . . 28

3-1 Optimal cache configurations for task phases. Each configuration is denoted by the total cache size in kilobytes (kb), followed by the associativity in number of ways (w), followed by the line size in bytes (b). . . . 38

3-2 (a) Static profile table and (b) Task list entry for task i for the conservative approach 45

3-3 (a) Static profile table and (b) Task list entry for task i for the aggressive approach 47

3-4 Benchmark task sets . . . 61

3-5 Task performance variations for conservative approach . . . 66

3-6 Task performance variations for aggressive approach . . . 66

3-7 Current phases of deadline violated tasks when they are discarded. . . . 67

3-8 Input variation exploration. . . . 71

3-9 Input pattern changes. . . . 72

3-10 Overhead of different lookup tables (180nm technology) . . . 73

3-11 Overhead of different lookup tables (65nm technology) . . . 73

3-12 Task sets consisting of real benchmarks. . . . 74

3-13 Cache hierarchy configuration explored using different exploration methods. . . 78

4-1 Task sets consisting of real benchmarks. . . . 117

4-2 Algorithm running time comparisons (in seconds). . . . 122

4-3 Task sets consisting of real benchmarks. . . . 124

5-1 Task sets consisting of real benchmarks. . . . 158

5-2 Task sets consisting of real benchmarks. . . . 163

6-1 TCEC results on different task sets . . . 186

7-1 Optimal partition factors for selected benchmarks . . . 202

7-2 Multi-task benchmark sets. . . . 207

LIST OF FIGURES

Figure page

1-1 Real-Time Systems. 16

1-2 System-wide power consumption breakdown of a typical SoC. 17

1-3 Optimization targets, objectives and techniques. 22

1-4 Dissertation outline. 23

3-1 Reconfigurable cache architecture: (a) base cache bank layout, (b) way concatenation, (c) way shutdown, and (d) configurable line size. 34

3-2 Cache configurations selected based on task phases 35

3-3 Dynamic cache reconfigurations for tasks T1 and T2 36

3-4 Task partitioning at n potential preemption points (P_i) resulting in n phases. Each phase comprises execution from the invocation/resumption point to task completion. C_i denotes the cache configuration used in each phase. 37

3-6 Effective range where a higher partition factor makes a difference 42

3-7 Task set and sample scheduling 49

3-12 Miss rate for epic under different cache configurations. 65

3-14 Normalized energy consumption of the searched energy-optimal cache configuration using heuristics. 75

3-15 Normalized execution time of the searched performance-optimal cache configuration using heuristics. 76

3-16 Cache hierarchy energy consumption using four heuristics. 77

4-1 Power consumption and clock cycle length of Crusoe processor (P_{dyn}, P_{sta} and P_{on} denote dynamic power, leakage power and the intrinsic power required to keep the processor on, respectively). 80

4-2 Inter-task DVS, PreDVS and Intra-task DVS. 85

4-3 Distinct block and distinct block set. 89

4-4 Profile table generation for distinct block set. 92

4-5 Aggregated profile table generation for each task. 94

4-6 Illustration of PreDVS heuristic without problem transformation. 103

4-7 Execution blocks after static slack allocation. 108

4-8 Dynamic slack generated by early finished task. 108

4-9 Dynamic slack allocation example. 110

4-10 Dynamic slack allocation with arrival time constraints. 111

4-11 Slack reclamation with task rescheduling. 112

4-12 Task rescheduling example. 113

4-13 Exploration window partitions into groups according to $MaxRS$. 114

4-16 PreDVS Approximation Algorithm Running Time Comparison. 122

4-17 Effect of Window size on the total energy savings. 125

4-18 Results for StrongARM processor (synthetic task sets). 126

4-19 Results for Transmeta Crusoe processor with constant effective capacitance values (synthetic task sets). 127

4-20 Results for Transmeta Crusoe processor with application-specific effective capacitance values (synthetic task sets). 128

4-21 Results for Transmeta Crusoe processor with application-specific effective capacitance values (real benchmark task sets). 129

4-22 Problem variations comparison. 130

4-23 Running time overhead. 130

5-1 Workflow of our approach. 135

5-2 Overview of our power estimation framework. 137

5-3 Conceptual system bus architecture. 138

5-4 Processor energy consumption E_{proc} for executing *cjpeg*: $E_{procDyn}$ is the dynamic energy, $E_{procSta}$ is the static energy and E_{procOn} is the intrinsic energy needed to keep processor on. 140

5-5 Overall system energy consumption E_{total} of the processor and L1 caches for executing *cjpeg*: E_{L1Dyn} and E_{L1Sta} are the dynamic and static L1 cache energy consumption, respectively. 140

5-6 Overall system energy consumption E_{total} of the processor, L1 caches and L2 cache (configured to 64KB,128B,8-way) for executing *cjpeg*: E_{L2Dyn} and E_{L2Sta} are the dynamic and static L2 cache energy consumption, respectively. 141

5-7 Overall system energy consumption E_{total} of the processor, L1/L2 caches and memory for executing *cjpeg*: E_{memDyn} and E_{memSta} are the dynamic and static memory energy consumption, respectively; E_{cache} represents the total energy consumption of both L1 and L2 caches. 142

5-8 Overall system energy consumption E_{total} of the processor, L1/L2 caches, memory and system buses for executing *cjpeg*: E_{busDyn} and E_{busSta} are the dynamic and static bus energy consumption, respectively. 143

5-9 Processor voltage scaling impact on various system components. 143

5-10 Total energy consumption across all L1 cache configurations (with L2 cache configured to 64KB,128B,8-way) for executing *cjpeg*. 145

5-11 Tasks and execution blocks. 150

5-12 ECBG model of $\wp$. 152

5-13 Illustration of our algorithm. 154

5-14 Ensuring the time constraints. 156

5-15 Illustration of the approximate version of our algorithm. 157

5-16 Total energy consumption of single-benchmark task sets. 160

5-17 System-wide overall energy consumption using different approaches. 161

5-19 Energy consumption compared with two heuristics: DVS+DCR. 165

5-20 Energy consumption compared with two heuristics: (a) DCR; (b) DVS. 166

5-21 Time discretization effect for DCR. 167

5-22 Time discretization effect for DVS. 168

5-23 Variable overhead aware effect in DVS. 168

5-24 Variable overhead aware effect in DCR. 169

5-25 Comparison of our exact approach and approximate approach: (a) energy consumption normalized to uniform slowdown heuristic; (b) running time. 170

5-26 Comparison of our exact approach and approximate approach under various reconfiguration overhead. 171

6-1 Workflow of our model checking approach. 178

6-2 TCEC problem modeled in extended timed automata. 182

6-3 Problem modeling when every task has own deadline (partial graph). 183

6-4 Running time with different constraints. 186

7-1 Typical multicore architecture with shared L2 cache. 190

7-2 Way-based cache partitioning example (four cores with a 8-way associative shared cache). 191

7-3 L1 DCR impact on L2 CP in performance. 193

7-4 Illustration of our algorithm. 197

7-5 Optimal partition factor variation with L1 caches of 4KB with 2-way associativity and 32-byte line size. 201

7-6 Task mapping effect. 202

7-7 Varying partitioning scheme. 203

7-9 Deadline effect on total energy consumption. 210

7-10 Task mapping heuristic effect on total energy consumption. 211

7-11 Varying partitioning scheme effect on total energy consumption. 211

7-12 Gated-V_{dd} cache lines effect on total energy consumption. 212

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

ENERGY-AWARE SCHEDULING AND DYNAMIC RECONFIGURATION IN
REAL-TIME EMBEDDED SYSTEMS

By

Weixun Wang

August 2011

Chair: Prabhat Mishra
Major: Computer Engineering

Energy is one of the key design considerations in embedded systems. Optimization techniques based on dynamic reconfiguration are widely employed for achieving various design objectives. Dynamic cache reconfiguration (DCR) is promising for improving both energy efficiency and performance of the memory hierarchy. Dynamic voltage scaling (DVS) is capable of reducing processor energy dissipation. While these techniques have been studied for general-purpose systems, it is a major challenge to apply them to real-time embedded systems. Applications in such systems have timing constraints that need to be satisfied during execution otherwise it can lead to performance degradation or even catastrophic consequences. This dissertation presents novel reconfiguration techniques and scheduling algorithms for energy optimization in real-time embedded systems. My research has made five major contributions: i) it proposes scheduling-aware cache reconfiguration algorithms and design space exploration techniques for soft real-time systems; ii) it proposes energy-aware scheduling algorithms based on DVS; iii) it effectively integrates DVS and DCR together for system-wide energy minimization; iv) it verifies task schedulability in temperature- and energy-constrained real-time systems; and v) it presents cache hierarchy energy optimization based on dynamic reconfiguration in multicore systems. Extensive experimental results demonstrate significant improvement in overall energy efficiency, performance and thermal control without affecting timing constraints.

CHAPTER 1
INTRODUCTION

Design and optimization of real-time multitasking systems has received significant attention from both academia and industry in recent years. Figure 1-1 illustrates the typical structure and application domains of real-time systems. These systems require unique design considerations since timing constraints are imposed on the workloads (i.e., tasks). In general, tasks could be heterogeneous in terms of timing constraints (e.g., deadlines, arrival times) and characteristics (e.g., periodic/sporadic, preemptive/non-preemptive etc.). Tasks have to complete execution by their deadlines in order to ensure correct system behavior. In hard real-time systems, such as safety-critical applications like medical devices and aircrafts, violating task deadlines could lead to catastrophic consequences. Due to these stringent constraints, the real-time scheduler must perform task *schedulability analysis* based on task characteristics such as priorities, periods, and deadlines [69]. A task set is considered to be *schedulable* only if there exists a valid schedule that satisfies all the deadlines. As embedded systems become ubiquitous, real-time systems with soft timing constraints also become widespread in applications such as gaming, housekeeping and multimedia equipments. Minor deadline violations may only result in temporary service quality degradation, but will not lead to incorrect system behavior. For example, users of video-streaming on mobile devices can tolerate occasional jitters caused by frame droppings.

Existing low-power computing techniques tune the system at runtime (*dynamically reconfigure*) to meet optimization goals by changing *tunable* system parameters. It is a major challenge to determine *when* and *how* to dynamically reconfigure the system in order to achieve lower power consumption, higher performance, lower peak temperature and balance system behavior. My research mainly focuses on energy optimization in real-time embedded systems based on dynamic scheduling and reconfiguration techniques. The rest of the chapter is organized as follows. Section 1.1 discusses various optimization

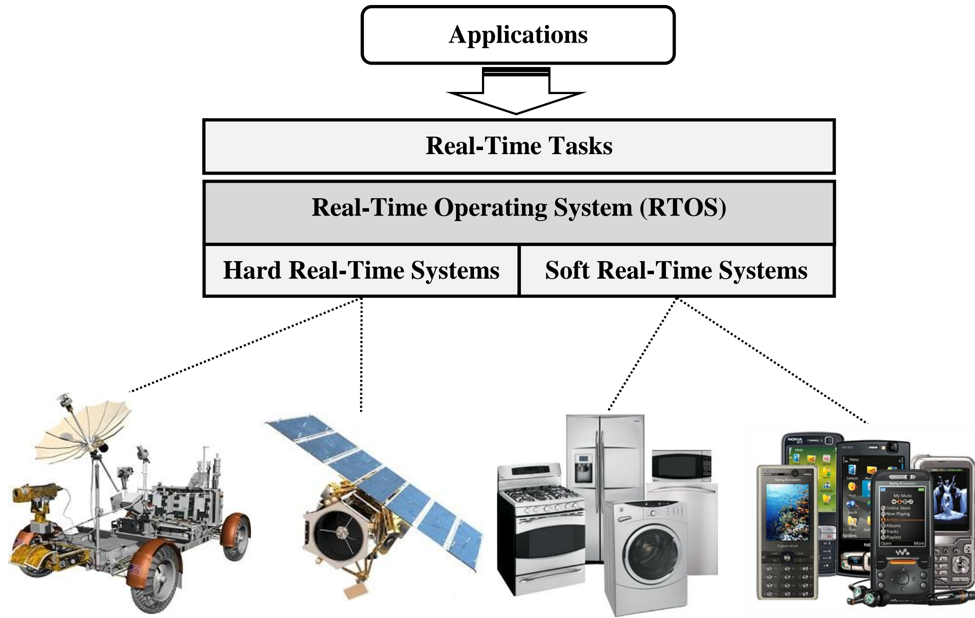


Figure 1-1. Real-Time Systems.

objectives in real-time embedded systems. Section 1.2 presents promising reconfiguration techniques, and studies potential improvement opportunities as well as major challenges in implementing them. Finally, Section 1.3 summarizes the contributions of this dissertation.

1.1 Optimizations in Real-Time Embedded Systems

Energy conservation is the primary optimization objective in almost every system design. It is important for desktop-based conventional computing since a significant amount of electricity is consumed by computers nowadays [1]. For embedded systems, which are generally driven by batteries with a limited energy budget, reduction in power and energy dissipation is even more critical. The benefits of such optimizations include battery life improvement, cost and area reduction due to less energy and cooling requirements as well as improved design of power supply, voltage regulators and interconnect dimensions.

Figure 1-2 shows system-wide energy consumption distribution for a typical System-on-Chip (SoC) [65]. It can be seen that processor, cache hierarchy, main memory and bus are the four main comparable contributors to the overall power consumption. The

processor is the primary contributor due to the most intensive switching activities in the circuit. Recent studies have shown that memory hierarchy, especially the cache subsystem, has become comparable to the processor with respect to energy consumptions [72] due to its increasing access frequency and on-chip areas. Therefore, they are the main targets for optimization in both uniprocessor systems and multicore architectures.

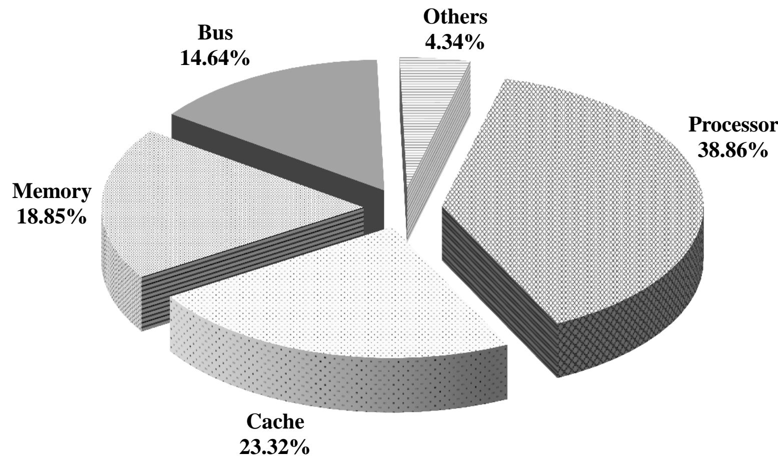


Figure 1-2. System-wide power consumption breakdown of a typical SoC.

In the past, leakage energy was negligible compared to its dynamic counterpart. However, in the last decade, we have observed a continuous CMOS device scaling in which higher transistor density and smaller device dimension have led to increasing leakage (static) power consumption. Therefore, energy optimization techniques should take both dynamic and static consumption into consideration, from various system components, to achieve overall energy reduction. This is also the primary focus of this dissertation.

Along with the performance improvement in state-of-art microprocessors, power densities are rising more rapidly due to the fact that feature size scales faster than voltages [105]. Since energy consumption is converted into heat dissipation, high heat flux increases the on-chip temperature. This trend is observed in both desktop and embedded processors [117] [138]. Thermal increase will lead to a series of adverse effects including reliability and performance degradation [131], frequent transient errors or even permanent damage, and higher leakage power dissipation [124]. Due to the severe detrimental impact,

we have to control the instantaneous temperature so that its peak value is minimized or does not go beyond a certain level. Thermal management schemes are widely studied for general-purpose systems. However, in the context of embedded systems, traditional packaging and cooling solutions are not applicable due to the limits on device size and cost. This dissertation also examines temperature management schemes.

1.2 Opportunities and Challenges

Tremendous optimization opportunities exist based on the design objectives. In this section, we briefly describe various reconfiguration techniques that we explore in this dissertation. Next, we discuss opportunities and challenges to employ them in real-time embedded system optimizations.

1.2.1 Dynamic Reconfiguration Techniques

Dynamic cache reconfiguration (DCR) offers the ability to tune the cache configuration parameters at runtime to reflect each application's memory access behavior and meet its unique requirement. Different applications may have distinct preferences for cache configurations with respect to both performance and energy efficiency. Specifically, the working set of the application decides the favored cache capacity, while the spatial and temporal locality reflect the cache line size and associativity requirements, respectively. Research shows that specializing the cache configuration for the application can lead to significant reduction of memory subsystem energy consumption [31] [33] [122]. In multicore architectures, cache partitioning (CP) is another form of reconfiguration which helps to eliminate interferences and improve performance.

Many general as well as specific-purpose processors support dynamic voltage/frequency scaling (DVFS, or simply DVS) nowadays [74] [54] supporting multiple operating voltage levels. DVS takes advantage of the fact that linear reduction in the supply voltage can quadratically reduce the power consumption while the operating frequency is approximately lowered linearly [38]. Therefore, it will be beneficial to reduce the supply voltage whenever possible to achieve energy savings. Using dynamic power management

(DPM) [8], the overall energy consumption can be reduced by putting the system into a low-power sleep mode when there is no valid activity. Research has shown that it is always advantageous to exploit DVS prior to DPM in the processor [52] while DPM could be beneficial after DVS is applied or when DVS is not available.

Task scheduling, along with DCR/DVS, also plays an important role in optimizations of real-time multitasking systems. For uniprocessor systems, DCR/DVS has to be considered systematically together with static scheduling, dynamic rescheduling as well as runtime task management. For multicore systems, task mapping and sequencing schemes should be exploited together with dynamic reconfigurations.

1.2.2 Potential Optimization Opportunities

Although real-time systems are constrained by task deadlines, idle time still exists when there is no task executing in the system. We denote system idle time as *time slack* or, simply, *slack*. There are two categories of time slack. Static slack is intrinsic for a given set of tasks assuming every task takes its worst-case execution time (WCET) to complete. Dynamic slack is generated at runtime due to early-finished tasks. DVS and DPM can take advantage of time slacks to either slow down or switch off the processor to save energy without violating any timing constraint. DCR, however, is even more promising since energy-efficient cache configurations do not necessarily have inferior performance. In other words, for a specific application, one cache configuration could possibly be superior in both energy efficiency and performance. Therefore, both DVS and DCR would be employed for energy optimization by wisely utilizing time slacks.

Real-time systems, especially those with hard timing constraints, normally have highly deterministic characteristics [89], e.g., release time, deadline, input and execution behavior. This fact provides great opportunities for energy optimization, especially for cache reconfiguration. Off-line analysis is most suitable for time consuming works (e.g., determining appropriate schedulings and/or cache configurations) which may not be feasible to compute at runtime for real-time systems. We believe that utilizing static

analysis information during runtime with minimum amount of overhead is the most appropriate and beneficial strategy for real-time system optimization.

1.2.3 Challenges

There are major challenges in achieving design objectives mentioned in Section 1.1. The key issue is ***when*** and ***how*** to reconfigure the system. Different strategies should be adapted for different techniques, systems and task characteristics. The problem difficulty varies depending on optimization scenarios. In certain cases, tradeoff has to be made between design quality (e.g., energy savings) and runtime complexity as well as reconfiguration overhead.

Cache behavior is very difficult to predict in terms of performance and energy efficiency. Since precision (i.e., timing constraints) is crucial for real-time systems, estimation based on program trace or dynamic evaluation is not acceptable. Aperiodic and sporadic tasks may arrive at any time and potentially preempt the currently running task. In this case, there is no way to know the exact preemption positions during design time. Since application's behavior and cache preference vary during execution, it is challenging to profile each task. It is also hard to utilize static profiling information at runtime along with task scheduling. Moreover, for multi-level cache hierarchy, the design space to be explored may become prohibitively large since the number of possible combinations of each individual cache configuration is huge.

Although DVS has been widely studied recently, we believe that there is enough potential for further energy conservation especially in preemptive real-time systems. By assigning multiple voltage levels to each task instance, we can no longer simply rely on the EDF schedulability condition to guide the DVS algorithm. Moreover, it is also important to minimize additional overhead. Since the original inter-task DVS problem is NP-hard, this aggressive strategy is even more difficult to solve, especially if close-to-optimal solutions are desired. It is also challenging to design algorithms to efficiently exploit

dynamic time slack during runtime given the fact that tasks may have distinct energy profile and thus shows different energy saving abilities.

While DCR and DVS can be successfully employed independently, effectively combining them simultaneously to achieve system-wide energy optimization remains an open question. It is even more challenging when leakage power become significant and all major system components are taken into account. In that case, the processor voltage level cannot be scaled down indefinitely to reduce dynamic energy even if the time slack is adequate since leakage power can dominate. As a result, DPM has to play a more important role in energy optimization by putting the system into sleep mode. In other words, collaboration of DVS, DCR, DPM and task rescheduling need to be explored in a systematic way.

While the task schedulability in terms of timing can be easily checked through well-established theorems, it remains a major challenge to verify the schedulability in a system which is constrained by both limited energy and tolerable operating temperature. An effective modeling method needs to be devised for this specific problem. The model needs to be compatible with formal techniques such as model checking. Moreover, the approach needs to address state space explosion problem for larger problem instances.

Multicore architecture imposes new challenges over uniprocessor systems and creates new optimization opportunities. Different optimization strategies should be applied to the private caches in each core and the shared cache of all cores. For example, private caches can benefit from DCR whereas the shared cache can take advantage of cache partitioning. It is challenging to efficiently integrate cache reconfiguration and partitioning techniques simultaneously for energy optimization. Moreover, given the real-time workloads, how to efficiently map tasks to each core also remains to be explored.

1.3 Research Contributions

My research proposes novel techniques to address design challenges mentioned in Section 1.2. The objective of my research is to develop efficient tools, scheduling

algorithms and reconfiguration techniques for real-time embedded system optimizations. Figure 1.3 summarizes the key contributions of this dissertation. It also outlines the comprehensive nature of my research. The proposed research will focus on major system components (i.e., processor, cache, memory) with various optimization objectives (i.e., power, energy, temperature and performance) using a wide variety of reconfiguration techniques (i.e., DCR, DVS and scheduling) for both single-core and multicore systems. This research achieves the goals by involving both dynamic and static approaches in terms of information collection, evaluation, analysis and decision-making.

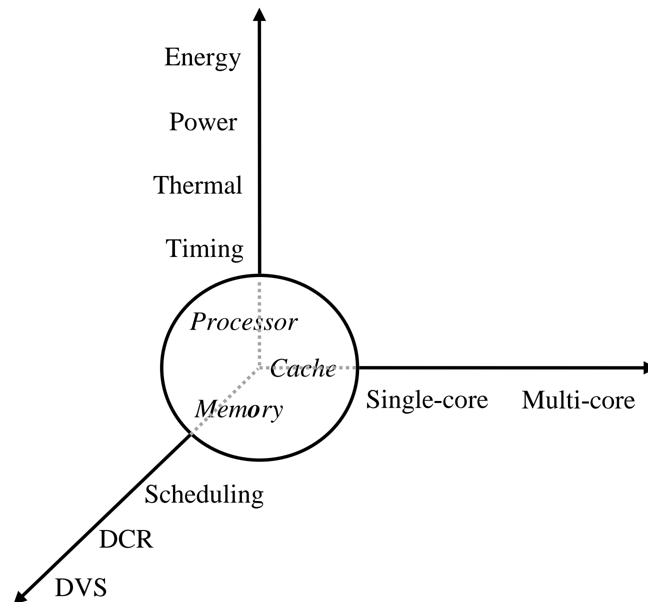


Figure 1-3. Optimization targets, objectives and techniques.

Figure 1-4 outlines the five major research contributions of this dissertation that are summarized as follows.

- **Dynamic Cache Reconfiguration:** This dissertation exploits dynamic cache reconfiguration in both statically and dynamically scheduled soft real-time systems. The research proposes design-time profiling techniques specifically for systems with reconfigurable cache and preemptive tasks. It also presents design space exploration heuristics for multi-level cache hierarchy tuning. The static profiling information is efficiently utilized during runtime for energy minimization while maintain the system's quality of service.

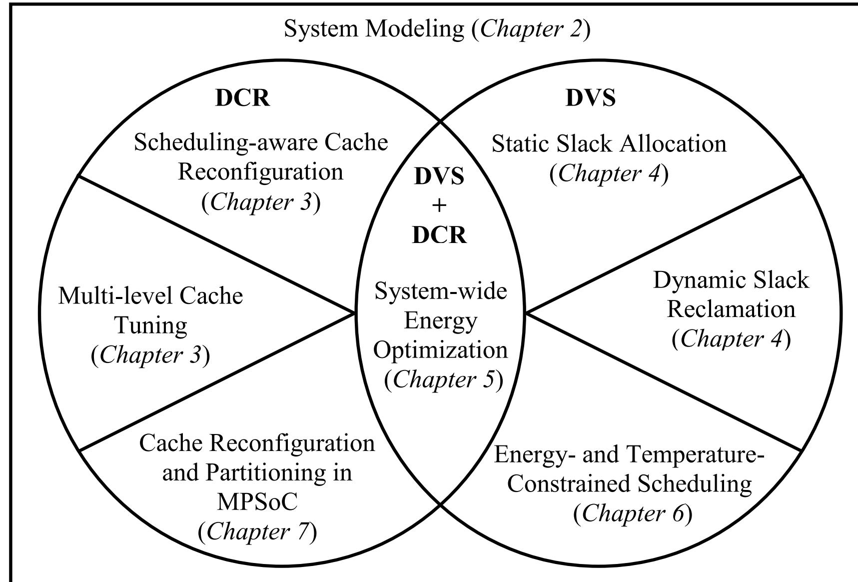


Figure 1-4. Dissertation outline.

- **Dynamic Voltage Scaling and Task Scheduling:** This dissertation proposes novel algorithms based on processor voltage/frequency scaling and task scheduling for preemptive hard real-time systems. The proposed approach outperforms existing inter-task techniques and is based on approximation algorithms that can guarantee to generate solutions within a specified quality bound (e.g., within 1% of the optimal). It also examines and resolves dynamic slack reclamation problem that involves slack reallocation and task rescheduling at runtime.

- **System-wide Energy Minimization:** This dissertation systematically employs DVS and DCR simultaneously for system-wide leakage-aware energy minimization. The proposed research approaches the problem by devising an energy estimation framework for major system components, studying their impact on DVS assignments and proposing algorithms for configuration selection as well as task procrastination. Based on these insights, we also develop a general algorithm for real-time dynamic reconfiguration which accounts for varying runtime overhead.

- **Temperature- and Energy-Constrained Scheduling:** This dissertation proposes a formal method based approach to verify the schedulability in temperature- and energy-constrained systems. Timed automaton is used to model the problem, which is automatically solved by the model checker. The research uses SAT solver to alleviate state explosion problems.

- **Energy Optimization in Multicore Systems:** This dissertation presents a novel energy optimization technique which employs both DCR and CP for real-time multicore systems. Our static profiling based algorithm is designed to judiciously find beneficial cache configurations (of private caches) for each task as well as partitioning

schemes (of the shared cache) so that the cache energy consumption is minimized while the task deadline is satisfied. We study both fixed and varying CP schemes. We also explore task mapping heuristics and Gated-V_{dd} cache line technique in our study.

The rest of this dissertation is organized as follows. Chapter 2 describes modeling of real-time multitasking systems as well as models for power, energy, performance and temperature. In Chapter 3, we present our scheduling-aware cache reconfiguration techniques for soft real-time systems. Chapter 4 presents our DVS-based energy-aware scheduling algorithms that can exploit both static and dynamic time slacks. Chapter 5 discusses our approaches for system-wide energy minimization. Scheduling problem in energy- and temperature-constrained systems is presented in Chapter 6. Chapter 7 presents our energy optimization techniques for cache hierarchy in multicore architecture. Finally, Chapter 8 discusses future directions and concludes the dissertation.

CHAPTER 2
MODELING OF REAL-TIME AND RECONFIGURABLE SYSTEMS

In this chapter, we describe various models used in this dissertation. Modeling plays an important role in developing real-time scheduling and dynamic reconfiguration techniques in real-time embedded systems. In this chapter, we first describe how to model a real-time system supporting dynamic reconfigurations. Next, we describe system-wide energy and thermal models. These models will be used in all subsequent chapters. In certain cases, some of these models will be modified and better reflect the specific context.

2.1 System Model

The target uniprocessor system can be modeled as:

- A highly configurable cache architecture which supports h different configurations C$\{c_1,c_2, ... ,c_h\}$ and/or,
- A voltage scalable processor which supports l discrete voltage levels V$\{v_1,v_2, ... ,v_l\}$,
- A set of m independent tasks T$\{\tau_1, \tau_2, ... ,\tau_m\}$.

The two main categories of real-time tasks that we consider are:

- Periodic task $\tau_i \in$ T has known attributes including worst-case workload, arrival time a_i, deadline d_i and period p_i,
- Aperiodic/sporadic task $\tau_i \in$ T has known has known attributes including workload and inter-arrival time.

In some problems, we consider frame-based tasks all of which have common arrival time and deadline. For dynamic algorithms, we assume the actual-cast execution time follows a distribution.

2.2 Energy Models

2.2.1 Cache Energy Model

For each individual cache, the energy consumption is modeled as the sum of dynamic energy E_{cache}^{dyn} and static energy E_{cache}^{sta}:

$$E_{cache} = E_{cache}^{dyn} + E_{cache}^{sta} \tag{2–1}$$

The number of cache accesses n_{cache}^{access}, cache misses n_{cache}^{misses} and clock cycles CC are obtained from microarchitectural simulation for given tasks and cache configurations. In multi-level cache subsystems, the L1 cache access are issued by the processor while L2 cache accesses are from L1 caches whenever there is a L1 cache miss. We use t_{cycle} to denote the clock cycle length. Let E_{access} and E_{miss} denote the energy consumed by one cache access and miss, respectively. Therefore, we have:

$$E_{cache}^{dyn} = n_{cache}^{access} \cdot E_{access} + n_{cache}^{misses} \cdot E_{miss} \quad (2–2)$$

$$E_{miss} = E_{offchip} + E_{\mu P_stall} + E_{block_fill} \quad (2–3)$$

$$E_{cache}^{sta} = P_{cache}^{sta} \cdot CC \cdot t_{cycle} \quad (2–4)$$

where $E_{offchip}$ is the energy required for fetching data from lower levels of memory hierarchy, $E_{\mu P_stall}$ is the energy consumed when the processor is stalled due to cache miss, E_{block_fill} is for cache block refilling after a miss and P_{cache}^{sta} is the static power consumption of cache. For system-wide energy optimization (Chapter 5) where lower-level memory and buses are modeled separately, $E_{offchip}$ and $E_{\mu P_stall}$ in Equation (2–3) are counted during the computation of corresponding components (e.g., for L2 cache misses, $E_{offchip}$ is incorporated in the energy consumption of off-chip buses and main memory). Here, values of E_{access}, P_{cache}^{sta} and E_{block_fill} for different cache configurations are collected from CACTI [41].

2.2.2 Processor Energy Model

Since short circuit power is negligible [115], the energy consumed in a processor mainly comes from dynamic and static power. The dynamic power can be computed as:

$$P_{proc}^{dyn} = C_{eff} \cdot V_{dd}^2 \cdot f \quad (2–5)$$

where C_{eff} is the total effective switching capacitance of the processor, V_{dd} is the supply voltage level and f is the operating frequency. We adapt the analytical processor energy model based on [73], whose accuracy has been verified with SPICE simulation. The

threshold voltage V_{th} is presented as:

$$V_{th} = V_{th1} - K_1 \cdot V_{dd} - K_2 \cdot V_{bs} \tag{2–6}$$

where V_{th1}, K_1, K_2 are all constants and V_{bs} represents the body bias voltage. Static current mainly consists of the subthreshold current I_{subth} and the reverse bias junction current I_j. Hence, the static power is given by:

$$P_{proc}^{sta} = L_g \cdot (V_{dd} \cdot I_{subth} + |V_{bs}| \cdot I_j) \tag{2–7}$$

where L_g is the number of devices in the circuit, I_j is approximated as a constant and I_{subth} can be calculated by:

$$I_{subth} = K_3 \cdot e^{K_4 V_{dd}} \cdot e^{K_5 V_{bs}} \tag{2–8}$$

where K_3, K_4 and K_5 are constant parameters. Obviously, to avoid junction leakage power overriding the gain in lowering I_{subth}, V_{bs} has to be constrained (between 0 and -1V). Let P_{proc}^{on} be the intrinsic energy needed for keeping the processor on (idle energy). The processor power consumption can be computed as:

$$P_{proc} = P_{proc}^{dyn} + P_{proc}^{sta} + P_{proc}^{on} \tag{2–9}$$

The cycle length, t_{cycle}, is given by a modified α power model:

$$t_{cycle} = \frac{L_d \cdot K_6}{(V_{dd} - V_{th})^{\alpha}} \tag{2–10}$$

where K_6 is a constant. L_d can be estimated as the average logic depth of all instructions' critical path in the processor. The constants mentioned above are technology and design dependent. Table 2-1 lists the constants for a 70nm technology processor.

The processor energy consumption then becomes:

$$E_{proc} = P_{proc} \cdot CC \cdot t_{cycle} \tag{2–11}$$

Table 2-1. Constants for 70nm technology

Const	Value	Const	Value	Const	Value
K_1	0.063	K_6	5.26×10^{-12}	V_{th1}	0.244
K_2	0.153	K_7	-0.144	I_j	4.80×10^{-10}
K_3	5.38×10^{-7}	V_{dd}	$[0.5, 1.0]$	C_{eff}	0.43×10^{-9}
K_4	1.83	V_{bs}	$[-1.0, 0.0]$	L_d	37
K_5	4.19	α	1.5	L_g	4×10^6

2.2.3 Bus Energy Model

The average dynamic power consumption of various system buses can be calculated by [28]:

$$P_{bus}^{dyn} = \frac{1}{2} \cdot C_{bus} \cdot V_{dd}^2 \cdot n_{trans} \cdot f \tag{2–12}$$

where C_{bus} is the load capacitance of the bus, V_{dd} is the supply voltage, f is the bus frequency and n_{trans} denotes the average number of transitions per time unit on the bus line, as shown below:

$$n_{trans} = \frac{\sum_{t=0}^{T-1} H(B^{(t)}, B^{(t+1)})}{T} \tag{2–13}$$

where T is the total number of discretized units of the system execution time and $H(B^{(t)}, B^{(t+1)})$ gives the Hamming distance between the binary values on the bus at two neighboring time units in T. Therefore, the total energy consumption of a bus is determined by its dynamic power P_{bus}^{dyn} and static power P_{bus}^{sta}:

$$E_{bus} = (P_{bus}^{dyn} + P_{bus}^{sta}) \cdot CC \cdot t_{cycle} \tag{2–14}$$

2.2.4 Main Memory Energy Model

Memory consists of DRAM has three main sources of power consumption: dynamic energy due to accesses E_{mem}^{dyn}, static power P_{mem}^{sta} and refreshing power P_{mem}^{ref}. Specifically, we have:

$$E_{mem}^{dyn} = n_{mem}^{access} \cdot E_{access} \tag{2–15}$$

where n_{mem}^{access} is the number of memory accesses and E_{access} denotes the dynamic energy required per access. Therefore, we have:

$$E_{mem} = E_{mem}^{dyn} + (P_{mem}^{sta} + P_{mem}^{ref}) \cdot CC \cdot t_{cycle} \tag{2–16}$$

Values of E_{access}, P_{mem}^{sta} and P_{mem}^{ref} can be collected from CACTI [41].

2.3 Thermal Model

A thermal RC circuit is normally utilized to model the temperature variation behavior of a microprocessor [138]. Here we introduce the RC circuit model proposed in [106], which is widely used in recent researches [138], to capture the heat transfer phenomena in the processor. If P denotes the power consumption during a time interval, R denotes the thermal resistance, C represents the thermal capacitance, T_{amb} and T_0 are the ambient and initial temperature, respectively, the temperature at the end of the time interval t can be calculated as:

$$T = P \cdot R + T_{amb} - (P \cdot R + T_{amb} - T_{init}) \cdot e^{\frac{-t}{RC}} \tag{2–17}$$

where t is the length of the time interval. If t is long enough, T will approach a steady-state temperature $T_s = P \cdot R + T_{amb}$.

2.4 Summary

This chapter presented system models for describing real-time reconfigurable systems as well as associated energy and thermal models. They will be extensively used in following chapters.

CHAPTER 3
DYNAMIC CACHE RECONFIGURATION FOR SOFT REAL-TIME SYSTEMS

Research has shown that cache subsystem has become significant contributor in the overall energy consumption comparable to other components of the processor [72][96]. Since different programs may have distinct requirements on cache configuration during execution, we can achieve significant energy efficiency as well as performance improvements by employing dynamic cache reconfiguration (DCR) in the system. Although reconfigurable caches are highly beneficial in general-purpose platforms such as desktop and embedded systems [136] [33] recently, it has not been considered in real-time systems due to several fundamental challenges. How to employ and make efficient use of reconfigurable caches in such systems remains unsolved. Determining the appropriate cache configuration typically requires time-consuming evaluation of different candidates. Furthermore, any change in cache configuration on-the-fly may arbitrarily alter task execution time. In hard real-time systems, the benefit of reconfiguration is limited since both of these facts can make scheduling decisions difficult and eventually may lead to unpredictable system behavior. However, soft real-time systems offer much more flexibility, which can be exploited to achieve considerable energy savings at the cost of minor impacts to user experiences.

This chapter presents a novel methodology for applying cache reconfiguration in soft real-time systems with preemptive task scheduling. The proposed approach provides an efficient scheduling-aware cache tuning strategy based on static profiling and is applicable for both statically and dynamically scheduled soft real-time systems. Generally speaking, this technique is useful in any multitasking systems. The goal is to optimize energy consumption with performance considerations via reconfigurable cache tuning while ensuring that the majority of the task deadlines are met. We first consider single-level cache reconfiguration. As shown in [114], L1 cache energy consumption can be a significant part in overall energy optimization. In fact, some small embedded systems

executing light-weight kernels are very likely to not even have L2 cache. Our approach is independent of the actual cache sizes and is applicable for both large systems with larger L1 caches and small systems with smaller L1 caches. Next, we study dynamic cache reconfiguration in systems with two-level cache hierarchy. We investigate the unique challenge in multi-level cache tuning and propose several design space exploration heuristics that can be employed to make tradeoff between energy savings and static profiling time.

The rest of this chapter is organized as follows. Section 3.1 discusses background and related work in cache reconfiguration. Section 3.2 presents our scheduling-aware cache reconfiguration techniques in detail. Section 3.3 describes multi-level cache tuning heuristics. Experimental results are presented in Section 3.4. Finally, Section 3.5 summarizes this chapter.

3.1 Related Work

3.1.1 Caches in Real-Time Systems

Cache systems are included in nearly all computing systems to temporarily store frequently accessed instructions and data. Since caches have a much faster access time compared to main memory, caches effectively alleviate the increasing performance disparity between the processor and memory by exploiting the temporal and spatial locality properties of programs. However, historically, incorporating caches into real-time embedded system faces serious difficulties due to the unpredictability imposed on the system. Cache affects data access pattern and hence creates variations in data access time. For example, in a preemptive system, since a task may be interrupted by a higher-priority task and resumed again at a later time, the data of preempted tasks may be evicted from the cache. This may result in a period of cold-start compulsory cache misses, many of which may have been cache hits if the task had not been preempted. This makes it difficult to calculate task's worst-case execution time (WCET), knowing which is a prerequisite of most traditional scheduling algorithms.

Since caches introduce intra-task interference so that a specific task's execution time becomes variable at runtime, a great deal of research efforts are directed at employing caches in real-time systems either by proving schedulability through WCET analysis or avoiding hazardous compulsory miss uncertainty altogether. Cache-aware WCET analysis is a static, design time analysis of tasks in the presence of caches to predict cache impact on task execution times [86]. Cache locking [87] is a technique in which useful cache lines are "locked" in the cache when a task is preempted so that these blocks will not be evicted to accommodate the new incoming task. Through cache line locking, the WCET and cache behavior becomes more predictable since the major delay from data replacement and access is avoided. Cache partitioning [125] is a similar but more aggressive approach where the cache is partitioned into reserved regions, each of which can only cache data associated with a dedicated task. However, a potential drawback to both cache locking and cache partitioning is per-task reduction of cache resources. To alleviate this limitation, cache-related preemption delay analysis [112][108] features tight delay estimation so that prediction accuracy is higher than traditional WCET analysis. This improved accuracy can in turn result in a durable task schedule. Scratch-pad memories, like caches, are also on-chip RAMs but mapped onto the address space of the processor at a specified range. Puant et al. [88] proposed an off-line content-selection algorithm for both scratch-pad memory and cache with line locking ability to improve both predicability and WCET estimation. Our approach is applicable to real-time systems that employ caches.

3.1.2 Reconfigurable Cache Architectures

There are many existing general or application specific reconfigurable cache architectures. Motorola M*CORE processor [72] provides way shut-down and way management, which has the ability to specify the content of each specific way (instruction, data, or unified way). Settle et al. [98] proposed a dynamically reconfigurable cache specifically designed for chip multi-processors. The reconfigurable cache architecture proposed by Zhang et al. [136] imposes no overhead to the critical path, thus cache

access time does not increase. Furthermore, the cache tuner consists of a small custom hardware or a lightweight process running on a co-processor, which can alter the cache configuration via hardware or software configuration registers. The underlying cache architecture consists of four separate banks as shown in Figure 3-1 (a), each of which acts as a separate way. The cache tuner can be implemented either as a small custom hardware or lightweight software running on a co-processor which changes the cache configuration through special registers. In order to reconfigure associativity, way concatenation, shown in Figure 3-1 (b), logically concatenates ways together so that the associativity can be changed accordingly without affecting total cache size. The required configure circuit consists of only eight logic gates and two single-bit registers. Varying cache size is achieved by shutting down certain ways, as shown in Figure 3-1 (c), using gated-V_{dd} technique. An extra transistor is used for every array of SRAM cells. Cache line size is configured by setting a unit-length base line size and then fetching subsequent lines if the line size increases as illustrated in Figure 3-1 (d). Therefore, the configurable cache architecture achieves configurability using rather simple hardware thus requires very minor overhead which makes this architecture especially suitable for embedded systems [136].

3.1.3 Caches Tuning Techniques

Given a runtime reconfigurable cache, determining the best cache configuration is a difficult process. Dynamic and static analysis are two possible techniques. With dynamic analysis, cache configurations are evaluated during runtime to determine the best configuration. Two methods are possible for runtime cache analysis. The first method is intrusive and physically changes the cache to each configuration in the design space, examines the effects of each configuration, and chooses the best cache configuration. This method is inappropriate for real-time systems since it imposes unpredictable performance overhead during exploration. To eliminate this performance overhead, a second method employs an N-experts based analysis [33]. In this technique, an auxiliary structure evaluates all cache configurations simultaneously. The best cache configuration is

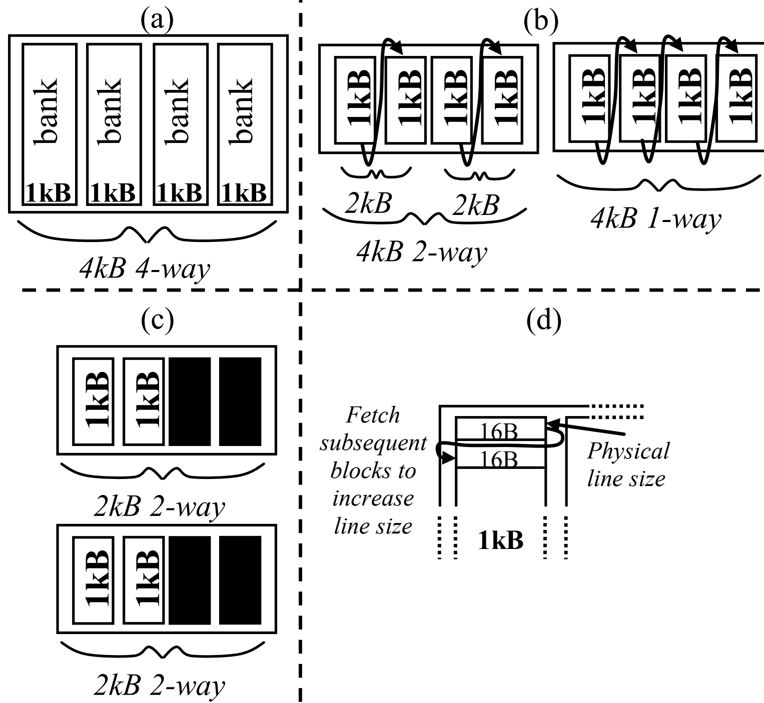


Figure 3-1. Reconfigurable cache architecture: (a) base cache bank layout, (b) way concatenation, (c) way shutdown, and (d) configurable line size.

determined by inspecting this auxiliary structure, allowing the cache to change to the best configuration in one-shot, without incurring any performance overhead. Even though this method is non-intrusive, the auxiliary data structure is too power hungry to continuously evaluate the system, and thus can only operate periodically.

With static analysis, various cache alternatives are explored and the best cache configuration is selected for each application in its entirety [32] – application-based tuning, or for each phase of execution within an application [99] – phase-based tuning. Since applications tend to exhibit varying execution behavior throughout its execution, phase-based tuning allows for the cache configuration to be specialized to each particular period, resulting in greater energy savings than application-based tuning. Regardless of the tuning method, the predetermined best cache configuration (based on design requirements) could be stored in a look-up table or encoded into specialized instructions. The static analysis approach is most appropriate for real-time systems due to its non-intrusive nature. Previous methods focus solely on energy savings or Pareto-optimal

points trading off energy consumption and performance. However, none of these methods consider task deadlines, which are imperative in real-time systems. In other words, the existing approaches were designed for desktop applications but not applicable for real-time systems.

3.2 SACR: Scheduling-Aware Cache Reconfiguration

3.2.1 Overview

This section presents a simple illustrative example to show how reconfigurable caches benefit real-time systems. This example assumes a system with two tasks, *T1* and *T2*. Traditionally if a reconfigurable cache technique is not applied, the system will use a *base cache* configuration Cache_{base}, which is defined in Definition 1.

Definition 1. *The term **Base cache** refers to the configuration selected as the optimal one for tasks in the target system with respect to energy as well as performance based on static analysis. Caches in such systems are chosen to ensure durable task schedules and their configurations are fixed throughout all task executions.*

In the presence of a reconfigurable cache, as shown in Figure 3-2, different optimal cache configurations are determined for every "*phase*" of each task. For ease of illustration, we divide each task into two phases: $phase_1$ starts from the beginning to the end, and $phase_2$ starts from the half position of the dynamic instruction flow (midpoint) to the end. The terms Cache^1_{T1}, Cache^2_{T1}, Cache^1_{T2}, and Cache^2_{T2} represent the optimal cache configurations for $phase_1$ and $phase_2$ of task *T1* and *T2*, respectively. These configurations are chosen statically to be more energy efficient (with same or better performance), in their specific phases, than the global base cache, Cache_{base}.

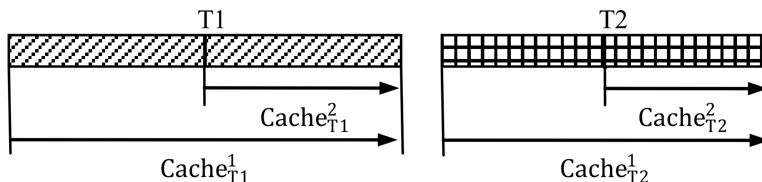


Figure 3-2. Cache configurations selected based on task phases

Figure 3-3 illustrates how energy consumption can be reduced by using our approach in real-time systems. Figure 3-3 (a) depicts a traditional system and Figure 3-3 (b) depicts a system with a reconfigurable cache (our approach). In this example, *T2* arrives (at time *P1*) and preempts *T1*. In a traditional approach, the system executes using Cache_{base} exclusively. With a reconfigurable cache, the first part of *T1* executes using Cache^1_{T1}. Similarly, Cache^1_{T2} is used for execution of *T2*. Note that the actual preemption point of *T1* is not exactly at the same place where we pre-computed the optimal cache configuration (midpoint) since tasks may arrive at any time. When *T1* resumes at time point *P2*, the cache is tuned to Cache^2_{T1} since the actual preemption point is closer to the midpoint compared to the starting point. The overall energy consumed using a reconfigurable cache results in the energy savings due to use of different energy optimal caches for each phase of task execution compared to using one global base cache in the traditional system. Our experimental results suggest that the proposed approach can significantly reduce energy consumption of the memory subsystem with only very little performance penalty.

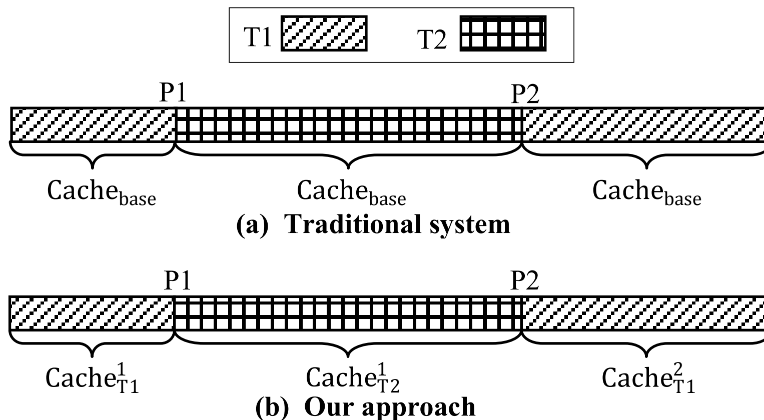


Figure 3-3. Dynamic cache reconfigurations for tasks T1 and T2

3.2.2 Phase-based Optimal Cache Selection

This section describes our static analysis approach to determine the optimal cache configurations for various task phases. In a preemptive system, tasks may be interrupted and resumed at any point of time. Each time a task resumes, cache performance for

the remainder of task execution will differ from the cache performance for the entire application due to its own distinguishing behaviors as well as cold-start compulsory cache misses. Therefore, the optimal cache configuration for the remainder of the task execution may be different.

Definition 2. ***Phase*** *is defined as the execution period between one potential preemption point (also called* ***partition points****) and task completion. The phase that starts at i^{th} partition point is denoted as phase* $\boldsymbol{p}_n^i$*, where* n *is the total number of phases of that task.*

Figure 3-4 depicts the general case where a task is divided by *n-1* predefined *potential preemption points* (P_1, P_2 ... P_{n-1}). P_0 and P_n are used to refer to the start and end point of the task, respectively. Here, C_0, C_1 ... C_{n-1} represent the optimal cache configuration (either energy or performance) for each phase, respectively. To observe the variation in cache requirements for each phase, Table 3-1 shows variation in energy-optimal and performance-optimal instruction and data caches for each phase[1] . For example, the energy-optimal cache configuration for the phase starting from the half point to the completion (C_2) of benchmark *cjpeg* has 2048-byte capacity, 16-byte block and 2-way associativity.

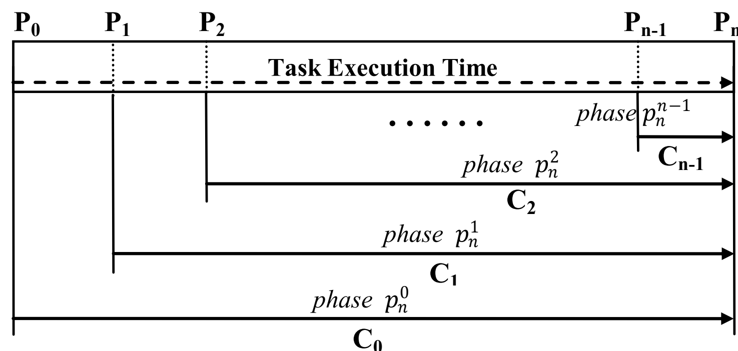


Figure 3-4. Task partitioning at n potential preemption points (P_i) resulting in n phases. Each phase comprises execution from the invocation/resumption point to task completion. C_i denotes the cache configuration used in each phase.

[1] In this dissertation, for example, 4KB_2W_16B means a cache configuration with 4096-byte capacity, 16-byte line size and 2-way associativity.

Table 3-1. Optimal cache configurations for task phases. Each configuration is denoted by the total cache size in kilobytes (kb), followed by the associativity in number of ways (w), followed by the line size in bytes (b).

	CJPEG			
	I-Cache		**D-Cache**	
	Energy Optimal	**Performance Optimal**	**Energy Optimal**	**Performance Optimal**
C0	4KB_2W_16B	4KB_4W_16B	4KB_4W_16B	4KB_4W_16B
C1	4KB_2W_16B	4KB_4W_32B	4KB_4W_16B	4KB_4W_16B
C2	2KB_2W_16B	4KB_4W_16B	2KB_2W_32B	4KB_4W_16B
C3	2KB_2W_16B	4KB_4W_16B	2KB_2W_32B	4KB_4W_16B
	RAWCAUDIO			
	I-Cache		**D-Cache**	
	Energy Optimal	**Performance Optimal**	**Energy Optimal**	**Performance Optimal**
C0	1KB_1W_16B	4KB_2W_64B	2KB_2W_16B	2KB_2W_16B
C1	1KB_1W_16B	2KB_2W_16B	2KB_2W_16B	4KB_4W_16B
C2	1KB_1W_16B	4KB_4W_16B	2KB_2W_16B	4KB_4W_16B
C3	1KB_1W_16B	4KB_2W_16b	2KB_2W_32B	4KB_4W_16B
	A2TIME01			
	I-Cache		**D-Cache**	
	Energy Optimal	**Performance Optimal**	**Energy Optimal**	**Performance Optimal**
C0	4KB_4W_16B	4KB_4W_16B	4KB_2W_32B	4KB_4W_16B
C1	4KB_4W_16B	4KB_4W_16B	2KB_2W_32B	4KB_4W_16B
C2	4KB_4W_16B	4KB_4W_16B	2KB_2W_16B	4KB_4W_16B
C3	4KB_4W_16B	4KB_4W_16B	2KB_2W_16B	2KB_2W_16B

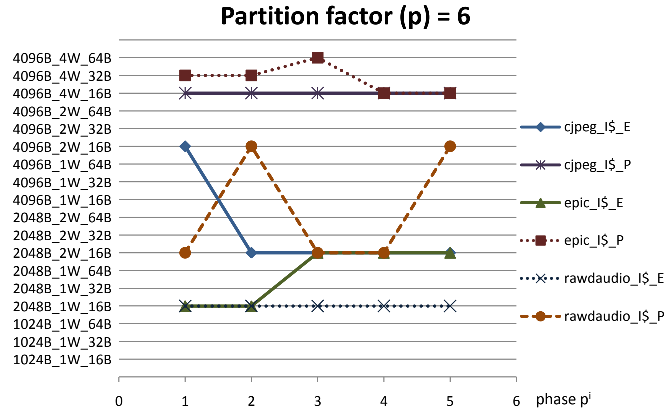
During static profiling, a *partition factor* is chosen that determines the number of potential preemption points and resulting phases. Partition granularity is defined as the number of dynamic instructions between two partition points and is determined by dividing the total number of dynamically executed instructions by the partition factor. Intuitively, the optimal partition granularity should be a single instruction, potentially leading to the largest amount of energy savings. However, such a tiny granularity would result in a prohibitively large look-up table, which is not feasible due to area as well as searching time constraints. Due to cache locality over time, the optimal performance cache is tend to be the largest cache [37] and the optimal energy cache is not necessarily

the smallest dynamic energy cache [30]. Thus, a trade-off should be made to determine a reasonable partition factor based on energy-savings potential and acceptable overheads. An important question one can raise is whether a larger partition factor (finer granularity) always reveals more energy savings. However, to answer this question, we need to address the following two issues.

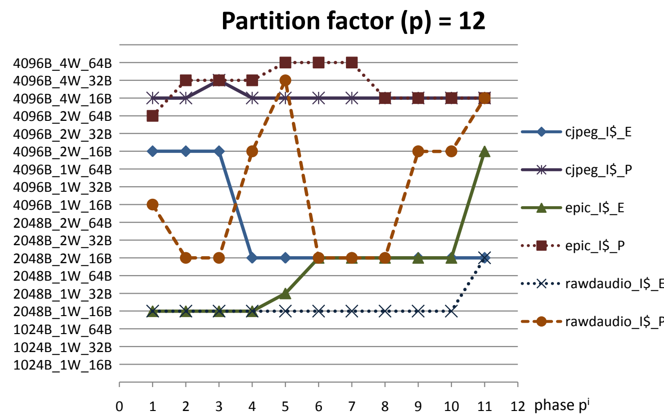
The first issue is how the optimal cache configuration for each phase varies when the partition factor increases. We noticed that, for each task, once the partition factor is larger than a certain threshold, more and more neighboring partitions share the same optimal cache configuration. We explored how partition factor can affect the variation of optimal (both energy and performance) cache configurations for each benchmark in MediaBench [66] and EEMBC [25] – the two benchmark suites we use in Section 3.4. Figure 3-5 shows the results for some of them (*cjpeg*, *epic* and *rawdaudio*) using partition factor 6, 12 and 18. For the same benchmark, the optimal cache configuration for each phase varies in a consistent pattern across different partition factors. For example, the energy-optimal instruction cache configuration for benchmark *cjpeg* (cjpeg_I$_E) is 4096B_2W_16B for the first several phases and then changes to 2048B_2W_16B starting from about one third of the program: *phase* p_6^2, *phase* p_{12}^4 and *phase* p_{18}^6 when partition factor is 6, 12 and 18, respectively. In other words, larger partition factor makes more and more phases share the same optimal cache configuration with their neighboring phases. Exception can happen when partition factor increases, different optimal cache configuration from lower partition factor case is found. For example, the performance-optimal instruction cache configuration of benchmark *cjpeg* (cjpeg_I$_P in Figure 3-5 (b)) with partition factor 12 differs at *phase* p_{12}^3 compared to the similar position when partition factor is 6 (Figure 3-5 (a)). Our experimental result shows that though discrepancies do happen, their impact on energy savings is normally negligible because the energy/performance difference between the newly picked cache configuration and the original one is usually very small. From this observation, one can derive the

fact that application behavior can sufficiently be captured by a certain partition factor. This is evident due to the well-established 90/10 law of execution – 90% of the execution time is spent in only 10% of the code – in which the 90% of the time is typically spent executing loops. For each loop iteration, except the first and last iterations, execution behavior is typically similar, thus resulting in the same optimal cache configuration for all other iterations. For a loop with N iterations, the partition factor only need to be large enough to capture all dynamic instructions of iterations 2 through (N - 1), as any smaller granularity would capture a subset of iterations, each of which may have the same optimal configuration. Thus, we define a *stage* of execution as a range of consecutive dynamic instructions in which a common optimal cache configuration exists.

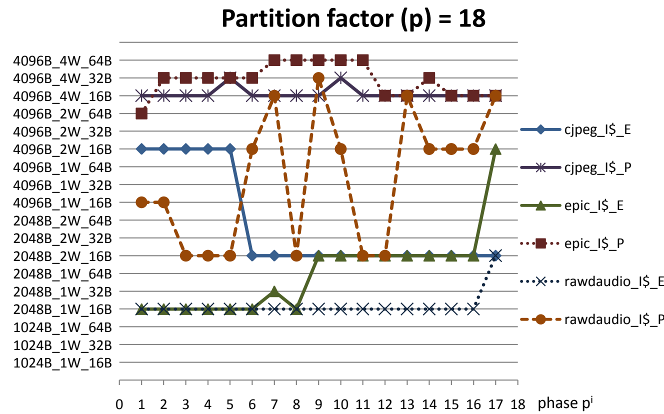
The second issue is whether finer partition granularity always brings more energy savings than a coarser one. With finer granularity, if there is no extra variation in the optimal cache configuration across phases, there will be no additional energy savings since the same cache configurations are being used. If variations can be observed, according to our experiments, they only happen at stage boundaries, which is a very limited portion in the entire program. Figure 3-6 gives an example explaining why this is the case. Suppose there are two tasks: *T1* and *T2* in the system and partition factor (p) can be chosen as 4 or 8. A valid schedule of them is shown in Figure 3-6. Since *T2* is executed as a whole, the cache configuration used is the optimal one for the entire task, which are the same using both partition factors. *T1* is preempted by *T2*. So when *T1* resumes, a different cache configuration should be picked based on the preemption point as well as the partition factor. As discussed in the first issue, higher partition factor shows consistent variation pattern of optimal cache configuration with only minor exceptions. Suppose when partition factor is 4, for task *T1*, the cache configuration picked for *phase* p_4^0, *phase* p_4^1, *phase* p_4^2 and *phase* p_4^3 are C_A, C_A, C_B and C_B, respectively. And when partition factor is 8, they are C_A, C_A, C_A, C_C, C_B, C_B, C_B and C_B for *phase* p_8^0 to *phase* p_8^7, respectively. Using the nearest-neighbor technique as discussed in Section 3.2.4.1, the advantage of



(a)



(b)



(c)

Figure 3-5. Optimal cache configuration variation under different partition factors (a) Partition factor = 6, (b) Partition factor = 12, (c) Partition factor = 18 (Here I$ represents instruction cache. 'E' stands for energy-optimal and 'P' stands for performance-optimal)

using partition factor 8 over 4 become effective only when the preemption happens within the range from 5/16 to 7/16 of *T1* (*effective area*) since C_C will be chosen instead of C_A or C_B. Note that C_C may be more energy/performance efficient for the rest part of *T1* than C_A and C_B. From the entire system's point of view, higher partition factor (8) does not help for *T2* as well as *T1* if them never get preempted or the preemption does not happen in the *effective area*. Based on our experiments, merely 3 - 8% additional energy saving (for that one task) is possible only if the preemption occurs within the *effective area* of the dynamic instruction flow. Empirically, the *effective area* is usually 5 - 8% of the task. Due to these two small probabilities multiplied together, merely 0.4% on average, finer granularity partition can bring only minor benefit.

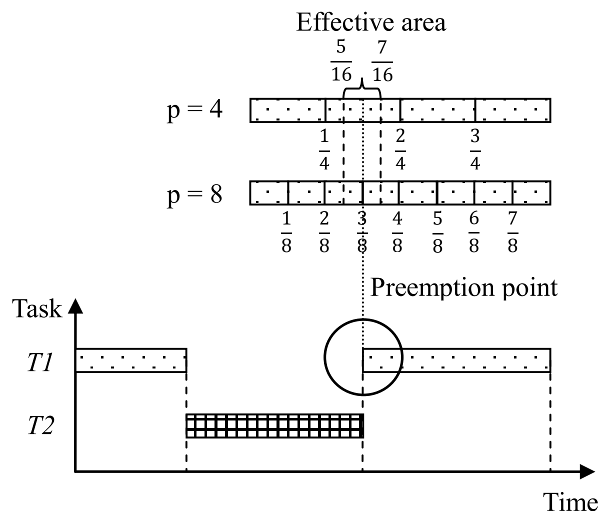


Figure 3-6. Effective range where a higher partition factor makes a difference

Thus, the goal of a system designer is to find a partition factor which leads to maximized energy reduction and minimizes the number of partition points that need to be stored. The rule of thumb is to find a partition factor minimizing the number of neighboring partitions that share the same optimal cache configuration. It could be a local optimal factor for each task if varying number of table entries for different tasks is allowed or it could be a global optimal factor for the task set. Based on our experience, a partition factor ranging from 4 to 7 is sufficient to make our technique working efficiently.

Static profiling generates a **profile table** that stores the potential preemption points and the corresponding optimal cache configurations for each task. Section 3.2.3 and 3.2.4 describe how this profile table is used during runtime in statically and dynamically scheduled systems.

3.2.3 Statically Scheduled Systems

With static scheduling, arrival times, execution times, and deadlines are known a priori for each task and this information serves as scheduler input. The scheduler then provides a schedule detailing all actions taken during system execution. According to this schedule, we can statically execute and record the energy-optimal cache configurations that do not violate any task's deadline for every execution period of each task. For soft real-time systems, global (system-wide) energy-optimal configurations can be selected as long as the configuration performance does not severely affect system behavior. After this profiling step, the profile table is integrated with the scheduler so that the cache reconfiguration hardware (cache tuner) can tune the cache for each scheduling decision.

3.2.4 Dynamically Scheduled Systems

With dynamic scheduling (online scheduling), scheduling decisions are made during runtime. In this scenario, task preemption points are unknown since new tasks may enter the system at any time with any time constraint. In this section, we present two versions of our technique based on the nature of the target system.

3.2.4.1 Conservative Approach

In some soft real-time systems where high service quality is required thus time constraints are pressing, only an extremely small number of violations are tolerable. The conservative approach could ensure that given a carefully chosen partition factor, almost every task could meet their deadlines with only few exceptions. To ensure the largest task schedulability, any reconfiguration decision will only change the cache into a lowest energy configuration whose execution time is not longer than that of the base cache. In other words, to maintain a high quality of service, only cache configurations with equal or higher

performance than the base cache are chosen for each task phase. Note that the chosen energy-optimal configuration may not be the global lowest energy configuration but is the one with lowest energy consumption given a specific time constraint. We denote them as deadline-aware energy-optimal cache configurations.

The scheduler chooses the appropriate cache configuration from the generated profile table that contains the energy-optimal cache configurations for each task phase. Table 3-2 (A) shows the profile table for task i with a partition factor p. $EO_i(n/p)$ represents the energy-optimal cache configuration for *phase* p_p^n of task i. Here, n/p represents the nth phase out of p phases. The total dynamic instruction count (TIN) refers to the number of dynamic instructions executed in a single run of that task.

During system execution, the scheduler maintains a task list keeping track of all existing tasks as shown in Table 3-2 (B). In addition to the static profile table Table 3-2 (A), runtime information such as arrival time (A_i), deadline (D_i), and number of already executed dynamic instructions (EIN) are also recorded. This information is stored not only for the scheduler, but also for the cache tuner. When a newly arrived task[2] begins execution for the first time, the deadline-aware energy-optimal cache configuration $EO_i(0/p)$ is obtained from the task list entry, and the cache tuner adjusts the cache appropriately. If preemption happens, the number of the preempted task's executed instructions (EIN) is calculated and stored in its task list entry.

As indicated in Section 3.2.2, potential preemption points are pre-decided during the profile table generation process. However, it is highly unlikely that the actual preemptions will occur precisely on these potential preemption points. Hence, a *nearest-neighbor* method is used to determine which cache configuration should be used. Essentially, if

[2] To be more specific, we actually mean "jobs" (execution instance of tasks). For simplicity, a more general term "task" is used in this chapter.

Table 3-2. (a) Static profile table and (b) Task list entry for task i for the conservative approach

Task ID: i	**Partition Factor: p**
Total Instruction Number (TIN)	
$EO_i(0/p)$	
$EO_i(1/p)$	
$EO_i(2/p)$	
......	
$EO_i(p-1/p)$	

(A)

Task ID: i	**Partition Factor: p**
Arrival time (A_i)	**Deadline (D_i)**
Total Instruction Number (TIN)	**Executed Instruction Number (EIN)**
$EO_i(0/p)$	
$EO_i(1/p)$	
$EO_i(2/p)$	
......	
$EO_i(p-1/p)$	

(B)

the preemption point falls between partition points *n/p* and *(n+1)/p*, the nearest point will be referred to select the current cache configuration. Algorithm 1 illustrates cache tuning algorithm for our conservative approach. This algorithm is called when a previously preempted task resumes its execution. It runs in a time complexity of $O(p)$, where p is the partition factor. Note that the returned cache configuration information is sent to the cache tuner.

As our experimental result shows, conservative approach obtains significant energy savings with little or no impact on quality of service. Minor number of time constraint violations are caused by cache behaviors in which optimal cache configuration for the period from the one preemption point to another preemption point and that for the pre-decided phase differ greatly. In other words, the chosen cache configuration may happen to be inefficient for the execution period between two actual preemption points such that the lost time is not reparable by the subsequent selected cache configurations in that task. Fortunately, this kind of behavior is relatively rare.

Algorithm 1: Selection of cache configuration for resumed preempted task in conservative approach

Input: Task list entry
Output: A deadline-aware cache configuration for the resumed task Tc.
for $i = 0$ to $p - 2$ **do**
  if $TIN_{Tc} \times i/p \leq EIN_{Tc} < TIN_{Tc} \times (i+1)/p$ **then**
    if $(EIN_{Tc} - TIN_{Tc} \times i/p) < (TIN_{Tc} \times (i+1)/p - EIN_{Tc})$ **then**
      $PHASE_{Tc} = i/p$;
    else
      $PHASE_{Tc} = (i+1)/p$;
    end if
  end if
end for
if $EIN_{Tc} \geq TIN_{Tc} \times (p-1)/p$ **then**
  $PHASE_{Tc} = (p-1)/p$;
end if
$Cache_{Tc} = EO_i(PHASE_{Tc})$;
Return: $Cache_{Tc}$

3.2.4.2 Aggressive Approach

For soft real-time systems in which only moderate service quality is needed, a more aggressive version of our approach can reveal additional energy savings at the cost of possibly violating several future task deadlines, but remain in an acceptable range.

Similar to the conservative approach, a profile table is associated with every task in the system; however this profile table contains the performance-optimal cache configuration (whose execution time is the shortest) in addition to the energy-optimal configuration (the one with lowest energy consumption among all candidates) for every task phase. In order to assist dynamic scheduling, the profile table also includes the corresponding phase's execution time (in cycles) for each configuration. Table 3-3 (A) shows the profile table for task i with a partition factor of p. The terms EO, EOT, PO, and POT stand for the energy-optimal cache configuration, the energy-optimal cache configuration's execution time, the performance-optimal cache configuration, and the performance-optimal cache configuration's execution time, respectively. Notice that, the performance and energy efficiency of a cache configuration is not in inverse proportion. The energy-optimal one does not necessarily have the worst performance. Compared to the base cache, it could have both better energy efficiency and performance.

Table 3-3 (B) shows the task list entry for the aggressive approach. The difference from the conservative approach (shown in Table 3-2 (B)) is that every task list entry also holds a Current Phase (CP_i) identifier. CP_i denotes the partition point that this task's execution just passed and is useful for cache reconfiguration upon task resumption. Note that newly inserted task's CP is initialized to 0. In addition to the task list, the scheduler also maintains another runtime data structure called the Ready Task List (RTL), which contains an identifier of each existing task currently ready to execute in the system.

Table 3-3. (a) Static profile table and (b) Task list entry for task i for the aggressive approach

Task ID: i		**Partition Factor: p**	
Total Instruction Number (TIN)			
$EO_i(0/p)$	$EOT_i(0/p)$	$PO_i(0/p)$	$POT_i(0/p)$
$EO_i(1/p)$	$EOT_i(1/p)$	$PO_i(1/p)$	$POT_i(1/p)$
$EO_i(2/p)$	$EOT_i(2/p)$	$PO_i(2/p)$	$POT_i(2/p)$
......			
$EO_i(p\text{-}1/p)$	$EOT_i(p\text{-}1/p)$	$PO_i(p\text{-}1/p)$	$POT_i(p\text{-}1/p)$

(A)

Task ID: i		**Partition Factor: p**	
Arrival time (A_i)		**Deadline (D_i)**	
Total Instruction Number (TIN)		**Executed Instruction Number (EIN)**	
Current Phase (CP)			
$EO_i(0/p)$	$EOT_i(0/p)$	$PO_i(0/p)$	$POT_i(0/p)$
$EO_i(1/p)$	$EOT_i(1/p)$	$PO_i(1/p)$	$POT_i(1/p)$
$EO_i(2/p)$	$EOT_i(2/p)$	$PO_i(2/p)$	$POT_i(2/p)$
......			
$EO_i(p\text{-}1/p)$	$EOT_i(p\text{-}1/p)$	$PO_i(p\text{-}1/p)$	$POT_i(p\text{-}1/p)$

(B)

To explain the aggressive approach, we use an illustrative example in which there are three tasks (jobs), *T1*, *T2*, and *T3*, with absolute deadlines D_{T1}, D_{T2}, and D_{T3}, where $D_{T2} < D_{T1} < D_{T3}$. According to EDF, the priority sequence is simply the opposite of the deadlines, which is $Pri_2 > Pri_1 > Pri_3$. Figure 3-7 shows a schedule for these tasks. Note that *P0*, *P1*, *P2*, and *P3* represent the time instances when any event (arrival, completion, etc.) occurs. At time point *P0*, *T1* arrives and the scheduler generates the task list entry

for *T1* and adds *T1* to the *RTL*. Since *T1* is currently the only task in the system, the scheduler instructs the cache tuner to configure the cache to $EO_{T1}(0/p)$ if and only if $P0 + EO_{T1}(0/p) < D_{T1}$, otherwise the cache will be tuned to $PO_{T1}(0/p)$, which ensures that *T1*'s deadline will be met. At time point *P1*, *T2* arrives with priority higher than the currently active task *T1*. The scheduler calculates *T1*'s current phase CP_{T1} and updates *T1*'s task list entry. Note that *T1*'s deadline may be violated if the following inequality holds:

$$P1 + POT_{T1}((CP_{T1} + 1) / p) + POT_{T2}(0 / p) > D_{T1} \tag{3–1}$$

This is obviously an underestimation of the execution time that the remaining portion of *T1* will take, thus more aggressive, but it favors tasks with higher priority (*T2*). However, if we use $POT_{T1}(CP_{T1}/p)$ in Equation 1, *T2* may have a lower chance of being accepted, but the lower priority task *T1* would more likely meet its deadline.

If Equation 1 does not hold, the scheduler determines *T2*'s cache configuration C_{T2} as follows (assuming $P_i + POT_i(0/p) < D_i$ for all tasks i otherwise task i is not schedulable in any situation):

if $(\mathrm{P1} + \mathrm{EOT}_{T2}(0/\mathrm{p}) > \mathrm{D}_{T2})$ **then**
  $\mathrm{C}_{T2} = \mathrm{PO}_{T2}(0/\mathrm{p})$
else if $(\mathrm{P1} + \mathrm{EOT}_{T2}(0/\mathrm{p}) + \mathrm{POT}_{T1}((\mathrm{CP}_{T1} + 1)/\mathrm{p}) < \mathrm{D}_{T1})$ **then**
  $\mathrm{C}_{T2} = \mathrm{EO}_{T2}(0/\mathrm{p})$
else if $(\mathrm{P1} + \mathrm{EOT}_{T2}(0/\mathrm{p}) + \mathrm{POT}_{T1}((\mathrm{CP}_{T1} + 1)/\mathrm{p}) > \mathrm{D}_{T1})$ **then**
  $\mathrm{C}_{T2} = \mathrm{PO}_{T2}(0/\mathrm{p})$

At time point *P2*, *T2* completes and *T1* resumes since it is the only ready task. The scheduler utilizes CP_{T1} to determine the appropriate partition to choose a cache configuration. This technique is similar in principle to the nearest neighbor method used in Section 3.2.4.1, except that a decision should be made whether to use the energy-optimal or performance-optimal configuration based on the remaining time budget. At some point during *T1*'s execution, *T3* arrives but since *T3* has a lower priority than

T1, *T3* begins execution after *T1* completes execution. By this time, *T3* is the only task and its cache configuration decision is made using the same method as task *T1* at time *P0*.

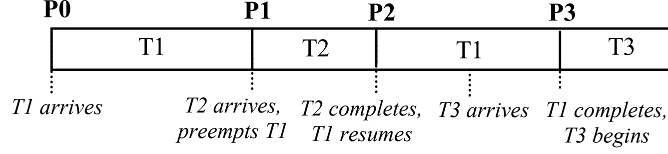


Figure 3-7. Task set and sample scheduling

Algorithm 2: Selection of cache configuration for aggressive approach

---

Input: Task list entry, ready task list and preemption point
Output: A appropriate cache configuration
Step 1: Calculate CP for the preempted task Tp. Insert Tp to RTL.
for $i = 0$ to $p - 1$ **do**
  if $TIN_{Tp} \times i/p \leq EIN_{Tp} < TIN_{Tp} \times (i+1)/p$ **then**
    $CP_{Tp} = i/p$;
  end if
end for
Step 2: Remove the task with maximum priority Tc from RTL.
Step 3: Sort all tasks in RTL by priority, T_1 to T_m, from highest to lowest. C represents the current time instant.
for $j = 1$ to m **do**
  if $C + POT_{Tc}(CP_{Tc}/p) + \sum_{i=1}^{j} POT_{Ti}((CP_{Ti}+1)/p) > D_{Tj}$ **then**
    Task D_{Tj} is subject to be discarded;
  end if
end for
Step 4: Select cache configuration for Tc. Let m' be the number of tasks in RTL left after **Step 3**.
if $C + EOT_{Tc}(CP_{Tc}/p) > D_{Tc}$ **then**
  $Cache_{Tc} = PO_{Tc}$;
else
  $EO_OK = true$;
  for $j = 1$ to m' **do**
    if $C + EOT_{Tc}(CP_{Tc}/p) + \sum_{i=1}^{j} POT_{Ti}((CP_{Ti}+1)/p) > D_{Tj}$ **then**
      $EO_OK = false$;
    end if
  end for
end if
if $EO_OK == true$ **then**
  $Cache_{Tc} = EO_{Tc}$;
else
  $Cache_{Tc} = PO_{Tc}$;
end if
Return: $Cache_{Tc}$

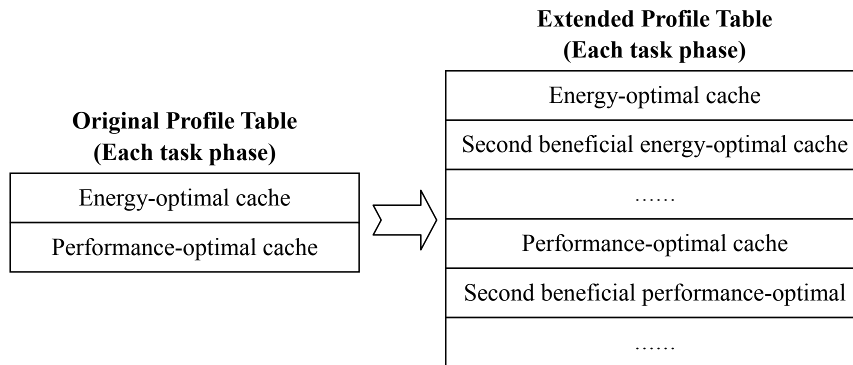
Algorithm 2 illustrates the general cache configuration selection algorithm for preempted tasks of our aggressive approach. This algorithm is called either when a new task with a higher priority than the current executing task arrives or when the current task finishes execution. In the former case, Step 1 uses the executed instruction number (EIN) to calculate the Current Phase (CP) for the preempted task. While in the latter case, this step should be omitted. Step 2 picks the highest priority[3] task Tc from RTL. In the former case, the newly arrived task in inserted into RTL and, obviously, Tc refers to that task. Step 3 checks the schedulability of all the tasks in RTL by iteratively checking whether each task can meet its deadline if all the preceding tasks, including itself, use performance-optimal cache configurations. This process is done in the order of tasks' priority (from highest to lowest) to achieve least discarded tasks. In Step 4, the appropriate cache configuration for Tc is selected based on whether it is safe to use energy-optimal cache configuration. This algorithm runs in time of O(max(p,m)) where p is the partition factor and m is the total number of tasks in RTL.

3.2.5 Impact of Storing Multiple Cache Configurations

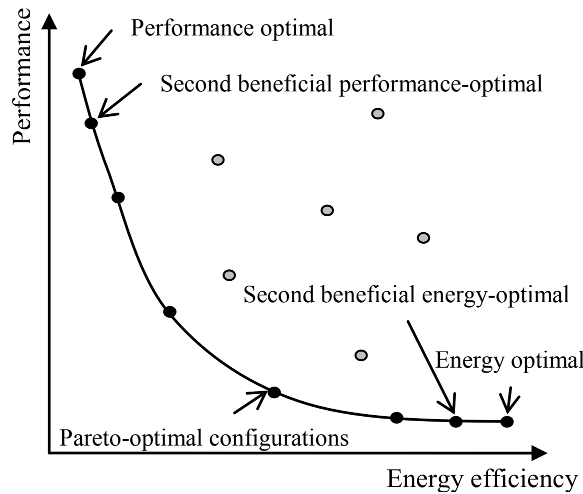
This section investigates the extent at which individual cache configuration candidates are required during scheduling. In the approaches proposed in Section 3.2.4.1 and 3.2.4.2, the scheduler only considers either the energy-optimal cache in the conservative approach, or the energy- and performance-optimal caches in the aggressive approach, for each task phase. As justified by our experiments, we can achieve considerable amount of energy savings at the cost of very low system overheads by just storing these cache configurations in the static profile table. However, there exists other configurations which offer Pareto-optimal tradeoff points. Simply because the energy-optimal cache cannot satisfy a particular task's deadline, it does not mean that there is no cache configuration of that task which can meet the deadline and consume less energy than the

[3] Here the priority means the dynamic scheduling priority decided by EDF.

performance-optimal cache. For example, as described in Algorithm 2, when the scheduler finds that using energy-optimal cache for a task is unsafe, it has no choice but to pick the performance-optimal cache. But if the second energy-optimal cache is also available to the scheduler and is able to meet the time constraint (has higher performance), the scheduler can pick that cache configuration to potentially save more energy. Figure 3-8 (a) illustrates this extension of the profile table.



(a)



(b)

Figure 3-8. (a) Storing multiple optimal cache configurations for each task phase, (b) Second beneficial optimal cache selection on the Pareto-optimal curve

Note that we use the phrase *second beneficial energy-optimal cache* and *second beneficial performance-optimal cache* in Figure 3-8 (a). Figure 3-8 (b) shows how we

choose them. We only consider those cache configurations on the Pareto-optimal curve which have either better energy efficiency or higher performance than other ones. In the extreme case, if we can store all these cache configurations for every task phase in the profile table, the scheduler will be capable of choosing the lowest energy cache configuration that is capable of meeting time constraints of all the existing task in the system. Thus this is a tradeoff between potential energy savings and system overhead in the form of table storage and scheduler complexity. Note that storing information for one more cache configuration in the table will potentially double the area overhead as well as increase power consumption and access time. Section 3.4.2.3 provides experimental results of this approach.

3.3 Design Space Exploration for Two-Level Cache Reconfiguration

So far, we have explored the use of one-level reconfigurable cache in soft real-time systems. It remains a challenge to dynamically tune multi-level caches since the exploration space is prohibitively large[4] . It is because a cross product of two configuration spaces of both cache levels needs to be considered. In this section, we efficiently employ cache reconfiguration in a unified two-level cache hierarchy. We develop four exploration heuristics for our static analysis to effectively decrease the exploration time while keeping the generated profile results beneficial. Our target architecture has separate level one caches – instruction L1 cache (IL1) and data L1 cache (DL1) – as well as a unified level two cache (L2).

[4] After static profiling, the idea of scheduling-aware multi-level cache reconfiguration is similar to single-level cache scenario. The only difference is that now L2 cache configuration needs to be considered with L1 configurations whenever applicable. For example, in phase-based optimal cache selection, C_i represents for three cache configurations (IL1, DL1 and L2). In profile tables, we also needs to store energy- and performance-optimal L2 cache configurations for each phase. Similarly, in Algorithm 1 and 2, the determined L2 cache configurations will be returned.

Tuning a two-level cache faces the difficulty of exploring an enormous configuration space. Here we examine typical exploration parameters of conventional embedded systems. We explore cache sizes of 1KB, 2KB and 4KB, line sizes of 16, 32 and 64 bytes, and direct-mapped, 2- and 4-way set associativity for the L1 cache. We use a 4KB cache architecture proposed in [135] with four banks each of which is 1KB. Since the reconfiguration of associativity is achieved by way concatenation as described in Section 3.1.2, 1KB L1 cache can only be direct-mapped as other three banks are shut down. For the same reason, 2KB cache can only be configured to direct-mapped or 2-way associativity. Therefore, there are 18 (=3+6+9) configuration candidates for L1 caches. Let S_{il1} and S_{dl1} denote the size of exploration space for IL1 cache and DL1 caches, respectively. So we have $S_{il1} = 18$ and $S_{dl1} = 18$. For simplicity, which is also practically true in most scenarios, IL1 and DL1 has the same exploration space which is denoted by S_{l1}. For L2 cache, we choose 8KB, 16KB and 32KB as cache sizes; 32, 64 and 128 bytes as line sizes; 4-, 8- and 16-way set associativity with a 32KB cache architecture composed of four separate banks. Similarly, there are 18 possible configurations ($S_{ul2} = 18$). For comparison, a ***base cache*** hierarchy is chosen which reflects a fixed configuration for all the tasks if cache reconfiguration is not available, consisting of two 2KB, 2-way set associative L1 caches with a 32 byte line size (2KB_2W_32B), and a 16KB, 8-way set associative unified L2 cache with a 64 byte line size (16KB_8W_64B). The remainder of this section describes the proposed exploration techniques.

3.3.1 Exhaustive Exploration

Intuitively, if the two levels of caches can be explored independently, one can easily profile one level at a time while holding the other level to a typical configuration, which will result in a much small exploration space. However, it is not reasonable to claim that the combination of three independently found energy-optimal configurations actually is or ever close to the global optimal one. The two cache levels affect each other's behavior in various ways. For instance, L2 cache's configuration determines the miss penalty of the L1

caches. Also, the number of L2 cache accesses directly depends on the number of L1 cache misses.

Therefore, the only way to obtain the optimal configuration is to search the entire space exhaustively. Since the instruction cache and the data cache could have different configurations, there are 324 (=S_{il1}*S_{dl1}) possible configurations for L1 cache. Addition of the L2 cache increases the design space size to 4752[5] . Moreover, the phase-based static profiling strategy we use makes this number even larger. For a single task, if the partition factor is 4, we have to explore for all four phases, leading to a total of 19008 task phase executions. Obviously it is infeasible. We use the exhaustive method for comparison with the heuristics presented in the following sections.

3.3.2 Same Level One Cache Tuning – SLOT

As discussed above, the design space explosion is resulted from the cross-product of three separate design spaces: IL1, DL1 and L2. The most straightforward optimization is to remove one dimension (i.e., space) so that the total exploration time is drastically reduced while the solution quality is mostly preserved. Our studies show that, for many real applications, the favored (both in terms of energy efficiency and performance) IL1 and DL1 cache configurations are similar to each other (at least in cache size).

Therefore, we propose SLOT – Same Level One Cache Tuning heuristic – during which IL1 and DL1 caches always use the same configuration while exploring with all L2 cache configurations. This method results in a total of 288 configurations – a considerable cut down (94%) of the original quantity (4752), though still not small enough.

3.3.3 Two-Step Tuning – TST

By examining the results generated by SLOT, we find that some very unprofitable L1 cache configurations are also explored 18 (=S_{ul2}) times with L2 cache, resulting in still

[5] Not equal to S_{il1} * S_{dl1} * S_{ul2} because candidates whose L2 cache's line size is smaller than L1 are eliminated

relatively inferior energy efficiency and performance when combined together as the cache hierarchy configuration. These non-beneficial configurations are likely to be discarded. Therefore, just like in single level cache tuning, we only have to consider configurations which offer Pareto-optimal tradeoff points. In other words, for each individual cache, candidates which have both lower performance and higher energy consumption than any other one(s) can be safely eliminated during exploration. Then, the design space which contains the cross-product of all three sets of Pareto-optimal points is explored. Our proposed Two-Step Tuning (TST) heuristic is summarized below:

1. Hold DL1 and L2 as the base cache. Tune IL1 and record all its Pareto-optimal configurations. Let P_{il1} denote the number of recorded IL1 configurations.
2. Hold IL1 and L2 as the base cache. Tune data cache and record all its Pareto-optimal configurations. Let P_{dl1} denote the number of recorded DL1 configurations.
3. Hold both L1 caches as the base cache. Tune L2 and record all its Pareto-optimal configurations. Let P_{ul2} denote the number of recorded L2 configurations.
4. Explore all the combinations from each set of Pareto-optimal configurations recorded in the previous steps and find the energy- and performance- optimal cache hierarchy configurations.

The first three steps explore 54 ($= S_{il1} + S_{dl1} + S_{ul2}$) candidates while the last step explores $P_{il1} * P_{dl1} * P_{ul2}$ candidates. Based on our experimental results, the number of Pareto-optimal configurations varies from application to application but normally around 3 to 5. Therefore, the total exploration space is reduced to 81 - 179 (a reduction of 38% to 72%), though in some worst cases the number could be larger than SLOT's space size (288).

3.3.4 Independent Level One Cache Tuning – ILOT

While different cache levels are dependent on each other, our experimental results demonstrate that instruction cache and data cache are relatively independent. In this study, we fix one's configuration while changing the other's to see whether the varying one

has impact on the fixed one. We observe that the profiling statistics for the instruction cache almost remain identical with different data caches and vice versa. It is mainly due to the fact that access pattern of L1 cache is purely determined by the application's characteristics, and the instruction and data streams are relatively independent from each other. Furthermore, factors affecting the instruction cache's energy consumption as well as performance (such as hit energy, miss energy and miss penalty cycles) have very little dependency on the data cache and vice versa.

This observation offers an opportunity to further reduce the exploration space. We can use the same configurations for IL1 and DL1 while L2 is fixed to base cache to find the "local optimal" configurations for L1 caches. Specifically, throughout the static analysis, we record the energy consumptions and miss cycles of each cache individually. The local energy-optimal IL1 cache is the one with the lowest energy consumption of itself (and same for DL1 cache and L2 cache). The local performance-optimal cache is determined by the number of miss cycles for each cache. ILOT is summarized as below:

1. Hold L2 as the base cache. Explore all L1 cache configurations during which IL1 and DL1 are always configured to the same configuration. Local optimal (both energy- and performance-) configurations for both IL1 and DL1 are recorded.
2. Hold IL1 and DL1 as the energy-optimal configurations found in the last step. Explore all L2 cache configurations and record local energy-optimal L2 cache configuration. The process is repeated for performance-optimal L2 configuration also.
3. The energy- (performance-) optimal configuration for the cache hierarchy is composed of the three local energy- (performance-) optimal caches for each separate cache.

Clearly, the first step simulates 18 ($= S_{l1}$) configurations while the second step requires 36 ($= S_{ul2} * 2$) explorations. If some local optimal IL1 and DL1 configurations happen to be identical, the second step may take less number of explorations. The last

step potentially takes 2 simulations. In total, discarding repeating configurations, ILOT has a exploration space of no more than 54 configurations.

3.3.5 Interlaced Tuning – ILT

Gordon-ross et al. [30] designed a tuning heuristic named TCaT – Two-level Cache Tuning – in a interlaced manner for desktop systems with unified level one and level two caches. In their approach, cache parameters are tuned in the order of their importance to the overall energy consumption, which is cache size followed by line size and finally associativity. TCaT claims to find the configuration with energy consumption close to the optimal one by only exploring tens of candidates. We adapt the strategy used in TCaT and propose ILT – Interlaced Tuning heuristic – which finds both energy- and performance- optimal parameters throughout the exploration. Therefore, as opposed to [30], in each step other than the first, we need to set the already-explored parameters to energy- and performance- optimal ones separately during the exploration of the current parameter. In order to increase the chances of finding optimal L2 cache size, which we found has the highest importance, we combine the exploration of L2 cache's size and associativity together. We sacrifice a certain amount of exploration time for better profiling results. ILT is summarized as below:

1. First, tune by cache size. Hold the IL1's line size, associativity as well as DL1 to the smallest configuration. L2 is set to the base cache. Explore all three instruction cache sizes (1KB, 2KB and 4KB) and find out the energy- and performance-optimal one(s). Same explorations are performed for DL1 cache size. In L2 size exploration, we try all the associativities (4W, 8W and 16W) with each L2 cache size (8KB, 16KB and 32KB) and repeat the process twice to find the energy- and performance-optimal size(s), separately. We set L1 sizes to the energy- (performance-) optimal ones in the corresponding process of finding energy- (performance-) optimal L2 size(s).

2. Next, tune by line size. We set the cache sizes and L2 cache's associativity to the energy- (performance-) optimal ones found in the first step during exploring energy- (performance-) optimal line sizes for each cache (16B, 32B and 64B for L1 caches while 32B, 64B and 128B for L2 cache), respectively. These two tasks are repeated for both L1 caches and L2.
3. Finally, tune by associativity. We set the cache sizes and line sizes to the energy- and performance-optimal ones when we explore for the energy- and performance-optimal associativity (1W, 2W and 4W), respectively. Note that we only explore associativities for L1 caches in this step. During the process of finding DL1's optimal associativities, we already have all the other parameters we need to compute the total numbers of execution cycles that are required in the profile table.

At the beginning of the first step, we do not have any explored parameter so the L1 cache size tuning is done in one-shot for both IL1 an DL1, which lead to 6 (=3+3) configurations. During L2 cache size tuning, there are 9 (=3*3) possible combinations with the associativity and the process has to be done twice for both energy- and performance- optimal L1 cache sizes. Hence, the first step requires to explore 24 (=6+9*2) configurations. Similarly, the second step explores all three lines sizes for each cache separately twice which leads to 18 (=3*2*3) candidates. The final step explores 12 (=3*2*2) configurations since L2 associativity has already been examined in the first step. Therefore, in the worst case, ILT explores 54 (=24+18+12) configurations. However, in most cases, we observe that there are a lot of repetitive configurations throughout the process which we only have to profile once. For example, the L1 configuration 2KB_1W_16B in the second step has already been explored in the first step. Furthermore, all the configurations composed of invalid cache parameter combinations are also discarded. In practice, ILT has a exploration space size of around 35 configurations.

3.4 Experiments

3.4.1 Experiments Setup

To quantify energy savings using the proposed approaches, selected benchmarks from MediaBench [66], MiBench [35] and EEMBC [25] benchmark suites, representing typical tasks that might be present in a soft real-time systems, are examined. These benchmarks are all specially designed for embedded systems and suitable for the cache configuration parameters described in Section 3.3. All applications were executed with the default input sets provided with the benchmarks suites.

We utilized the configurable cache architecture described in [136]. The access latency (e.g., in ns) to read particular data from the cache remains the same when we reconfigure the cache because the clock frequency is fixed determined by the base cache size. The data transfer time during a cache miss is determined by the cache line size as well as the bandwidth between memory levels. In general, larger line sizes will lead to more data transfer cycles thus higher access latencies. This variance, for both L1 and L2 caches, are incorporated in our model considering different miss cycles for cache configurations with various line sizes. We adopt these values from the study in [136]. To obtain cache hit and miss statistics, we used the SimpleScalar toolset [14] to simulate the applications. We assume an in-order issue core with a four-stage pipeline. It supports out-of-order completion but the pipeline is stalled whenever a data hazard is detected. It also supports speculation and a branch predictor with 2-bit saturating counter. We use PISA architecture in our experiments and the compiler is the default little-endian PISA compiler (sslittle-na-sstrix-gcc) which comes with SimpleScalar 3.0, with cc options CFLAGS= -O -I$(srcdir). To populate the static profile tables for each task, we utilize SimpleScalar's external I/O trace files (eio file), checkpointing, and fastforwarding capabilities. This method allows for every benchmark phase to be individually profiled via fastforwarding execution to each potential preemption point. Once we have the profile tables for all the tasks, we use a task scheduler to simulate the system. The scheduler calls another

script which contains the cache configuration selection algorithm (e.g., Algorithm 2) to reconfigure the cache.

For single-level scheduling-aware cache reconfiguration evaluation (Section 3.2), we assume a four-bank cache of base size 4 KB which offers the same configuration space as described in Section 3.3 for L1 caches. For comparison purposes, we define the *base cache* configuration to be a 4 KB, 2-way set associative cache with a 32-byte line size, a reasonably common configuration that meets the needs of the benchmarks studied. The L2 cache is fixed to a 64K unified cache with 4-way associativity and 32B line size. We used partition factors ranging from 4 to 7. Driven by Perl scripts, the design space of 18 L1 cache configurations is exhaustively explored during static analysis to determine the energy-, performance-, and deadline-aware energy-optimal cache configurations for each phase of each benchmark. For multi-level cache reconfiguration, we implemented the cache tuning heuristics using Perl scripts, which are then used to drive SimpleScalar to do the phase-based task profiling.

3.4.2 Results: Single-level SACR

To model sample real-time embedded systems, we created seven different task sets as shown in Table 3-4. In each task set, the three selected benchmarks have comparable dynamic instruction sizes in order to avoid behavioral domination by one relatively large task. For system simulation, task arrival times and deadlines are randomly generated. To achieve effective and fair comparison, we make the system utilization ratio close to the schedulability condition [69]. We examine varying preempting points and average these values so that our results represent a generic degree of scheduling decisions.

3.4.2.1 Energy Saving

We compare the energy consumption for each task set using different schemes: a fixed base cache configuration, the conservative approach, and the aggressive approach. Energy consumption is normalized to the fixed base cache configuration such that value of

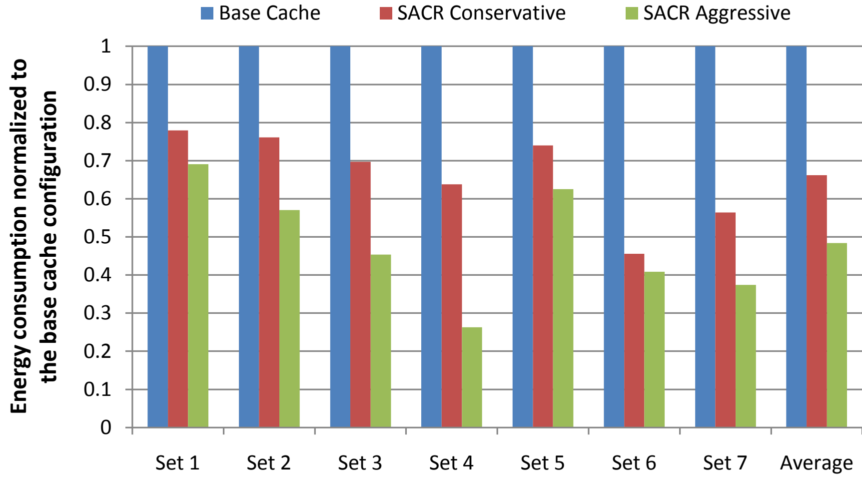
Table 3-4. Benchmark task sets

	Task 1	**Task 2**	**Task 3**
Task Set 1	epic*	pegwit*	rawcaudio*
Task Set 2	cjpeg*	toast*	mpeg2*
Task Set 3	A2TIME01**	AIFFTR01**	AIFIRF01**
Task Set 4	BITMNP01**	IDCTRN01**	RSPEED01**
Task Set 5	djpeg*	rawdaudio*	untoast*
Task Set 6	BaseFP01**	CACHEB01**	IIRFLT01**
Task Set 7	TBLOOK01**	TTSPRK01**	PUWMOD01**

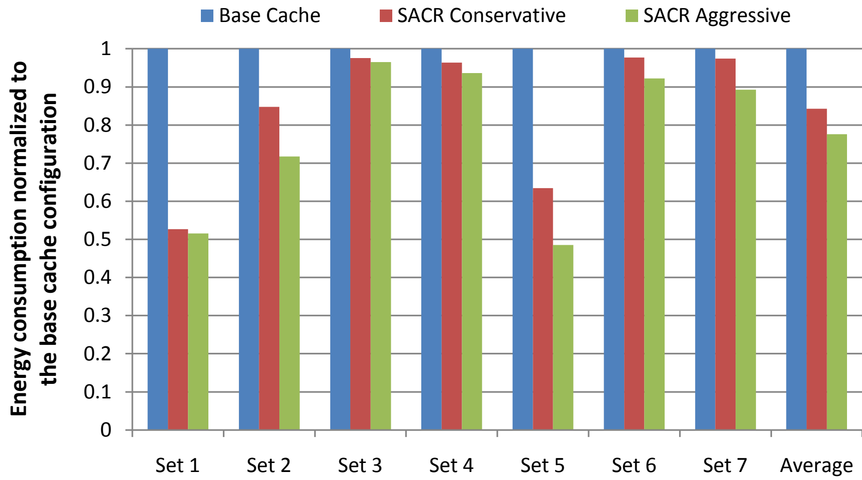
**MediaBench **EEBMC*

1 represents our baseline. Figure 3-9 presents energy savings for the instruction and data cache subsystems. Energy savings in the instruction cache subsystem ranges from 22% to 54% for the conservative approach, while it reaches as high as 74% for the aggressive approach. Energy savings average 33% and 52% for the conservative and aggressive approaches, respectively. In the data cache subsystem, energy saving is generally less than that of the instruction cache subsystem due to less variation in cache configuration requirements. In the data cache subsystem, energy savings range from 15% to 47% for the conservative approach, while it reaches as high as 64% for the aggressive approach, and the average are 16% and 22% for the conservative and aggressive approaches, respectively.

It is worth investigating the insights behind the experimental results: why instruction cache and data cache reveal such different energy savings when executing tasks from different benchmark suites (MediaBench and EEMBC)? Note that we use benchmarks from MediaBench in task sets 1 and 2 while benchmarks from EEMBC in task sets 3 and 4. As shown in Figure 3-9, task set 1, for example, has more energy savings in data cache than in instruction cache using aggressive approach. By looking at the properties of each benchmark in that task set, we found that they have common characteristics in their energy-optimal and performance-optimal cache configurations stored in the profile table. To illustrate this, we sort each task's all cache configurations by their energy consumption as well as performance. Figure 3-10 depicts the layout of each configuration



(a)



(b)

Figure 3-9. Cache subsystem energy consumption normalized to the base cache configuration for each task set (a) Instruction cache (b) Data cache

ranking by its energy and performance for benchmark epic[6] . We can see that in data cache, the chosen energy-optimal cache's performance and performance-optimal cache's energy consumption are relatively much better than in instruction cache. The higher the performance an energy-optimal cache configuration has, the higher the chance that it will

[6] Due to space limit, results for one task in test set 1 is shown here. But other tasks in that set also have the similar pattern. For the same reason, though only results for the entire benchmark is shown there, other phases also show the same property.

be chosen by the scheduler. On the other side, the less energy an performance-optimal cache configuration consumes, the less penalty (extra energy consumption) it has to pay when the scheduler has to choose the performance-optimal one due to tight timing constraints. These two factors explain why for test case 1, data cache reveals more energy savings than instruction cache. For those task sets containing benchmarks from EEBMC, the situation is just the opposite. Task sets 3 and 4 do very well in instruction cache but show very little energy saving in data cache. Figure 3-11 illustrates the reason for this observation. Again, though only A2TIME01 is shown, we found almost all the benchmarks in EEMBC have the same property. In instruction cache, the performance of the energy-optimal cache is very close to that of the performance-optimal one. Similarly, the energy consumption of the performance-optimal cache is very close to that of the energy-optimal one. Interestingly, in many cases they are the same cache configuration, for example, A2TIME01 in Figure 3-11 (a). However, in data cache, the energy-favored caches and performance-favored caches differ tremendously. For this reason, benchmarks from EEMBC are doing extremely bad in data cache.

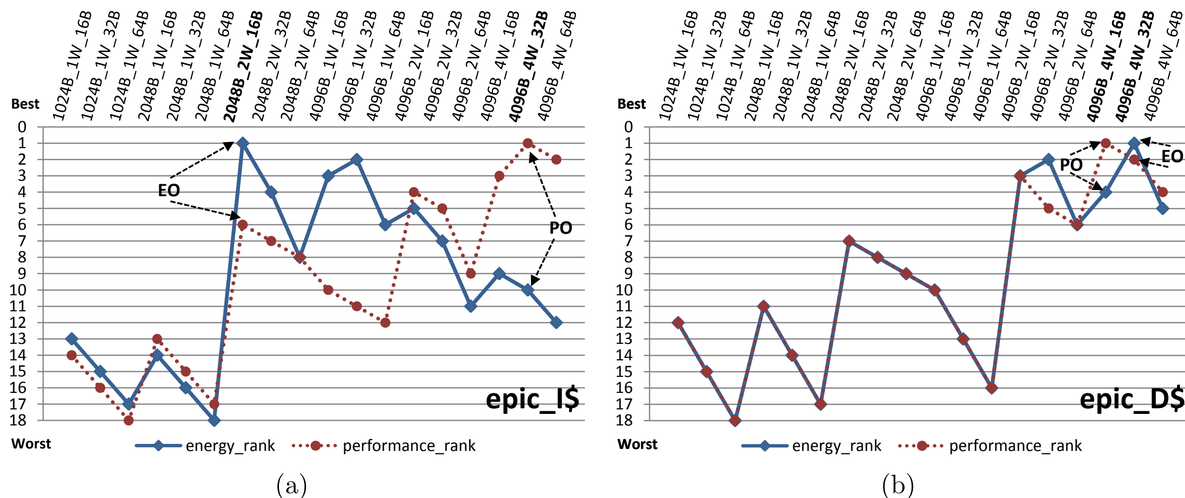
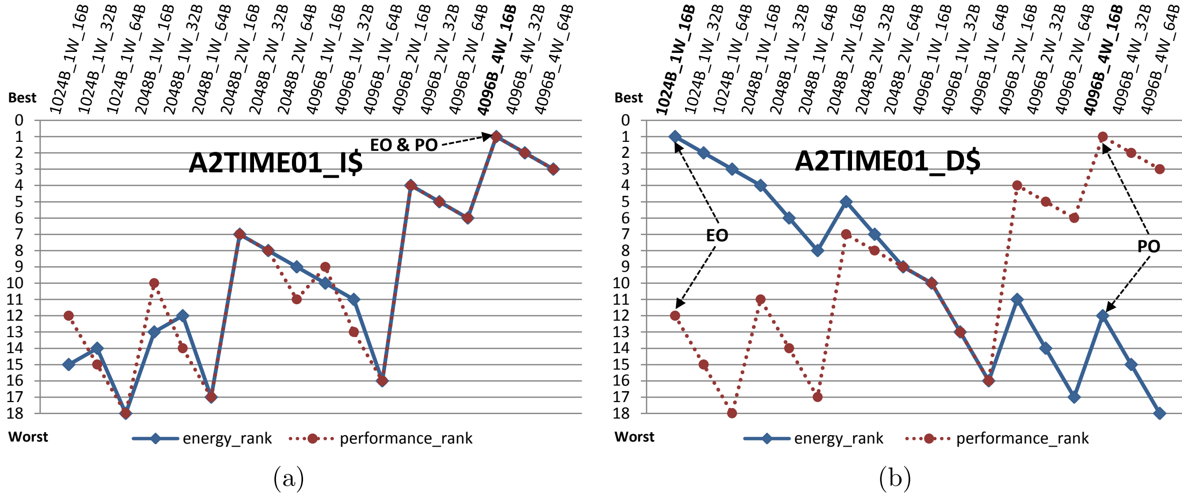


Figure 3-10. Cache configuration candidate's energy and performance rank layout (a) Instruction Cache (epic), (b) Data Cache (epic)

It is also helpful to discuss how miss rate plays its role in the cache model and thus affects the optimal cache configuration variations. Figure 3-12 shows the miss rates for



(a) (b)

Figure 3-11. Cache configuration candidate's energy and performance rank layout (a) Instruction Cache (A2TIME01), (b) Data Cache (A2TIME01)

epic, which explains the insights behind Figure 3-10 showing each data cache configuration behaving similarly in terms of both performance and energy efficiency (e.g. Figure 3-10 (b)) while the instruction caches behaves just the opposite. The reason for $4K_4W_16B$ and $4K_4W_32B$ being superior in both energy and performance is the following. On one hand, the benchmark's data region in the footprint is relatively large and thus the capacity of the data cache is critical. In other words, configurations with smaller sizes cannot satisfy the benchmark's footprint and thus suffer from high miss rates. Therefore, with same associativity, configurations with larger capacity always win over those with smaller size in both performance and energy. On the other hand, as shown in Figure 3-12, 1K configurations with 2-way associativity[7] have similar miss rates as 2K direct-mapped caches while 2K and 2-way associativity configurations have lower miss rates than 4K direct-mapped caches. Therefore, temporal locality of the benchmark which reflects in the number of conflict misses (which further reflect in the desired cache associativity) also play an important role in deciding optimal cache configurations. The code region in the

[7] Although 1K cache with 2-way associativity is not valid in our reconfigurable cache architecture, we include here for illustration purpose only.

footprint is relatively small and thus can be easily satisfied, each configuration will show similar low miss rates thus smaller configurations could win in energy efficiency due to its low power dissipation.

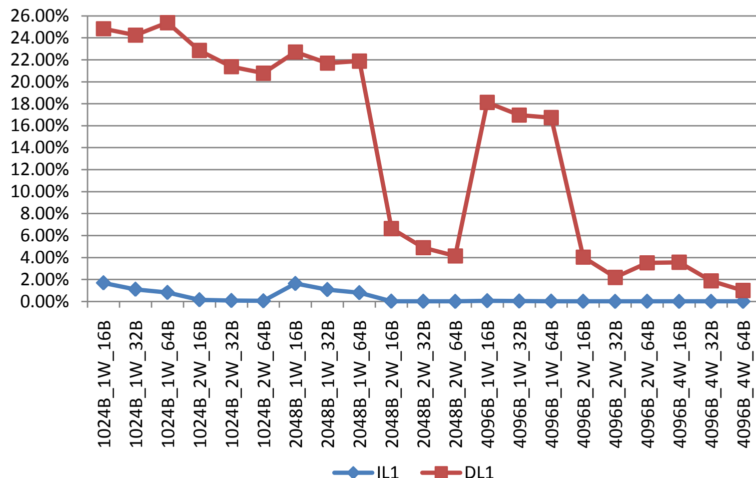


Figure 3-12. Miss rate for epic under different cache configurations.

3.4.2.2 Suitability of Statically Determined Configurations

System's performance variations when using our approaches are shown in Table 3-5 and Table 3-6. We keep tracking of each task's performance during the system execution and find the percentage of those jobs of that task whose performance are higher (and lower but deadlines are met) using the selected cache configuration compared to the base cache. As discussed in Section 3.2.4.1, the cache configuration selected by our approach (nearest-neighbor nature) may possibly be inefficient in performance for the execution period between the actual preemption points. The percentage deadline misses are also provided for each task to evaluate the system service level. Though lower performance jobs do potentially have impact on the system performance, they are not harmful since no task deadline is missed. As the results show, our approach achieves significant energy savings at a very low cost of small amount of task deadline misses which are acceptable in soft real-time systems. For example, among *epic*'s all executions (jobs), 75% of them took shorter time using the cache configurations selected by conservative approach than using base cache while 21% of them took longer time but still met the time constraints.

Only 4% of all its jobs actually miss their deadlines. As Table 3-5 demonstrates, our conservative approach leads to very minor deadline misses (0 - 4%). Our aggressive approach can generate drastic reduction in energy requirements with slight higher deadlines misses (1% - 18%).

Table 3-5. Task performance variations for conservative approach

Task Sets	Tasks	Higher performance jobs	Lower performance jobs	Deadline misses
1	epic	75%	21%	4%
	pegwit	99%	1%	0%
	rawcaudio	94%	3%	3%
2	cjpeg	94%	5%	1%
	toast	89%	4%	7%
	mpeg2	94%	2%	4%
3	A2TIME01	98%	2%	0%
	AIFFTR01	82%	15%	3%
	AIFIRF01	99%	1%	0%
4	BITMNP01	100%	0%	0%
	IDCTRN01	96%	2%	2%
	RSPEED01	99%	1%	0%

Table 3-6. Task performance variations for aggressive approach

Task Sets	Tasks	Higher performance jobs	Lower performance jobs	Deadline misses
1	epic	63%	29%	8%
	pegwit	89%	10%	1%
	rawcaudio	76%	12%	12%
2	cjpeg	90%	6%	4%
	toast	72%	16%	12%
	mpeg2	75%	17%	8%
3	A2TIME01	94%	2%	3%
	AIFFTR01	52%	30%	18%
	AIFIRF01	97%	2%	1%
4	BITMNP01	62%	27%	11%
	IDCTRN01	94%	3%	3%
	RSPEED01	91%	2%	7%

To show how early/late in the execution the deadlines are missed, for each low-priority job that is discarded, we collected its current phase (CP) as defined in Section 3.2.4.2, as shown in Table 3-7. In other words, among all the jobs that missed their deadlines (e.g. 4% of all jobs), different jobs are dropped at different stages of execution (CP). For example, in case of epic, among the 8% of jobs that are dropped, 23% of them have executed over one-fourth (CP = 1), 54% of them have executed over half (CP = 2) and 23% of them are over three-fourth (CP = 3).

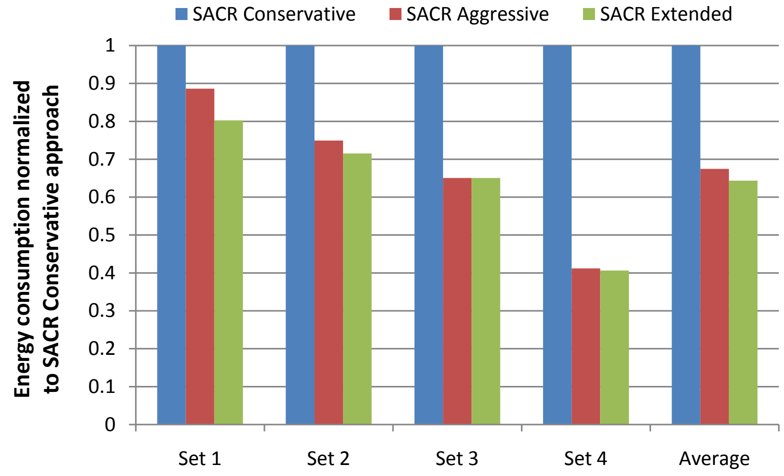
Table 3-7. Current phases of deadline violated tasks when they are discarded.

Task Sets	**Tasks**	**CP = 1**	**CP = 2**	**CP = 3**
1	epic	23%	54%	23%
	pegwit	0%	0%	100%
	rawcaudio	33%	34%	33%
2	cjpeg	14%	34%	52%
	toast	10%	25%	65%
	mpeg2	18%	32%	50%

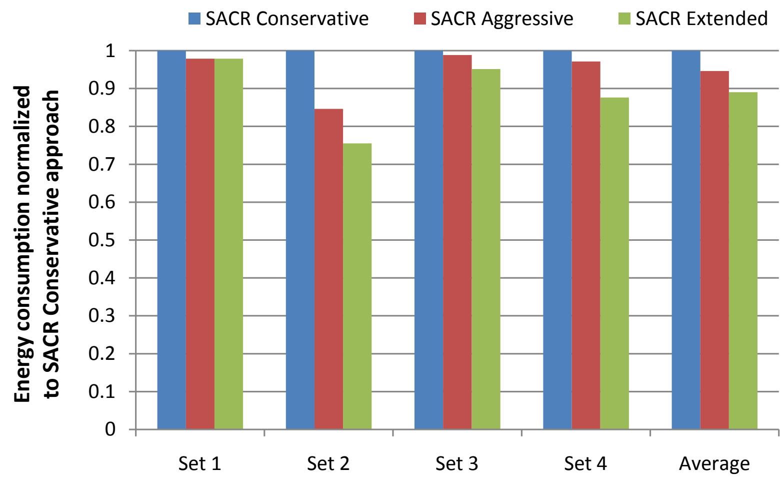
3.4.2.3 Impact of Storing Multiple Cache Configurations

As discussed in Section 3.2.5, storing multiple beneficial cache configurations may lead to more energy savings. We explore the effect of using extended profile table by running task set 1 - 4 in Table 3-4. The profile table size is doubled to accommodate the second beneficial energy- and performance-optimal cache configuration. Algorithm 2 is modified to be aware of this extension. We call this method *Extended* approach and Figure 3-13 shows its energy consumption compared to the conservative and aggressive approaches. On average, the extended approach achieves 4.6% more energy savings than aggressive approach in instruction caches while 5.9% more in data caches. In some cases, like set 3 in instruction cache and set 1 in data cache, no extra energy saving is observed due to lack of beneficial cache configurations.

As already discussed in Section 3.2.5, extended profile table will cause exponential increase in system overheads. Energy overhead of the profile table can be safely ignored because it only accounts for a very less proportion of the gained energy savings. However,



(a)



(b)

Figure 3-13. The effect of using extended profile table: cache subsystem energy consumption normalized to conservative approach for each task set (a) Instruction cache (b) Data cache

increase in area and access time of the table affect the feasibility of applying this extended approach. When the number of different tasks is relatively small such that system overhead is not serious, extended approach is favorable than other two approaches. Since it is common to have large number of tasks in the system, applying extended approach may not be a good idea in these cases because the profile table's area could exceed the chip area constraints and increased access time may impact the system's critical path. In

extreme cases, it may lead to longer clock cycle length and lower system frequency, which should obviously be avoided.

3.4.2.4 Analysis of Input Variations

Program's cache behavior, especially the data cache, can vary when being fed with different inputs. Essentially, input varies in its size, structure, and its contents. For example, different inputs may drastically affect the program's dynamic execution path (such as number of loop iterations), thus both energy- and performance-optimal caches may differ from what are stored in the profile table.

Obviously, it is impossible to exhaustively explore all possible inputs. Energy-aware task scheduling techniques face the same problem. In real-time systems, scheduler should be fed with the task set information which includes task's *execution time* (in cycles). The potential solutions include use of i) fixed input set (execution time is known beforehand) [42] [94], ii) Worst Case Execution Time (WCET) [137] [51] [97] [101] and iii) probabilistic execution time distribution [83] [139] [40].

It is worth exploring how varying input would impact each task's cache behavior. In our experiments, we examine inputs with different sizes and observe the variation of optimal cache configurations. For the instruction cache, the energy optimal cache configuration parameters (cache size, line size and set associativity) reduce as the input size decreases. Results are similar for the data cache. The performance optimal instruction cache configuration's line size reduces as input size decreases, but cache size and associativity remain the same. However, in this case, the data cache shows non-deterministic behavior. The reason for such kind of variations in instruction cache is the size of critical data processing code sections which accounts for 90% of the time (loops etc.) may be a comparatively small segment of the entire program due to the 90/10 law. Since critical data processing code sections (instruction cache working set) remains in the cache, the line size tends to be smaller in order to reduce the time spent on cache misses, and thus static energy consumed. For the data cache, as the input size increases,

spatial locality is more critical than temporal locality, thus, the cache size nearly remains the same, or even decreases, but line size increases. It is important to note that drastic changes in input size is not usual in real-time systems. We also studied the impact of changing input pattern on our approach. We observed that a change in input pattern (data structure and the absolute values change but not the size) shows a minor impact on the cache behavior. Both energy and performance optimal cache configurations show very little variation.

Here are the experimental results that support our arguments. We examined cjpeg benchmark from MediaBench. In the first set of experiments, we selected six differently sized input image files (a.ppm, b.ppm, c.ppm, d.ppm, e.ppm, f.ppm) and found that the energy/performance optimal cache configurations for both instruction and data caches, with partition factor of 4, as shown in Table 3-8. In the second experiment, we selected two similarly sized images files (man.ppm and woman.ppm) with different content and exploited the cache behavior under partition factor of 4, 5 and 6. As shown in Table 3-9, there is very little variation in terms of the optimal cache configuration selection for the two inputs. Therefore, our approach is applicable when the input for each task is known during design time so that it can be statically profiled. Our approach is also applicable when there are changes in input pattern. This is a realistic assumption for real-time systems.

3.4.2.5 Hardware Overhead

This section describes the overhead of implementing the profile table in hardware. The profile table is stored in SRAM and accessed by the cache tuner to fetch the cache configuration information. The size of the table depends on the number of tasks in the system and the partition factor used. For conservative approach, each table entry consists of five bits since the configurable cache architecture used in this study offers 18 possible cache configurations. We have implemented the profile table using Verilog HDL and

Table 3-8. Input variation exploration.

a.ppm: Size of input: 8431 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
4096B_2W_16B	4096B_4W_16B	4096B_2W_16B	4096B_4W_16B
4096B_2W_16B	4096B_4W_16B	4096B_2W_16B	4096B_4W_16B
4096B_2W_16B	4096B_4W_16B	4096B_2W_16B	4096B_4W_16B
b.ppm: Size of input: 101484 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
4096B_2W_16B	4096B_4W_16B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
c.ppm: Size of input: 306915 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
d.ppm: Size of input: 530895 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	4096B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
e.ppm: Size of input: 1476015 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_16B	4096B_2W_16B	4096B_4W_16B
2048B_2W_16B	4096B_2W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_2W_64B	2048B_2W_32B	4096B_4W_16B
f.ppm: Size of input: 3832336 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_2W_64B	4096B_2W_16B	4096B_4W_16B
2048B_2W_16B	2048B_2W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	2048B_2W_64B	2048B_2W_32B	4096B_4W_16B

synthesized using Synopsis Design Compiler with TSMC 0.18 cell library. We estimate a table lookup frequency of once per three million nanoseconds during dynamic power computation, which means that there is a table lookup every one million instructions using a 500 MHz CPU with an average CPI of 1.5. It is clearly an overestimation (which is safe) since the benchmarks we used have around 10 to 200 million dynamic instructions. Table 3-10 illustrates our results. Each row in the table indicates the area, dynamic power, leakage power, and critical path length for profile table with different sizes. We also calculate overhead using 65nm technology as shown in Table 3-11. We observed that

Table 3-9. Input pattern changes.

man.ppm: Size of input: 336165 bytes			
Partition factor p = 4			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
Partition factor p = 5			
EO_icache	PO_icache	EO_dcache	PO_dcache
4096B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
Partition factor p = 6			
EO_icache	PO_icache	EO_dcache	PO_dcache
4096B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_2W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
woman.ppm: Size of input: 312999 bytes			
Partition factor p = 4			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
Partition factor p = 5			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
Partition factor p = 6			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_2W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B

on average for each task set, the energy overhead of our approach only account for less than 0.02% (450 nJ compared to 2825563 nJ) of the total energy savings. Admittedly,

aggressive approach requires more bits per lookup table entry (74 bits[8]). However, Table 3-10 and 3-11 illustrate that the power dissipation is about linearly proportional to the table size. Therefore, even if the table entry size is increased by 15 times (5 bits to 74 bits), the total energy overhead is no more than 0.3% of the total energy savings. Therefore, we can safely conclude that the overhead for profile tables are negligible compared to the energy saving for both conservative and aggressive approaches.

Table 3-10. Overhead of different lookup tables (180nm technology)

Table size (# of entries)	**Area (μm^2)**	**Dynamic Power (μW)**	**Leakage Power (μW)**	**Critical Path Length**
64	61416	38.13	114.37	0.91
128	121200	84.25	224.90	0.91
256	244520	187.68	461.30	1.08
512	483416	327.90	904.70	1.20

Table 3-11. Overhead of different lookup tables (65nm technology)

Table size (# of entries)	**Area (μm^2)**	**Dynamic Power (μW)**	**Leakage Power (μW)**
64	6756	12.23	154.52
128	13332	27.02	303.86
256	26897	60.19	623.25
512	53176	105.16	1222.32

3.4.3 Results: Multi-level SACR

To evaluate our exploration heuristics and scheduling algorithm, we selected six benchmarks from each of the following benchmark suits: MediaBench [66] (*cjpeg, epic, pegwit, rawcaudio, mpeg2, toast*), MiBench [35] (*CRC32, dijkstra, FFT, pktflow, qsort, rijndael, susan*) and EEMBC [25] (*A2TIME01, AIFFTR01, AIFIRF01, BITMNP01,*

[8] 74 bits are needed to store both energy- and performance-optimal cache configurations (5 + 5 bits) as well as the corresponding execution times (32 + 32 bits).

IDCTRN01, RSPEED01). Table 3-12 shows our seven task sets, each of which consists of six selected benchmarks. Task set 1 consists of benchmarks from MediaBench, set 2 from MiBench, set 3 from EEMBC and set 4 - 7 are mixtures from all threes suites. In order to avoid the situation where one or two tasks dominate the total energy consumption, tasks in each set are chosen to have comparable sizes. All the tasks are executed with the default input sets provided with the benchmark suites.

Table 3-12. Task sets consisting of real benchmarks.

Sets	**Tasks**
Set 1	cjpeg, epic, pegwit, rawcaudio, mpeg2, toast
Set 2	CRC32, dijkstra, FFT, pktflow, qsort, rijndael
Set 3	A2TIME01, AIFFTR01, AIFIRF01, BITMNP01, IDCTRN01, RSPEED01
Set 4	cjpeg, pegwit, qsort, susan, A2TIME01, IDCTRN01
Set 5	epic, rawcaudio, dijkstra, CRC32, AIFFTR01, BITMNP01
Set 6	mpeg2, toast, pktflow, rijndael, AIFIRF01, RSPEED01
Set 7	pegwit, mpeg2, qsort, FFT, BITMNP01, IDCTRN01

3.4.3.1 Optimal Cache Configuration Selection

First we evaluate our proposed design space exploration heuristics by comparing the energy- (performance-) optimal cache configurations found using each heuristic to the exhaustive approach. This comparison directly reflects the effectiveness of each heuristic (the closer to the exhaustive approach the better). Since these design space exploration results are used to construct the profile table, it will have impact on the scheduling-aware reconfiguration algorithm.

Figure 3-14 and 3-15 show the heuristic searching results for selected benchmarks. From Figure 3-14, we can observe that, for most of the time, all four heuristics behaves well in finding energy-optimal cache hierarchy configurations. For example, for benchmark *dijkstra*, *cjpeg*, *rawcaudio* and *RSPEED01*, all four heuristics are able to find configurations which are very close to the optimal. However, in certain cases, some heuristics may lead to inferior exploration results. For example, both ILOT and ILT do not work well for *pegwit*.

Generally speaking, with respective to energy consumption, SLOT and TST behave consistently well among all benchmarks. ILOT behaves very close to TST, sometimes even better (e.g., *cjpeg*, *AIFIRF01*), but could be inferior in other cases. ILT, though having the smallest exploration space and thus being fastest, is only able to find the optimal configurations with the quality 30% away from the optimal on average.

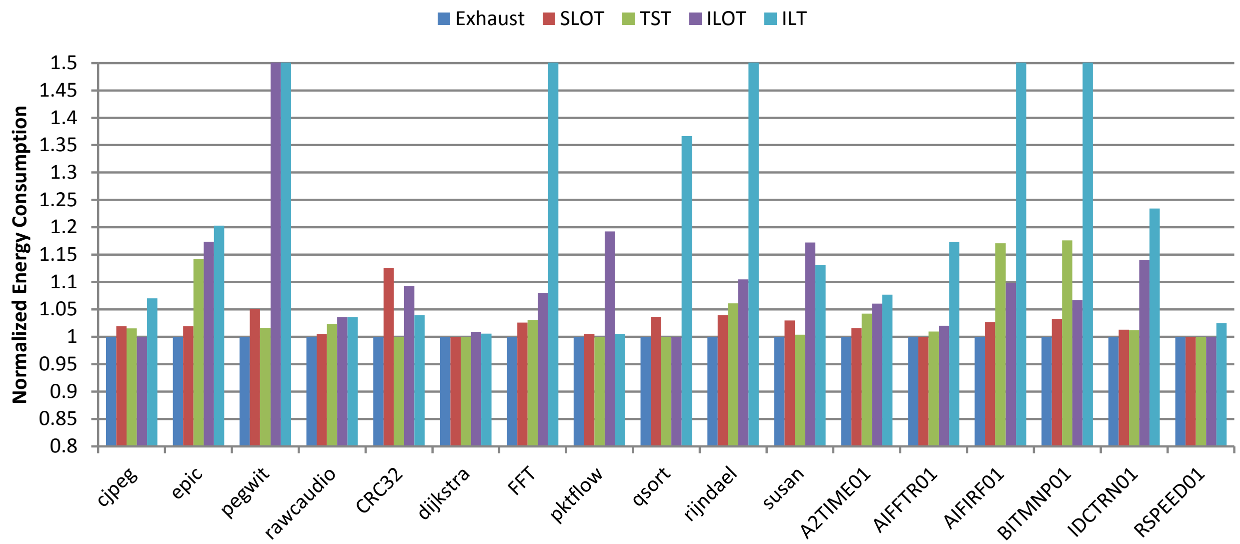


Figure 3-14. Normalized energy consumption of the searched energy-optimal cache configuration using heuristics.

Figure 3-15 shows the exploration results in terms of performance. In other words, the execution time of the performance-optimal cache configuration found by each heuristic is compared with the exhaustive search. It can be observed that SLOT and TST are able to consistently find the actual performance-optimal configurations or at least very close ones. On the other hand, although behaves very well in terms of energy consumption, ILOT is not good at finding the performance-optimal configuration for a number of benchmarks. In this aspect, ILT outperforms ILOT.

3.4.3.2 Energy Saving

We quantify the cache subsystem energy savings using our approach by comparing to the base cache scenario. We use five cache exploration methods – exhaustive, SLOT, TST, ILOT and ILT – to generate profile tables for all the task sets. Figure 3-16 presents the

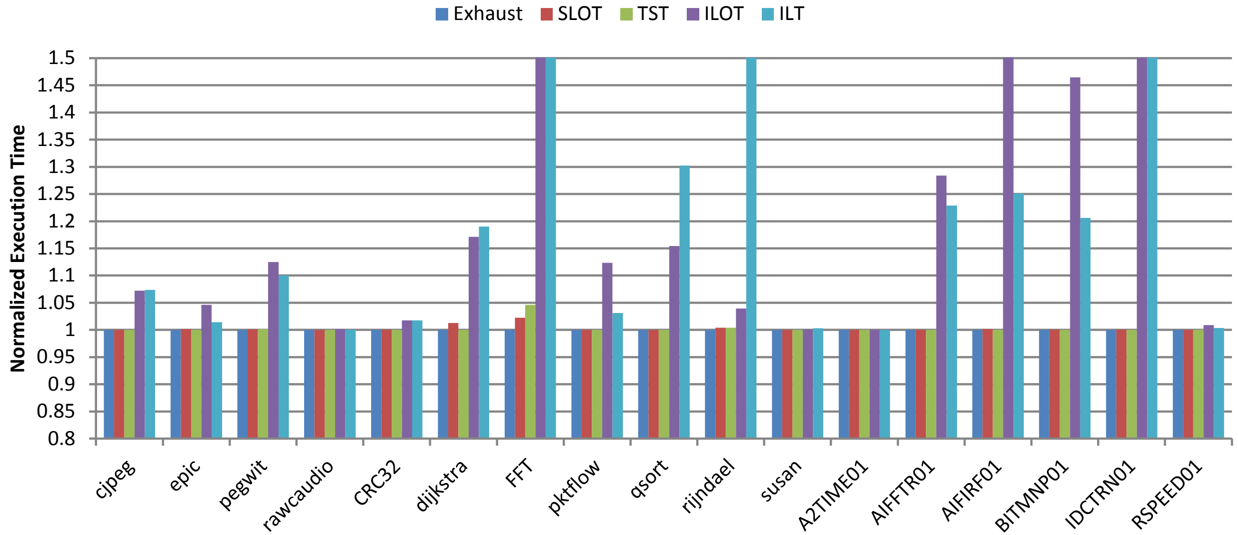


Figure 3-15. Normalized execution time of the searched performance-optimal cache configuration using heuristics.

total cache hierarchy energy consumption normalized to the base cache for all the seven task sets using each exploration technique. As expected, exhaustive exploration generated the highest energy saving (58% on average). SLOT achieves 56% average energy saving which is comparable to the exhaustive approach. TST outperforms SLOT in some task sets but on average saves 52% of the energy consumption. While ILOT and ILT perform the worst, we can still achieve 46% and 40% of energy savings, respectively. Figure 3-16 also shows the relative comparison of each heuristic. On an average, SLOT, TST, ILOT and ILT make the system consume 2.8%, 9.1%, 25.6% and 43.1% more energy than the exhaustive method.

3.4.3.3 Insights behind Results

It is helpful to examine some insights behind the results shown above. SLOT simply discards the flexibility and benefit of running IL1 and DL1 cache separately. Therefore, when optimal configurations for IL1 and DL1 are different, SLOT will have to suffer from decreased energy efficiency and/or performance in either IL1 or DL1. TST only considers Pareto-optimal configurations at the cost of losing the chance of finding more efficient cache combinations which actually consists of non-beneficial ones. Specifically, when

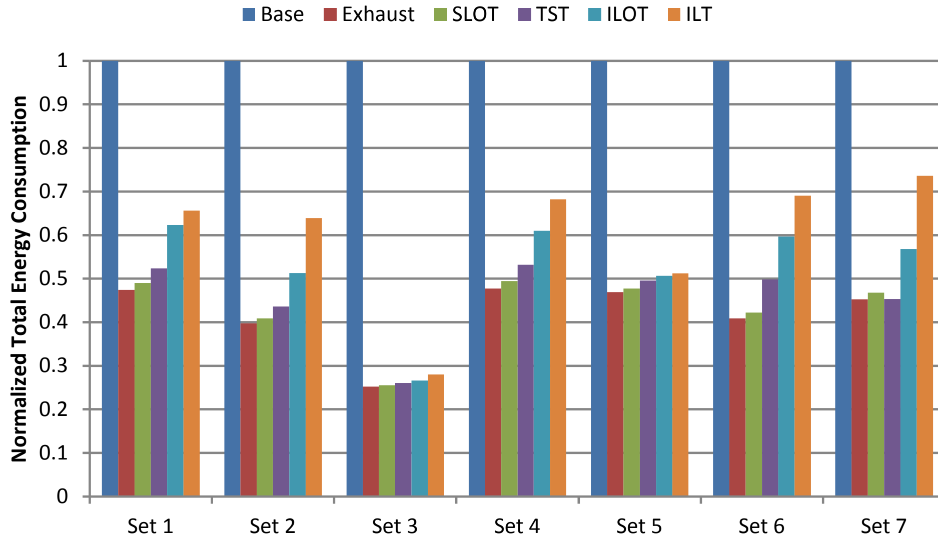


Figure 3-16. Cache hierarchy energy consumption using four heuristics.

searching for the Pareto-optimal points for each cache, the other two caches are fixed to the base case. In other words, it is assumed that the Pareto-optimal configuration set for each individual cache is independent of the other cache's configuration. However, the assumption does not always hold. One of the reason is that a less energy efficient (due to oversize) L1 cache may cause less accesses to L2 cache. Hence an appropriate L2 cache may make this non-beneficial L1 cache overall better. The reason for ILOT not finding the optimal configurations is that, although relatively independent from each other, IL1 and DL1 both have impact on the L2 cache which has effect back on L1 caches. So they are essentially indirectly dependent on each other through the L2 cache. Furthermore, varying one of them, say DL1, will lead to different total execution time and thus the static power consumption of the other (IL1) is also going to change. Therefore, although miss rate is unaffected, IL1 and DL1 do have impact on each other in terms of energy consumption as well as performance. ILT behaves worst due to the fact that it could miss the optimal parameter easily when exploring with other unknown but fixed parameters.

3.4.3.4 Exploration Efficiency

The four heuristics, though exhibits less energy savings, are much more efficient than exhaustive method in the static profiling stage. Table 3-13 presents the total number of

cache configurations explored by each exploration heurstics[9] for each benchmark. Our experience is that it normally takes days to profile a task using exhaustive method while a few minutes if ILT is employed. For example, exhaustive exploration of all configurations for *qsort* takes about 5 days and 16 hours while only 44 minutes are required for ILT heuristic. Designers can decide which heuristic to use based on the profiling time they have and the overall energy savings.

Table 3-13. Cache hierarchy configuration explored using different exploration methods.

	Exhaust	**SLOT**	**TST**	**ILOT**	**ILT**
cjpeg	4752	288	192	54	31
epic	4752	288	70	54	31
pegwit	4752	288	128	36	36
rawcaudio	4752	288	452	54	33
CRC32	4752	288	318	54	33
dijkstra	4752	288	92	54	32
FFT	4752	288	165	52	36
pktflow	4752	288	114	54	37
qsort	4752	288	116	54	37
rijndael	4752	288	58	54	31
susan	4752	288	352	54	33
A2TIME01	4752	288	92	54	34
AIFFTR01	4752	288	120	54	31
AIFIRF01	4752	288	79	54	38
BITMNP01	4752	288	68	54	38
IDCTRN01	4752	288	84	54	36
RSPEED01	4752	288	116	53	37

3.5 Summary

Dynamic cache reconfiguration is a promising approach to improve both energy consumption and overall performance in embedded systems. This chapter presented a novel scheduling-aware dynamic cache reconfiguration technique for soft real-time systems.

[9] For simplicity, these numbers only count for the task on the whole in each set but not for every phase.

This methodology employs an ideal combination of static analysis and dynamic tuning of cache parameters with minor or no impact on timing constraints. We also presented a novel methodology for tuning two-level configurable cache hierarchy in soft real-time systems. Four cache exploration heuristics, which greatly improve the static analysis efficiency, are designed and compared with the exhaustive method. Experimental results demonstrated a 50% reduction on average in the overall energy consumption of the single-level cache subsystems and up to 40 - 58% of the multi-level cache hierarchy in soft real-time embedded systems.

CHAPTER 4
ENERGY-AWARE SCHEDULING WITH DYNAMIC VOLTAGE SCALING

Dynamic voltage scaling (DVS) [38] is widely acknowledged as one of the most effective processor energy saving techniques. The reason behind its capability to save energy is that linear reduction in the supply voltage leads to approximately linear slow down of performance while the power can be decreased quadratically. Many general as well as specific-purpose processors nowadays support DVS [74] [75] [54] with multiple available voltage levels. Figure 4-1 shows how power consumption and clock cycle length vary on the Crusoe processor in 70nm technology. The processor supply voltage (V_{dd}) is varied from 1V to 0.5V in one step of 0.05V. We can observe that both dynamic and static power reduce along with the voltage while the operating frequency drops at a lower pace (down to 0.60V in this case). Therefore, it will be beneficial to reduce the supply voltage whenever possible to achieve energy savings. Processor idle time (i.e., time slack) also provides a unique opportunity to reduce the overall energy consumption by putting the system into a low-power sleep mode using Dynamic Power Management (DPM) [8]. Research has shown that DVS should be used as the primary low-power technique for processor [52] while DPM could be beneficial after applying DVS.

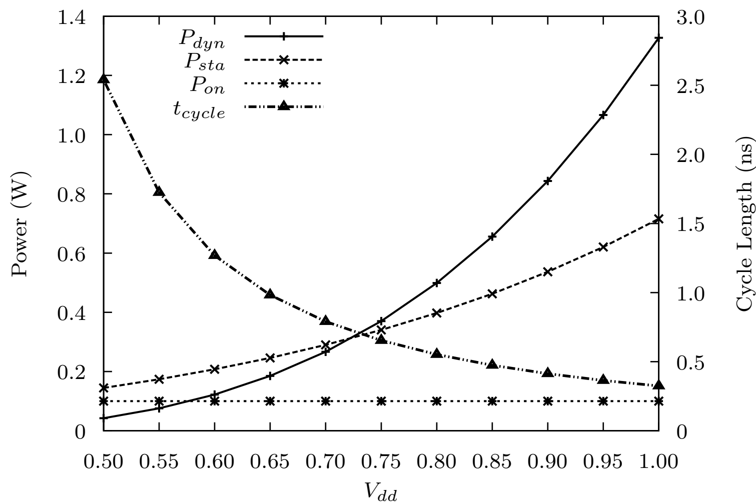


Figure 4-1. Power consumption and clock cycle length of Crusoe processor (P_{dyn}, P_{sta} and P_{on} denote dynamic power, leakage power and the intrinsic power required to keep the processor on, respectively).

Employing DVS in real-time multitasking systems involves assigning different processor voltage levels to tasks based on characteristics of the task set as well as the target system. In this chapter, we present novel algorithms for energy-aware processor voltage scaling and task scheduling in real-time systems. The proposed research mainly focuses on hard real-time systems with preemptive task sets. Generally, the objective is to minimize energy consumption without violating any task timing constraint. We first present a novel DVS scheme – PreDVS – which assigns multiple voltage levels to each task instance based on preemptive scheduling. PreDVS targets on static slack allocation and is called during design time. Our approach is based on an approximation scheme that can ensure the solution quality to be within a specified boundary of the optimal. Next, we describe our dynamic slack reclamation algorithm (DSR) which runs at runtime to adjust voltage assignments and reschedule tasks for further energy savings. The two methodologies proposed in this chapter can be employed together to achieve maximum energy optimizations.

The rest of this chapter is organized as follows. Section 4.1 summarizes existing research works on DVS. Section 4.2 describes PreDVS in details. Section 4.3 presents the dynamic slack reclamation algorithm. Experimental results are presented in Section 4.3. Finally Section 4.5 concludes this chapter.

4.1 Related Work

A great deal of research has been carried out on dynamic voltage scaling in real-time systems. Early researches [130] [39] [102] [4] assumed an ideal processor with continuously variable voltage/frequency (i.e. infinite number of voltage levels) which is unpractical in real systems. Ishihara et al. [45] relieved this limitation by using two available neighboring voltage levels above and below the desired one for only a single task. Kwon et al. [64] combined the approaches from [130] and [45] to give optimal voltage schedule for given task sets which allows voltage scaling at specific calculated points during task execution. However, it suffers from the large number of voltage transitions (nearly double the

number needed by inter-task DVS and PreDVS). Furthermore, monitoring whether the pre-determined optimal scaling point has been reached at runtime also requires certain overhead as well as hardware complexity. Another way to convert the solution for ideal processors to the practical discrete voltage scenario is always choosing the next available voltage level above the desired one. Aydin et al. [6] and Quan et al. [89] adopted this strategy for dynamic-priority periodic tasks and fixed-priority mixed tasks, respectively. However, it is very inefficient and could result in 15 - 17% more energy consumption [6].

Most of the recent works focused on inter-task DVS, which uses a single voltage level throughout each task's execution, using heuristic algorithms [128] [78] [76] [110] [139] [142] [79] [50] [44] while others are based on approximation algorithms [16] [140] [137]. Mejia-Alvarez et al. [76] first proposed DVS scheduling problem as Multiple-Choice Knapsack Problem (MCKP) and presented a greedy heuristic algorithm. Swaminathan et al. [110] modeled and solved the problem using network flow techniques to assign a single voltage level to every job. Zhong et al. [139] considered voltage scaling for sporadic task sets whose characteristics are not known a priori and provided statistical real-time guarantees. Zhuo et al. [142] employed DVS to achieve system-wide (with a processor supporting continuous voltage levels) energy saving by combining static speed setting and preemption minimization together. Irani et al. [44] used both DVS and DPM and presented heuristics that give solutions within constant factor of the optimal algorithm. Chen et al. [16] formulated inter-task scaling problem as a Subset Sum problem and proposed an approximation algorithm with performance guarantee. Zhong et al. [140] considered processor and peripheral devices energy consumptions at the same time. They proposed pseudo-polynomial time optimal algorithm and fully polynomial approximation algorithm for both periodic and sporadic task sets. Zhang et al. [137] solved the same problem with an approximation algorithm which has lower complexity than the one in [140].

Intra-task DVS, which assigns multiple voltage levels to each task instance based on runtime information, also gained significant research interest [45] [141] [83] [127] [97] [126] [100]. Ishihara et al. [45] formulated the intra-task DVS problem as an integer linear programming problem but no more than two voltage levels can be assigned to each task. Seo et al. [97] presented a comprehensive technique based on execution profile of each task to determine the voltage scheduling of each individual block. However, only minimum average energy consumption is guaranteed. Zhu et al. [141] considered task execution time variation and introduced a DVS scheme with feedback control which handles dynamic workloads. Xie et al. [126] took switching costs into consideration and proposed exponential-time optimal algorithm as well as a linear-time heuristic with various scaling granularity. Shin et al. [100] analyzed the program profile and data-flow to improve the estimation of remaining execution cycles and optimize the voltage scaling points.

Dynamic slack reclamation techniques are proposed in [5] [84] [62] [49] [48]. Aydin et al. [5] presented an online algorithm for utilizing unused task running time and a more aggressive speculative mechanism based on expected workload. Pillai et al. [84] adjusted the processor voltage on each job arrival based on the current system utilization or future task's WCET. Kim et al. [62] considered this problem on an ideal continuously scalable processor. Jejurikar et al. [49] presented an algorithm for non-preemptive task sets. Leakage-aware dynamic slack reclamation technique is proposed in [48]. It is based on the theorem proved in [5] that every task instance can fully reclaim slacks with higher or equal scheduling priority.

4.2 PreDVS: Preemptive Dynamic Voltage Scaling

4.2.1 Overview

In this section, we present a novel voltage scaling technique which generates a voltage assignment based on the preemptive schedule of the target task set. We develop a fully polynomial approximation scheme which can guarantee to give solutions within specified quality bounds. We also propose two efficient heuristics which can lead to

comparable energy savings in certain cases. Our approach, named PreDVS, differs from inter-task and intra-task DVS as illustrated in Figure 4-2. Specifically, PreDVS differs from existing inter-task scaling techniques in that inter-task DVS assigns only single profitable voltage level to all instances of each task, whereas PreDVS can adjust processor voltage level multiple times throughout each task instance's execution, without introducing any extra scaling overhead, to potentially achieve more energy savings. In Figure 4-2, if the deadline of task τ_1 is 7, inter-task DVS cannot further lower any task's voltage level otherwise deadline will be missed since it requires reduction of voltage for all task instances. However, PreDVS is able to further reduce the energy consumption by lowering the voltage level for the first segment of τ_1 from 0.75 to 0.50. Note that, although only an illustrative example is given here, the number of segments for one task instance could be excessive due to preemptions in real systems (e.g., tasks with long execution time and period are preempted by short tasks many times). PreDVS also differs from existing intra-task DVS techniques [97] [100] [83] in the following ways. Existing intra-task scaling methods assume that static slack allocation has already been done. They only consider one task instance (local optimization) and focus on exploiting dynamic time slacks generated at runtime due to early finished task execution. They require excessive analysis, runtime tracking and modification of the task source code, which makes them difficult to be implemented and thus not always feasible in real systems [110]. Furthermore, they normally result in large number of additional voltage switching points and most of them assume continuous voltage levels. PreDVS aims at static slack exploitation and is carried out during design time. In fact, our technique is complementary to existing intra-task DVS techniques. Any intra-task scaling can be applied after PreDVS to further reduce energy consumption at runtime.

PreDVS considers the problem globally and find a voltage assignment for all task instances so that the total energy consumption can be minimized while no deadline is violated. Off-line analysis (i.e., static slack exploitation) is of great importance in

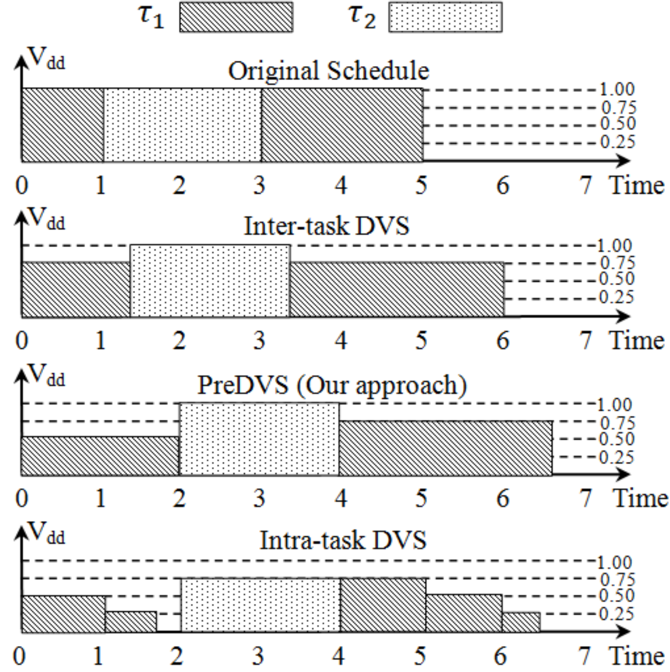


Figure 4-2. Inter-task DVS, PreDVS and Intra-task DVS.

energy-efficiency scheduling techniques in real-time systems [89]. Since our approach focuses on static slack exploitation, we assume every task takes its worst-case execution time to complete as most existing inter-task DVS works do. We focus on hard real-time systems with preemptive periodic task sets in this work. The system can be executed on any processor with discrete voltage/frequency levels. We formulate the problem in Section 4.2.2. The problem transformation scheme and an pseudo-polynomial algorithm which can give optimal solutions are presented in Section 4.2.3. We then propose the fully polynomial-time approximation scheme in the same section. In Section 4.2.4, we describe two heuristics with and without using problem the transformation scheme.

4.2.2 Problem Formulation

In this section, we formulate our problem, prove its NP-hardness and then discuss its unique difficulties. Specifically, we are given a set of m independent periodic tasks T$\{\tau_1,\tau_2, \ldots ,\tau_m\}$ with each task $\tau_i \in$ T has known period p_i, deadline d_i and worst-case execution cycles (WCEC) c_i. Task $\tau_i \in$ T has energy consumption e_i^k and execution time

t_i^k at processor voltage $v_k \in$ V. Note that we use WCEC c_i here to reflect the worst-case workload of each task since it is independent of processor frequency level. e_i^k and t_i^k can be computed based on the underlying processor energy model. The average switched capacitance of each task could be constant or variable, although research has shown that there is very little variation among real-time tasks in practice [104]. In order to avoid leakage power consumption compromising the energy saving, we can eliminate all the voltage levels in V which have the frequency level below the critical speed [51]. We assume that each task is released at the beginning of every period and the relative deadline is equal to the period. We prove that our voltage scaling problem is NP-hard by considering a **simplified version** of our problem – inter-task scaling – in which each task $\tau_i \in$ T is uniquely assigned a fixed voltage level throughout all its jobs. The system schedulability can be guaranteed by restricting the total utilization rate of task set under the scheduler's bound U. Note that it is sufficient to consider the task scheduling over its hyper-period P (equal to the least common multiple of all tasks' periods) since periodic task set has repetitive execution pattern during every P. To be more specific, the simplified problem can be stated as:

$$min(E = \sum_{i=1}^{m} \sum_{k=1}^{l} \frac{P}{p_i} \cdot x_i^k \cdot e_i^k) \tag{4–1}$$

subject to,

$$\sum_{i=1}^{m} \sum_{k=1}^{l} x_i^k \cdot \frac{t_i^k}{p_i} \leqslant U \tag{4–2}$$

$$\forall i \sum_{k=1}^{l} x_i^k = 1 \tag{4–3}$$

In Equation (4–1), x_i^k is a 0/1 variable which denotes whether task τ_i is assigned with voltage level v_k and Equation (4–3) presents the temporary assumption we make that only one voltage level is used throughout each task's execution. Equation (4–2) shows the sufficient condition of schedulability which must be satisfied. According to [137], the inter-task DVS problem formulated above is NP-hard, as shown below:

Theorem 1. *The simplified version of our problem stated above by Equation (4–1), (4–2) and (4–3) is NP-hard.*

Proof. This problem can be proved to be NP-hard by showing that it is reducible from an existing NP-hard problem – Multiple-Choice Knapsack Problem (MCKP) [58]. This reduction can be performed in polynomial time by a transformation of the goal from energy consumption minimization to energy saving maximization. It is done by changing the objective to maximize in Equation (4–1) and replacing e_i^k with (e_i^{max} - e_i^k) where e_i^{max} = max(e_i^k), $v_k \in$ V. Finding the optimal voltage assignments for each task is equivalent to finding an optimal solution for MCKP which picks one and only one object from each class. □

Now let's switch back to our **original problem**. By assigning multiple voltage levels at different places throughout each task's execution, more energy savings can be achieved since we have more flexibility during decision making. Nevertheless, the issue of when and how to apply scaling remains to be solved. Clearly, it is not feasible to consider all possible positions. Task preemption, which creates multiple segments of a single job, provides natural opportunities to assign different voltage levels to each task. In this work, we examine the EDF schedule[1] when the system is executed without DVS and change the processor frequency whenever a job starts execution or resumes after being preempted. Since inter-task scaling techniques also have to perform voltage switching in all these occasions, our strategy does not introduce any additional runtime overhead. It is important since voltage scaling overhead can have significant negative impact on overall energy consumption as well as performance [97] [83]. Note that PreDVS leads to distinct

[1] Our approach is also similarly applicable with RM scheduling but is not discussed in this dissertation.

energy consumption and execution time for each task's different jobs. As the simplified version has been shown to be NP-hard, the original problem is NP-hard as well.

4.2.3 Approximation Scheme

Since PreDVS problem has shown to be NP-hard (and thus does not admit a polynomial time optimal algorithm), the best option is to devise an efficient method that can lead to approximate-optimal solutions. As described in Section 4.2.2, our original problem (PreDVS) essentially adds another dimension (voltage/frequency selection for a task's each segment) to the simplified version (Inter-task DVS). This fact prevents us from solving the problem directly by adapting approximation algorithms for MCKP or inter-task DVS [137]. In this section, we present our two primary contributions. First, we develop a problem transformation scheme that can eliminate this complexity. Next, we propose an fully polynomial-time approximation algorithm which can efficiently solve the problem.

4.2.3.1 Problem Transformation

This section describes four important steps of our problem transformation scheme.

Step 1: As in traditional systems without a voltage scalable processor, all the tasks are executed at a fixed frequency. We use the case in which the processor is running solely at the highest voltage as the baseline and further assume that the given task set is schedulable in this case otherwise applying DVS is not meaningful. We simulate and generate an EDF schedule of the task set on the target system. Each task is set to take its WCEC c_i to finish. During simulation, we let the scheduler generate the distinct block list and distinct block set list of each task, which are defined as:

Definition 3. *A **distinct block** is an execution segment (interval) of a task with a distinct pair of start and end point.*

Definition 4. *A **distinct block set** is a set of distinct blocks which compose a whole job of a task. Every distinct block set has a different set of distinct block(s).*

Let b_i^j and s_i^j denote the jth distinct block and distinct block set of task τ_i, respectively. Figure 4-3 illustrates these two terms. In this example, we consider three tasks: $\tau_1\{3,3,1\}$[2] , $\tau_2\{5,5,2\}$ and $\tau_3\{12,12,4\}$. For task τ_1, it never gets preempted and has only one distinct block b_1^1 that is of its entirety, which forms its only distinct block set $s_1^1 = \{b_1^1\}$. Task τ_2, however, has three distinct blocks: b_2^1 appears from time 1 to 3, which is of its entirety; b_2^2 appears from time 5 to 6, which is its first half; b_2^3 appears from time 7 to 8, which is its second half. Therefore, τ_2 has two distinct block sets: $s_2^1 = \{b_2^1\}$ and $s_2^2 = \{b_2^2, b_2^3\}$. Task τ_3, during its first period, experiences two preemptions which result in three distinct blocks: b_3^1 which is its first quarter, b_3^2 which is its second quarter and b_3^3 which is the rest half. These three distinct blocks compose a distinct block set $s_3^1 = \{b_3^1, b_3^2, b_3^3\}$. Note that τ_3 may have more distinct blocks (and sets) since only first 12 time units are shown here. In practice, one needs to consider the whole hyper-period P to collect all the distinct blocks (and sets) for each task. We denote $|s_i^j|$ as the number of blocks in s_i^j and δ_i as the number of distinct block sets for task τ_i.

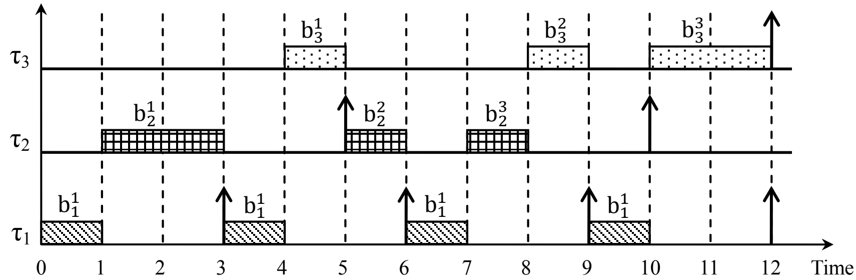


Figure 4-3. Distinct block and distinct block set.

Step 2: Once we have a list of all distinct blocks for each task, static profiling is carried out to collect the energy consumption as well as execution time for each block. We calculate these two values for each distinct block under all voltage levels in V. Let $e_i^{j,k}$ and

[2] The three elements in the tuple here denote period, deadline and worst-case execution time, respectively.

$t_i^{j,k}$ denote these two values of task τ_i's j^{th} distinct block b_i^j under voltage v_k. Note that energy and time overhead for voltage transition are incorporated in them, respectively. In other words, we generate a *profile table* for every b_i^j which stores $e_i^{j,k}$ and $t_i^{j,k}$ for all l voltage levels in V.

Step 3: For each distinct block set s_i^j, different voltage assignments for every distinct block in it will effect the total energy consumption as well as execution time for the entire set, which essentially forms a whole job. In order to take all possible scenarios into account, we calculate the total energy consumption and execution time of all voltage level combinations, each of which comprises of one voltage level chosen for each distinct block in s_i^j. Let $E_i^{j,h}$ and $T_i^{j,h}$ stand for the total energy consumption and execution time of s_i^j using voltage level combination h. Specifically, $E_i^{j,h} = \sum_{b \in s_i^j} \sum_{k=1}^{l} x_i^{b,k}\ e_i^{b,k}$ and $T_i^{j,h} = \sum_{b \in s_i^j} \sum_{k=1}^{l} x_i^{b,k}\ t_i^{b,k}$ where s_i^j is used to represent the index set comprising of distinct blocks in itself, and $x_i^{b,k}$ has the same meaning as x_i^k in Equation (4–1). Each pair of $E_i^{j,h}$ and $T_i^{j,h}$ are stored in the *profile table* for s_i^j. Furthermore, non-beneficial voltage combinations, whose energy consumption and execution time are dominated by another combination in the same set, are eliminated. We use a dynamic programming based algorithm to generate the profile table for each distinct block set as shown in Algorithm 3. The outermost loop iterates over all the blocks in s_i^j. In each iteration, we maintain a list L_b containing all beneficial voltage combinations of the first b blocks. We enumerate all combinations by merging l lists of $L_k{}'$ which consist of all the elements in the previous list L_{b-1} with each of them having their energy and execution time value added by the currently considered block's energy and execution time under voltage v_k. We use $\bigoplus$ to denote the element-wise addition. The merged list is then sorted in non-decreasing order of the total energy. We examine each of the elements in the merged list in the sorted order and append it to L_b only when it has shorter execution time than the latest added entry in L_b to ensure it is beneficial. Obviously, the number of Pareto-optimal combinations in s_i^j's profile table is $\pi_i^j \leqslant l^{|s_i^j|}$.

Algorithm 3: Profile table generation for distinct block set s_i^j.

1: $L_0 = \{(0, 0)\}$
2: **for** $b = 1$ to $|s_i^j|$ **do**
3:     **for all** $v_k \in$ V **do**
4:         $L_k{}' = L_{b-1} \bigoplus (e_i^{b,k}, t_i^{b,k})$
5:     **end for**
6:     Merge all lists of $L_k{}'$ into one profile table $L_b{}' = \{(\overline{e}'_r, \overline{t}'_r)\}$
7:     Sort $L_b{}'$ in non-decreasing order of energy sum $\overline{e}'_r$
8:     $L_b = \varnothing$
9:     Add the first element of $L_b{}'$, $(\overline{e}'_1, \overline{t}'_1)$, into L_b
10:     **for all** elements in $L_b{}'$, starting from $r = 2$ **do**
11:         Let $(\overline{e}'', \overline{t}'')$ denote the latest added entry in L_b
12:         **if** $\overline{t}'_r < \overline{t}''$ **then**
13:             **if** $\overline{e}'_r == \overline{e}''$ **then**
14:                 Replace $(\overline{e}'', \overline{t}'')$ with $(\overline{e}'_r, \overline{t}'_r)$ in L_b
15:             **else**
16:                 Add element $(\overline{e}'_r, \overline{t}'_r)$ to the end of L_b
17:             **end if**
18:         **end if**
19:     **end for**
20: **end for**
21: **return** $L_{|s_i^j|}$

Figure 4-4 shows a simple illustrative example. Here we show the profile table generation for task τ_3's distinct block set s_3^1 shown in Figure 4-3. For simplicity, assume only two voltage levels are available. For instance, the 3rd entry in the final profile table, $\{E_3^{1,3}$=18, $T_3^{1,3}$=22$\}$, is derived by selecting voltage level $\{e_3^{1,1}$=5, $t_3^{1,1}$=8$\}$ for b_3^1, $\{e_3^{2,1}$=3, $t_3^{2,1}$=6$\}$ for b_3^2 and $\{e_3^{3,2}$=10, $t_3^{3,2}$=8$\}$ for b_3^3 and then by adding them together.

Step 4: So far we have obtained complete profiling information for all the jobs of each task. In other words, we know each task occurrence's total energy and execution time under all possible voltage level assignments. In the original schedulability condition $\sum_{i=1}^{m} \frac{t_i}{p_i} \leqslant 1$ [69], t_i represents task τ_i's worst-case execution time. In order to guarantee that the task set is still schedulable after applying PreDVS, we have to ensure task τ_i's each job does not execute longer than some specific value t_i.

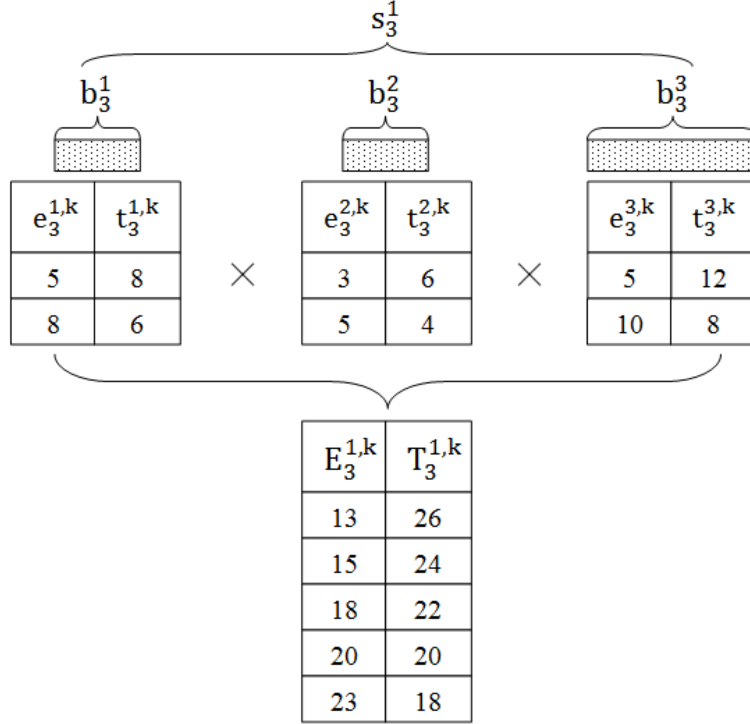


Figure 4-4. Profile table generation for distinct block set.

In order to ensure schedulability, we decide a *threshold* execution time t_i^{th} for each task to be used in the above schedulability condition as t_i that will act as the upper bound on each job's execution time. Now the original problem has become how to select one voltage combination for each distinct block set of a task so that the total energy consumption $E = \sum_{i=1}^{m} \sum_{j=1}^{\delta_i} \lambda_i^j \cdot E_i^{j,h'}$ (where h' is the selected profile table entry's index for s_i^j and λ_i^j is the number of times s_i^j occurs in the hyper-period P) is minimized while $\forall i\ \forall j\ T_i^{j,h'} \leqslant t_i^{th}$ is satisfied. An important question is how to decide t_i^{th}. In this step, we give every voltage combination in the profile tables a chance to use its execution time as the threshold for the corresponding task. Once a voltage combination of one distinct block set is picked as the threshold for task τ_i, all the other distinct block set's decisions can be made in a greedy manner, that is, the one with lowest energy consumption while execution time is less than or equal to the chosen threshold is selected. Note that if, for some other distinct block sets, no voltage combination can make its execution time under the threshold, the chosen threshold is infeasible and thus discarded. After all the decisions

are made, we calculate the total energy consumption of that entire task and then put them along with the threshold execution time into the *aggregated profile table*, which is used as input to our approximation algorithm. Algorithm 4 illustrates this process. Note that h denote the index of the distinct block set profile table entry which is currently chosen to act as the threshold.

Algorithm 4: Aggregated profile table generation for τ_i.

1: Sort each s_i^j's profile table in the ascending order of $E_i^{j,h}$
2: **for** $j = 1$ to δ_i **do**
3:   **for** $h = 1$ to π_i^j **do**
4:     $isFeasible =$ **true**
5:     $h'_j = h$ {h'_j denotes the selected index of s_i^j.}
6:     **for** $k = 1$ to δ_i; $k \neq j$ **do**
7:       **for** $u = 1$ to π_i^j **do**
8:         **if** $T_i^{j,u} \leqslant T_i^{j,h}$ **then**
9:           $h'_u = u$
10:           break
11:         **end if**
12:       **end for**
13:       **if** h'_u is not updated **then**
14:         $isFeasible =$ **false**
15:         break {$T_i^{j,h}$ is an infeasible threshold.}
16:       **end if**
17:     **end for**
18:     **if** $isFeasible$ is **true then**
19:       $e_i = \sum_{j=1}^{\delta_i} \lambda_i^j \cdot E_i^{j,h'_j}$ {Total energy consumption of τ_i}
20:       Add $(e_i, T_i^{j,h})$ into task τ_i's aggregated profile table
21:     **end if**
22:   **end for**
23: **end for**

Figure 4-5 shows an illustrative example of aggregated profile table generation. Suppose task τ_3 in Figure 4-3 has three distinct block sets in P and for each of them we have generated profile table in Step 3 as shown on the top-part of Figure 4-5. For simplicity, we assume that each of them only occurs once in P. The first entry in τ_3's aggregated profile table is calculated as follows. We choose execution time (26) of the first entry in s_3^1's profile table as the threshold. For s_3^2, the first entry cannot be selected since

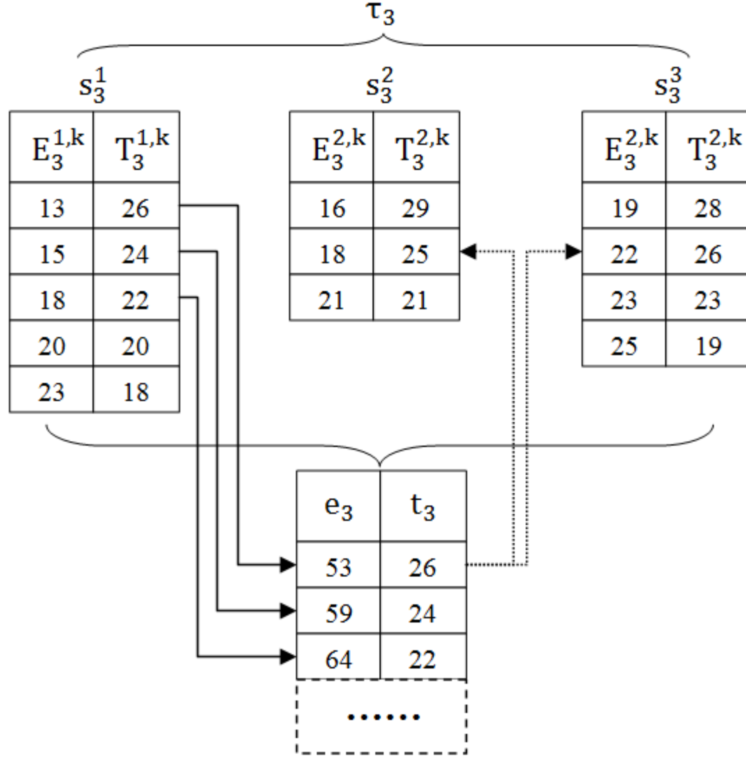


Figure 4-5. Aggregated profile table generation for each task.

its execution time is higher than the threshold. Hence the second entry is chosen. For the same reason, the second entry is selected for s_3^3. The total energy consumption and the selected threshold execution time for τ_3 are inserted into its aggregated profile table. The rest of the table can be generated similarly.

After applying Algorithm 4, non-beneficial entries in the aggregated profile table are filtered out. Note that Algorithm 4 assigns the same voltage combination to every occurrence (job) of each distinct block set. Theorem 2 shows that it is reasonable and safe to do so in finding optimal assignments.

Theorem 2. *An optimal solution of our problem must assign the same voltage combination for each occurrence of a distinct block set.*

Proof. Suppose in the optimal assignment, distinct block set s_i^j is assigned two different voltage combinations vc_1 and vc_2. Assume that the threshold execution time chosen is t_i^{th}.

Thus the execution time of both vc_1 and vc_2 are less than t_i^{th}. Since it is always safe to use voltage combinations with execution time under threshold, one can get a better solution by replacing the higher energy consuming one with the lower one, which contradicts the fact that the solution is optimal. □

Complexity Analysis: We now analyze the complexity of our problem transformation scheme. *Step 1* performs scheduling of the task set. *Step 2* requires $\sum_{i=1}^{m}\sum_{j=1}^{\delta_i}|s_i^j|$ calculations. *Step 3* has a time complexity of $O(\sum_{i=1}^{m}\sum_{j=1}^{\delta_i}|s_i^j|\cdot|L_b{}'|\cdot\log(|L_b{}'|))$, where $|L_b{}'|$ is the upper bound of the length of list $L_b{}'$ in Algorithm 3. *Step 4* needs a running time of $O(\sum_{i=1}^{m}\sum_{j=1}^{\delta_i}\delta_i\cdot(\pi_i^j)^2)$. Each step takes only polynomial time. It is important to note that the problem transformation introduces only design-time computation overhead. Although the static overhead depends on the nature of the input task set, our experiments show that it normally takes only in the order of minutes for common task sets.

4.2.3.2 Approximation Algorithm

The program transformation results in an aggregated profile table for each task. Each entry of the aggregated profile table (say j^{th} entry of task τ_i) represents one possible voltage assignment of all distinct block sets for task τ_i and keeps the corresponding total energy consumption as well as execution time, denoted by e_i^j and t_i^j, respectively. Note that here t_i^j is actually the selected threshold time. We divide each t_i^j by period p_i to represent the utilization rate (t_i^j/p_i) of the task. Furthermore, let ρ_i denote the number of entries in task τ_i's aggregated profile table.

We now convert our problem from a minimization version to a maximization one. Let $e_i^{max}=\max\{e_i^1, e_i^2, \ldots, e_i^{\rho_i}\}$. For each e_i^j, we calculate energy saving $\overline{e}_i^j = e_i^{max} - e_i^j$. Now the objective becomes to maximize total energy saving $\overline{E}=\sum_{i=1}^{m}\overline{e}_i^{r_i}$ while satisfy the schedulability condition $\mathrm{T}=\sum_{i=1}^{m}t_i^{r_i}\leqslant U$ by choosing one and only one entry from the aggregated profile table for each task (here r_i is the index of the chosen entry).

Dynamic Programming Dynamic programming gives the optimal solution to our problem. Let $\overline{e}_i^{max}$ defined as $\max\{\overline{e}_i^1, \overline{e}_i^2, \ldots, \overline{e}_i^{\rho_i}\}$. Clearly, $\overline{E}\in[1, m\overline{e}_i^{max}]$. Let $S_i^{\overline{E}}$ denote

a solution in which we make decisions for the first i tasks and the total energy saving is equal to $\overline{E}$ while the utilization rate T is minimized. A two-dimensional array is created where each element T[i][$\overline{E}$] stores the utilization rate of $S_i^{\overline{E}}$. Therefore, the recursive relation for dynamic programming is:

$$T[i][\overline{E}] = min_{j \in [1,\rho_i]}(T[i-1][\overline{E} - \overline{e}_i^j] + t_i^j) \tag{4–4}$$

Using this recursion, we fill up T[i][$\overline{E}$] for all $\overline{E} \in [1, m\overline{e}_i^{max}]$. Finally, the optimal energy saving $\overline{E}^*$ is found by:

$$\overline{E}^* = \{max\ \overline{E} \mid T[m][\overline{E}] \leqslant U\} \tag{4–5}$$

Dynamic programming achieves the optimal energy saving by iterating over all the tasks (1 to m), all possible total energy saving value (from 1 to $m\overline{e}_i^{max}$) and all entries in each task's aggregated profile table (from 1 to ρ_i). This algorithm fills the array in order so that when calculating the ith row (T[i][]), all the previous (i - 1) rows are all filled. Hence, the time complexity is $O(m^2 \cdot max\{\rho_i\} \cdot \overline{e}_i^{max})$, which is pseudo-polynomial since the last term is unbounded.

Approximation Algorithm The approximation algorithm proposed in this section is based on dynamic programming. It reduces the time complexity by scaling down every $\overline{e}_i^j$ value by a constant K such that $\overline{e}_i^{max}/K$ can be bounded by m (as well as the approximation ratio ε) – which reduces the complexity to polynomial. By doing so, we actually decrease the size of the design space. Our goal is to guarantee that the energy saving achieved by our approximation algorithm is no less than $(1-\varepsilon)\overline{E}^*$.

In order to obtain the constant K, we need to get the lower and upper bound on $\overline{E}^*$. This is done by employing a LP-relaxation version of our problem by removing the integral constraint (choose only one entry out of each task's aggregated profile table), that is, "fractional" entries are allowed to be chosen. Algorithm 5 shows the polynomial-time greedy algorithm which can give the optimal solution to the LP-relaxation problem. Note that $\widetilde{e}_i^j$ is the incremental energy saving – a measure of how much more energy saving

can be gained if the jth entry is chosen instead of the $(j-1)^{th}$ entry from task τ_i's table. Here, $\widetilde{p}_i^j$ represents the incremental energy saving efficiency $(\overline{e}_i^j/t_i^j)$. $\widetilde{U}$ keeps the residual utilization rate. The algorithm terminates when $\widetilde{U}$ is exhausted. The LP-relaxation optimal choice for task τ_i, found by the algorithm, is r_i where $x_i^{r_i} = 1$. The split task, τ_s, has two fractional entries being picked: r_s and $r_{s'}$ ($\widetilde{p}_s^{r_s} < \widetilde{p}_s^{r_{s'}}$). We have the following lemma:

Lemma 1. *If the optimal solution S^{LP} to the LP-relaxation version of our problem has no split task, it is already the optimal solution to our original problem. Otherwise, S^{LP} has at most one split task τ_s in which the two chosen fractional entries must be adjacent in its aggregated profile table.*

Proof. Since this scenario can be mapped to MCKP, we can reuse the proof shown in [58]. □

Let $\overline{E_0}$ be the maximum of the following three values: 1) total energy saving when the split task τ_s is discarded; 2) energy saving generated by the first fractional entry; 3) energy saving generated by the second fractional entry. That is,

$$\overline{E_0} = max\{\sum_{i=1;i\neq s}^{m} \overline{e}_i^{r_i} \ , \ \overline{e}_s^{r_s} x_s^{r_s} \ , \ \overline{e}_s^{r_{s'}} x_s^{r_{s'}}\} \tag{4–6}$$

Note that according to Lemma 1, the last two terms belong to the same task. Now we can give the upper and lower bound of $\overline{E}^*$, as shown in the following lemma:

Lemma 2. *$\overline{E_0}$ dictates the lower and upper bound of the optimal energy saving as: $\overline{E_0} \leqslant \overline{E}^* \leqslant 3\overline{E_0}$.*

Proof. If $\overline{E_0} = \sum_{i=1;i\neq s}^{m} \overline{e}_i^{r_i}$, we can safely obtain higher overall energy saving by adding $\overline{e}_s^{r_s}$. If $\overline{E_0}$ equals to either of the other two terms, more energy saving can be achieved by adding $\sum_{i=1;i\neq s}^{m} \overline{e}_i^{min}$ where $\overline{e}_i^{min} = min\{\overline{e}_i^1, \overline{e}_i^2, ..., \overline{e}_i^{\rho_i}\}$. Hence, we have $\overline{E}^* \geqslant \overline{E_0}$. Clearly, the solution to the LP-relaxation version must not be worse than the one for the original

Algorithm 5: Greedy algorithm for LP-relaxation problem.

1: **for** $i = 1$ to m **do**
2: Sort τ_i's aggregated profile table in ascending order of t_i^j.
3: $\widetilde{p}_i^1 = \overline{e}_i^1 / t_i^1$
4: **for** $j = 2$ to ρ_i **do**
5: $\widetilde{e}_i^j = \overline{e}_i^j - \overline{e}_i^{j-1}$
6: $\widetilde{t}_i^j = t_i^j - t_i^{j-1}$
7: $\widetilde{p}_i^j = \widetilde{e}_i^j / \widetilde{t}_i^j$
8: **end for**
9: **end for**
10: $\widetilde{U} = U - \sum_{i=1}^{m} t_i^1$
11: $\widetilde{E} = \sum_{i=1}^{m} \overline{e}_i^1$
12: Sort all the entries from each task's aggregated profile table together in descending order of $\widetilde{p}_i^j$, associating with the original indices i and j.
13: **for** each entry (i,j) in the sorted order of $\widetilde{p}$ **do**
14: **if** $\widetilde{U} - \widetilde{t}_i^j < 0$ **then**
15: $s = i$; $t = j$ {Indices of the entry to be split.}
16: break {Utilization rate exceeds the bound.}
17: **end if**
18: $\widetilde{E} = \widetilde{E} + \widetilde{p}_i^j$
19: $\widetilde{U} = \widetilde{U} - \widetilde{t}_i^j$
20: $x_i^j = 1$; $x_i^{j-1} = 0$ {entry (i,j) is chosen instead of (i,j-1)}
21: **end for**
22: $x_s^t = \widetilde{U} / \widetilde{t}_s^t$; $x_s^{t-1} = 1 - x_s^t$
23: $\widetilde{E} = \widetilde{E} + \widetilde{p}_s^t x_s^t$
24: **return** $\widetilde{E}$

problem. In other words, $\widetilde{E} \geqslant \overline{E}^*$. Since $\widetilde{E} = \sum_{i=1; i \neq s}^{m} \overline{e}_i^{r_i} + \overline{e}_s^{r_s} x_s^{r_s} + \overline{e}_s^{r_s'} x_s^{r_s'} \leqslant 3\overline{E_0}$, we have $\overline{E}^* \leqslant 3\overline{E_0}$. □

Now we decide the scaling down factor K using the bounds described above. We prove that approximation ratio and polynomial time complexity can be guaranteed if we assign $K = \frac{\varepsilon \overline{E_0}}{m}$, as shown in the following lemmas:

Lemma 3. *The K-scaled dynamic programming algorithm generates a* $(1 - \varepsilon)$ *approximation voltage assignment.*

Proof. Let the scaled energy saving value $\overline{e}'^j_i = \lfloor \overline{e}^j_i / K \rfloor$, we have $K\overline{e}'^j_i \leqslant \overline{e}^j_i < K(\overline{e}'^j_i + 1)$, hence $\overline{e}^j_i - K\overline{e}'^j_i < K$. Therefore, by accumulating all m tasks, we have:

$$\sum_{i=1}^{m} \overline{e}^{h_i}_i - K \sum_{i=1}^{m} \overline{e}'^{h_i}_i < Km \tag{4–7}$$

where h_i is the index of the selected aggregated profile table entry for task τ_i.

Note that the left term of Equation (4–7) is the approximation scaling error. Since $K = \frac{\varepsilon \overline{E_0}}{m}$, we have $Km = \varepsilon \overline{E_0}$. Since $\overline{E}^* \geqslant \overline{E_0}$, according to Lemma 2, we have $Km \leqslant \varepsilon \overline{E}^*$. Therefore, the approximation error $\sum_{i=1}^{m} \overline{e}^{h_i}_i - K \sum_{i=1}^{m} \overline{e}'^{h_i}_i < Km \leqslant \varepsilon \overline{E}^*$. Hence the approximation ratio ε holds. □

Lemma 4. *The time complexity of the K-scaled dynamic programming algorithm is* $O(\frac{m^2 \cdot max\{\rho_i\}}{\varepsilon})$.

Proof. Given the upper bound of $\overline{E}^*$ ($\overline{E}^* \leqslant 3\overline{E_0}$), the dynamic programming method can be improved to search in the range of $[1, 3\overline{E_0}]$, resulting in a time complexity of $O(m \cdot max\{\rho_i\} \cdot \overline{E_0})$. For the scaled version, the complexity is reduced to $O(m \cdot max\{\rho_i\} \cdot \frac{\overline{E_0}}{K})$. Given $K = \frac{\varepsilon \overline{E_0}}{m}$, we have $\frac{\overline{E_0}}{K} = \frac{m}{\varepsilon}$, thus the complexity becomes $O(\frac{m^2 \cdot max\{\rho_i\}}{\varepsilon})$, which is independent of any pseudo-polynomial energy values. □

Theorem 3. *The proposed algorithm is a fully polynomial time (1 - ε) approximation scheme for the maximization version of our problem.*

Proof. Directly follows from Lemma 3 and 4. □

Now let's evaluate the quality of the solution generated by the converted problem with respect to our original problem. Let E^* denote the optimal result (the minimum energy consumption) and α denote the approximation ratio for the original problem. Given an approximation ratio ε for the maximization version, α can be quantified as:

$$(1 + \alpha)E^* = \sum_{i=1}^{m} e^{max}_i - (1 - \varepsilon)\overline{E}^* \tag{4–8}$$

Hence,

$$\alpha = \frac{\sum_{i=1}^{m} e_i^{max} - \overline{E}^* + \varepsilon\overline{E}^* - E^*}{E^*} \tag{4–9}$$

Since for a specific solution, according to our conversion strategy, we have: $E^* = \sum_{i=1}^{m} e_i^{max} - \overline{E}^*$. Therefore,

$$\alpha = \frac{\overline{E}^*}{E^*}\varepsilon \tag{4–10}$$

Equation (4–10) illustrates that α is related to ε by the factor of $\overline{E}^*/E^*$, which is the ratio of the total energy saving to total energy consumption over all tasks. In the worst case, when the overall utilization is low enough so that entries with the lowest energy consumption are selected for each task, this ratio reaches maximum. Let v_{max} and v_{min} denote the maximum and minimum voltage available, respectively. We have,

$$\alpha \leqslant \frac{\sum_{i=1}^{m}(e_i^{max} - e_i^{min})}{\sum_{i=1}^{m} e_i^{min}}\varepsilon \leqslant \frac{{v_{max}}^2 - {v_{min}}^2}{{v_{min}}^2}\varepsilon \tag{4–11}$$

Let γ denote this maximum ratio (thus $\alpha \leqslant \gamma \cdot \varepsilon$). In practice, given a voltage scalable processor, we first calculate its γ value. If $\gamma \leqslant 1$, it means that by solving the converted maximization problem using approximation ratio ε, we can get a solution with an equal or better quality bound ($\leqslant \varepsilon$) to the original problem. Otherwise, if needed, we can set $\varepsilon = \alpha/\gamma$ so that the specified approximation ratio (α) to the original problem can be achieved firmly. As a result, the time complexity of our approximation scheme with respect to the original problem is $O(\frac{m^2 \cdot max\{\rho_i\} \cdot \gamma}{\alpha})$. As shown in the experimental results, for common voltage scalable processors, γ is usually very small and in some cases (e.g., StrongARM [74]) is less than 1.

Now we have obtained an approximated optimal solution based on the original EDF schedule which is generated without voltage scaling. Note that running the task set with the new voltage assignment given by PreDVS could potentially result in a slightly different schedule since we essentially changed the execution time of each block. Therefore, we need to store all the voltage scaling points (along with their corresponding voltage levels) in a lookup table which can be easily accessed by the operating system to change the

voltage level at each point in the order of occurrence time. As described in Section 4.2.3.1, we have ensured that the utilization bound of EDF is observed, the modified task set is guaranteed to be schedulable. The solution we give is essentially with respect to the original schedule. Certainly, more iterations can be carried out based on the new schedule until it becomes steady. Based on our observations, such costly iterations contribute very little in overall energy savings, and therefore not beneficial.

4.2.4 Efficient PreDVS Heuristics

In this section, we propose two heuristics for PreDVS that can be used as alternatives to the approximation algorithm. The goal is to trade off design quality for running time compared to the approximation algorithm. The first heuristic does not require problem transformation steps described in Section 4.2.3.1. Therefore, it is the fastest approach and, as shown by our experimental results, can achieve better energy savings than the optimal inter-task DVS in certain scenarios. The second heuristic is based on the aggregated profile table that is generated by the problem transformation scheme. Hence it can lead to better solutions than the first one but with higher time complexity.

4.2.4.1 Heuristic Without Problem Transformation

Let S$\{s_1, s_2, ..., s_l\}$ denote the set of processor operating speeds (i.e., frequencies), in descending order, corresponding to the voltage levels in V. Suppose all the speeds are normalized to the highest one $s_1 = 1.0$. Aydin et al. [6] proved that, in an ideal system where we have continuously scalable processor speed, the constant and optimal speed for all tasks is $s_{opt} = max\{s_l, \eta\}$. Note that η represents the utilization ratio of the task set in the base case (e.g., all tasks execute at the highest voltage level). For real systems with discrete speed levels, using the two neighboring available speeds in S above and below s_{opt}, denoted by s_{above} and s_{below}, has shown to be sufficient to minimize the energy consumption optimally in practice [45] [64]. Specifically, for each task instance, we run t_{above} and t_{below} of time using the two neighboring speeds which are calculated as:

$$t_{above} = \frac{s_{opt} - s_{below}}{s_{above} - s_{below}} \cdot t_{opt} \tag{4–12}$$

$$t_{below} = t_{opt} - t_{above} \tag{4–13}$$

where t_{opt} is the time required to execute the task using s_{opt}. For example, as illustrated in Figure 4-6 (a), suppose two tasks τ_1 and τ_2 have optimal speed at $s_{opt} = 0.60$. However, the processor only supports four speed levels of 0.25, 0.50, 0.75 and 1.00. According to Equation (4–12) and (4–13), task τ_1 should execute at s_{above} (i.e., 0.75) for $t_{above} = 1.6$ time units and s_{below} (i.e., 0.50) for $t_{below} = 2.4$ time units, as shown in Figure 4-6 (b). Using the method in [64], we end up with the schedule shown in Figure 4-6 (c). However, as pointed out in Section 4.1, this strategy suffers from a large number of scaling points which may not be feasible in real systems [89].

In our PreDVS heuristic, we first simulate the task set and generate the schedule using the optimal speed s_{opt} for all tasks. Then, we simply assign each distinct block in the schedule with s_{above} or s_{below} depending on whether it starts before or after the optimal scaling point decided by t_{above}, respectively. In other words, when a task instance resumes after being preempted, it lowers the speed to s_{below} if the preemption happens after the optimal scaling point (in terms of workload). Otherwise, in order to guarantee all deadlines, s_{above} is used. Figure 4-6 (d) gives an example. Here, the first part of task τ_1 runs at s_{above} since it starts at cycle 0 in τ_1. Task τ_2 is not preempted thus can only use s_{above}. The second part of τ_1, however, can run at the lower level s_{below} since it starts after the optimal scaling point. There is only negligible extra runtime overhead since the schedule can be determined off-line. Obviously, it can achieve more energy savings than the uniform slowdown heuristic which simply round-up to the next available higher speed level [5].

For those blocks assigned s_{above}, it creates a time slack with the length of $t_{opt} - t_{above}$. This time slack can be potentially reclaimed by the very next block to further lower its voltage level to achieve more energy savings. In order to satisfy all the deadlines while

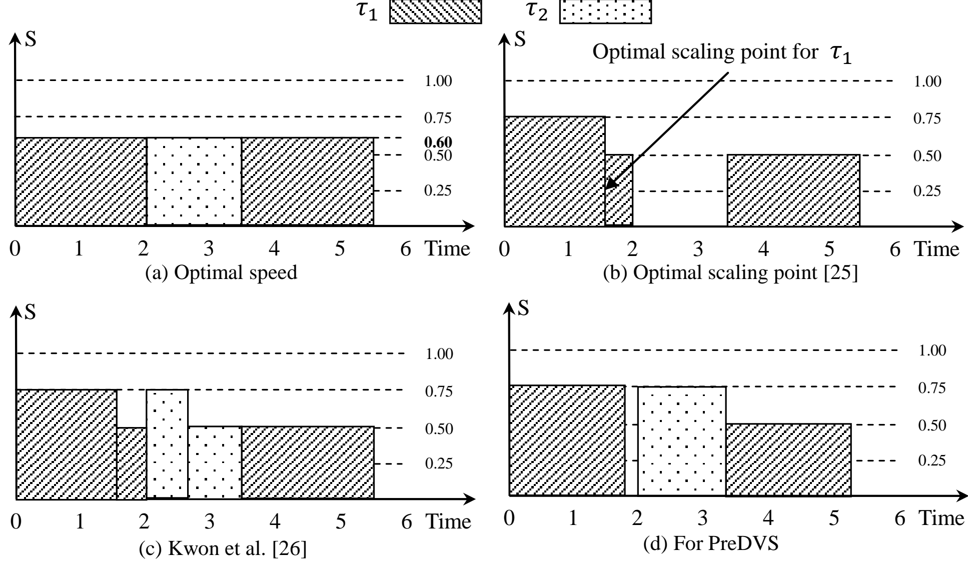


Figure 4-6. Illustration of PreDVS heuristic without problem transformation.

keeping our heuristic simple, if the next block starts at the corresponding job's arrival time and thus is not able to start earlier, we do not reclaim the slack since otherwise task rescheduling is required. Whenever the next block can be moved up, we lower its voltage level to the next available one if the time slack is sufficient to do so.

Algorithm 6 illustrates this heuristic in detail. The task set is scheduled under the optimal speed s_{opt}. Similarly as step 1 of the problem transformation scheme, let δ_i and $|s_i^j|$ denote the number of distinct block sets of task τ_i and the number of blocks in s_i^j. t_{opt}^i and t_{above}^i represent the time required to execute the task τ_i using s_{opt} and s_{above}, respectively. c_{above}^i denotes the number of cycles needs to be executed under s_{above}^i. For each block $b_i^{j,k}$ (k^{th} block in the jth distinct block set of task τ_i), we compare its start cycle with c_{above}^i and lower down the speed if the former one is larger. Note that dumping out each block's start cycle is easy during the simulation under s_{opt}. Clearly, it has a time complexity of $O(n)$ where $n = \sum_{i=1}^{m} \sum_{j=1}^{\delta_i} |s_i^j|$ is the total number of distinct blocks for all tasks in the system.

The voltage assignment given by Algorithm 6 will not affect the schedulability of the task set. This is because, compared with the method in [64], we use the same schedule

Algorithm 6: Heuristic based on ideal optimal voltage level.

1: $\eta = \sum_{i=1}^{m} \frac{t_i}{p_i}$ {Utilization ratio when there is no DVS.}
2: $s_{opt} = max(s_l, \eta)$
3: **for** $i = 2$ to l **do**
4: **if** $s_i \leqslant s_{opt}$ **then**
5: $s_{above} = s_{i-1}$
6: $s_{below} = s_i$
7: break
8: **end if**
9: **end for**
10: Schedule the task set under fixed s_{opt}.
11: **for** $i = 1$ to m **do**
12: $t^i_{opt} = \frac{c_i}{s_{opt}}$
13: $t^i_{above} = \frac{s_{opt} - s_{below}}{s_{above} - s_{below}} \cdot t^i_{opt}$
14: $c^i_{above} = t^i_{above} \cdot s_{above}$
15: **for** $j = 1$ to δ_i **do**
16: **for** $k = 1$ to $|s^j_i|$ **do**
17: **if** $b^{j,k}_i.startCycle \leqslant c^i_{above}$ **then**
18: $b^{j,k}_i.speed = s_{above}$
19: **else**
20: $b^{j,k}_i.speed = s_{below}$
21: **end if**
22: **end for**
23: **end for**
24: **end for**
25: Reclaim available slacks by a linear scan of n blocks.

and the execution time of each execution block in the schedule is no longer than [64] after assignment of s_{above} and s_{below}. Note that the slack reclamation phase does not make any execution block finish later than before. Therefore, we can safely conclude that the task set is ensured to be schedulable.

4.2.4.2 Heuristic With Problem Transformation

The problem transformation scheme, as described in Section 4.2.3.1, generates the aggregated profile table for each task. Each entry in the aggregated profile table represents one possible voltage assignment for all the distinct block sets of that task. The corresponding threshold execution time of the entry is essentially the worst-case execution time for all the task instances under that voltage schedule. Substantially, the aggregated

profile table provides much finer granularity in decision making for the entire task set than inter-task DVS. In other words, we have much more choices within the schedulability bound. This advantage can be efficiently exploited combining with the simple uniform slowdown method.

As shown in Algorithm 7, we first calculate base case utilization ratio η. Here t_i denote the execution time for task τ_i under the highest voltage level. For each task, we calculate the execution time required if s_{opt} is used (line 3), denoted by t_i^{opt}. After problem transformation is applied, we choose the aggregated profile table entry with the minimum total energy consumption while the threshold execution time is lower than t_i^{opt}. This heuristic has time complexity of $O(m \cdot max\{\rho_i\})$ plus the time required by problem transformation.

Algorithm 7: Uniform slowdown after problem transformation.

1: $\eta = \sum_{i=1}^{m} \frac{t_i}{p_i}$ {Utilization ratio when there is no DVS.}
2: **for all** task $\tau_i \in$ T **do**
3: $\quad t_i^{opt} = t_i/\eta$;
4: $\quad$ Do problem transformation for τ_i and generate the aggregated profile table using Algorithm 4.
5: $\quad$ Select the entry r_i in the aggregated profile table with: 1) Minimum total energy consumption $e_i^{r_i}$;
$\quad$ 2) Threshold execution time $t_i^{r_i} \leqslant t_i^{opt}$;
6: **end for**
7: **return** $r_i, \forall i \in [1, m]$

4.3 DSR: Dynamic Slack Reclamation

4.3.1 Overview

DVS stretches the clock cycle length (thus increases task execution time) resulting in energy reduction whenever time slack is available. *Static slack* is determined based on the Worst Case Execution Time (WCET) of each task. It is analyzed and exploited during the off-line scheduling process. However, in many cases, task's execution time may vary and thus complete earlier than expected at runtime. This could be caused by different input parameter values, environmental conditions, variable execution paths or mix of the

above. *Dynamic slack* created due to early completed tasks can be exploited to further reduce the power dissipation of subsequent tasks. Existing dynamic slack reclamation techniques are either based on certain assumptions or for dynamic energy minimization only. Furthermore, they did not consider various energy saving potentials across different tasks which our approach takes advantage of to utilize the slacks more efficiently. In [48], a priority queue is maintained for dynamic slacks generated and each newly arrival task simply fetches all the eligible slacks and scales down the voltage level until the critical speed[3] is reached, which is then followed by procrastination.

In this section, we develop a dynamic slack reclamation (DSR) algorithm for energy-aware scheduling in uniprocessor multitasking systems with the following innovative properties.

1. Our approach iteratively considers multiple tasks for utilizing the dynamic slack available along with necessary task rescheduling. This leads to higher energy savings compared to existing techniques (e.g. [48]), especially when tasks have different power characteristics [4].
2. Our approach can be parameterized to limit the search space of tasks (the number of subsequent tasks) to be considered for slack allocation. This effectively allows tradeoffs between energy saving versus runtime overhead.

Same as PreDVS, this algorithm also adjusts the voltage level multiple times within each task instance (i.e. job). It carefully allows task rescheduling to make more benefit from the available slack. Furthermore, our approach is relatively independent of the system characteristics and scheduling policy. It works, for example, either with or without earliest start time constraints. It can also incorporate scaling overhead if necessary. This leads to an extensive and flexible approach and is shown to lower the energy requirements

[3] Critical speed is the point lower than which the total energy per cycle will start increasing rather than decreasing [48] [120].

as compared to [48] with a minor runtime overhead. Extensive experimental results show that our technique can achieve significant reduction in energy requirements after applying static DVS (e.g., PreDVS). It also outperforms existing techniques for dynamic slack allocation by 2 - 12%.

4.3.2 Dynamic Slack Reclamation Algorithm

Energy optimization techniques dedicated to static slack allocation derive a scheduling scheme which minimizes energy consumption while guaranteeing all task deadlines. If we execute the tasks under this scheme assuming each of them requires its worst-case workload and let the scheduler record the execution trace, we can get a series of *execution blocks* each of which is a piece of task execution. For example, in Figure 4-7, we show a preemptive schedule of three periodic tasks[4] . The execution blocks are linearly indexed, e.g. b_1, b_2, ... , b_{10} in Figure 4-7. Specifically, the input to our problem can be further refined as:

- A set of n execution blocks $\mathcal{B}\{b_1, b_2, \ldots, b_n\}$. Each block is associated with its corresponding task id and job id.
- Each block $b_i \in \mathcal{B}$ has its arrival time (earliest start time) a_i if it is the first block in the corresponding job and an absolute deadline constraint d_i if it is the last block.
- Each block b_i has its current voltage assignment (thus start time and finish time) after applying the static slack allocation.
- Each block b_i has execution time t_i^k and energy consumption e_i^k at processor voltage level $v_k \in \mathcal{V}$ in the worst-case scenario.

As part of the static analysis, we calculate t_i^k and e_i^k for $\forall i \in [1, n]$ and $\forall k \in [1, h]$ based on either the existing processor datasheets or the energy model described in

[4] Although the example shown in this section is for a preemptive periodic task set, our approach is applicable to other kinds of tasks as long as the characteristics are known a priori and thus the static slack allocated schedule is pre-determined.

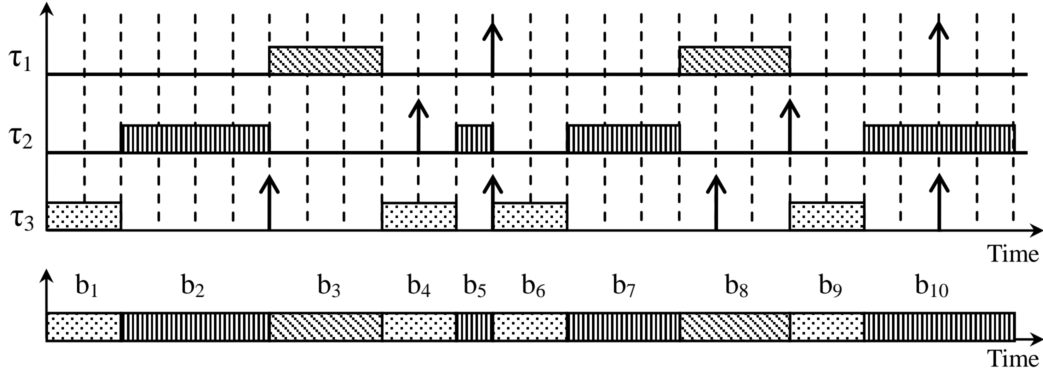


Figure 4-7. Execution blocks after static slack allocation.

Chapter 2. We store all the entries for each block with t_i^k lower than the execution time corresponding to its critical speed in a *profile table* with an increasing order of t_i^k (thus decreasing order of e_i^k). In other words, non-beneficial voltage levels are eliminated so that the increase in leakage energy will not compromise the reduction in dynamic energy consumption. Note that varying task's power characteristics lead to different critical speeds. This information is exploited at runtime by our algorithm.

As discussed in Section 4.3.1, during actual execution, task instances may take less dynamic instructions (hence shorter time) to complete than the worst-case scenario. The difference between ACET and WCET hence is the generated dynamic slack, as shown in Figure 4-8 where the first job of task τ_2 (b_2) finishes earlier by 3 time units. Note that if one job consists of multiple blocks due to preemption, its earlier completion can result in multiple discrete pieces of dynamic slack.

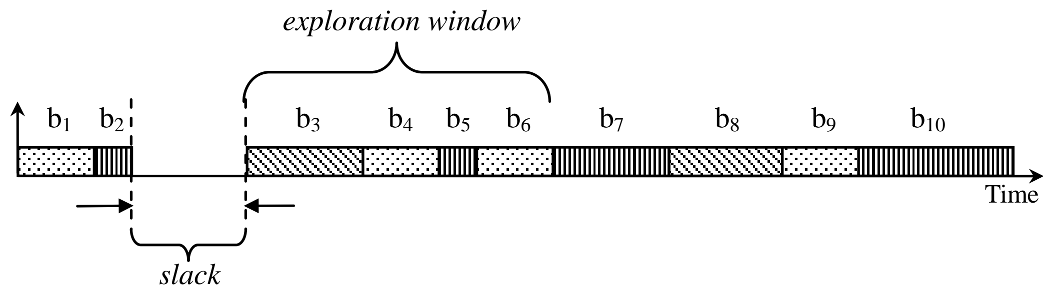


Figure 4-8. Dynamic slack generated by early finished task.

To reclaim dynamic slack, we reassign the voltage levels of one or more subsequent blocks after the slack at runtime. We define *exploration window* as the range of subsequent execution blocks from which the targets of slack reclamation are selected. In other words, we look forward within the exploration window and try to allocate the generated dynamic slack to these tasks in the most beneficial way. Let w denote the size of the exploration window. Clearly, since different blocks may have variable potential for energy reduction (based on the power characteristic and current voltage level assignment), larger w should generally result in better solution but introduce longer time overhead.

There are several design considerations which lead to several variations and finally the full description of our algorithm: 1) whether the tasks have earliest start time (or arrival time) constraints, and 2) whether the preemption schedule of a task is allowed to be modified at runtime (i.e. decomposition/agglomeration of the execution blocks). We describe each of these variations in the following subsections.

4.3.2.1 Tasks without Arrival Time Constraints

In order to lower some subsequent tasks' voltage level (i.e. stretch their execution time) to reduce energy requirements, they have to be able to start earlier by the same amount of time. The basic idea of our algorithm is to bring forward (start earlier) every block which receives slack by the difference between the execution time of its previous and new voltage assignments. By doing this, we ensure that no block (in the exploration window) after dynamic slack reclamation finishes later than before. Otherwise, deadline constraints may be violated in the future since it is always possible that all subsequent jobs finish in their WCET. Consider the case when there is no arrival time constraint (Figure 4-8). If b_4 and b_6 are selected to be assigned the dynamic slack, b_4 and b_6 as well as all the blocks between them and the one which creates the slack (b_2), which are b_3 and b_5 in this case, should be started earlier as illustrated in Figure 4-9. Clearly, no deadline will be violated since we ensure no block in the exploration window gets its completion delayed. Note that when making the decision, we assume that b_4 and b_6 still require

WCET to complete. However, they may also finish earlier and create additional slacks later.

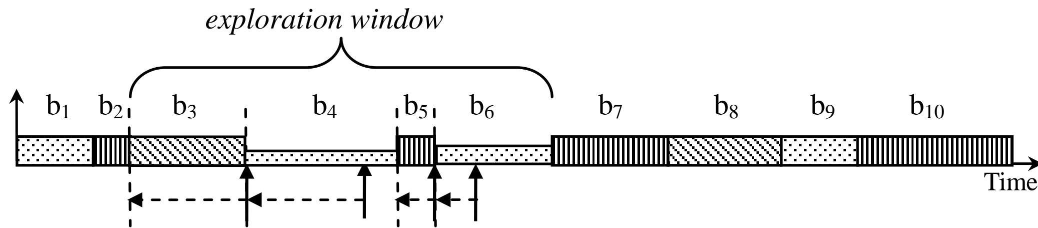


Figure 4-9. Dynamic slack allocation example.

Clearly, in the scenario where there is no arrival time constraint (e.g. all tasks are ready when the system begins), it is allowed to freely make any subsequent block start earlier to assign the slack within the exploration window. In other words, all the blocks within exploration window have equal opportunity to take the advantage of reclaiming full amount of slack. Algorithm for assigning the slack will be described in Section 4.3.3.

4.3.2.2 Tasks with Arrival Time Constraints

When tasks have arrival time constraints, e.g. periodic tasks, we may not have the freedom to start the execution of a subsequent block earlier to fully reclaim the slacks, i.e., it is possible that not all the blocks within the exploration window have the same capability to receive the slack. In the example shown in Figure 4-10, if b_2 finishes earlier to create 3 units of time slack, b_5, unlike b_3 and b_4, is not able to receive the full benefit since it can only be started earlier by at most 1 time unit. Similarly, b_6 cannot be further slowed down without affecting subsequent tasks since it starts right at its arrival time. We define the term *maximum reclaimable slack* ($MaxRS$) for each block as the maximum amount of available slack it can exploit. In this example, b_3 and b_4 have $MaxRS$ of 3 units but b_5 has only 1 unit. This observation leads to two variations of our approach.

Without Task Rescheduling As discussed above, in order to let one block start earlier, all the preceding blocks should also be moved up by the same amount of time. Therefore, within the exploration window, every block's $MaxRS$ is no more than any of its predecessors. If it is not allowed to change the original schedule (i.e. block execution

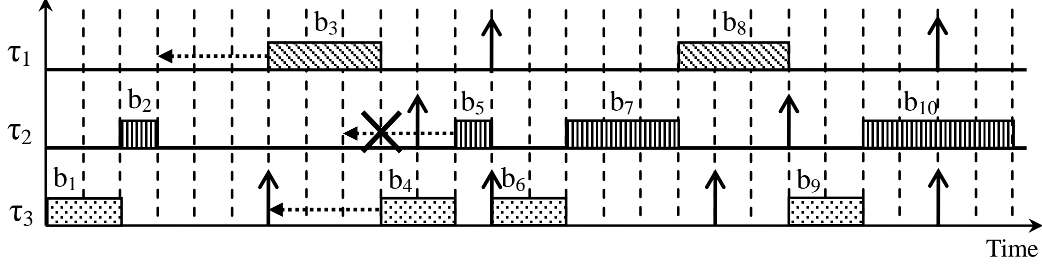


Figure 4-10. Dynamic slack allocation with arrival time constraints.

order), once a block B's $MaxRS$ gets reduced and becomes lower than its precedent block, all the blocks after B will also have their $MaxRS$ reduced to the same amount. In other words, even if some subsequent blocks can be moved up by the extent more than B can, they will still end up with their $MaxRS$ at most equal to B's since they can only start after B finishes. For example, in Figure 4-10, b_6 and all the subsequent blocks are not capable of using any dynamic slack since none of them can start earlier without rescheduling.

With Task Rescheduling We can prevent the $MaxRS$ of block b_i ($MaxRS_i$) from being reduced due to arrival time constraints by changing the task execution order. It is beneficial since it can increase the number of eligible blocks that can receive more slack in the exploration window. By doing this, potentially more energy savings can be achieved. This can be done by bringing forward the execution of some subsequent blocks (or part of them), say b_j where $j > i$, before b_i. As illustrated in Figure 4-11, for example, some block before b_1 finishes earlier and creates a piece of slack with length s. While we have $MaxRS_1 = s$, however, b_2 is not able to take any advantage since it starts at its arrival time and thus cannot be moved up ($MaxRS_2 = 0$). It will be inferior in terms of energy reduction in this case if b_2 has higher potential in energy reduction by claiming the time slack than b_1. Therefore, we can let the job consisting of b_1 and b_4 start earlier so that b_2 can be eligible to slowdown without affecting any other block's deadline. Essentially, we need to move part of b_4's execution with a length of s before b_2's arrival time. In this example, note that moving up b_3 does not help as b_3 itself arrives later than b_2. But b_3 also

benefits from task rescheduling: it now can reclaim full amount of the slack. As another example, using the previous scenario, as shown in Figure 4-12, b_6 to b_{10} are now legal to reclaim 1 unit of slack by moving one unit of b_7 before b_6. In general, by making the suggested changes in the schedule, $MaxRS$ values of blocks in the exploration window will be larger than the case when no task rescheduling is applied. Effectively, it is equal to judiciously changing the priority of the dynamic slack (defined in [48]) so that it can be better utilized as compared to a strategy that reclaims it as soon as possible by allocating it to the very next task [48]. Note that rescheduling is attempted for deciding $MaxRS$ but only actually happens when the corresponding block is selected as the target of slack reclamation.

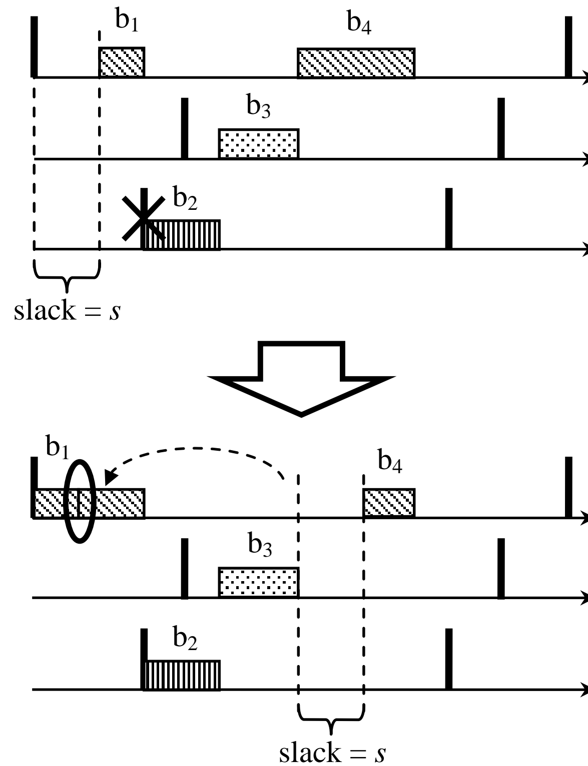


Figure 4-11. Slack reclamation with task rescheduling.

Obviously, in case of Figure 4-11, the amount of slack that b_2 as well as b_3 can reclaim depends on how much of b_4 can be brought forward. It is possible that b_1 itself has smaller $MaxRS$ than what is available due to its own arrival time constraint. Besides,

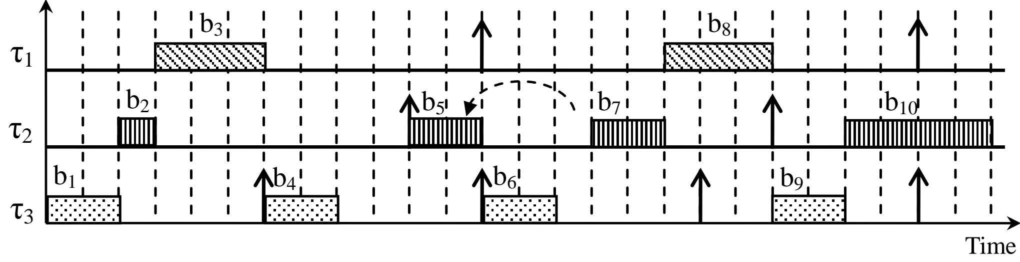


Figure 4-12. Task rescheduling example.

if the length of b_4 (under its current voltage assignment), say t_4^k, before rescheduling is shorter than the slack s, b_2 can only accept a slack with length of t_4^k after rescheduling. However, on the other hand, it is also possible that b_2 and b_3 themselves are not able to slowdown by s due to their own deadline constraints[5] . These scenarios will all result in reduced $MaxRS$ for subsequent blocks inevitably. In general, there may be multiple candidates that can be moved forward for maintaining higher $MaxRS_i$. We can simply choose the one which would result in maximum value of $MaxRS_i$. Note that if b_1 and b_4 have different voltage assignments before rescheduling, an extra scaling is needed which may lead to certain amount of overhead and need to be taken into account during decision making. It is also possible that, for some block b_i, no subsequent block of it has earlier arrival time. In this case, there is no remedy and b_i as well as all subsequent blocks can only receive a reduced $MaxRS$.

4.3.3 Algorithm

In this section, we describe the details of our algorithm. For tasks without arrival time constraints, all blocks in the exploration window share the same $MaxRS$ which is equal to the total amount of slack. However, for tasks with arrival time, there will be a series of n' groups with all blocks in each group having equal $MaxRS$ and the groups' $MaxRS$ values are in decreasing order as shown in Figure 4-13. This is because $MaxRS$

[5] Our study shows that it is rare and only happens when there are very few tasks (e.g., 2).

remains the same for each block, but may monotonically decrease for consecutive blocks due to additional constraints discussed above. Therefore, we have $MaxRS_1 > MaxRS_2 > ... > MaxRS_{n'}$.

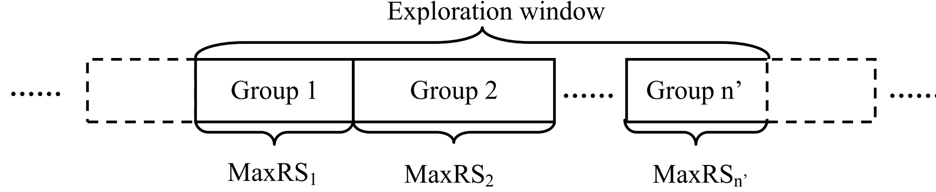


Figure 4-13. Exploration window partitions into groups according to $MaxRS$.

We define the minimum amount of slack time that can allow block b_i to lower down its current voltage level to the next available lower level as *minimum reclaimable slack* ($MinRS_i$). Note that a slack smaller than $MinRS_i$ will have no benefit for b_i since no energy saving can be achieved. If the block is already in the lowest voltage level of its profile table (thus further slow down will drop below its critical speed), its $MinRS$ is set to ∞. This process is applied iteratively for the available slack. A greedy approach is used in which the energy saving per unit of slack ($ESpU$) is maximized in each iteration. Specifically, for block b_i, we have:

$$ESpU_i = \frac{e_i^{h_i} - e_i^{h_i+1}}{MinRS_i} \tag{4–14}$$

where h_i is the index of the current voltage level of b_i and $MinRS_i = t_i^{h_i+1} - t_i^{h_i}$. We assign $MinRS$ units of slack to the block which has the maximum $ESpU$ value, but has $MinRS \leqslant MaxRS$. After each iteration, the target block's $MinRS$ is recalculated and each group's $MaxRS$ needs to be updated sequentially in a cascading fashion. Specifically, if b_i in group i' is allocated $MinRS_i$ units of slack, we let $MaxRS_j = MaxRS_j - MinRS_i$ for all blocks b_j in group i' (including b_i) as well as all the groups before i' along the timeline. If the blocks in group i' still have their common $MaxRS$ larger than the ones in the next group, no update is required for all the subsequent groups. If the $MaxRS$ value for group i' drops below its next group $i'+1$, we have to make them equal. Since group

$i' + 1$'s $MaxRS$ also gets changed, the update process repeats until it reaches the last group or the next group has lower $MaxRS$.

Algorithm 8: Dynamic slack reclamation algorithm.

1: **Input:** $startIdx$, s, w.
2: **Output:** New scheduling for subsequent blocks.
3: **Step 1:** Calculate $MaxRS$ for all the blocks in the window.
4: **Step 2:** Dynamic slack reclamation.
5: $endIdx \leftarrow startIdx + w$;
6: $minMinRS \leftarrow min(MinRS_i), \forall i \in [startIdx, endIdx]$;
7: Calculate $MinRS_i$ and $ESpU_i$, $\forall i \in [startIdx, endIdx]$;
8: **while** $s \geqslant minMinRS$ **do**
9: Find b_j in the window with:
10: 1) $MinRS_j \leqslant s$;
11: 2) $MinRS_j \leqslant MaxRS_j$;
12: 3) $ESpU_j$ is the maximum for $\forall j \in [startIdx, endIdx]$;
13: Allocate $MinRS_j$ units of slack to b_j and apply task rescheduling if needed;
14: $s \leftarrow s - MinRS_j$;
15: Update $MinRS_j$, $ESpU_j$, $minMinRS$ and $MaxRS$ for all the blocks in the window;
16: **end while**

Algorithm 8 shows the outline of our approach. Let $startIdx$ denote the index of the early finished block that creates slack with duration s. Here w represents the exploration window size. Note that line 13 and 15 are done based on the problem requirements (with/without arrival time constraints, allow/deny task rescheduling) accordingly. If multiple slack pieces are created due to one early-finished job, Algorithm 8 is called separately in a reverse order starting from the latest slack with the same size of exploration window. In this case, we use a simple scheme that all the blocks in the examined windows are procrastinated by the residual amount of slack if possible. By doing that, the unused slacks tend to combine together to form a larger idle period. For single piece slack reclamation, our algorithm inherently maintains all unused slack before all the subsequent blocks. Since our approach considers multiple candidates for slack allocation, the residual slack is normally very small (i.e. the system utilization U is close to 1). Earlier work has shown that static procrastination has no benefit [51] and dynamic procrastination can at most improve the total energy efficiency by 1% when U

is larger than 60% [48]. Therefore, our scheme that does not consider procrastination during scheduling will only lead to negligible solution quality degradation since there is no need to apply dynamic slack reclamation when U is smaller than 50%. This is because static scheduling already makes each task operating at or near the critical speed. We only consider the scenarios where U is no smaller than 0.6, which are reasonable and practical cases.

4.4 Experiments

4.4.1 PreDVS

4.4.1.1 Experimental Setup

To demonstrate the effectiveness of PreDVS, two DVS-capable processors are considered: StrongARM [74] and XScale [75]. The former one supports four voltage - frequency levels (1.5V - 206MHz, 1.4V - 192Mhz, 1.2V - 162MHz and 1.1V - 133MHz) with $\gamma = 0.86$[6] and the latter one supports five levels (2.05V - 1000MHz, 1.65V - 800MHz, 1.3V - 600HMz, 0.99V - 400MHz and 0.7V - 200MHz) with $\gamma = 7.58$. We compare our results with two scenarios: when no DVS is used and when optimal inter-task scaling is employed [137]. In the former scenario, every task is running under the highest voltage level. While in the latter scenario, a dynamic programming based algorithm is used to obtain the optimal solution as discussed in Section 4.2.3.2. Approximation ratio α of 0.01, 0.05, 0.10, 0.15 and 0.20 are considered[7] . We implemented the EDF scheduling simulator along with all the algorithms in C++.

Real Benchmarks Four task sets are constructed for evaluation with each of which consists of real benchmark applications selected from typical embedded system benchmark suites (MediaBench [66], EEMBC [25] and MiBench [35]) as shown in Table 4-1. Task Set

[6] As a remainder, γ is defined in Section 4.2.3.2.

[7] Approximation ratio ε for the maximization version of our problem is calculated as described in Section 4.2.3.2

1 consists of tasks from MediaBench, Set 2 from EEMBC, Set 3 from MiBench, and Set 4 is a mixture from all three suites. We set each task's utilization rate (under the highest voltage level) randomly in the interval of $[\frac{0.5}{m}, \frac{1.5}{m}]$. The accumulated overall utilization rate is controlled to be within [0.7,0.9] for StrongARM and [0.5,0.7] for XScale.

Table 4-1. Task sets consisting of real benchmarks.

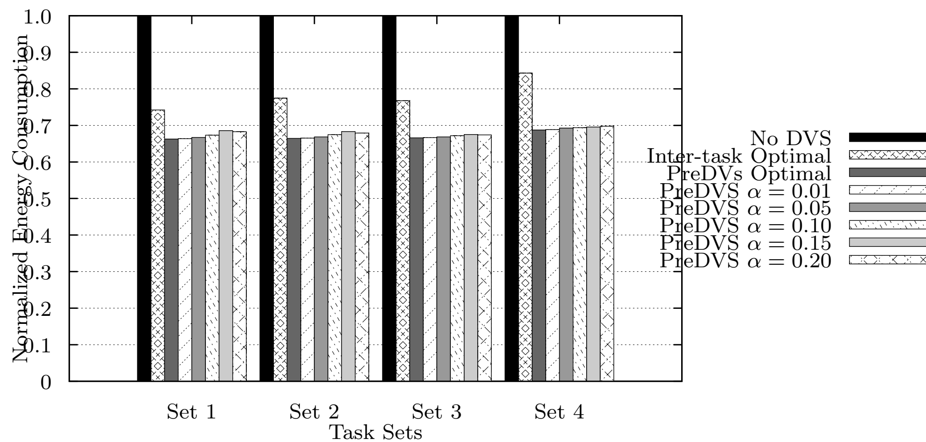
Task sets	**Tasks**
Set 1	cjpeg, djpeg, epic, mpeg2, pegwit, toast, untoast, rawcaudio
Set 2	A2TIME01, AIFFTR01, AIFIRF01, BaseFP01, BITMNP01, IDCTRN01, RSPEED01, TBLOOK01
Set 3	qsort, susan, dijkstra, patricia, rijndael, adpcm, CRC32, FFT, stringsearch
Set 4	cjpeg, epic, pegwit, A2TIME01, RSPEED01, qsort, susan, dijkstra

Synthetic Tasks PreDVS is also evaluated by randomly generated synthetic task sets with 5 to 10 tasks per set with different overall utilization rates. We define the *effective bound* of a DVS processor as the following: any task set with an overall utilization rate equal to or lower than the effective bound can achieve the optimal voltage assignment by trivially choosing the lowest voltage for all the tasks. Clearly, the effective bound is the ratio of the lowest frequency to the highest one. In other words, the effective bound is 0.64 for StrongARM and 0.2 for XScale. Hence, in the former case, we vary the overall utilization rate of each task set from 0.65 to 0.95 at one step of 0.05 while from 0.3 to 0.9 at one step of 0.01 for the latter one. Given each overall utilization rate, we randomly generate task periods in the interval of [100,30000]. Similarly, each task's utilization rate is evenly distributed between $[\frac{0.5*U}{m}, \frac{1.5*U}{m}]$.

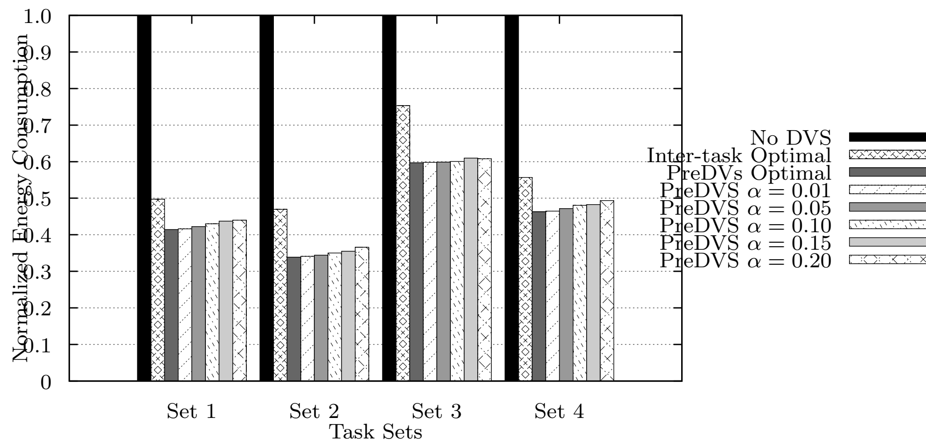
4.4.1.2 Results

Real Benchmarks Figure 4-14 shows the results for real benchmark task sets under both processors. The energy consumption values are normalized to no-DVS case. On StrongARM processor, our approximation scheme saves up to 34% energy compared to no-DVS and outperforms the optimal inter-task scaling by up to 17% even when the

approximation ratio is set to 0.2 ($\alpha = 0.2$). On XScale processor, due to larger span between available voltage levels and lower overall utilization rate, up to 67% energy saving is achieved over no-DVS scenario and on average 19% extra saving than optimal inter-task voltage scaling.



(a) StrongARM processor



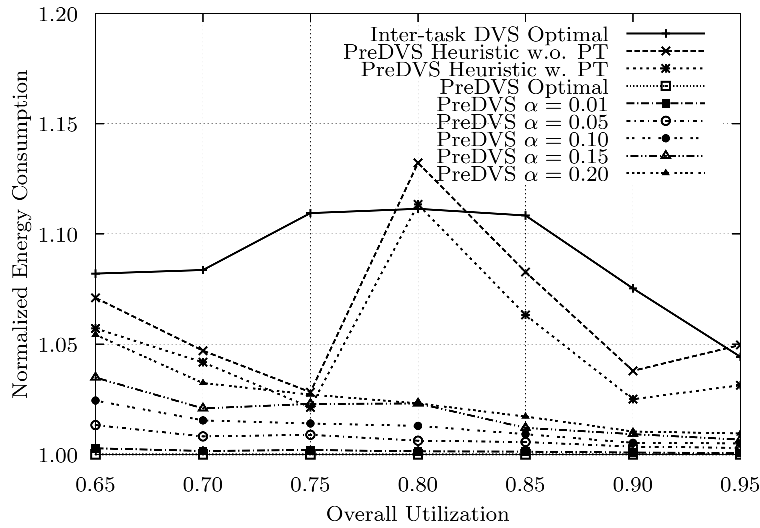
(b) XScale processor

Figure 4-14. Results for real benchmark task sets.

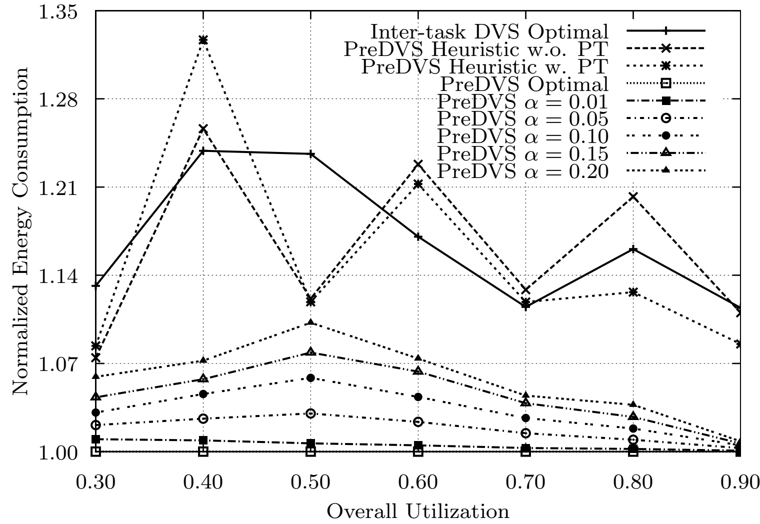
Synthetic Tasks Figure 4-15 shows the results which are the average of 10 randomly generated task sets for each utilization rate on both DVS processors. Here, energy consumption values are normalized to PreDVS optimal solutions. Clearly, in all cases, our approximation algorithm achieves closely approximated overall energy consumption with respect to the optimal solution and outperforms inter-task optimal scaling consistently

up to 12% for StrongARM and 24% for XScale. On average of all scenarios, PreDVS can save 8.7% and 16.7% for both processors, respectively. Figure 4-15 also reveals that our approximation algorithm is capable of generating solutions that are very close to the optimal. For example, for $\alpha = 0.01$, the total energy consumption of the approximated solution is on average merely 0.15% more than the optimal case for StrongARM and 0.51% for XScale. Even for large $\alpha = 0.20$, the actual bound is 2.48% and 5.67% on average, respectively. As shown in the next section, our approximation algorithm is efficient enough in terms of running time when $\alpha = 0.01$. Therefore, we can always expect PreDVS solutions that are no worse than the optimal by 1%.

Our two heuristics proposed in Section 4.2.4 seem to show unpredictable performance in Figure 4-15 at the first glance. At some utilization ratios, they are more energy efficient than optimal inter-task scheduling while in other scenarios are similar or even worse. The reason behind is that the energy saving achieved by heuristics actually depends on the relative position of the ideal optimal speed between its two neighboring available levels s_{above} and s_{below}. For example, the normalized speed levels for StrongARM processor is S$\{1.00, 0.932, 0.786, 0.646\}$. When the system utilization ratio is 0.80, $s_{opt} = 0.80$ is very far away from its $s_{above} = 0.932$ and close to $s_{below} = 0.786$. Although in this case the optimal scaling point is relatively early in each task thus there is more chance to use s_{below}, the sacrifice when s_{above} is used is extremely high that can easily compromise the achieved energy savings. Therefore, generally, in such scenarios when s_{opt} lies in the lower part of $[s_{below}, s_{above}]$, inter-task DVS shows its advantages over PreDVS heuristics. On the other hand, when the system utilization ratio (thus s_{opt}) is close to s_{above}, PreDVS heuristics perform better than optimal inter-task DVS in all task sets. It is because, just as the opposite, using s_{above} only consumes a little bit more energy while lower down to s_{below} saves a lot. In other words, when s_{opt} lies in the upper part of $[s_{below}, s_{above}]$, PreDVS heuristics are preferable over optimal inter-task DVS. Note that optimal inter-task DVS can only be achieved in exponential time while our heuristics have either linear or



(a) StrongARM processor



(b) XScale processor

Figure 4-15. Results for synthetic task sets.

polynomial time complexity. Since the base case utilization η is known a priori, we should easily decide which approach to use.

It is also interesting to observe that the heuristic without problem transformation behaves similarly or sometimes even better than the other heuristic when the overall utilization is low. For example, as shown in Figure 4-15 (b), the former heuristic achieves more energy savings than the later one when $U = 0.4$. However, the scenario becomes

just opposite when $U = 0.8$. The reason behind is that schedule the task set under the optimal speed s_{opt} in Algorithm 6 will lead to more execution blocks compared with the base case schedule particularly when the utilization is low. With higher system utilization, there is less extra number of blocks and the heuristic with problem transformation shows its advantage of finer voltage assignment granularity.

Algorithm Running Time Comparison Table 4-2 compares the running time of optimal inter-task DVS, optimal PreDVS, PreDVS approximation algorithm ($\alpha = 0.01$) and two PreDVS heuristics for four randomly selected task sets. The time spent on problem transformation, which has been included in the relevant columns (heuristic with problem transformation, PreDVS optimal algorithm and PreDVS approximation algorithm), is also listed separately for illustration purpose. For optimal solution, our PreDVS algorithm, as expected, requires longer running time than inter-task DVS. It is because the number of entries (ρ_i) in each task's profile table used by PreDVS is much larger than inter-task DVS which only requires l entries. However, our approximation algorithm can cut down the running time drastically. For example, PreDVS optimal algorithm takes 1.5 hour for a task set with 10 tasks, while the approximation algorithm only requires less than 6 minutes. As a matter of fact, the time spent on problem transformation turns out to be a dominating factor for PreDVS approximation algorithm which itself takes merely a few seconds. Therefore, our approximation scheme takes slightly longer running time than the optimal inter-task DVS. However, as shown in Figure 4-15, it is able to achieve much more energy savings thus is worth the effort. In fact, for the same reason, PreDVS approximation algorithm under different approximation ratio requires very similar running time as shown in Figure 4-16. Hence, small approximation ratio (e.g., $\alpha = 0.01$) can always be efficiently used to minimize energy consumption without excessive increase in running time. The heuristic without problem transformation for PreDVS (Section 4.2.4.1) takes almost negligible running

time while the other one (Section 4.2.4.2) requires problem transformation thus consumes running time close to the approximation algorithm.

Table 4-2. Algorithm running time comparisons (in seconds).

Task Set	Inter-task Optimal Algorithm	Heuristic w.o. PT	Heuristic w. PT	PreDVS Optimal Algorithm	PreDVS Approximation Algorithm ($\alpha = 0.01$)	(Problem Transformation)
1	29.8	0.4	294.9	2418.0	299.1	(294.7)
2	36.2	0.5	385.5	4986.8	393.8	(385.2)
3	31.2	0.3	115.1	2215.8	121.2	(114.9)
4	39.7	0.4	374.6	5465.0	382.9	(374.3)

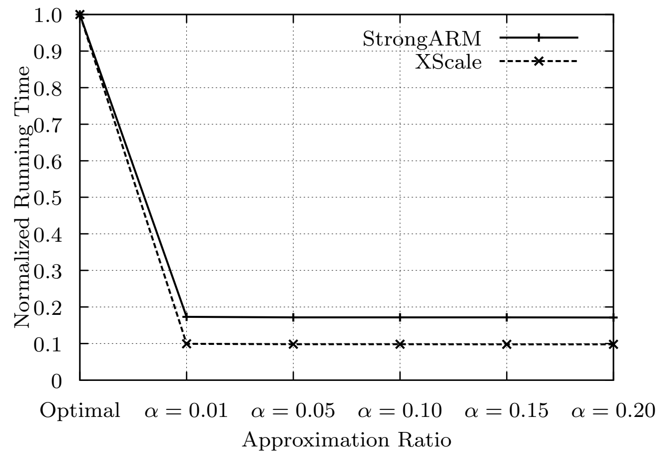


Figure 4-16. PreDVS Approximation Algorithm Running Time Comparison.

Discussion While choosing among proposed techniques, designers can trade-off between solution quality and running time. In case of small problems, it may be possible to generate PreDVS optimal solution in reasonable amount of time. However, for large piratical systems, our PreDVS approximation algorithm is recommended. In case the time requirement for approximation algorithm is not acceptable (e.g., during early design space exploration), our first heuristic is most suitable unless optimal speed is in the lower part of the range $[s_{below}, s_{above}]$. If s_{opt} is in the lower part, one can choose between optimal inter-task DVS and PreDVS approximation algorithm depending on expected design quality and running time. When problem transformation time dominates, our

approximation algorithm is preferable, otherwise, in some extreme case, our second heuristic is suitable.

4.4.2 DSR

4.4.2.1 Experimental Setup

We evaluate our dynamic slack reclamation algorithms through simulation using two DVS-capable processors: Marvell StrongARM processor [74] and Transmeta Crusoe processor [113]. The former one supports four voltage - frequency levels (1.5V - 206MHz, 1.4V - 192Mhz, 1.2V - 162MHz and 1.1V - 133MHz) and its characteristics are collected from manufacturer's datasheets. The latter one has scalable voltage level from 1.1V to 1.5V in steps of 0.1V. Its operating frequency and power consumption values are calculated by the detailed energy model described in Chapter 2. We use a static slack allocation algorithm adapted from [5]. Voltage/frequency assignment for task τ_i is the one with minimum energy consumption but has execution time no longer than $min(t_i^h/U, t_i^{crit})$, where t_i^h and t_i^{crit} is the execution time under the highest frequency level and the critical speed, respectively. Our proposed dynamic slack reclamation algorithm is implemented with a discrete-event simulator written in C++.

Real Benchmarks We also evaluate our approach using real benchmarks selected from MediaBench [66], MiBench [35] and EEMBC [25] to from four task sets as shown in Table 4-3. Task Set 1 consists of tasks from MediaBench, Set 2 from EEMBC, Set 3 from MiBench and Set 4 is a mixture of all three suites. In Set 4, the two benchmarks from EEMBC are set to iterate 100 times in order to make their size comparable with others. We use SimpleScalar [14] as the underlying micro-architectural simulator to get the number of cycles for each task execution to act as its WCET. We define λ as the probability for a job to finish earlier than its WCET. If a job completes earlier, its $ACET$ is generated using a normal distribution with a mean of $(BCET + WCET)/2$ and a standard deviation of $(WCET - BCET)/6$. $BCET$ for each task is based on a percentage

of its $WCET$ and is varied from 10% to 100% in steps of 10%. Let δ denote the value of $BCET/WCET$.

Table 4-3. Task sets consisting of real benchmarks.

Sets	Tasks
Set 1	cjpeg, pegwit, untoast, epic, mpeg2
Set 2	A2TIME01, BaseFP01, BITMNP01, RSPEED01, TBLOOK01
Set 3	susan, dijkstra, rijndael, qsort, stringsearch
Set 4	cjpeg, pegwit, A2TIME01, RSPEED01, pktflow, dijkstra

Synthetic Tasks We consider seven randomly generated synthetic task sets. Each set consists of 3 to 10 tasks. The workload of each task under the highest voltage level and the period (for periodic tasks) or inter-arrival time (for non-periodic tasks) are randomly chosen within pre-determined ranges so that at any moment U is maintained under the schedulability constraint (e.g. 1.00 for EDF). For each task set, we vary U from 0.6 to 0.9 in steps of 0.1. Task sets are formed with similar characteristics as real benchmark task sets.

4.4.2.2 Results

Window Size Effect We first show the effect of adjusting the exploration window size in DSR algorithm. Here, the window size is varied from 1 to 10 with $U = 0.8$ and $\delta = 20\%$. Figure 4-17 shows the average results over all synthetic task sets assuming no arrival time constraints on StrongARM processor. It shows that window size of 4 or 5 is good enough to capture most of the energy savings. Furthermore, larger window size also lead to more overhead and a higher chance that some blocks that are allocated slacks finish earlier than expected. This can compromise the total energy saving achieved. Therefore, we use window size of 4 in the following experiments.

Energy Saving Comparison To illustrate effectiveness of our approach, we compare the following three techniques across different values of U and δ as discussed above:

- **No-DSR**: Tasks are executed based on the static scheduling and no dynamic slack reclamation is utilized.

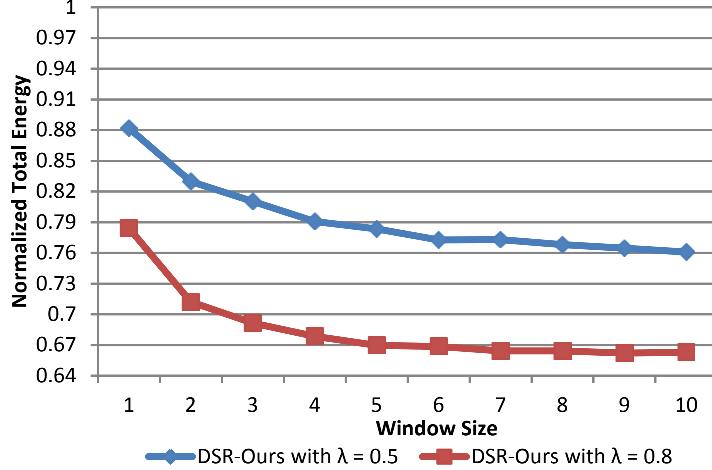
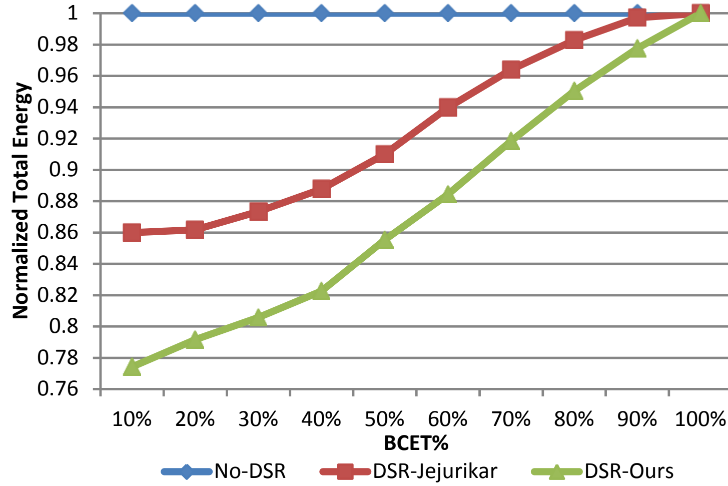


Figure 4-17. Effect of Window size on the total energy savings.

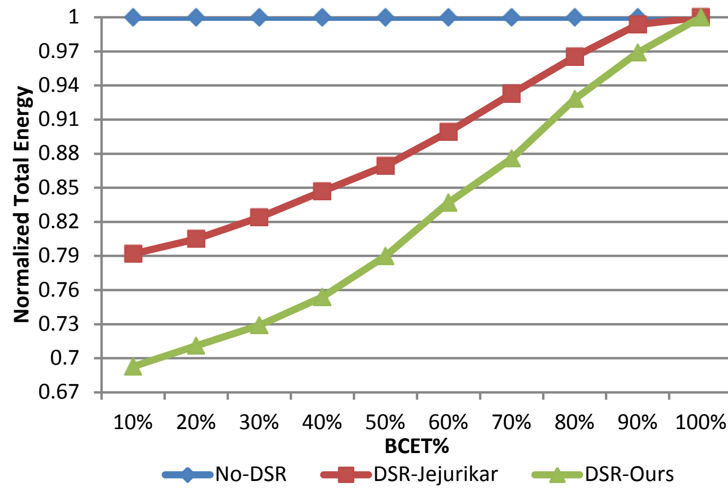
- **DSR-Jejurikar**: Dynamic slack reclamation algorithm proposed in [48].
- **DSR-Ours**: Our approach on dynamic slack reclamation.

Synthetic Tasks: Figure 4-18 shows energy comparison results using synthetic task sets on StrongARM processor. We examine both scenarios of (a) $\lambda = 0.5$ and (b) $\lambda = 0.8$ with window size of 4. The total energy consumption values for all techniques are normalized to No-DSR scenario. These have been averaged over multiple runs of all task sets and all values of U, where each run consists of a combination of a task set and a value of U. For both values of λ, our approach can achieve average energy savings of 14% and 18% over No-DSR (can be as large as 23% and 31% when $\delta = 10\%$). Our approach also outperforms DSR-Jejurikar across all δ values by 2 - 12% in terms of total energy requirements. In practice, the ACET of a program is smaller than its WCET by at least 80% (i.e. $\delta = 20\%$) [5], especially when the WCET estimation is pessimistic. In such cases, our technique can reduce the energy consumption by more than 10% compared with the state-of-the-art algorithm.

Figure 4-19 and 4-20 show the results for the same set of experiments on Transmeta Crusoe processor with constant effective capacitance and application-specific effective capacitance, respectively. For the latter case, we randomly generate $\mathcal{K}$ in the energy model within a range of $[0.2, 1.0]$ for each task. In other words, we have $C_{eff} \in [0.2 \cdot C_{total}, C_{total}]$.



(a) $\lambda = 0.5$

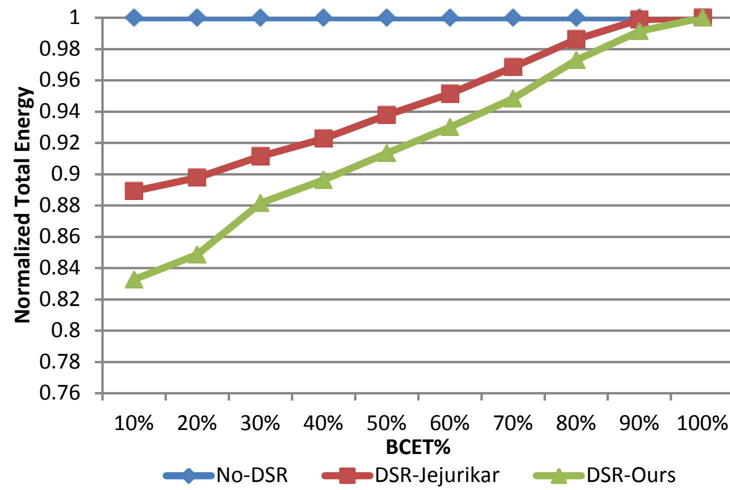


(b) $\lambda = 0.8$

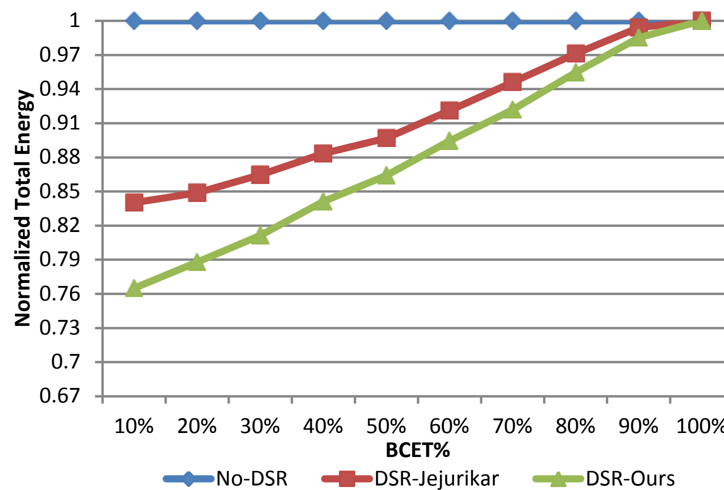
Figure 4-18. Results for StrongARM processor (synthetic task sets).

In both scenarios, it can be observed that energy savings are less significant than StrongARM processor. It is possibly due to the fact that leakage energy consumption is much higher in 70nm technology. Therefore, the energy reduction created by DSR (lower subsequent job's voltage level) decreases. However, our approach still consistently outperforms DSR-Jejurikar. Another important observation is that in the scenario where tasks have different effective capacitance (C_{eff}), our approach can result in more additional energy savings compared with DSR-Jejurikar. The reason is that

application-specific C_{eff} leads to more variation in task's energy saving potential during dynamic slack reclamation, which clearly makes our approach more beneficial.



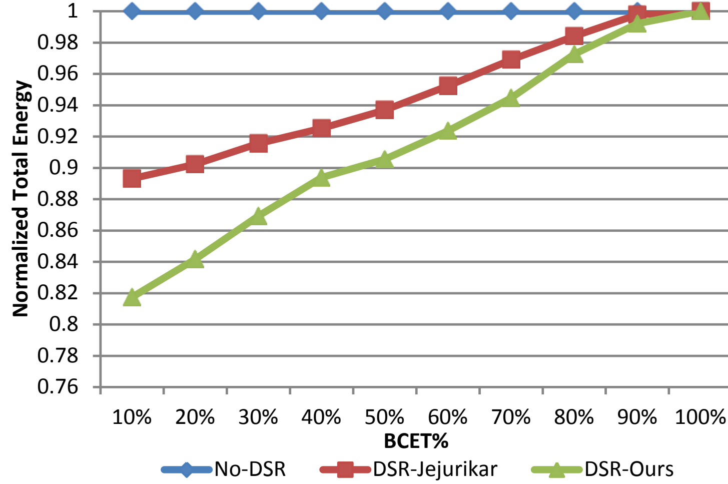
(a) $\lambda = 0.5$



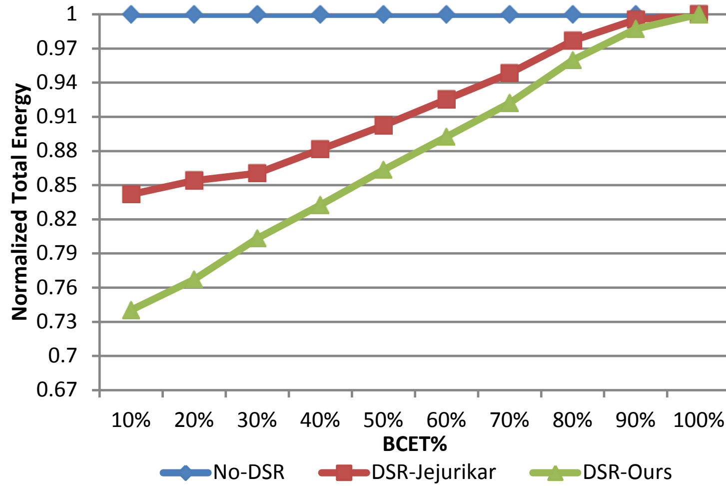
(b) $\lambda = 0.8$

Figure 4-19. Results for Transmeta Crusoe processor with constant effective capacitance values (synthetic task sets).

Real Benchmarks: Figure 4-21 shows total energy consumption comparisons across four real benchmark task sets with $\delta = 10\%$ and (a) $\lambda = 0.5$, (b) $\lambda = 0.8$ on Transmeta Crusoe processor. Here, similar observation can be made as shown in Figure 11. On average, 7% and 10% extra savings in total energy consumption can be achieved in both scenarios, respectively.



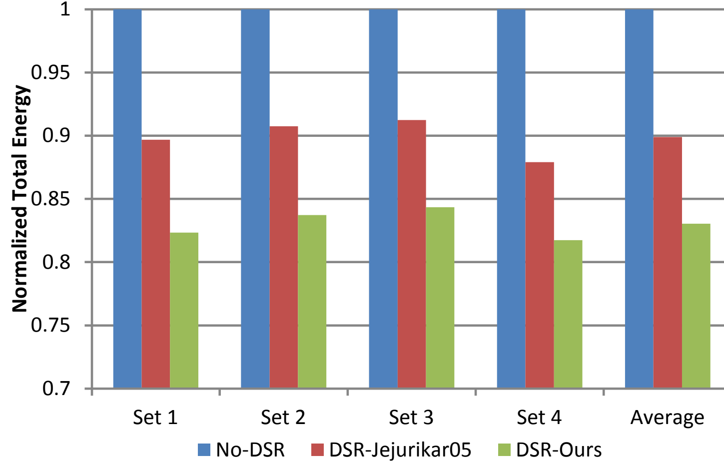
(a) $\lambda = 0.5$



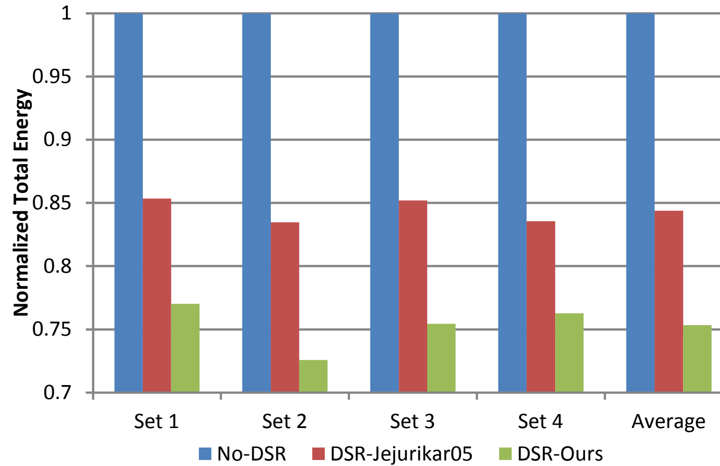
(b) $\lambda = 0.8$

Figure 4-20. Results for Transmeta Crusoe processor with application-specific effective capacitance values (synthetic task sets).

Problem Variations To demonstrate the breadth of applicability of our approach, we compare the experimental results for the following three scenarios: 1) **No-AT**: task sets without arrival time constraints (Section 4.3.2.1); 2) **AT-NoRS**: Tasks with arrival time constraints but task rescheduling is not allowed (Section 4.3.2.2); 3) **AT-RS**: Tasks with arrival time constraints but task rescheduling is applied (Section 4.3.2.2). λ and U are set to 0.8. It can be observed from Figure 4-22 that task rescheduling is very effective and can achieve energy savings very close to No-AT. Thus, our approach is able to exploit the



(a) $\lambda = 0.5$



(b) $\lambda = 0.8$

Figure 4-21. Results for Transmeta Crusoe processor with application-specific effective capacitance values (real benchmark task sets).

available slack effectively even when significant constraints on task rescheduling and arrival times are considered.

Running Time Overhead We also investigated the runtime overhead of our DSR algorithm for all three scenarios above. Figure 4-23 shows the average running time requirement (of one time of dynamic slack reclamation) over all task sets with λ and δ equal to 0.8 and 0.2, respectively. The window size is varied from 1 to 10. We can observe that the running time overhead of AT-RS is very low (e.g. less than one fourth millisecond

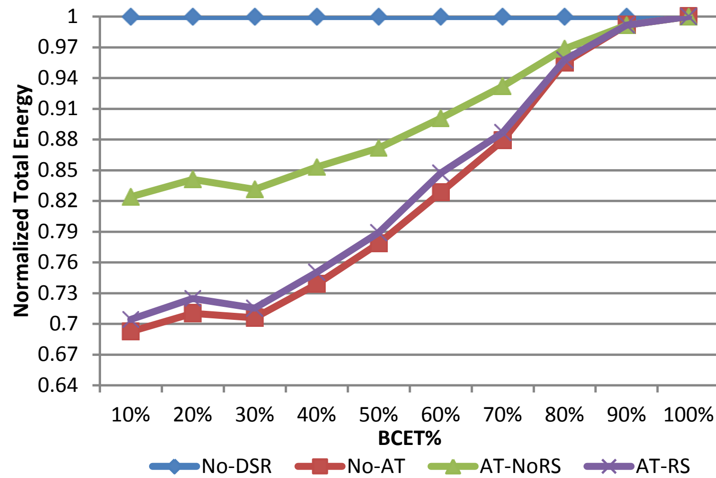


Figure 4-22. Problem variations comparison.

for window size of 4). Therefore, our algorithm is efficient enough at runtime for normal task sets which normally takes hundreds of milliseconds [138].

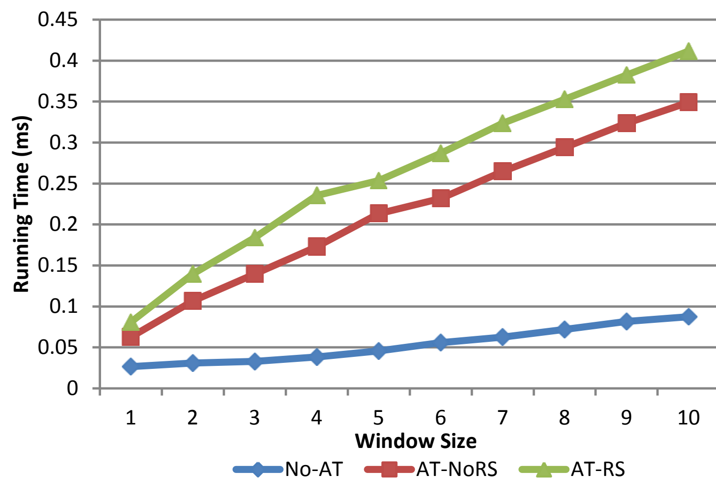


Figure 4-23. Running time overhead.

4.5 Summary

This chapter presented processor energy optimization techniques based on dynamic voltage/frequency scaling and task scheduling in real-time systems. Approaches are proposed for both static slack allocation and dynamic slack reclamation.

For the former problem, we presented a preemptive dynamic voltage scaling scheme – PreDVS – which can achieve significant energy savings by assigning different voltage

levels to each task instance. PreDVS does not introduce any additional voltage switching overhead compared to inter-task scaling techniques. Moreover, it exploits static time slack only and thus can be employed together with any existing intra-task scaling techniques. We showed that the problem is NP-hard and presented an approximation scheme by developing a novel transformation mechanism and a fully polynomial time approximation algorithm. We also proposed two efficient heuristics that can lead to significant energy improvement over optimal inter-task DVS in certain predictable cases. The approximate solutions given by our approach outperforms optimal inter-task scaling techniques by up to 24%. Experimental results demonstrated that PreDVS can generate solutions very close to the optimal.

For the later problem, we presented a dynamic slack reclamation algorithm for energy-aware scheduling in real-time multitasking systems. Our approach aims at minimizing total energy consumption, both dynamic and leakage, when some tasks finish earlier than their worst case. Unlike existing techniques, we systematically allocate the available slack among multiple jobs and apply task rescheduling whenever it is beneficial. By restricting the exploration window, tradeoffs can be made between solution quality and runtime overhead. Experimental results show that our approach can achieve significant energy saving over static energy-aware scheduling and also outperforms state-of-the-art technique by up to 12%.

CHAPTER 5
SYSTEM-WIDE ENERGY OPTIMIZATION WITH DVS AND DCR

In Chapter 3 and 4, we elaborate our approaches and algorithms for employing DCR and DVS in real-time systems separately. However, as shown in Figure 1-2, both processor and cache subsystem as well as other components contribute to the system's overall power dissipation. Therefore, it will be important and promising to employ DVS and DCR simultaneously to achieve system-wide energy optimization.

In the last decade, we have observed a continuous CMOS device scaling process in which higher transistor density and smaller device dimension have led to increasing leakage (static) power consumption. This is mainly due to the proportionally reduced threshold voltage level with the supply voltage which decreases along with the power supply at a speed of 0.85X per generation [24]. Lower threshold voltage results in larger leakage current which mainly consists of subthreshold current [15] and reverse bias junction current [73]. Study has shown that leakage power is increased by about five times in each technology generation [11]. It is responsible for over 42% of the overall power dissipation in the 90nm generation [55] and can exceed above half of the total in recent 65nm technology [24] [59]. On-chip caches nowadays contribute a significant share of the system leakage power. Static energy is projected to account for near 70% of the cache subsystem's budget in 70nm technology [59]. Furthermore, higher temperature have adverse impact on leakage power in both processor [133] and cache [81]. Leakage power also constitute a major fraction of the total consumption from on-chip buses – almost comparable to dynamic part even when leakage control scheme is employed [91]. Memory modules, both with DRAM and SRAM, can also consume significant amount of leakage power (i.e., standby power) [53] [41], especially when DVS is employed so that the standby time of those components increases [47] [140]. Therefore, decisions should be made judiciously on whether to slow down the system to save dynamic power or to finish task execution faster and switch the system to sleep mode to reduce static power.

While existing techniques try to control the leakage power along with DVS [51], extra consideration needs to be taken when DCR is also employed and other system components are taken into account.

The proposed research in this chapter integrates DVS and DCR together in hard real-time systems to minimize system-wide energy consumption. The main contribution is that, unlike existing DVS approaches which either ignore various system components other than the processor or assume application-independent constant power consumption values, we systematically incorporate power consumptions from the processor, cache hierarchy, system buses and main memory based on the same set of application simulation statistics. The power estimation framework that we propose uses separate power analyzers for different system components. We take a step forward by examining the correlation among the energy models of all the components and find that they have significant impact on the decision making of both DVS and DCR. Based on the analysis and observations, DVS and DCR decisions can be made at design time, and task procrastination is carried out at runtime to achieve more idle energy savings. We also propose a general algorithm for dynamic reconfiguration in real-time multitasking systems. This algorithm takes varying runtime reconfiguration overhead into account and can be flexibly parameterized to make tradeoff between energy saving and running complexity.

The rest of this chapter is organized as follows. Related works are discussed in Section 5.1. Section 5.2 presents our system-wide leakage-aware energy optimization algorithm based on both DVS and DCR. Section 5.3 describes the general algorithm for dynamic reconfiguration in real-time multitasking systems. Section 5.4 provides experimental results for evaluating the proposed approaches. Section 5.5 summarizes this chapter.

5.1 Related Work

A great deal of research work exists on applying DVS in real-time systems. A lot of them focus on minimizing dynamic power consumption and ignoring the static portion by

slowing down the processor as much as possible through various directions including task scheduling, voltage selection and worst-case execution time estimation [76] [16] [50] [137] [83]. Meanwhile, a number of existing works pay attention to control processor leakage power in real-time systems [67] [51] [47] [140] [19] [20] [121]. Lee et al. [67] propose a scheduling algorithm to minimize leakage energy consumption by procrastinating currently ready tasks to enlarge system idle periods based on a non-DVS platform. Jejurikar et al. [51] present a leakage-aware DVS scheme which does not allow to slow down the processor speed below a certain level called *critical speed* to avoid growing static energy consumption to compensate the reduction in dynamic energy. They also propose a procrastination scheduling technique to maximize processor idle intervals. Chen et al. [19] address the same problem in a rate-monotone scheduling system. They also propose a procrastination scheme based on energy consumption evaluation [20]. However, none of the above techniques considered DCR. Furthermore, they did not take other system components including cache hierarchy, bus and memory into account which potentially limits the benefit of their approaches. Jejurikar et al. [47] and Zhong et al. [140] proposed leakage-aware DVS techniques for system-wide energy minimization. However, the system components considered in their work (e.g. memory, flash drives) are only mock units whose power consumptions are assumed to be application-independent constants. In other words, their approaches use over-simplified models and cannot incorporate other power-hungry components including cache and buses whose power dissipations are determined by the application executing at runtime. Moreover, cache reconfiguration is not considered by any of them.

Micro-architecture level techniques are proposed at the aim of saving leakage energy in cache subsystem by switching unused cache sub-arrays into low-power mode [85] [60]. Chi et al. [22] applied these techniques in hard real-time systems. Data compression is also proposed for cache energy reduction in [116] [36]. However, none of these approaches takes processor voltage scaling or other system components into consideration. Nacul et al.

[80] presented preliminary results to demonstrate the benefit of combining DVS and DCR together in real-time systems but they did not consider leakage power which may make their solution inferior when leakage energy dominates the total consumption.

5.2 System-wide Leakage-aware DVS and DCR

Our approach addresses major challenges including design space exploration, system-wide energy analysis, configuration selection and task procrastination to significantly reduce overall energy consumption while meeting all task deadlines. Figure 5-1 illustrates the workflow of our approach. Each real application in the task set is fed into a simulation process which is driven by a design space exploration heuristic. For each simulation, its total system energy consumption is calculated by our energy estimation framework and put into the task's profile table along with the corresponding execution time. Based on the task set characteristics and the profile tables as well as the scheduling policy, processor voltage level and cache configuration can be selected for each task. Task DVS/DCR assignments and procrastination algorithm are then used in a one-pass task scheduling which produces the total energy consumption of the task set during its hyper-period P. Extensive experiments show that our approach can result on average 47.6% energy savings compared to DVS-only systems and up to 23.5% extra savings compared to leakage-oblivious DVS + DCR technique [80]. This section describes each of these steps in detail.

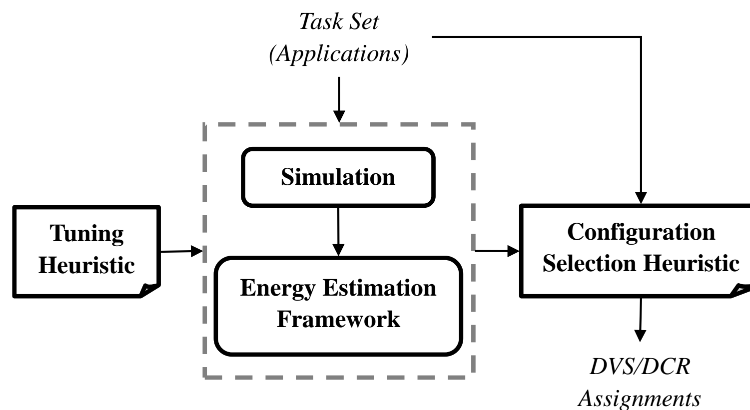


Figure 5-1. Workflow of our approach.

5.2.1 Power Estimation Framework

Since we do not focus on system design which requires to minimize development time and costs, our energy estimation framework, as shown in Figure 5-2, targets at a specific SoC micro-architecture and is able to trade more design time for higher accuracy than the one proposed in [29] [111]. We use SimpleScalar [14] as the underlying micro-architectural simulator in our approach. For each application (task) and cache configuration, we run a simulation and collect the execution statistics, memory access statistics and bus activity traces. These information, along with the processor voltage levels, are provided to energy models for each system components, based on which the total system energy can be computed. Note that in our framework, the inputs to each energy model are all from one single micro-architectural simulation thus are more comprehensive and systematic, as opposed to [29] and [111] in which the inputs are collected separately using instruction-set simulator, memory trace profiler, cache simulator and bus simulator. Furthermore, by doing this, the impact on DVS/DCR decisions from other system components as well as their correlations, which is not considered in [29] and [111], can be reflected in an accurate manner. This framework still provides flexibility to allow different energy models and analyzers to be used.

We consider on-chip separate L1 caches, an off-chip unified L2 cache and DRAM memory. As shown in Figure 5-3, system buses consist of a 32-bit address bus and a 32-bit data bus between processor and L1 caches, L1 caches and L2 cache, as well as L2 cache and the main memory. Hence, there are totally 8 bus lines. For on-chip bus lines, their frequency must match with the processor otherwise instructions and data cannot be fetched on time from L1 caches. Off-chip bus lines are assumed to have constant and lower frequencies. As indicated by the arrows in Figure 5-3, bit streams flowing between those components have their directions. For example, the data bus between processor and IL1 cache only carries fetched instructions while the data bus between processor and DL1 cache also carries the data to be written issued by processor. We modified

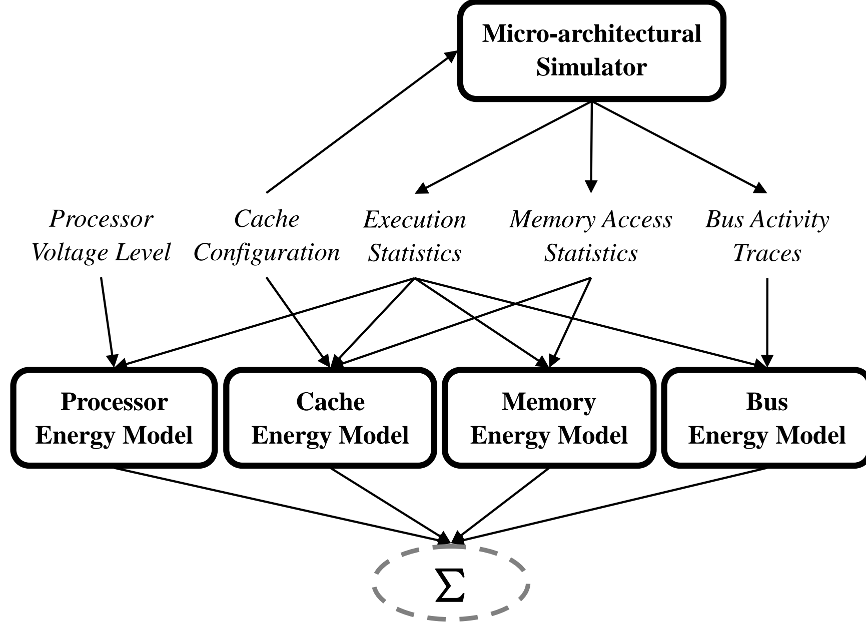


Figure 5-2. Overview of our power estimation framework.

SimpleScalar [14] to record activity traces (bits transmitted) for all bus lines during program execution, which is used in the bus energy model to calculate average transitions n_{trans} in Equation (2–12). Note that bus energy model calculates the energy consumption of each bus line separately. Obviously, which is also shown by our studies, DCR have major impact on off-chip bus's activities which mainly come from L1/L2 cache misses. However, on-chip bus activities, which are sourced from the processor, are not affected by DCR.

5.2.2 Two-Level Cache Tuning Heuristic

As discussed in Chapter 3, it is a major challenge to employ multi-level cache reconfiguration since the exploration space is prohibitively large. Section 3.3 describes efficient tuning heuristics for two-level cache hierarchy which can be applied here. We use L1 cache sizes of 4KB, 8KB and 16KB, line size of 16, 32 and 64 bytes with set associativity of 1-way, 2-way and 4-way. For L2 cache, the capacity is selected to be 32KB, 64KB and 128KB with line sizes of 64, 128 and 256 bytes and set associativity of 4-way, 8-way and 16-way. Therefore, there are 18 configurations for each individual cache and totally 5832 different configurations for the cache hierarchy [119]. We employ ILOT –

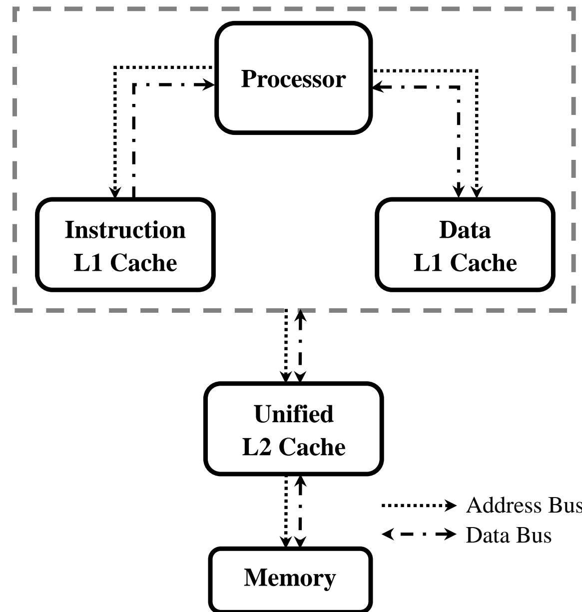


Figure 5-3. Conceptual system bus architecture.

Independent Level One Cache Tuning – here to reduce the simulation time while still preserve the most amount of accuracy. Note that other heuristics described in Section 3.3 are also applicable.

5.2.3 Critical Speed

The *critical speed* for processor voltage scaling defines a point beyond which the processor speed cannot be slowed down otherwise DVS will no longer be beneficial [51]. The dynamic power consumption of processor, which is exclusively considered in traditional DVS, is usually a convex and increasing function of the operating frequency. However, since lowering processor speed makes the task execution time longer which leads to higher static energy consumption, the total energy consumed per cycle in the processor will start increasing due to further slowdown.

By taking DCR into consideration, we find that cache configuration has significant impact on the critical speed with respect to the overall system energy consumption. Note that as described in Chapter 2, there exists strong correlation among the energy models of processor, cache subsystem and other system components. Since different cache configurations lead to different miss ratios and miss penalty cycles, the number of clock

cycles (CC) required to execute an application is decided by the cache configuration, which directly affects the energy consumption of other components, as shown in Equation (2–11), (2–14) and (2–16). On the other hand, the length of each clock cycle (t_{cycle}), which is determined by the processor voltage/frequency level, also directly affects the energy consumption of other components as shown in Equation (2–3), (2–14) and (2–16). In other words, DVS and DCR will affect the overall system energy consumption. On the other hand, due to leakage power, all system components will have impact on decision making of DVS/DCR, especially the critical speed. Specifically, when the processor is slowed down by DVS, increasing static energy consumed by cache hierarchy, bus lines and memory will compromise the benefit gained from reduced processor dynamic energy thus lead to higher system-wide energy dissipation. Therefore, considering DCR and other system components effects, we found that the critical speed is going to increase drastically.

5.2.3.1 Processor + L1 Cache

We use a motivating example in which a single benchmark[1] (*cjpeg* from MediaBench [66]) is executed under all processor voltage levels. It can be observed that in Figure 5-4, when only processor energy is considered, the critical speed is achieved at $V_{dd} = 0.7V$, which matches the results in [51]. However, as shown in Figure 5-5, with respect to the total amount of energy consumption, combining processor and L1 caches (both configured to 8KB of capacitance, 32B line size and 2-way associativity) increases the critical speed slightly to around $V_{dd} = 0.75V$, due to the effect from L1 cache's leakage power dissipation. This highlights the importance of considering other system components for accurate analysis when applying DVS. In other words, if L1 caches are incorporated, $V_{dd} = 0.7V$ is no longer a beneficial choice with respect to the overall energy savings. Note

[1] Although results for *cjpeg* is shown in this section, similar observations have been made for other benchmarks.

that in Figure 5-5, dynamic energy consumption of L1 caches only includes access energy E_{access} and block refilling energy E_{block_fill}. Energy consumed on buses and lower-level memory hierarchy during L1 cache misses will be incorporated when we gradually add the corresponding components into consideration, as shown in following sections.

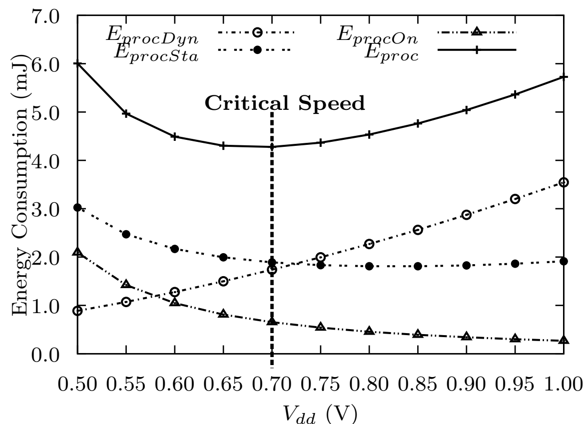


Figure 5-4. Processor energy consumption E_{proc} for executing *cjpeg*: $E_{procDyn}$ is the dynamic energy, $E_{procSta}$ is the static energy and E_{procOn} is the intrinsic energy needed to keep processor on.

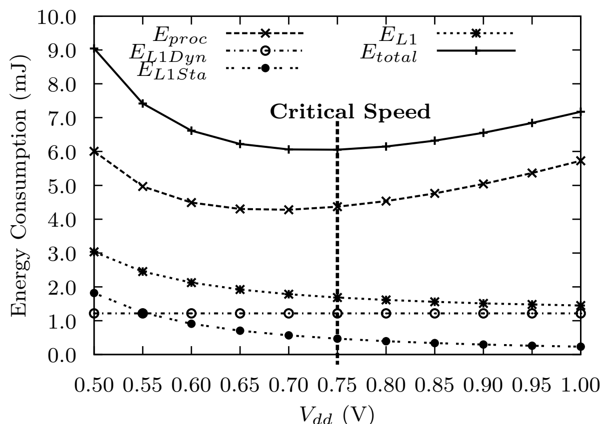


Figure 5-5. Overall system energy consumption E_{total} of the processor and L1 caches for executing *cjpeg*: E_{L1Dyn} and E_{L1Sta} are the dynamic and static L1 cache energy consumption, respectively.

5.2.3.2 Processor + L1/L2 Cache

Figure 5-6 shows the impact on the critical speed if L2 cache (with capacity of 64KB, line size 128B and 8-way associativity) is considered in the overall energy consumption.

The critical speed increases to the frequency corresponding to $V_{dd} = 0.85V$. For L1 caches, as shown in Figure 5-5, dynamic energy dominates and leakage energy becomes comparable only when the processor voltage level drops below $0.6V$. However, in L2 cache, for *cjpeg*, leakage energy dissipation dominates while dynamic energy is almost negligible. It is expected since L1 access rate is much higher than L2 while the capacity, thus leakage power, of L2 cache is much larger than L1. Note that, although some other benchmarks (e.g. *qsort* from MiBench [35]) shows non-negligible dynamic energy consumption in L2 cache, the leakage part still dominates when the voltage level goes below a certain point. Therefore, when processor voltage decreases, the total leakage energy consumption increases drastically due to the L2 cache. Generally, when DCR is applied, different cache configurations will lead to different critical speed variations.

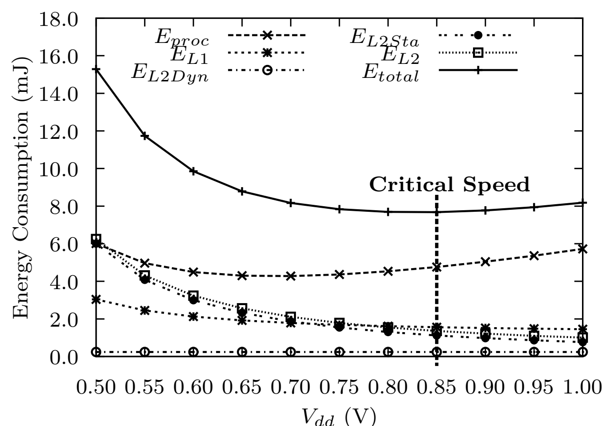


Figure 5-6. Overall system energy consumption E_{total} of the processor, L1 caches and L2 cache (configured to 64KB,128B,8-way) for executing *cjpeg*: E_{L2Dyn} and E_{L2Sta} are the dynamic and static L2 cache energy consumption, respectively.

5.2.3.3 Processor + L1/L2 Cache + Memory

Figure 5-7 illustrates the fact that memory energy consumption also makes the critical speed increase. The memory we considered is modeled as a common DRAM with size of 8MB. It can be observed that memory has a similar effect on the critical speed as L2 cache. In fact, for the configurations we used, the static energy consumptions are comparable for L2 cache and the memory. Although DRAM needs to have its

capacitor charge refreshed all the time (which consumes relatively negligible power in 70nm technology [41]), it requires only one transistor to store one bit. Therefore, it consumes much less leakage power per bit compared to cache, which is smaller but made of more power expensive SRAM.

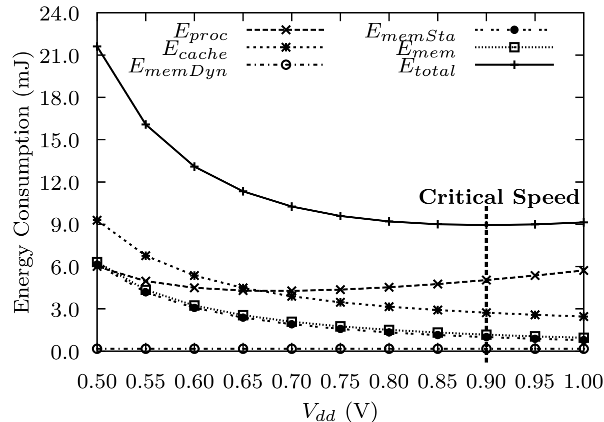


Figure 5-7. Overall system energy consumption E_{total} of the processor, L1/L2 caches and memory for executing *cjpeg*: E_{memDyn} and E_{memSta} are the dynamic and static memory energy consumption, respectively; E_{cache} represents the total energy consumption of both L1 and L2 caches.

5.2.3.4 Processor + L1/L2 Cache + Memory + Bus

System bus lines, as described in Section 5.2.1, have double effect on the critical speed in overall system energy consumption. On one hand, since on-chip buses should have equal frequency as the processor (which makes them dominate in terms of energy among all system buses), DVS will lead to dynamic energy reduction in them. On the other hand, like other system components, static power dissipation on system buses is also going to increase along with voltage scaling down, which compensates the dynamic energy reduction. As a result, system buses make very minor impact on critical speed as shown in Figure 5-8.

For ease of demonstration, we show how energy consumption (both dynamic and static) of each system components vary with voltage scaling in Figure 5-9. When DVS is not applied ($V_{dd} = 1V$), the processor accounts for over half of overall energy consumption while others also take considerable share. This observation matches what we have shown

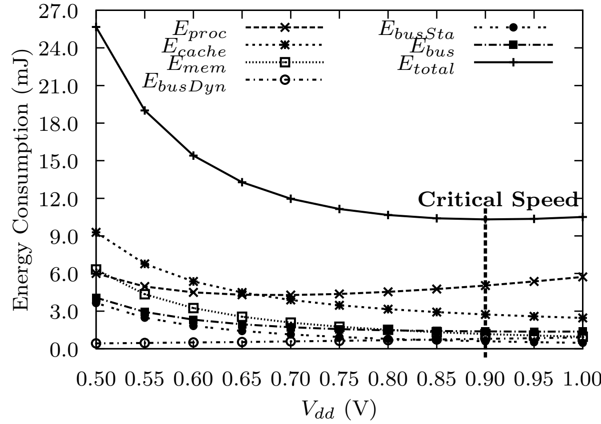


Figure 5-8. Overall system energy consumption E_{total} of the processor, L1/L2 caches, memory and system buses for executing *cjpeg*: E_{busDyn} and E_{busSta} are the dynamic and static bus energy consumption, respectively.

in Figure 1-2. When the voltage level decreases, we can see that the energy consumed by the cache hierarchy and memory subsystem increases drastically and, after certain point, it becomes comparable with the processor or even larger. For system buses, due to the reason explained above, this effect is less significant compared to cache and memory.

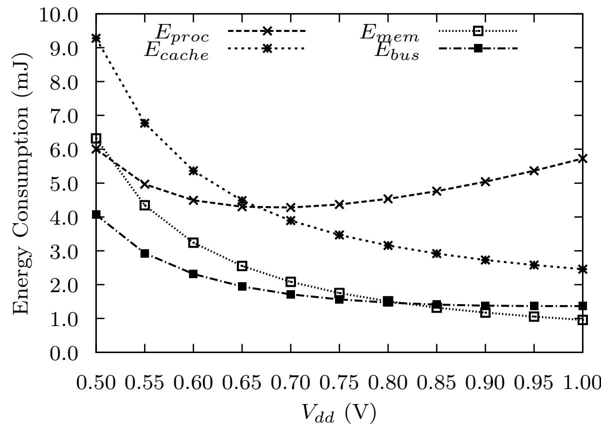


Figure 5-9. Processor voltage scaling impact on various system components.

In conclusion, the discussion above leads to several interesting observations and important questions. The critical speed is going to change as different system components are considered – increases when leakage energy dominant components are added and decreases when dynamic energy dominant components (DVS-controllable) are added. One

would wonder whether DVS is really practically beneficial since our case study shows that the critical speed is at $V_{dd} = 0.9V$ and potentially adding more components may increase it further (possibly close to $1.0V$)? A simple answer is yes but it has to be evaluated using leakage-aware DVS and DCR. It is also important to notice that system properties, application characteristics and reconfiguration decisions together will affect the critical speed, which typically varies between $V_{dd} = 0.65V$ and $0.9V$ in our case.

5.2.4 Real-Time Voltage Scaling and Cache Reconfiguration

This section briefly describes three important aspects in performing real-time DVS and DCR: profile table creation, configuration selection and task procrastination.

5.2.4.1 Profile Table

We define a *configuration point* as a pair of processor voltage level and cache hierarchy configuration: (v_j,c_k) where $v_j \in$ V and $c_k \in$ C. Note that each c_k represents a configuration of the cache hierarchy which includes L1 caches and L2 cache. For each task, we can construct a *profile table* which consists of all possible configuration points as well as the corresponding total energy consumption $E_i(v_j, c_k)$ and execution time $T_i(v_j, c_k)$. Clearly, all points with the voltage level lower than the critical speed are eliminated. Furthermore, non-beneficial configuration points, which is inferior in both energy and time compared to some other points, are also discarded. In other words, we only consider those Pareto-optimal tradeoff points.

An important observation is that cache configurations behave quite consistently across different processor voltage levels. For example, as shown in Figure 5-10, the L1 cache configuration favored by *cjpeg*, 8KB cache size with 32B line size and 2-way associativity, outperforms all the other configurations with respect to the total energy consumption. Similar observations can be made when we fix L1 cache configuration while vary L2 cache. Therefore, the profile table for each task actually consists of favored cache configuration combinations with voltage levels equal to or higher than the corresponding critical speed. In fact, we find that only the most energy-efficient cache hierarchy configuration with

the voltage level equal to or higher than the critical speed exist in the profile table with slightly lower energy consumption but much longer execution time compared with other entries. It can be explained, for example, as shown in Figure 5-8, that E_{total} only decreased by 1.78% when V_{dd} is lowered down from $1V$ to $0.9V$ but the execution time is increased by 27.45%. In other words, generally speaking, the energy reduction caused by DVS is not worth the loss in performance when the voltage level is close to the critical speed.

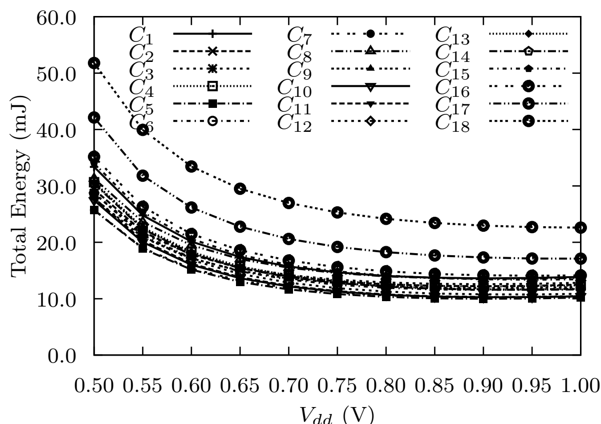


Figure 5-10. Total energy consumption across all L1 cache configurations (with L2 cache configured to 64KB,128B,8-way) for executing *cjpeg*.

5.2.4.2 Reconfiguration Selection Heuristics

Existing DVS algorithms for hard real-time systems are not applicable when DCR is employed since the energy consumption as well as the impact on task's execution time from system components other than the processor cannot be simply calculated from energy models. DCR algorithms proposed in Chapter 3 are also not applicable since they only support soft task deadlines. Given a static slack allocation, we can assign the most energy efficient configuration point which does not stretch the execution time over the allocated slack. As long as the slack allocation is safe, we can always ensure that the schedulability condition is satisfied. Therefore, we use a simple heuristic algorithm motivated by the uniform constant slowdown scheme which is proved to be optimal in continuous voltage scaling [5]. The optimal common slowdown factor η is given by the

system utilization rate. In our approach, we only consider a finite number of discrete configuration points as defined above. Therefore, as shown in Algorithm 9, for each task, we select the configuration point with minimum energy consumption but equal or shorter execution time compared to the one decided by the optimal slowdown factor. Note that we use each task's execution under the highest voltage in V and largest cache configuration in C as the base case (v_{base}, c_{base}), which is used in the optimal slowdown factor calculation.

Algorithm 9: Configuration selection heuristic.

$\eta = \sum_{i=1}^{m} \frac{T_i(v_{base}, c_{base})}{p_i}$
for all task $\tau_i \in$ T **do**
  $T_i^{bound} = T_i(v_{base}, c_{base})/\eta$;
  Assign τ_i with (v_{j_i}, c_{k_i}) which satisfied:
   1) $E_i(v_{j_i}, c_{k_i})$ is the minimum;
   2) $T_i(v_{j_i}, c_{k_i}) \leqslant T_i^{bound}$;
end for
return (v_{j_i}, c_{k_i}), $\forall i \in [1, m]$

5.2.5 Procrastination

To further control static energy consumption, it is beneficial to put the system into a sleep mode instead of keep it idle since the static power could be lower by order-of-magnitude. As discussed in Section 5.2.3, taking various system components into consideration leads to much higher critical speed compared to leakage-aware DVS-only scenario. In other words, for the same task set, the idle periods during which there is no active task are getting longer. However, bringing the system into sleep mode and vice versa requires certain amount of overhead in terms of energy and time. In order to reduce the number of processor mode switches, we need to make the busy/idle periods as long as possible. One way to achieve this is to procrastinate task execution when no time constraint will be violated. We adapt the task procrastination algorithm from [67] into our EDF scheduler. We ensure that when the system gets shut down, there is no unfinished job in the system. This avoids cold start penalty being introduced since otherwise resumed task after wakeup has to refetch its data from memory. Hence, the shutdown overhead

consists of the energy consumed for circuit logic recharging and dirty data flushing-back in the cache subsystem provided write-back policy is used.

Algorithm 10 outlines our procrastination scheme. A timer is enabled when idle period starts and disabled when busy period starts. A newly arrived task during idle period will update the timer if it has earlier absolute deadline compared to the current earliest deadline. Upon timeout, all delayed ready tasks are executed in EDF order. Arriving tasks during busy period are allowed to preempt as normal EDF scheduler does. Note that *time* represents the current time instant and (v_{j_i}, c_{k_i}) stands for the chosen configuration point for task τ_i. Here, $isEarlier[i]$ records whether the current job of τ_i's deadline is earlier than all the pending tasks in the system at the time when it arrives. Also, $p_r \cdot \lceil \frac{time}{p_r} \rceil$ and $p_r \cdot \lfloor \frac{time}{p_i} \rfloor$ are essentially the absolute deadline and the arrive time of τ_i's current job.

Algorithm 10: Task procrastination algorithm.

```
isEarlier[i] is initialized to be all false;
Current earliest deadline of delayed jobs δ = 0;
On arrival of a new job of task τ_r:
d_r = p_r · ⌈time / p_r⌉;
actUtil = Σ_{i=1}^{m} T_i(v_{j_i}, c_{k_i}) / p_i;
if System is in sleep mode or is idle then
   if timer is disabled then
      timer = ⌊(1 − actUtil) · p_r⌋;
      δ = d_r; isEarlier[r] = true;
   else
      if d_r < δ then
         for all τ_i in ready task queue do
            if isEarlier[i] is true then
               delayed = delayed + (time − p_i · ⌊time / p_i⌋) / p_i;
            end if
            timer = ⌊(1 − actUtil − delayed) · p_r⌋;
            δ = d_r; isEarlier[r] = true;
         end for
      end if
   end if
end if
```

5.3 A General Dynamic Reconfiguration Algorithm

5.3.1 Overview

Existing techniques, as well as previous approaches presented by this dissertation, are either designed for specific systems (e.g., soft real-time systems in which missing task deadlines are tolerable) or specific task characteristics (e.g., periodic tasks). Moreover, they are also based on certain assumptions which do not always hold, e.g., negligible or fixed reconfiguration overhead. Swaminathan et al. [110] modeled the uniprocessor voltage scaling for real-time system as a generalized network flow problem and solved it using network flow algorithms. However, their method does not support cache reconfiguration and only considered voltage switching at task boundaries. Moreover, their method cannot incorporate variable runtime overhead nor make tradeoff between running time and design quality. We address these limitations in the methods proposed in this section.

We develop a general algorithm that comprehensively solves energy-aware reconfiguration problems in uniprocessor multitasking systems. The contribution can be summarized as:

1. The algorithm assumes that each task can be executed under one or multiple configurations and finds the optimal configuration assignment to minimize energy consumption while ensuring all the time constraints. Each configuration could correspond to one cache configuration, one voltage level or a combination of them. Therefore the algorithm can either separately or simultaneously accommodate DCR and DVS techniques.
2. It allows differential cost of switching from one configuration to another. Thus, it has advantages over existing techniques that it can effectively take variable runtime overhead into account.
3. The algorithm can be flexibly parameterized to tradeoff between algorithm running time and solution quality. Our experimental results show that the running time can be drastically reduced while only minor quality degradation is observed.

Furthermore, this algorithm is relatively independent of the scheduling policy and task properties. It can support tasks with/without time constraint, preemptive/non-preemptive scheduling or periodic/aperiodic tasks.

5.3.2 Algorithm

The proposed approach accepts a trace of *execution blocks* as the input. Given a task set and a scheduling policy, we first execute all the tasks under the base case (under the base cache configuration in DCR or the highest voltage level for DVS) assuming each of them requires its worst-case workload. The scheduler generates the execution blocks in temporal order. Note that for non-preemptive scheduling, execution blocks are essentially a sequence of task instances (jobs) with each of them having an absolute deadline and earliest start time (arrival time). In preemptive systems, however, execution blocks can be segments of tasks produced by preemptions. Figure 5-11 illustrates the relation between execution blocks and the tasks which they belong to. Suppose there are three periodic tasks τ_1, τ_2 and τ_3 with the characteristics of $(1, 3, 3)$[2] , $(2, 5, 5)$ and $(4, 12, 12)$. Under EDF schedule, there are 10 execution blocks (b_1, b_2, ... , b_{10}) before time unit 12. Our algorithm makes reconfiguration decisions on the granularity of each execution block. Thus, it is optimal in non-preemptive systems with inter-task manner DVS/DCR. It can also generate more energy savings in preemptive systems without introducing additional runtime overhead since a context switching has to be carried out during task preemption.

Static profiling for each execution block can be similarly carried out using techniques described in Section 5.2 as well as previous chapters. Only Pareto-optimal configurations, which are not dominated by any other configuration in terms of both energy consumption and performance, are considered for each block. Specifically, for DVS, since leakage power is considered, the minimum voltage level is lower bounded by the critical speed

[2] Here the three numbers stand for execution time, period and relative deadline, respectively.

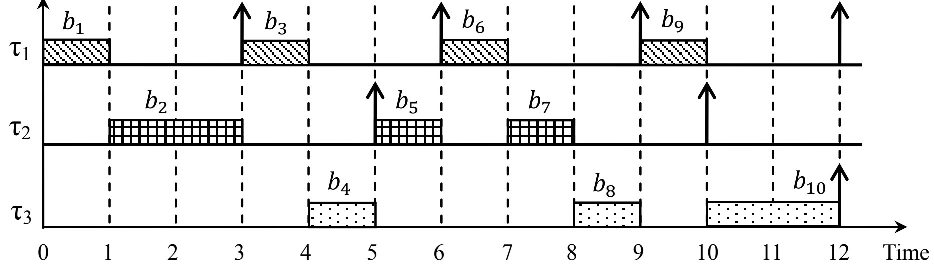


Figure 5-11. Tasks and execution blocks.

as discussed in Section 5.2. In this section, we define a general term *configuration* which could be a cache configuration, a voltage level, a combination of them (i.e., configuration point in Section 5.2.4.1) or any other form of system configuration. Let h and h_i denote the total number of available configurations and the number of Pareto-optimal configurations for the ith execution block, respectively.

We model the runtime reconfiguration overhead as variables depending on the transition from one configuration to another. For example, the overhead for reconfiguring a 4KB cache to a 8KB cache is generally larger than just changing the line size from 16 bytes to 32 bytes since the former requires waking up cache banks but the later is done by line concatenation. The input to our algorithm can be formally represented as:

- A set of n execution blocks B$\{b_1, b_2, \dots, b_n\}$.
- Execution block $b_i \in$ B has an arrival time a_i if it is the first block in the task instance and an absolute deadline d_i if it is the last block.
- Execution block b_i has execution time t_i^k and energy consumption e_i^k under configuration k (c_k).
- Reconfiguration energy overhead $\rho(i,j)$ and time overhead $\sigma(i,j)$ for converting from configuration c_i to configuration c_j.

Note that a_i and d_i correspond to the task to which the execution block belongs. a_i and d_i are set to -1 when they are not applicable to block b_i. If we denote t_i as the start

time and k_i as the index of the configuration assigned to block b_i given in the solution, the general dynamic reconfiguration problem $\wp$ can be formulated as[3] :

$$minimize\ \ E = \sum_{i=1}^{n}(e_i^{k_i} + \rho(c_{k_{i-1}}, c_{k_i})) \tag{5–1}$$

subject to,

$$t_i \geqslant a_i, \forall a_i \geqslant 0 \tag{5–2}$$

$$t_i + \sigma(c_{k_{i-1}}, c_{k_i}) + t_i^{k_i} \leqslant d_i, \forall d_i \geqslant 0 \tag{5–3}$$

$$t_{i+1} \geqslant t_i + \sigma(c_{k_{i-1}}, c_{k_i}) + t_i^{k_i}, \forall i \in [1, n) \tag{5–4}$$

Equation (5–2) represents the timing constraint that all the execution blocks must start executing after the task instance's arrival time. Equation (5–3) ensures deadline is not violated for any task. Note that time overhead is accounted at the beginning of task execution. Since we stick to the original schedule, Equation (5–4) guarantees the execution order of all the blocks in the final solution. The goal is to find k_i for all blocks in B so that Equation (5–1) can be achieved. The described modelling method makes our approach generally applicable – it does not depend on any task set characteristic or scheduling algorithm.

5.3.2.1 Extended Complete Bipartite Graph

We formulate the dynamic reconfiguration problem $\wp$ as a minimum-cost path finding problem in an extended complete bipartite graph (ECBG) as shown in Figure 5-12. Unlike traditional complete bipartite graph, an ECBG has multiple (specifically, n) disjoint sets $\{V_1, V_2, ..., V_n\}$ and a single source node as well as a single destination node. Every node in one set is connected to every node in its neighboring sets. The source node is fully

[3] c_{k_0} denotes the initial configuration.

connected with all the nodes in the first set and all the nodes in last set is connected to the destination node. Formally, an ECBG can be defined as ECBG$\{V_1 + V_2 + ... + V_n,\ E\}$ such that for any two nodes $v_i^k \in V_i$ and $v_{i+1}^j \in V_{i+1}$, there is an edge (v_i^k, v_{i+1}^j) in E.

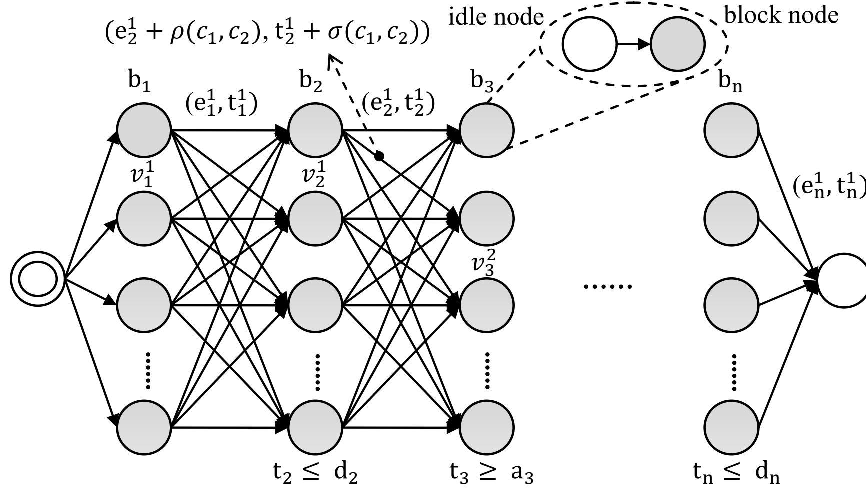


Figure 5-12. ECBG model of $\wp$.

Semantically, each disjoint set V_i represents an execution block b_i in B. Each node in the disjoint set stands for one configuration for that block. Hence, the number of nodes in set V_i is h_i. Each edge (v_i^k, v_{i+1}^j) in E is associated with two values: e_i^k and t_i^k. It means that, by moving from set V_i to V_{i+1} through this edge (choosing c_k), it requires t_i^k time units and e_i^k units of energy to execute block b_i. The runtime overhead is also taken into account on each edge. Specifically, edge (v_i^k, v_{i+1}^j) carries a pair of values: $(e_i^k + \rho(c_k, c_j), t_i^k + \sigma(c_k, c_j))$. Therefore, the objective shown in Equation (5–1) is algorithmically equal to finding a path from the source node to the destination node in the ECBG which has the minimum accumulative energy E. This path contains one and only one node from each disjoint set (choosing one edge between neighboring sets), which corresponds to selecting one configuration for each block. Moreover, all the constraints shown in Equation (5–2), (5–3) and (5–4) have to be satisfied in the path. For those nodes with arrival time constraint, say b_i, it is possible that the finish time of its previous node b_{i-1} is earlier than a_i. To ensure $t_i \geqslant a_i$, there is an idle node before every block node to represent the possible idle intervals. Note that edge (v_1^1, v_2^1) does not involve any overhead

since no reconfiguration is carried out (i.e. $k_1 = k_2$). However, edge (v_2^1, v_3^2) includes reconfiguration overhead $\rho(c_1, c_2)$ and $\sigma(c_1, c_2)$.

5.3.2.2 Minimum-Cost Path Algorithm

In this section, we employ a dynamic programming based algorithm to find the minimum-cost path. Let E_i and T_i denote the total energy consumption (cost) and execution time up to node b_i. Starting from the first node, for each node b_i, we find the lowest cost E_i under each possible value of T_i and possible configuration choice for b_i (i.e. c_{k_i}), in a node by node manner until the destination node is reached. If there is no such partial path which has an accumulative execution time no larger than a specific value of T_i and ends up with a specific configuration for b_i, the corresponding E_i is set to infinity. The calculation of all E_i values for each node is based on the lowest cost values of its previous node calculated in last step. At each step, say b_i, we know the lowest total energy of last $i-1$ nodes under each possible value of T_{i-1} and configuration for b_{i-1}. Based this information and various overhead, we can easily find the minimum E_i under all possible T_i and c_{k_i}.

Since the execution time is continuous but the design space is actually discrete (consists of finite number of choices), it is neither possible nor necessary to consider all possible values of T_i. Hence, we discretize T_i into a finite set of values. The interval between two adjacent discretized values is regarded as one time unit, which could be as small as one clock cycle or as large as one millisecond in practice. To reduce running time, we can limit T_i within the rage of $[T_{min}, T_{max}]$. We set $T_{min} = \sum_{i=1}^{n} t_i^h$ where t_i^h is the execution time under the most performance efficient configuration. T_{max} can be set to the deadline constraint of last task instance or the common deadline for all tasks. In other words, all blocks need to be completed before T_{max}. A three-dimensional array D is created for dynamic programming in which each element $D[i][\tau][j]$ stores the lowest total cost for nodes $b_1, b_2, ..., b_i$ while total execution time T_i is equal or less than τ $(T_i \leqslant \tau)$ and configuration choice for b_i is c_j. As a result, there are n rows in D with each

row consisting of $(T_{max} - T_{min})$ vectors and each vector has h elements. Therefore, the recursive relation for our dynamic programming scheme can be represented as:

$$D[i][\tau][j] = \min_{k \in [1, h_{i-1}]} \{D[i-1][\tau - t_i^j - \sigma(c_k, c_j)][k] + e_i^j + \rho(c_k, c_j)\} \tag{5–5}$$

D is filed up in a row by row manner and in an order so that all the previous $i-1$ rows are filled when the ith row is being calculated. Note that only those elements corresponding to the Pareto-optimal configuration of b_i is calculated in each vector of $D[i][\tau][\,]$. Finally, the solution quality is decided by $min\{D[n][\tau][j]\}$, for $\tau \in [T_{min}, T_{max}]$ and $j \in [1, h_n]$, which is the lowest value in last row of D. Figure 5-13 provides a pictorial representation of our algorithm. A possible solution and one of the configuration on the path are shown for illustrative purpose.

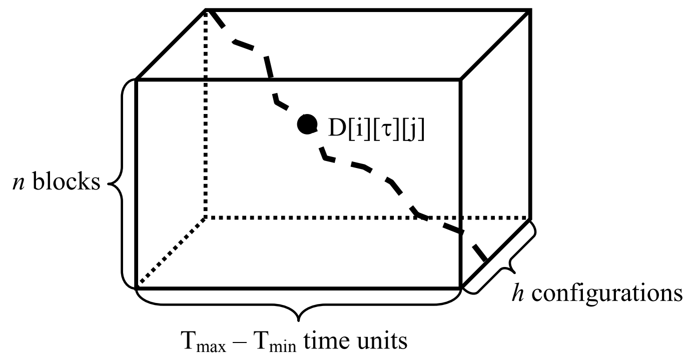


Figure 5-13. Illustration of our algorithm.

Complexity Analysis: Our algorithm iterates over all the nodes (1 to n). In other words, the input size of our algorithm is actually the number of execution blocks. In each iteration, all discretized T_i values $(T_{max} - T_{min})$ as well as all Pareto-optimal configuration points (1 to h_i) for current and previous nodes are examined. Hence the time complexity is $O(n \cdot (max\{h_i\})^2 \cdot (T_{max} - T_{min}))$. The memory requirement of our algorithm is determined by the size of D, which stores $n \cdot (T_{max} - T_{min}) \cdot h \cdot sizeof(element)$ bytes. To reduce the memory complexity, in each entry of D, we can simply use minimum number of bits to remember the configuration choice instead of real E_i values. For calculation purposes, two

two-dimensional arrays are used for temporarily storing E_i values for current and previous nodes.

Deadline Constraint: To ensure that the solution we find does not violate any task's deadline, during each step of the dynamic programming process, if b_i has deadline constraint, all the entries with T_i value larger than d_i are set to infinity. As a result, in the next step, those entries will be regarded as invalid.

Arrival Time Constraint: In the final solution, we have to guarantee that none of the initial blocks of each task instance starts execution before the task's arrival time as shown in Equation (5–2). However, since it is possible that one execution block finishes earlier than its very next block (thus creating an idle interval), the entries (each of which is a vector) with $T_i \leqslant a_{i+1}$ in the ith row of D are valid. One important observation is that, for block b_{i+1}, it does not really matter when exactly b_i ends if b_i finishes before b_{i+1}'s arrival time. In other words, the T_i values of these entries have no impact on the decision making in b_{i+1}. Hence, in the final solution, if b_i actually ends before a_{i+1}, the choice we make for b_i must be the one that results in the lowest E_i value.

We partition the ith row into three ranges by the next block's arrive time a_{i+1} and the current block's deadline d_i as shown in Figure 5-14. The first range, named range A, in which entries with finish time earlier than a_{i+1}, are all valid but not all are needed during decision making. The ones with minimum E_i, for each configuration choice of b_i, are selected and stored in the vector $D[i][a_{i+1}][\,]$. All entries in range A are then set to infinity. By doing this, without losing any precision, we force b_{i+1} to start no earlier than its arrival time. The second range (range B) in which entries with T_i values between a_{i+1} and d_i are all valid for the calculation of next iteration since they make b_{i+1} start after a_{i+1}. The last range are all discarded due to deadline constraint of b_i.

For periodic task set, if each task's deadline is equal to its period, a_{i+1} is always earlier than d_i. It can be proved by contradiction. If a_{i+1} is larger than d_i, it implies that the next job of the task corresponding to b_i arrives before b_{i+1} does. Therefore,

there exists a ready-to-execute task between b_i and b_{i+1}, which contradicts the fact that b_{i+1} is the very next execution block of b_i. In cases where a_{i+1} may be after d_i (e.g. for aperiodic task set), range B vanishes and, as a result, the problem essentially becomes two independent subproblems (one consists of blocks before b_i while the other consists of blocks after b_{i+1}, inclusively).

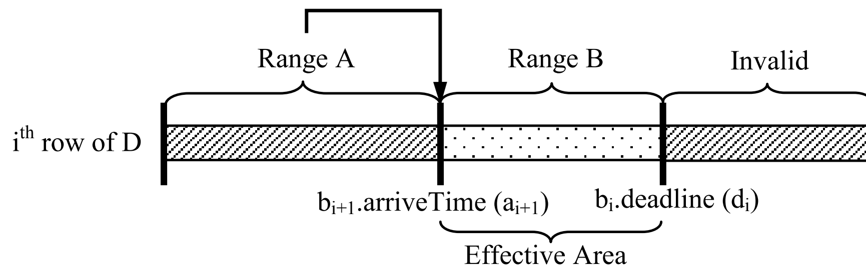


Figure 5-14. Ensuring the time constraints.

Tradeoff by Time Discretization: As discussed above, the time complexity of our algorithm is dominated by the term $(T_{max} - T_{min})$. A tradeoff can be made between solution quality and algorithm performance by further discretizing the execution time T_i. During the dynamic programming, instead of calculating for every time unit, we can compute in interval of multiple units. We define this number of time units as a parameter δ. For example, if $\delta = 2$, every row of D will contain $\lceil \frac{T_{max} - T_{min}}{\delta} \rceil$ vectors which are $\{T_{min}, T_{min} + 2, T_{min} + 4, ..., T_{max}\}$. The time complexity is reduced to $O(n \cdot (max\{h_i\})^2 \cdot \frac{T_{max} - T_{min}}{\delta})$. By doing this, we actually examine every possible path at a coarser granularity. Our experimental results demonstrate that time discretization only brings minor design quality degradation in terms of energy consumption while the algorithm efficiency can be greatly improved.

Approximate Approach: To further reduce the algorithm complexity, we can use an approximate version of our approach by storing only one element instead of a vector in $D[i][\tau][\,]$. In other words, D is now a two-dimensional array in which each element $D[i][\tau]$ stores the lowest E_i for nodes $b_1, b_2, ..., b_i$ while $T_i \leqslant \tau$, disregarding the end configuration (for b_i) of that specific path. As a result, the approximate version cannot support variable

time overhead since we do not know the configuration of the previous block without knowing the variable time overhead (which contradictorily depends on the previous block's configuration) during each step. Although variable energy overhead is used in actual calculation, we do not consider it for all possible configurations of the previous block in order to make tradeoff for efficiency. Therefore, the recursive relation becomes:

$$D[i][\tau] = \min_{j \in [1,h]} \{D[i-1][\tau - t_i^j - \sigma] + e_i^j + \rho(c_k, c_j)\} \tag{5–6}$$

where σ represents the constant time overhead and c_k stands for the configuration of $D[i-1][\tau - t_i^j - \sigma]$. For safety, σ can be set to the worst case time overhead. Similarly, the solution quality is decided by $min\{D[n][\tau]\}$, $\tau \in [T_{min}, T_{max}]$. The time complexity is reduced to $O(n \cdot max\{h_i\} \cdot \frac{T_{max}-T_{min}}{\delta})$. This is reduction of a factor of $max\{h_i\}$ over the exact algorithm. Figure 5-15 shows a pictorial illustration of our approximate approach.

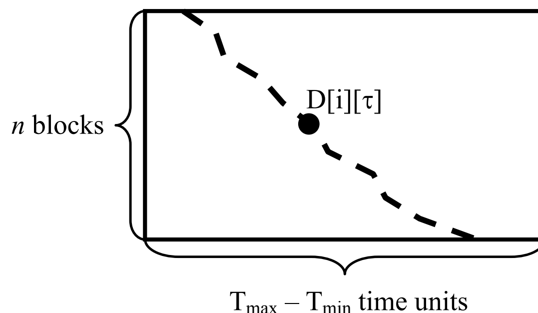


Figure 5-15. Illustration of the approximate version of our algorithm.

5.4 Experiments

5.4.1 System-wide Energy Optimization

5.4.1.1 Experiments Setup

To evaluate the effectiveness of our system-wide energy optimization approach, we select benchmarks from MediaBench [66], MiBench [35] and EEMBC [25] to from four task sets with each consists of 5 to 8 tasks. While DVS techniques usually use synthetic tasks for evaluation, we choose real benchmarks so that bus and memory hierarchy behaviors of real applications can be revealed. Table 5-1 lists our task sets. Task Set 1 consists of tasks

from MediaBench, Set 2 from EEMBC, Set 3 from MiBench and Set 4 is a mixture of all three suites. In Set 4, the two benchmarks from EEMBC are set to iterate 100 times in order to make their size comparable with others.

Table 5-1. Task sets consisting of real benchmarks.

Sets	**Tasks**
Set 1	cjpeg, djpeg, mpeg2, pegwit, rawcaudio
Set 2	A2TIME01, BaseFP01, BITMNP01, RSPEED01, TBLOOK01
Set 3	CRC32, susan, dijkstra, rijndael, adpcm, qsort, FFT, stringsearch
Set 4	cjpeg, rawdaudio, pegwit, A2TIME01, RSPEED01, pktflow, FFT, dijkstra

Processor constants described in Chapter 2 are adapted from [51]: $V_{bs} = -0.7V$, $L_d = 37$, $\alpha = 1.5$. The on-chip buses and off-chip buses have capacitance of $5pF$ and $60pF$, respectively. We believe they are reasonable numbers based on the bit line capacitance estimation described in [9] as well as the per unit of length capacitance estimation [21]. Similar set of numbers are also used in [28]. The on-chip buses have equal frequency as the processor (decided by the current voltage level) while off-chip buses (from L1 to L2 and from L2 to memory) have a frequency of $400MHz$ and $200MHz$, respectively. The bus static power is assumed to be 50% of the average dynamic power consumption, which is a conservative estimation [91].

We assume cache dirty data write back and circuit logic recharging penalty for shutdown to be $85\mu J$ and $300\mu J$. Therefore, the total shutdown overhead is $385\mu J$ [51]. Based on our energy model, idle power dissipation for the system, which comes from the static energy consumption of processor, cache hierarchy, bus lines and memory, is assumed to be $240 + 200 + 58 + 291 = 789mW$. System in sleep mode is assumed to consume $80\mu W$ of power. Hence, the shutdown threshold interval is $0.49ms$ and any interval whose length is shorter than this threshold will not lead to a shutdown. The energy estimation framework (whose input is gathered from SimpleScalar [14]) as well as the scheduling simulator are implemented in C++.

5.4.1.2 Results

We consider the following techniques:

- **DVS**: Traditional DVS without DCR which assigns the lowest feasible[4] voltage level at the aim of minimizing the processor dynamic energy consumption.
- **CS-DVS**: Leakage-aware DVS without DCR which assigns lowest feasible voltage level above the critical speed decided by processor energy consumption.
- **CS-DVS-G**: Leakage-aware DVS without DCR which assigns lowest feasible voltage level above the critical speed decided by system-wide energy consumption[5] .
- **DVS-DCR**: Traditional DVS + DCR which assigns the configuration point for minimizing the dynamic energy consumption of processor and cache subsystem.
- **CS-DVS-DCR**: Leakage-aware DVS + DCR which assigns the most energy-efficient while feasible configuration point above the critical speed decided by the energy consumption of processor and cache subsystem.
- **CS-DVS-DCR-G**: Leakage-aware DVS + DCR which assigns the most energy-efficient while feasible configuration point above the critical speed decided by system-wide energy consumption.
- **CS-DVS-DCR-G-P**: Leakage-aware DVS + DCR for system-wide energy minimization which also employs task procrastination.

Single Benchmark Figure 5-16 shows comparison of the overall energy consumption using three different techniques (CS-DVS, CS-DVS-G and CS-DVS-DCR-G) assuming that there is only one task in the system (i.e. no limitation on available slack). In other words, it represents the system-wide energy consumption of the configuration point with

[4] By saying "feasible", it refers to the configuration points that satisfy the slack allocation described in Section 5.2.4.

[5] In other words, the difference of CS-DVS-G from CS-DVS is that it considers other system components including caches, buses and memory in determining processor voltage level.

longest execution time in the profile table generated by each technique. Six selected benchmarks from Table 5-1 are considered and results are normalized to CS-DVS. It can be observed that considering other system components (CS-DVS-G) reduces the overall energy consumption compared to original leakage-aware DVS (CS-DVS) by 27.5% on average. Furthermore, by reconfiguring the cache hierarchy (CS-DVS-DCR-G), significant additional energy savings can be achieved (46.6% averagely).

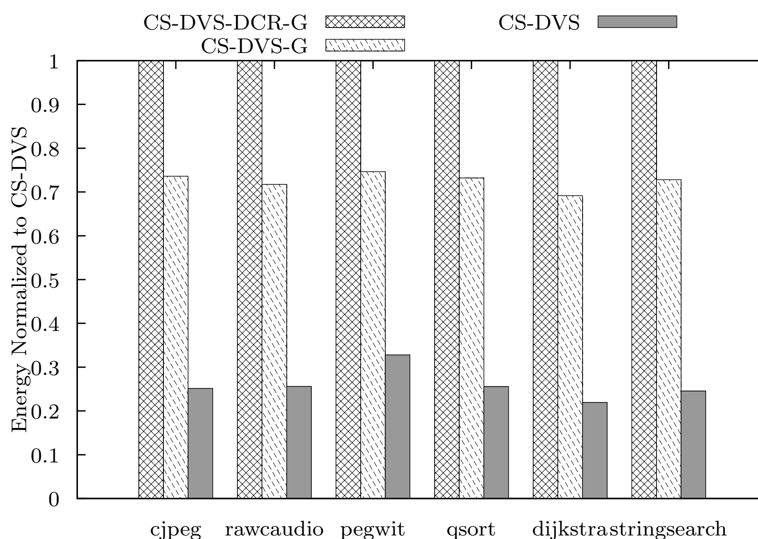


Figure 5-16. Total energy consumption of single-benchmark task sets.

Task Set For all the task sets described in Table 5-1, we compare all the above listed techniques across various system utilizations (from 0.1 to 0.9 in a step of 0.1). All the results are the average of all task sets and are normalized to **DVS** scenario. Figure 5-17 shows the normalized system-wide overall energy consumption using different approaches. The first observation is that, for DVS-only approaches, considering other system components (CS-DVS-G) can achieve 12.8% additional energy savings on average (up to 26.6%) compared with traditional leakage-aware DVS (CS-DVS). Generally, applying DVS and DCR together (DVS-DCR) outperforms traditional DVS (DVS) and CS-DVS-G across all utilization rates by 66.3% and 42.1% on average, respectively. Our approach, system-wide leakage-aware DVS + DCR (CS-DVS-DCR-G), outperforms CS-DVS-G by 47.6% on average. It can be observed that leakage-aware and leakage-oblivious DVS +

DCR approaches behave similarly when the system utilization ratio is beyond 0.5. It is because both of them are inclined to select similar configuration points which have voltage levels above the critical speed (V_{dd} is around 0.8 to 0.9). In other words, in these scenarios, DVS-DCR does not make inferior DVS decisions, which can lead to dominating leakage power, due to limited available slack. However, when the utilization ratio is low, CS-DVS-DCR-G can achieve around 4.6 - 23.5% more energy savings than DVS-DCR since CS-DVS-DCR-G does not lower down the voltage level below the critical speed.

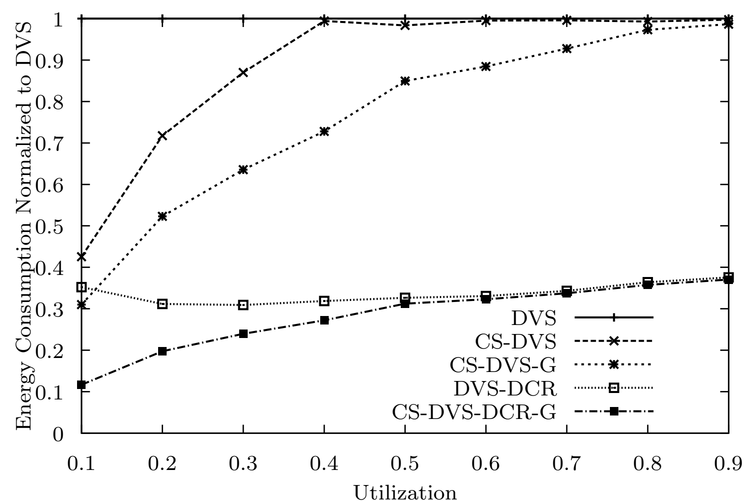
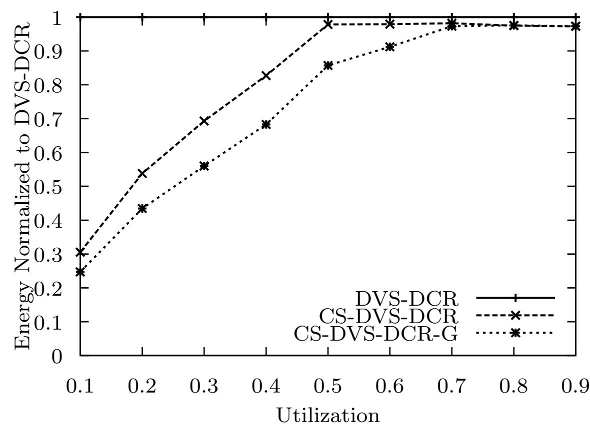


Figure 5-17. System-wide overall energy consumption using different approaches.

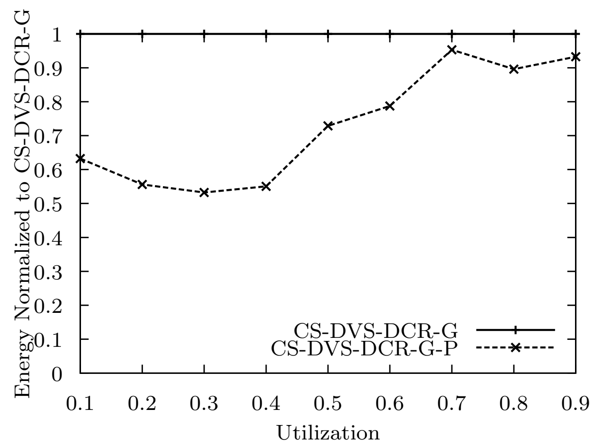
Figure 5-18 (a) shows the reduction in static energy consumption by using CS-DVS-DCR-G compared to DVS-DCR as well as CS-DVS-DCR. CS-DVS-DCR-G gains averagely about 26.5% static energy savings over DVS-DCR across all utilizations and around 44.4% in low utilization cases. Compared with CS-DVS-DCR, taking memory and system buses into consideration results in 7.1% static energy savings on average (up to 14.4%). This improvement is not as significant as the difference between CS-DVS and CS-DVS-G since, as shown in Section 5.2.3, memory and bus lines have relatively less impact on the critical speed compared with cache subsystem.

In our study, dynamic procrastination does not bring remarkable savings with respect to overall energy consumption. The reason is that the shutdown threshold is relatively

short compared with the execution time of real benchmarks in our approach. Therefore, even without procrastination, the idle periods during system execution normally are longer than the threshold which makes them beneficial to shutdown the system. In other words, the total sleep time for both CS-DVS-DCR-G and CS-DVS-DVS-G-P are close. It is expected, however, if the task sizes are small, reductions of overall energy will be more significant [51]. To illustrate the effectiveness of procrastination, Figure 5-18 (b) shows the result in idle energy savings. It can be observed that 26.9% savings on average can be achieved across all utilization rates by using CS-DVS-DCR-G-P.



(a)



(b)

Figure 5-18. Results: (a) Static energy consumption using DVS-DCR and cs-DVS-DCR; (b) Idle energy consumption using cs-DVS-DCR and cs-DVS-DCR-P.

5.4.2 General Algorithm for Dynamic Reconfiguration

5.4.2.1 Experiments Setup

DCR: To demonstrate the effectiveness of our algorithm on DCR, we use selected benchmarks from MediaBench [66], MiBench [35] and EEMBC [25] to form four task sets, each consisting 4 to 7 tasks, as shown in Table 5-2. In order to avoid scenarios where some task dominates the others in terms of energy consumption, we select the benchmarks such that all the tasks in the same set have comparable sizes. For each task set, we consider both cases of periodic and aperiodic/sporadic tasks. In the former scenario (periodic tasks), we assign the period and task's worst-case workload so that the system utilization varies in the range of 0.3 to 0.9[6] in incremental step of 0.1. In the later scenario (aperiodic/sporadic tasks), for each task, all the jobs are randomly generated with total accumulative system utilization at any moment under the schedulability constraint (e.g., 1). The job inter-arrival time is generated based on an exponential distribution. Note that, since we consider a preemptive system (although the simpler case, non-preemptive system, is also supported), the input size of our algorithm is actually the number of execution blocks as described in Section 5.3.2. Different task set characteristics will result in drastically different number of blocks.

Table 5-2. Task sets consisting of real benchmarks.

Sets	Tasks
Set 1	ospf, susan, pegwit, pktflow
Set 2	cjpeg, epic, dijkstra, FFT, qsort
Set 3	CANRDR01, PUWMOD01, AIFIRF01, BITMNP01, CACHEB01, AIFFTR01
Set 4	stringsearch, ospf, CRC32, pegwit, untoast, qsort, toast

[6] This is a practical and reasonable range since below 0.3 the solution can be trivially found by selecting most energy-efficient configurations for all tasks.

The reconfigurable cache architecture is a four-bank cache with tunable cache sizes of 4KB, 8KB and 16KB, line sizes of 16 bytes, 32 bytes and 64 bytes and associativity of 1-way, 2-way and 4-way. Therefore, we have $h = 18$ different cache configurations. Empirically, there are around 3 to 5 Pareto-optimal cache configurations for conventional applications [119]. Runtime reconfiguration overhead is dependent on the original cache configuration (c_i) and the one tuned to (c_j). We use SimpleScalar [14] to collect the static profiling information.

DVS: To evaluate our algorithm for DVS, we consider Marvell's StrongARM [74] as the underlying DVS-enabled processor as described in Section 4.4.1.1. We randomly generate four synthetic task sets, with similar characteristics for evaluating DCR, both for periodic and aperiodic/sporadic scenarios.

DVS+DCR: We also evaluate our approach in the case where both DVS and DCR are employed in the system. We use the same task sets as described in Table 5-2. The total energy consumption is therefore $E_{total} = E_{cache} + E_{processor}$. Task execution time is dependent on both voltage level (which decides the length of each cycle) and cache configuration (which decides the total number of cycles). Runtime overhead is thus the sum of both the cache reconfiguration overhead and the voltage scaling overhead.

5.4.2.2 Results

Energy Reduction We compare our algorithm with two heuristics which are applicable to both DVS and DCR, namely Uniform Slowdown and Greedy Repairing, since the techniques proposed in [122] and [119] are only for soft real-time systems and thus not applicable to our case. These two heuristics are adapted from DVS techniques [5] [95]. Generally, in uniform slowdown, we choose the configuration for task τ_i which consumes minimum energy while has equal or less execution time compared to t_i^{base}/η, where t_i^{base} is the execution time under base case and η is the system utilization. In greedy repairing, we first assign the most energy efficient configuration to every task. If the task set becomes unschedulable, we run a greedy repairing phase, during which the next more performance

efficient configuration for one of the tasks is selected which leads to minimum ratio of energy increase to system utilization decrease. The process repeats until the task set becomes schedulable. This heuristic is also used in [47]. Note that these two heuristics assign only one configuration per task and are not able to consider variable overhead. Figure 5-19 and 5-20 show the comparison results for both the scenarios where DVS and DCR are employed simultaneously and separately, respectively. The time discretization parameter δ is set to 1, 2, 4 and 8 milliseconds[7] . As normalized to the uniform slowdown heuristic, 25% of energy savings for DCR and 17% for DVS on average can be achieved using our approach. Compared with the greedy repair method, the energy savings are 17% and 11% for DCR and DVS, respectively. When both techniques are employed, the energy saving achieved are less compared with employing them separately.

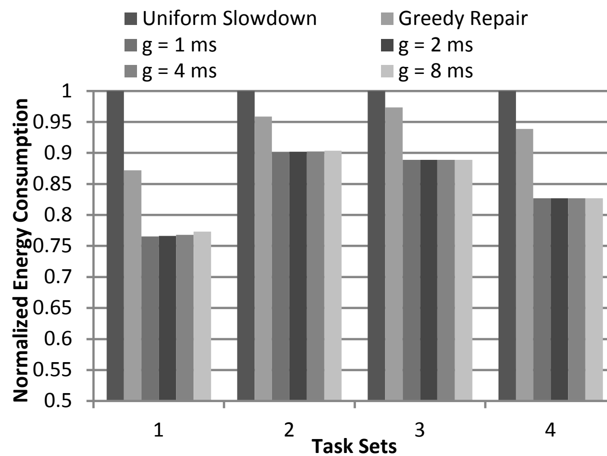
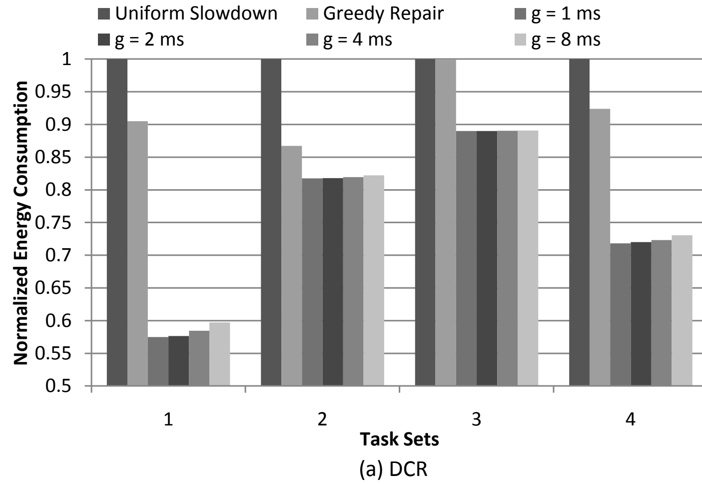


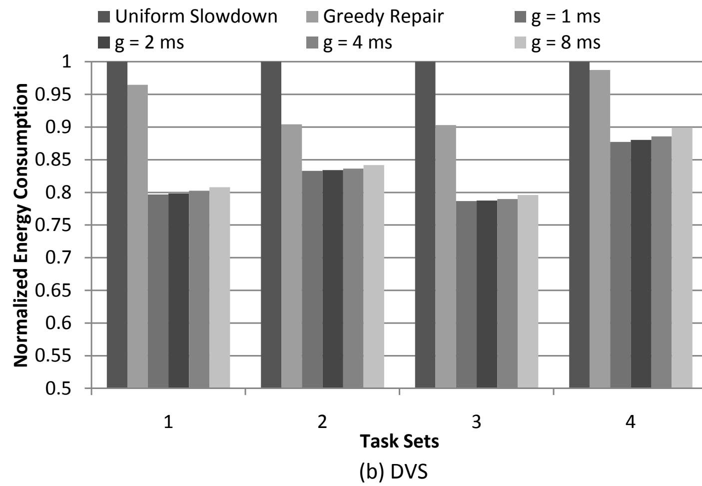
Figure 5-19. Energy consumption compared with two heuristics: DVS+DCR.

Time Discretization Effect Figure 5-21 and 5-22 illustrates the flexibility of our algorithm by varying the time discretization. Results are the average of both periodic and aperiodic scenarios and normalized to the $\delta = 1ms$ scenario. δ is increased exponentially from 1 millisecond to 128 milliseconds. The important observation is that, although

[7] In DCR, since tasks in set 3 has smaller sizes in terms of energy consumption and execution time than other sets, the unit of δ is microsecond.



(a) DCR



(b) DVS

Figure 5-20. Energy consumption compared with two heuristics: (a) DCR; (b) DVS.

our algorithm running time is drastically reduced, the design quality (total energy consumption) is only slightly sacrificed and still very close to the case where $\delta = 1ms$. For example, for task set 4 in DCR which has 679 execution blocks in the hyper-period, our algorithm gives the solution in 1.5 seconds with $\delta = 128ms$. The energy consumption of this solution is only 7% worse than the one generated with $\delta = 1ms$, which requires 19 seconds of execution time.

Variable Overhead Aware Effect For both DVS and DCR, we compare two different versions of our algorithm: one is aware of variable reconfiguration overhead and the other assumes constant overhead (which is the average of all variable overhead values). For DVS, the variable overhead matrix is generated so that each value depends on and is in

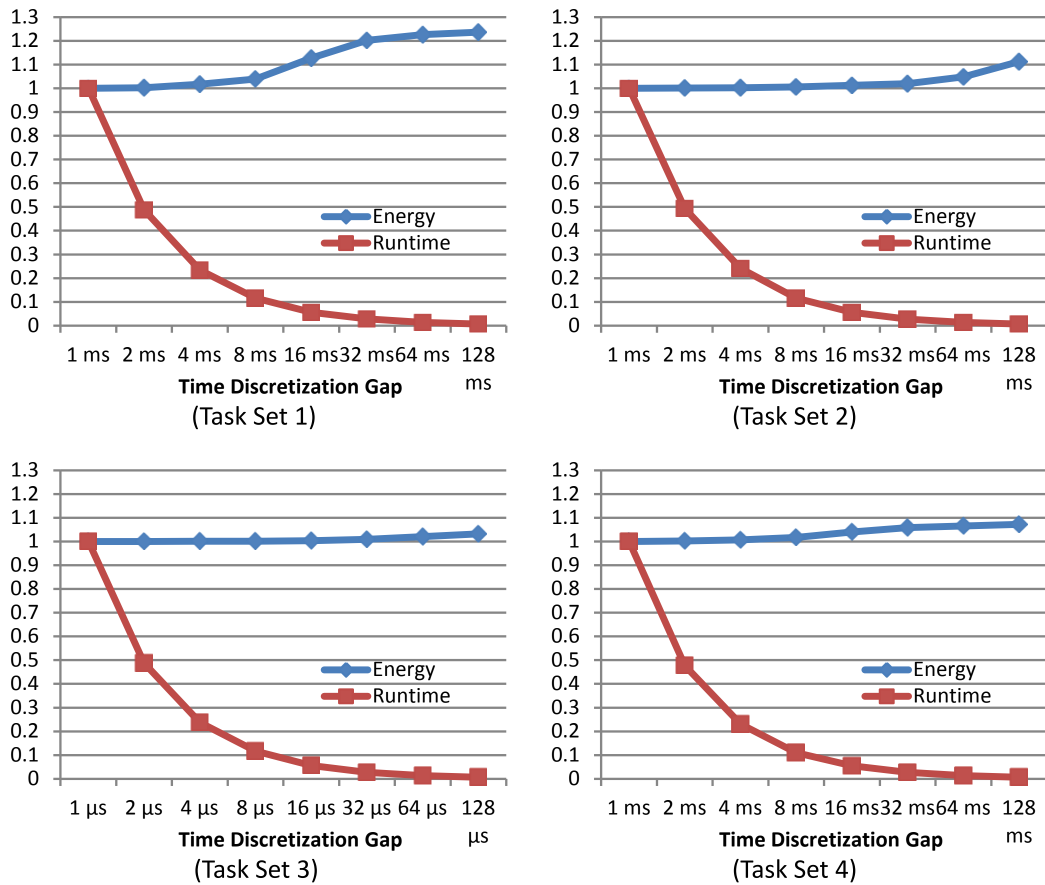


Figure 5-21. Time discretization effect for DCR.

proportion to how much voltage/frequency is increased or decreased. For DCR, the matrix is similarly generated except that the overhead for tuning the cache capacity from one level to another is 10 times larger than tuning the line size and associativity. Therefore, the actual overhead is the sum of all three cache parameters.

First we show how the amount of overhead affect the design quality in DVS. We vary the average of the variable energy overhead from 5% to 30% of the average of all block's energy consumption. Figure 5-23 shows the result averaged over all task sets. Clearly, variable overhead awareness brings more benefit when the amount of overhead is larger. Figure 5-24 demonstrates that effectively utilize the variable overhead can lead to substantial energy saving improvements for all task sets in DCR. Same observation can be made for DVS scenario. However, variable overhead awareness in DCR can lead to averagely 10% more energy savings than in DVS, which is because the size and

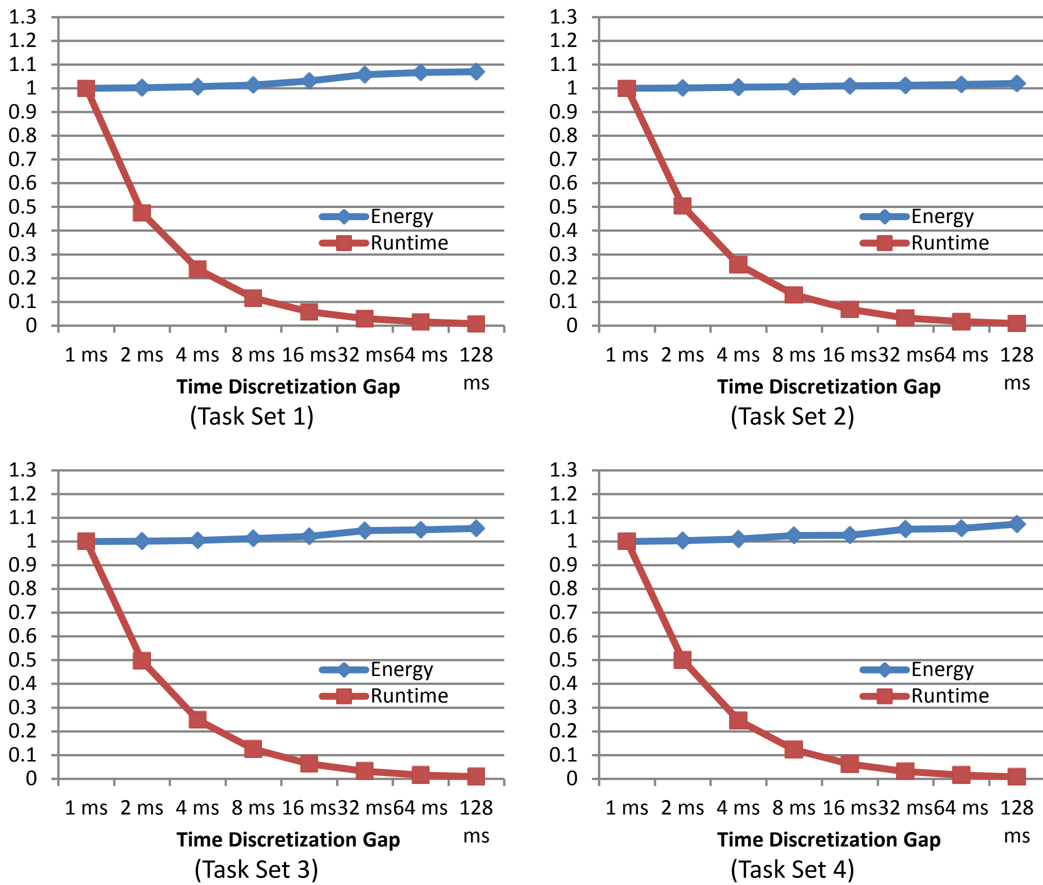


Figure 5-22. Time discretization effect for DVS.

variability of DCR's design space is much larger than DVS. Note that although DVS and DCR are used as the examples here, our approach is generally applicable to any kind of optimization problem based on reconfiguration – where the actual overhead of reconfiguration could be substantial.

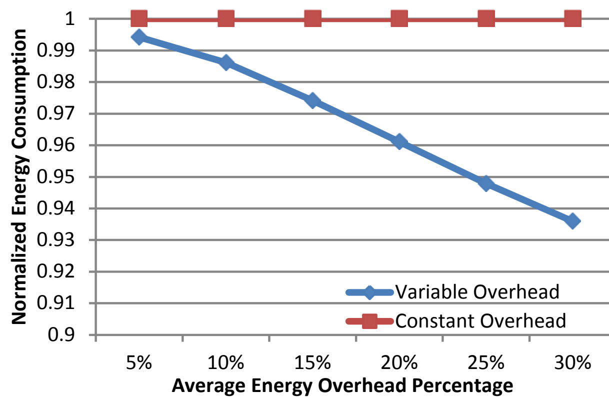


Figure 5-23. Variable overhead aware effect in DVS.

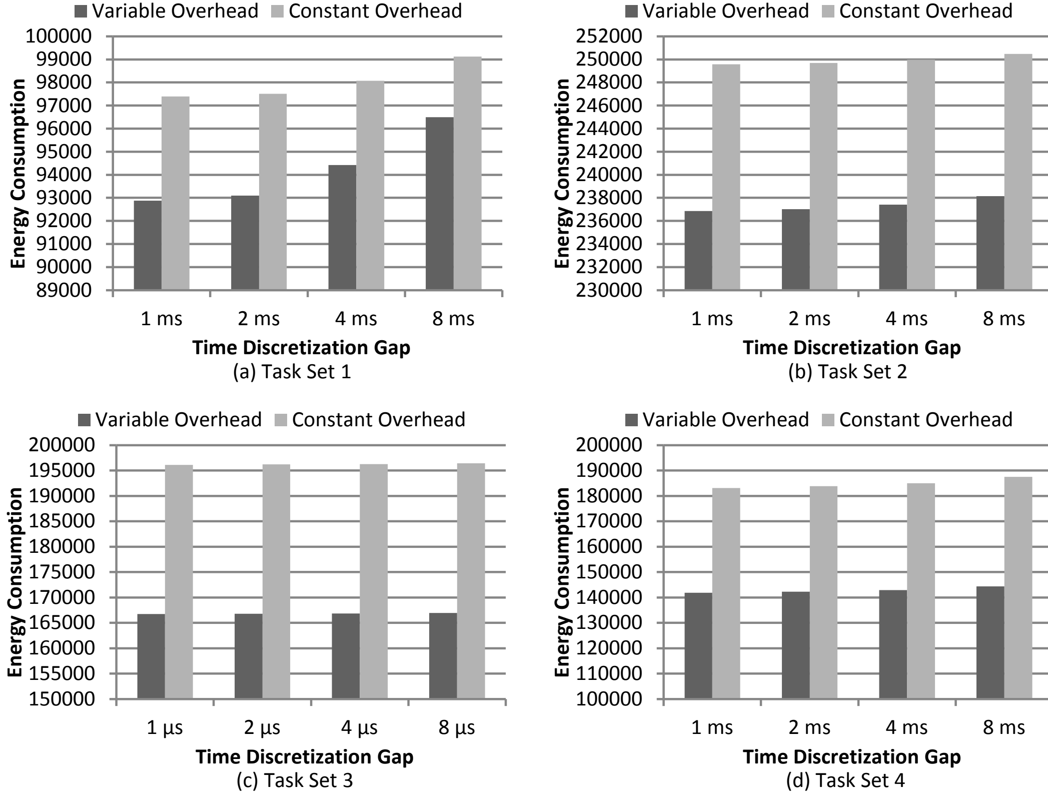
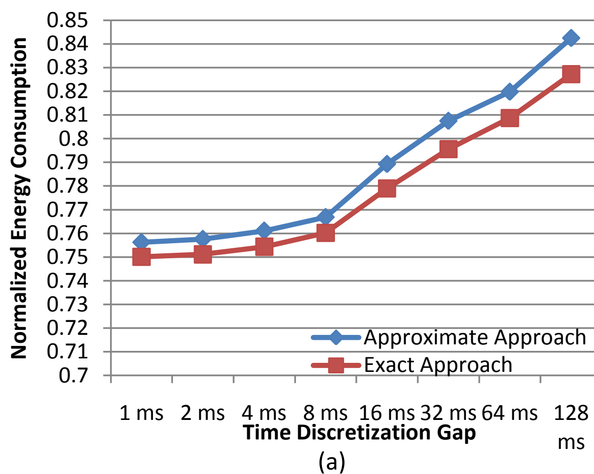


Figure 5-24. Variable overhead aware effect in DCR.

Approximate Approach Effect We study the performance of the approximate version of our approach using a 2-D array dynamic programming with respect to the exact approach using a 3-D array, as discussed in Section 5.3.2. Figure 5-25 (a) and (b) demonstrate the normalized energy consumption and absolute running time, respectively, under different δ values considering DCR. It can be observed that the approximate approach requires only 1 to 1.5% more total energy consumption (averaged over all task sets) but requires significantly less running time (for task set 1 with utilization of 0.8). However, exact approach is observed to experience relatively less design quality degradation with larger time discretization (δ).

We also investigate the impact from various reconfiguration overhead on the relative energy efficiency of the our approximate approach and exact approach. Figure 5-26 shows the comparison in energy consumption (normalized to the exact approach with $\delta = 1ms$) using DCR under various cache reconfiguration overhead values. We vary the overhead,



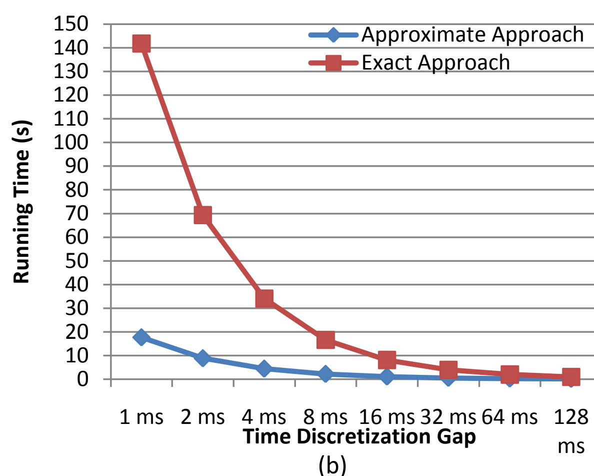


Figure 5-25. Comparison of our exact approach and approximate approach: (a) energy consumption normalized to uniform slowdown heuristic; (b) running time.

both energy and time, for tuning the cache size from one level to its neighboring one (e.g., from 4K to 8K or vise versa) as 1%, 2%, 4%, 8%, 12% and 16%[8] (as shown in Figure 13 (a), (b), (c), (d), (e) and (f), respectively) of the average consumption of all the blocks. The overhead matrix is generated as described above.

The important observation here is that when the reconfiguration overhead increases, the approximate version of our approach consumes more energy than the exact approach. Specifically, when only 3% differences is observed in Figure 5-26 (c), it becomes as large as

[8] In other words, the overhead for changing the line size and associativity is 0.1%, 0.2%, 0.4%, 0.8%, 1.2% and 1.6%, respectively.

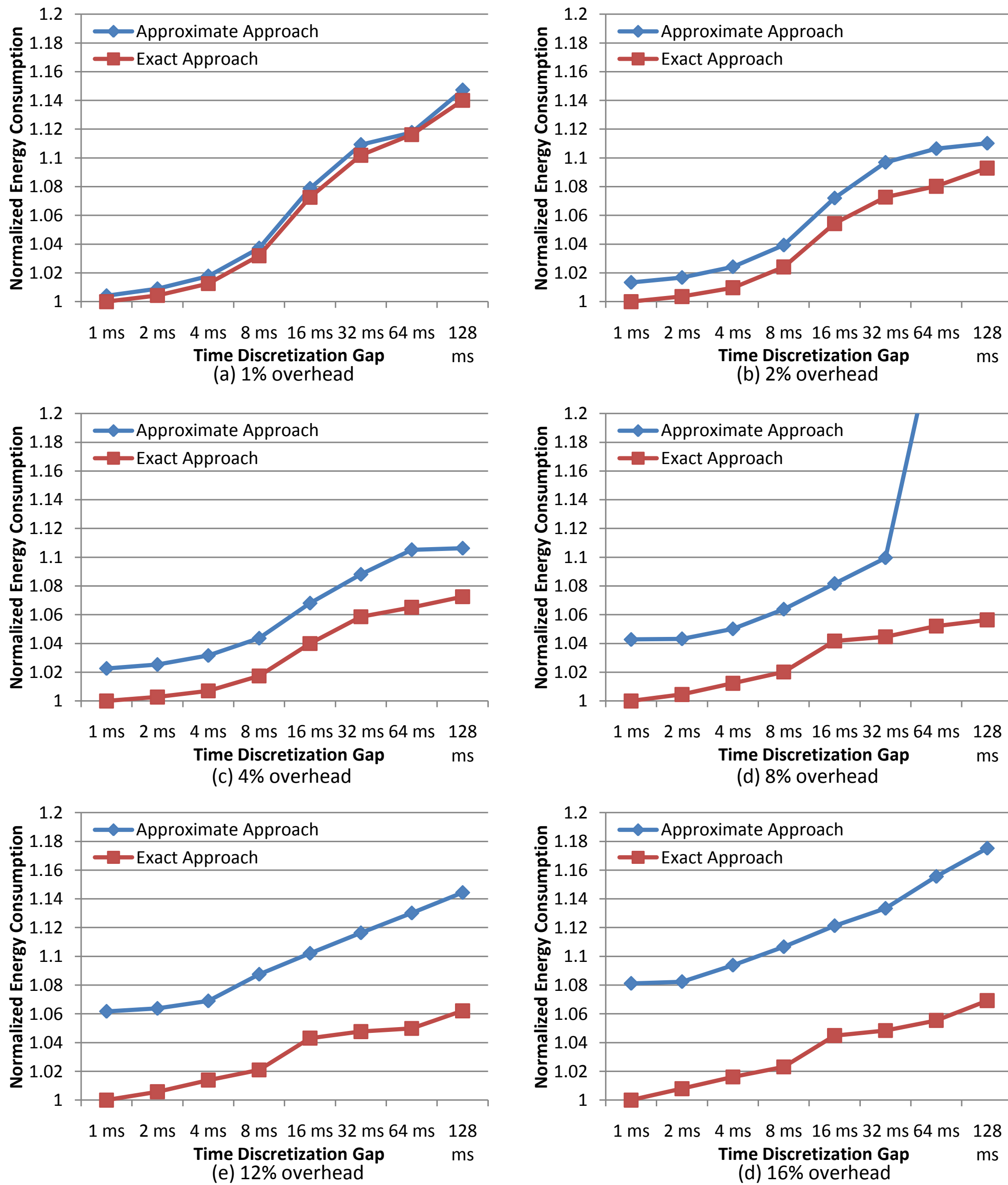


Figure 5-26. Comparison of our exact approach and approximate approach under various reconfiguration overhead.

10% when the overhead percentage increases. It is because the approximate approach does not consider all possible end configuration of the last step (i.e., block) during dynamic programming process and thus variable overhead is not fully incorporated. Moreover, the approximate approach also scales worse when δ increases. For example, in Figure 5-26 (d),

when δ becomes $128ms$, the approximate approach gives a solution with very bad quality while the exact approach behaves acceptably.

Another interesting observation is that, as shown in Figure 5-25 and 5-26, in certain scenarios, the exact algorithm with larger δ may have very similar or lower running time while achieve comparable or even better energy saving than the approximate approach with smaller δ. For example, in Figure 5-26 (c), the exact approach with $\delta = 8ms$ outperforms the approximate one with $\delta = 1ms$ in terms of energy using almost identical running time. In general, when the overhead is significant, our exact approach is preferable over the approximate version. However, when the overhead is small or negligible (e.g., Figure 5-26 (a) and (b)), the approximate approach is more efficient since it can achieve almost identical energy savings at small δ as the exact approach while requires significantly short time frame.

5.5 Summary

Leakage power can adversely impact any system energy optimization techniques including both dynamic voltage scaling and cache reconfiguration. Employing both DVS and DCR together can lead to greater system energy savings than using them independently. This chapter presented an efficient approach to integrate DVS and DCR that is aware of leakage power. Our studies demonstrate that considering only one optimization aspect (e.g., dynamic energy) or one component (e.g., DVS-capable processor) can lead to inaccurate conclusion in terms of overall energy, since critical speed will vary depending on the various components in the system. The proposed approach focuses on reducing system-wide energy consumption. It also integrates task procrastination to further save the energy consumption when the system is idle. Our approach is shown to be superior than both leakage-aware DVS techniques and leakage-oblivious DVS + DCR techniques.

This chapter also proposed a general algorithm for employing dynamic reconfiguration in multitasking systems with timing constraints. Our approach has the following

advantages. First, it can lead to more energy savings than inter-task manner DVS/DCR techniques. Secondly, it can effectively take variable reconfiguration overhead into consideration. Finally, our algorithm can be flexibly parameterized so that only slight solution quality degradation can be traded for drastically reduced running time requirement. It is also independent of task characteristics and scheduling policy. Extensive experiments demonstrates the effectiveness of our approach.

CHAPTER 6
TEMPERATURE- AND ENERGY-CONSTRAINED SCHEDULING

Along with the performance improvement in state-of-art microprocessors, power densities are rising more rapidly due to the fact that feature size scales faster than voltages [105]. In last five years, though the processor frequency is only improved by 30%, the power density is more than doubled and expected to reach over $250W/cm^2$ [26]. Since energy consumption is converted into heat dissipation, high heat flux increases the on-chip temperature. The “hot spot” on current microprocessor die, caused by nonuniform peak power distribution, could reach up to 120°C [12]. This trend is observed in both desktop and embedded processors [117] [138].

Thermal increase will lead to reliability and performance degradation since CMOS carrier mobility is dependent on the operating temperature. High temperature can result in more frequent transient errors or even permanent damage. Industrial studies have shown that a small difference in operating temperature (10-15°C) can make 2 times difference in the device lifespan [117]. Yeh et al. [131] also estimate that more than half of the electronic failures are caused by over-heated circuits. Furthermore, leakage power is exponentially proportional to temperature, which potentially results in more thermal runaway [124]. Studies also show that cooling cost increases super-linearly with the thermal dissipation [34].

Since high on-chip thermal dissipation has severe detrimental impact, we have to control the instantaneous temperature so that it does not go beyond a certain threshold. Thermal management schemes at all levels of system design are widely studied for general-purpose systems. However, in the context of embedded systems, traditional packaging and cooling solutions are not applicable due to the limits on device size and cost. Moreover, embedded systems normally have limited energy budgets since most devices are driven by batteries. Multitasking systems with real-time constraints add another level of difficulty since tasks have to meet their deadlines. Since such systems

normally have well-defined functionalities, this multi-objective problem admits design-time algorithms.

Dynamic voltage scaling (DVS) is acknowledged as one of the most efficient techniques used in both energy optimization [18] and temperature management [138]. In existing literatures, ***temperature (energy)- constrained*** means there is a temperature threshold (energy budget) which cannot be exceeded, while ***temperature (energy)-aware*** means there is no constraint but maximum instantaneous temperature (total energy consumption) needs to be minimized. In this chapter, we propose a formal method based on model checking for temperature- and energy- constrained (**TCEC**) scheduling problems in multitasking systems. We extend the classical timed automata [2] with notions of task scheduling, voltage scaling, system temperature and energy consumption. This approach is the first attempt on solving TCEC problem which is meaningful (especially in embedded systems) and as difficult as other problems including temperature-constrained (**TC**) scheduling, temperature-aware (**TA**) scheduling, temperature- constrained energy-aware (**TCEA**) scheduling and energy- constrained temperature-aware (**TAEC**) scheduling. A novel contribution of this approach is the development of a flexible and automatic design flow which models the TCEC problem in timed automata and solves it using formal verification techniques. Our approach is also capable of solving other problem variations mentioned above. Furthermore, it is applicable to a wide variety of system and task characteristics. Runtime voltage scaling overhead and leakage power consumption can also be easily incorporated.

The rest of this chapter is organized as follows. Section 6.1 presents the related works in this field. Section 6.2 provides background information on timed automata. Section 6.3 describes the proposed approach in details. Experimental results are presented in Section 6.4. Finally Section 6.5 concludes this chapter.

6.1 Related Work

Temperature-aware scheduling in real-time systems has drawn significant research interests in recent years. Wang et al. [118] introduced a simple reactive DVS scheme aiming at meeting task timing constraints and maintaining processor safe temperature. Zhang et al. [138] proved the NP-hardness of temperature-constrained performance optimization problem in real-time systems and proposed an approximation algorithm. Yuan et al [134] considered both temperature and leakage power impact in DVS problem for soft real-time systems. Chen et al. [17] explored temperature-aware scheduling for periodic tasks in both uniprocessor and homogeneous multiprocessor DVS-enabled platforms. Liu et al. [70] proposed a design-time thermal optimization framework which is able to solve problem variants EA, TA and TCEA scheduling in embedded system with task timing constraints. Jayaseelan et al. [46] exploited different task execution orders, in which each task has distinct power profile, to minimize peak temperature. However, none of these techniques solves TCEC problem. Moreover, they all make certain assumptions on system characteristics that limits their applicability.

Timed automata [2] has been widely adapted in real-time system researches. Norstorm et al. [82] first extended timed automata with a notion of real-time tasks and showed that the traditional schedulability analysis can be transformed to a decidable reachability problem in timed automata, which can be solved using model checking tools. Fersman et al. [27] further generalized [82] with asynchronous processes and preemptive tasks in continuous-time model. Abdeddaïm et al. However, none of these techniques considered energy or temperature related issues.

There are several studies on dynamic power management (DPM) using formal verification methods for embedded systems [103] and multiprocessor platforms [71]. Shukla et al. [103] provided a preliminary study on evaluating DPM schemes using an off-the-shelf model checker. Lungo et al. [71] tried to incorporate verification of DPM schemes in the early design stage. They showed that tradeoffs can be made between design quality and

verification efforts. None of these approaches considers temperature management in such systems. Moreover, they did not account for energy and timing constraints, which makes our methodology different from theirs.

6.2 Background

A classical timed automaton [2] is a finite-state automaton extended with notion of time. A set of clock variables are associated with each timed automaton and elapse uniformly with time in each state (i.e., location). Transitions (i.e., edges) are performed instantaneously from one state to another. Each transition is labeled with a set of guards which are Boolean constraints on clock variables and must be satisfied in order to trigger the transition. Transitions also have a subset of clock variables that need to be reset by taking the transition. Formally, we can define it as follows:

Definition 5. *A timed automaton $\mathcal{A}$ over clock set $\mathcal{C}$, state set $\mathcal{S}$ and transition set $\mathcal{T}$ is a tuple $\{\mathcal{S}, \mathcal{C}, \mathcal{T}, s_0\}$ where s_0 is the initial state. Transition set is represented as $\mathcal{T} \subseteq \mathcal{S} \times \Phi(\mathcal{C}) \times 2^{\mathcal{C}} \times \mathcal{S}$, where each element ϕ in clock constraint (guard) set $\Phi(\mathcal{C})$ is a conjunction of simple conditions on clocks ($\phi := c \leqslant t \mid t \leqslant c \mid \neg\phi \mid \phi_1 \wedge \phi_2$ where $c \in \mathcal{C}, t \in R$). $2^{\mathcal{C}}$ represents the subset of clock variables that will be reset in each transition and we term it as ρ.*

Semantically, the current configuration of a timed automaton $\mathcal{A}$ is decided by a state $s \in \mathcal{S}$ and the clock valuations $\mathcal{V}$ in the form of $\mathcal{C} \rightarrow R_+ \bigcup \{0\}$. Therefore, a legal execution of $\mathcal{A}$ consists of a sequence of transitions:

$$(s_0, \mathcal{V}_0) \xrightarrow{\phi,\rho} (s_1, \mathcal{V}_1) \xrightarrow{\phi,\rho} \cdots \xrightarrow{\phi,\rho} (s_n, \mathcal{V}_n) \tag{6–1}$$

6.3 TCEC Scheduling Approach

6.3.1 Overview

Figure 6-1 illustrates the workflow of our approach. The task information describes the characteristics of the tasks running in the system and is fed into the scheduler along with the scheduling policy. Any scheduling algorithm is applicable in our approach. The

scheduler executes the task set under the highest voltage level and produces a trace of *execution blocks*. Here, an execution block is defined as a piece of task execution in a continuous period of time under a single processor voltage/frequency level. Each execution block is essentially a whole task instance in non-preemptive systems. However, in preemptive scheduling, tasks could be preempted during execution hence one block can be a segment of one task. The scheduler records runtime information for each block including its corresponding task, required workload, arrival time and deadline, if applicable.

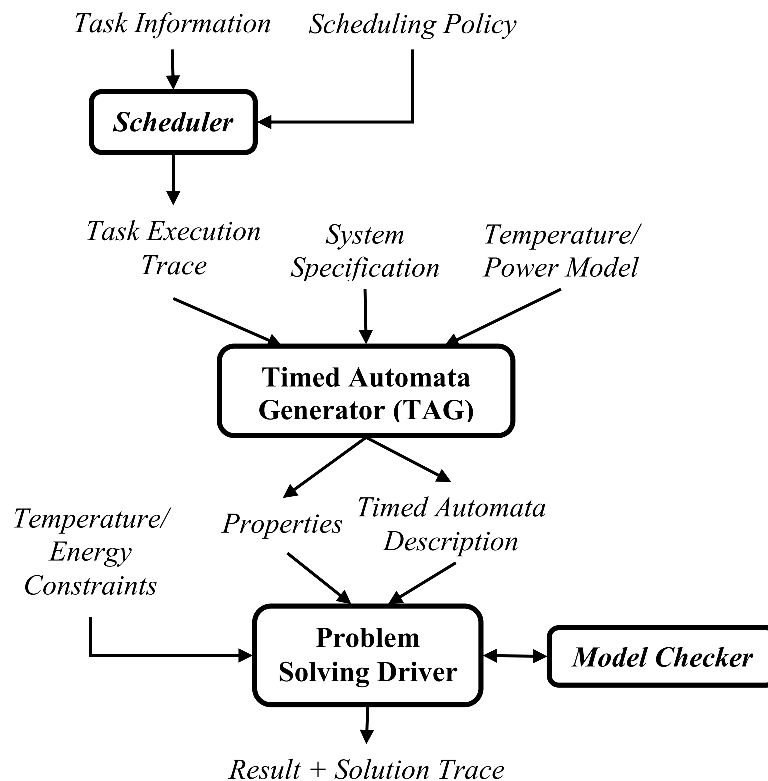


Figure 6-1. Workflow of our model checking approach.

The task execution trace, along with system specification (processor voltage/frequency levels), thermal/power models and design objective (not shown in Figure 6-1), are fed into the timed automata generator (TAG) that we have developed. Here the design objective decides the nature of the problem, e.g. TCEC. TAG generates two important outputs. One is the description of our timed automata model, which will be discussed in

Section 6.3.2, and the other one contains the properties reflecting the design objectives. We use a script based program to drive the model checker to solve the problem. Finally, the results and/or solutions are collected. Our methodology is flexible, completely automatic, based on formal technique and hence suitable in early design stages.

6.3.2 Modeling with Extended Timed Automata

Our approach scales the processor voltage level on the granularity of each execution block. In other words, the frequency level is changed at the beginning of each execution block. This strategy can lead to more flexible energy and temperature management in preemptive systems since decisions are made upon a finer granularity compared to inter-task manner [138]. We utilize timed automata to model the voltage scaling problem in the execution trace and extend the original automata with notions of temperature and energy consumption. Our model supports both scenarios in which task set has a common deadline and each task has its own deadline. For ease of discussion, the terms of *task*, *job* and *execution block* refer to the same entity in the rest of this chapter.

Task set with common deadline: TAG is given a trace of n execution blocks B$\{b_1, b_2, ..., b_n\}$. If tasks are assumed to have the same power profile (i.e. α is constant), the energy consumption and execution time for b_i under voltage level $v_k \in V$, denoted by e_i^k and t_i^k respectively, can be calculated based on the given processor model. Otherwise, they can be collected through static profiling by executing each task under every voltage level. Let ψ_{v_i,v_j} and ω_{v_i,v_j} denote runtime energy and time overhead, respectively, for scaling from voltage v_i to v_j. Since power is constant during a execution block, temperature is monotonically either increasing or decreasing [46]. We denote T_i as the final temperature of b_i. If the task set has a common deadline D, the safe temperature threshold is T_{max} and the energy budget is $\mathcal{E}$, TCEC scheduling problem can

be represented as finding a voltage assignment $\mathcal{K}\{k_1, k_2, ..., k_n\}$[1] such that:

$$\sum_{i=1}^{n}(e_i^{k_i} + \psi_{v_{k_{i-1}},v_{k_i}}) \leqslant \mathcal{E} \tag{6–2}$$

$$T_i \leqslant T_{max}, \forall i \in [1, n] \tag{6–3}$$

$$\sum_{i=1}^{n}(t_i^{k_i} + \omega_{v_{k_{i-1}},v_{k_i}}) \leqslant D \tag{6–4}$$

where T_i is calculated based on Equation (2–17). Here Equation (6–2), (6–3) and (6–4) denote the energy, temperature and common deadline constraints, respectively.

For illustration, an extended timed automata $\mathcal{A}$ generated by TAG is shown in Figure 6-2 assuming that there are three tasks and two voltage levels. Generally, we use l states for each task, forming disjoint sets (horizontal levels of nodes in Figure 6-2) among tasks, to represent different voltage selections. We also specify an error state which is reached whenever there is deadline miss. There are also a source state and a destination state denoting the beginning and the end of the task execution. Therefore, there are totally $(n \cdot l + 4)$ states. There is a transition from every state of one task to every state of its next task. In other words, the states in neighboring disjoint sets are fully connected. There are also transitions from every task state to the error state. All the states of the last task have transitions to the end state.

The system temperature and cumulative energy consumption are represented by two global variables, named T and E, respectively. The execution time for every task under each voltage level is pre-calculated and stored in a global array $c[\,]$. The common deadline D is stored in variable *deadline*. Constants such as processor power values, thermal capacitance/resistance, ambient temperature and initial temperature are stored in respective variables. There are two clock variables, **time** and **exec**, which represent the

[1] k_i denote the index of the processor voltage level assigned to b_i.

global system time and the local timer for task execution, respectively. The **time** variable is never reset and elapses uniformly in every state. Both clock variables are initially set to 0.

The transition from the source state carries a function *initialization()* which contains updates to initialize all the variables and constants. Each state is associated with an invariant condition, in the form of **exec** $\leqslant c[\,]$, which must be satisfied when the state is active. This invariant represents the fact that the task is still under execution. Each transition between task states carries a pair of guard: **time** $\leqslant$ *deadline* && **exec** $== c[\,]$. The former one ensures that the deadline is observed and the latter one actually triggers the transition, reflecting the fact that the current task has finished execution. Note that the overhead can be incorporated here since we know the start and end voltage level, if they are different. Each transition is also labeled with three important updates. The first one, $T = calcTemperature(P[\,], T, c[\,])$, basically updates the current system temperature after execution of one task based on the previous temperature, average power consumption and the task's execution time. The second one, $E = calcEnergy(P[\,], c[\,])$, adds the energy consumed by last task to E. The third update resets clock *exec* to 0. All the transitions to the error state are labeled with a guard in the form of **time** $>$ *deadline*, which triggers the transition whenever the deadline is missed during task execution. Note that not all the transition labels are shown in Figure 6-2.

The extended timed automata's current configuration is decided by valuations of clock variables (**time** and **exec**) and global variables (T and E). Therefore, the system execution now is transformed into a sequence of states from the source state to the destination state[2] . The sequence consists of one and only one state from each disjoint set which represents a task. Solving the TCEC problem as formulated above is equal to finding such a sequence with the following properties. First, the final state is the

[2] The sequence of states follows the same characteristics of Equation (6–1).

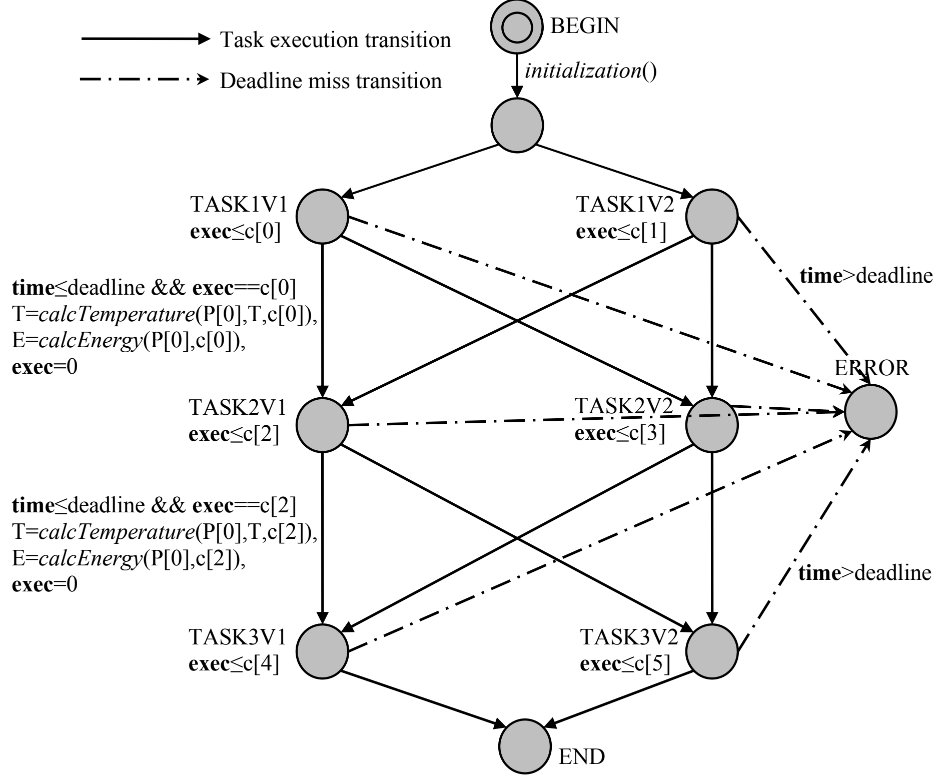


Figure 6-2. TCEC problem modeled in extended timed automata.

destination state which guarantees the deadline constraint. Next, the temperature T is always below T_{max} in every state. Finally, the energy consumption E is no larger than $\mathcal{E}$. We can write this requirement as a property in computation tree logic (CTL) [23] as:

$$\mathbf{EG}((T < T_{max} \;\wedge\; E < \mathcal{E}) \;\mathbf{U}\; \mathcal{A}.end) \tag{6–5}$$

where $\mathcal{A}.end$ means the destination state is reached. Now, we can use the model checker to verify this property and, if satisfied, the witness trace it produces is exactly the TCEC scheduling that we want.

However, it is possible that the model checker's property description language does not support the operator of "until" (**U**), e.g. UPPAAL [7]. In that case, we can add two Boolean variables, $isTSafe$ and $isESafe$, to denote whether T and E are currently below the constraints. These two Boolean variables are updated in functions *calcTemperature*() and *calcEnergy*(), respectively, whenever a transition is performed. Once the corresponding

constraint is violated, they are set to $false$. We can express our requirement in CTL as:

$$\mathbf{EF}(isTSafe \ \wedge \ isESafe \ \wedge \ \mathcal{A}.end) \tag{6–6}$$

Note that in the timed CTL that UPPAAL uses, the above property can be written as follows, where $Proc$ represents the timed automata $\mathcal{A}$, which is called a "Process" in UPPAAL.

$$\mathbf{E} <> (Proc.End \ \textbf{and} \ Proc.isTSafe \ \textbf{and} \ Proc.isESafe) \tag{6–7}$$

Task set with individual deadlines: In the scenario where each task has its own deadline, e.g. periodic tasks, we have to make sure the execution blocks finish no later than their corresponding task's deadline. A global array, $d[\,]$, is used to store the deadline constraints of each execution block. If not applicable, i.e. the block does not end that task instance, its entry in $d[\,]$ is set to -1. Therefore, instead of Equation (6–4), we have:

$$\sum_{j=1}^{i}(t_j^{k_j} + \omega_{v_{k_{j-1}},v_{k_j}}) \leqslant d[i], \forall d[i] > 0 \tag{6–8}$$

Figure 6-3 shows part of the new timed automata. The difference lies in the guard of transitions. Instead of **time** $\leqslant$ *deadline*, the guard for transitions between task states is in the form of $((d[\,] > 0 \ \&\& \ \mathbf{time} \leqslant d[\,]) \ || \ d[\,] < 0)$. The transition from task state to error state now carries a guard of $(d[\,] > 0 \ \&\& \ \mathbf{time} > d[\,])$.

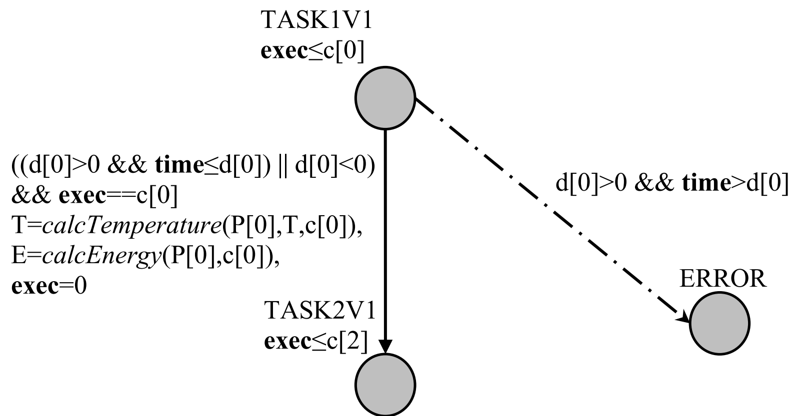


Figure 6-3. Problem modeling when every task has own deadline (partial graph).

6.3.3 Problem Variants

Our approach is also applicable to other problem variants by modifying the property and making suitable changes to invocation of the model checker.

TC: Temperature-constrained scheduling problem is a simplified version of TCEC. It only needs to ensure that the maximum instantaneous temperature is always below the threshold T_{max}. Therefore, the property can be written in CTL as:

$$\mathbf{EG}(T < T_{max}\ \mathbf{U}\ \mathcal{A}.end) \tag{6–9}$$

TA: To find a schedule so that the maximum temperature is minimized, we can employ a binary search over the temperature value range. Each iteration invokes the model checker to test the property (6–9) parameterized with current temperature constraint T_{max}. Initially, T_{max} is set to the mid-value of the range. If the property is unsatisfied, we search in the range of values larger than T_{max} in the next iteration. If the property is satisfied, we continue to search in the range of values lower than T_{max} to further explore better results. This process continues until the lower bound is larger than the upper bound. The minimum T_{max} and associated schedule, which makes the property satisfiable during the search, is the result. Note that the temperature value range for microprocessors is small in practice, e.g. $[30°C, 120°C]$. Hence, the number of iterations is typically no more than 7.

TAEC: TAEC has the same objective as TA except that there is an energy budget constraint. Therefore, we can solve the problem by using property (6–5) during the binary search.

TCEA: TCEA can be solved using the same method as TAEC except that the binary search is carried on energy values and temperature acts as a constant constraint. Since energy normally has a much larger value range, to improve the efficiency, we can discretize energy value to make trade-off between solution quality and design time. Since

the number of iterations has a logarithmic relationship with the length of energy value range, only moderate discretization is enough.

6.4 Experiments

6.4.1 Experiments Setup

We evaluate our approach assuming a StrongARM processor [74] as described in Section 4.4.1.1. We use synthetic task sets which are randomly generated with each of them having execution time in the range of 100 - 500 milliseconds. These are suitable and practical sizes to reflect variations in temperature, and millisecond is a reasonable time unit granularity [138]. We adopt the thermal resistance (R) and thermal capacitance (C) values from [46], which are $1.83°C/Watt$ and $112.2mJoules/°C$, respectively. The ambient temperature and initial temperature of the processor are set to $32°C$ and $60°C$, respectively. The scheduler and TAG shown in Figure 6-1 are both implemented in C++.

6.4.2 Results

6.4.2.1 Solving TCEC Problems

Table 6-1 shows the results on task sets with different number of blocks and constraints. The first and the second column are the index and number of blocks of each task set, respectively. The next three columns present the temperature constraint (TC, in $°C$), energy constraint (EC, in mJ), and deadlines (DL, in ms) to be checked on the model. The sixth column indicates wether the there exists a schedule which satisfies all the constraints. The last three columns give the actual maximum temperature (AT), total energy cost (AE), and time required to finish all blocks (AD) using the schedule found by the model checker (UPPAAL). It can be observed that our approach can find the solution (if exists) which satisfied all the constraints.

6.4.2.2 Running Time Variations

We have studied the impact of constraint variations on the running time required by UPPAAL. To achieve this, we measure the model checking time using task set 2 with

Table 6-1. TCEC results on different task sets

TS	#Blk	TC	EC	DL	Found?	AT	AE	AD
1	10	85	180000	7000	Y	77	171612	6865
		85	150000	8000	Y	77	149623	7966
		80	140000	8000	N			
2	12	85	70000	2500	Y	79	66375	2499
		85	60000	2700	Y	76	59911	2667
		80	60000	2500	N			
3	14	90	90000	2600	Y	90	81287	2540
		85	80000	2800	Y	79	71649	2702
		90	80000	2700	N			

two constraints kept constant while let the third one vary (TC, EC and DL). Figure 6-4 summarizes the results.

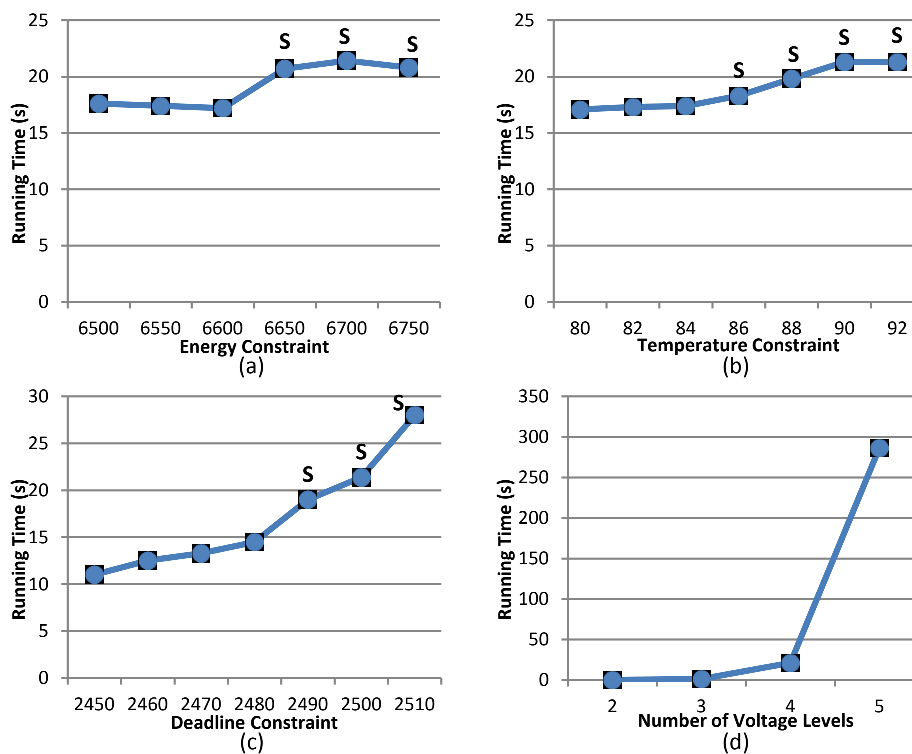


Figure 6-4. Running time with different constraints.

For energy constraint (Figure 6-4(a)) and temperature constraint (Figure 6-4(b)), we can observe that time requirement are not notably affected by the variation of these constraints. In general, it takes more time when the constraint can be satisfied (labeled "**S**" in Figure 6-4). When the constraint goes below or beyond the range shown in

Figure 6-4, the running time remains the same or slightly decreases in both cases because the constraint will either be easily falsified or no longer limit the search space, respectively. However, for the deadline constraint (Figure 6-4(c)), our experimental results show that the running time requirement will increase with the deadline, because larger time budget yields a larger solution search space for the model checker. We have also investigated the relation between the number of voltage levels and the time required for model checking. As shown in Figure 6-4(d), model checker's running time grows rapidly when more voltage levels are employed. This is due to the exponential growth of the search space.

6.5 Summary

This chapter proposed a model checking approach for temperature and energy-constrained scheduling problem in multitasking systems based on processor voltage scaling. We modeled the problem using extended timed automata which is solved by a model checker. We proposed a flexible and automatic framework which makes our approach applicable to temperature or energy-constrained problem as well as other variants and independent of any system characteristic. Extensive experimental results demonstrate the effectiveness of our approach.

CHAPTER 7
ENERGY OPTIMIZATION OF CACHE HIERARCHY IN MULTICORE SYSTEMS

Computation using single-core processors has hit the power wall on its way of performance improvement. Chip multiprocessor (CMP) architectures, which integrates multiple processing units on a single chip, have been widely adopted by major vendors like Intel, AMD, IBM and ARM in both general-purpose computers (e.g., [43]) as well as embedded systems (e.g., [3] [77]). Multicore processors are able to run multiple threads in parallel at lower power dissipation per unit of performance. Despite the inherent advantages, energy conservation is still a primary concern in multicore system optimization. While power consumption is a key concern in designing any computing devices, energy efficiency is especially critical for embedded systems. Real-time systems that run applications with timing constraints require unique considerations. Due to the ever growing demands for parallel computing, multicore processors are commonly employed in real-time systems [129] [123].

As discussed in Chapter 3 and 5, the prevalence of on-chip cache hierarchy has made it a significant contributor of the overall system energy consumption. For uniprocessor systems, DCR is an effective technique for cache energy reduction by tuning the cache configuration at runtime. For multicore systems, L2 cache typically acts as a shared resource. Recent research has showed that shared on-chip cache may become a performance bottleneck for CMP systems because of contentions among parallel running tasks [92] [56]. To alleviate this problem, cache partitioning (CP) techniques judiciously partition the shared cache and maps a designated part of the cache to each core. CP is designed at the aim of performance improvement [90], inter-task interference elimination [92], thread-wise fairness optimization [61], off-chip memory bandwidth minimization [132] and energy consumption reduction [93].

In this chapter, we present novel energy optimization techniques which efficiently integrate cache partitioning and dynamic reconfiguration in multicore architectures. Tasks

with timing constraints are considered in our approach. To the best of our knowledge, this is the first work that employs DCR and CP simultaneously. Our contributions can be summarized as:

1. We find that DCR in L1 caches has great impact on decisions of CP in shared L2 and vice versa. Moreover, both DCR and CP play important roles in energy conservation.
2. Our approach can minimize the cache hierarchy energy consumption while guarantee all timing constraints.
3. We propose efficient static profiling techniques and algorithms to find beneficial L1 cache configurations and L2 partition factors for each task.
4. Our approach considers multiple tasks on each core thus is more general than existing CP techniques which assume only one application per core [56] [93] [132].
5. We study both fixed and varying CP scheme along with DCR for multicore architectures.
6. We also study the effect on design quality from different deadline constraints, task mappings and gated-V_{dd} cache lines.

The remaining part of this chapter is organized as follows. Related works are discussed in Section 7.1. Section 7.2 describes the architecture model and motivation of our work. Section 7.3 presents our approach for CMPs in detail, followed by experimental results in Section 7.4. Finally, Section 7.5 concludes this chapter.

7.1 Related Work

Cache partitioning techniques are widely studied for various design objectives for multicore processors. Initially, majority of them focused on reducing cache miss rate to improve performance. Suh et al. [109] utilized hardware counters to gather runtime information which is used to partition the shared cache through the replacement unit. Qureshi et al. [90] proposed a low-overhead CP technique based on online monitoring and cache utilization of each application. Kim et al. [61] focused on fair cache sharing using

both dynamic and static partitioning. Recently, CP is employed for low-power system designs. Reddy et al. [92] [93] showed that, by eliminating inter-task cache interferences, both dynamic and leakage energy can be saved. Bank structure aware CP in CMP platforms is studied in [56]. Yu et al. [132] targeted at minimizing off-chip bandwidth through off-line profiling. Lin et al. [68] verified the effectiveness of CP in real systems. Meanwhile, CP is also beneficial for real-time systems to improve worst-case execution time (WCET) analysis, system predictability and cache utilization [13] [93]. Nevertheless, existing CP techniques only focus on shared L2 cache and ignore the impact as well as the optimization opportunities from private L1 caches.

7.2 Background and Motivation

In this section, we show important features of the underlying architecture. We also present an illustrative example to motivate the need and usefulness of our approach.

7.2.1 Architecture Model

Figure 7-1 illustrates a typical CMP platform with private L1 caches (IL1 and DL1) in each core and a shared on-chip L2 cache. Here, both instruction L1 and data L1 cache associated with each core are highly reconfigurable in terms of total capacity, line size and associativity as discussed in Section 3.1.2.

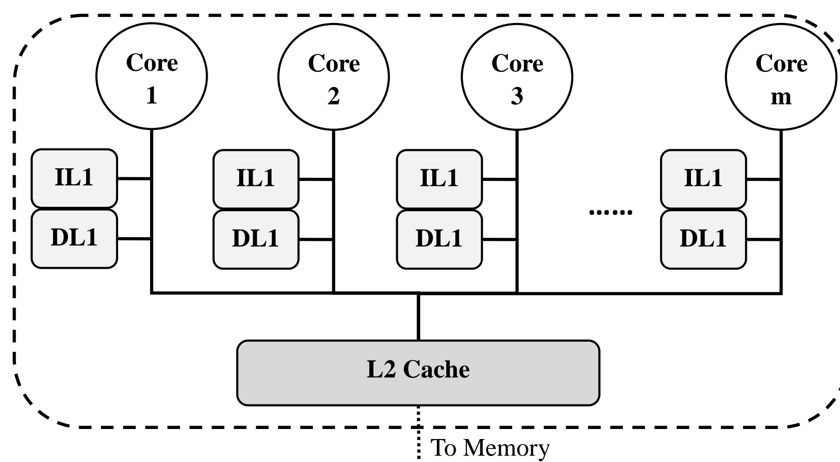


Figure 7-1. Typical multicore architecture with shared L2 cache.

Unlike traditional LRU replacement policy which implicitly partitions each cache set on a demand basis, we use a way-based partitioning in the shared cache [98]. As shown in Figure 7-2, each L2 cache set (here with a 8-way associativity) is partitioned in the granularity of ways. Each core is assigned a group of ways and will only access that portion in all cache sets. LRU replacement is enforced in each individual group which is achieved by maintaining separate sets of "age bit". It is also possible to divide the cache by sets (set-based partitioning) in which each core is assigned a number of sets and each set retains full associativity [132]. However, since real-time embedded systems usually have small number of cores, way-based partitioning is beneficial enough for exploiting energy efficiency. We refer the number of ways assigned to each core as its *partition factor*. For example, the L2 partition factor for Core 1 in Figure 7-2 is 3.

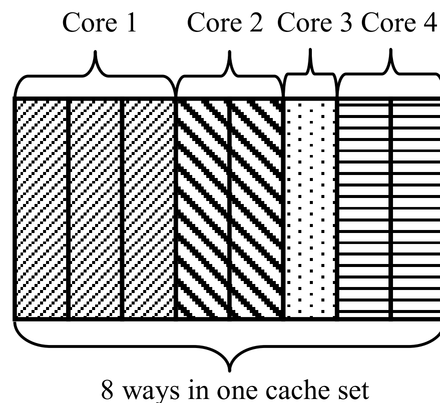


Figure 7-2. Way-based cache partitioning example (four cores with a 8-way associative shared cache).

In this work, we use static cache partitioning. In other words, L2 partitioning scheme for each core are pre-determined during design time and remain the same throughout the system execution. Dynamic partitioning [90] requires online monitoring, runtime analysis and sophisticated OS support thus is not feasible for embedded systems. Furthermore, real-time systems normally have highly deterministic characteristics (e.g., task release time, deadline, input) which make off-line analysis most suitable [89]. By static profiling, we can potentially search much larger design space and thus achieve better optimization results.

7.2.2 Motivation

Figure 7-3 shows the number of L2 cache misses and instruction per cycle (IPC) for two benchmarks (*qsort* from MiBench [35] and *vpr* from SPEC CPU2000 [107]) under different L2 cache partition factors p (with a 8-way associative L2 cache) and randomly chosen four L1 cache configurations[1] . Unallocated L2 cache ways remain idle. We observe that changing L1 cache configuration will lead to different number of L2 cache misses. It is expected because L1 cache configuration determines the number of L2 accesses. System performance (i.e., IPC) is also largely affected by L1 configurations. Notice that for larger partition factors (e.g., 7), the difference in L2 misses is negligible but IPC shows great diversity. It is because not only L2 partitioning but also L1 configurations determine the performance.

It is also interesting to see that *vpr* shows larger variances at the same L2 partition factor than *qsort*. For example, the number of L2 cache misses becomes almost identical (although there is times of differences in the number of L2 accesses) at $p = 4$ for *qsort* while it starts to converge for *vpr* only after $p = 6$. The reason behind this is that, for *qsort*, there are almost only compulsory misses for $p \geqslant 4$. In other words, $p = 4$ is a sufficient L2 partition factor for *qsort* in terms of performance. However, increasing p and reducing number of accesses continue to bring benefit for *vpr* as shown in Figure 7-3(c) due to the fact that *vpr* has more capacity and conflict misses.

Given the above observations, we see that L1 DCR has major impact on L2 CP and there are interesting trade-offs that can be explored for optimizations. Therefore, both DCR and CP should be exploited simultaneously for energy conservation in real-time multicore systems. Our experimental results in Section 7.4.2.1 confirms our conjecture.

[1] Here c_{18} and c_9, for example, stands for the 18th and 9th configuration for IL1 and DL1, respectively.

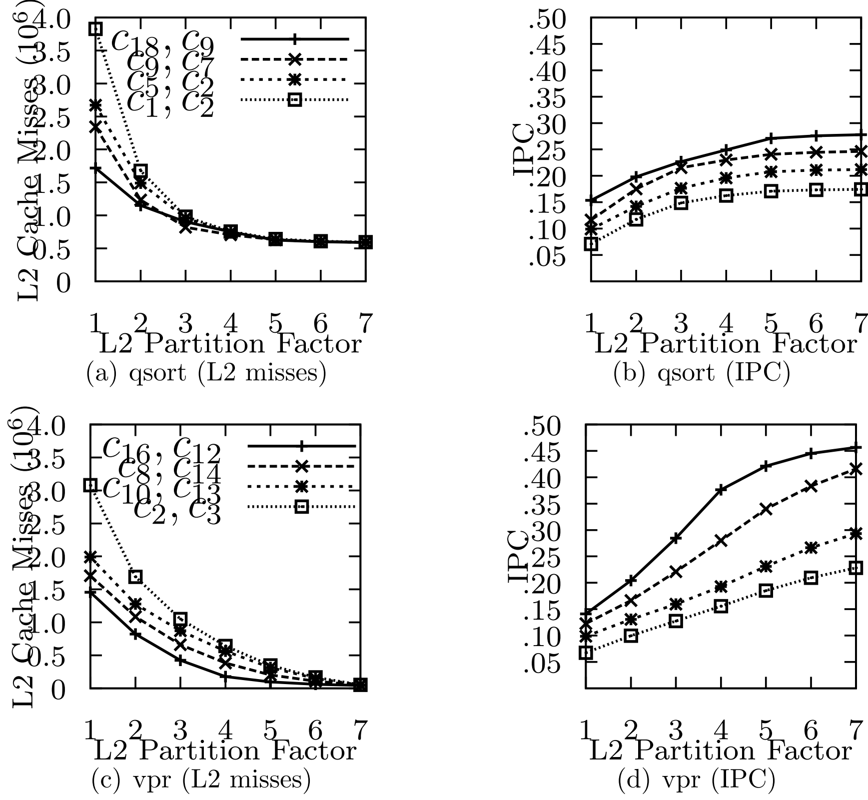


Figure 7-3. L1 DCR impact on L2 CP in performance.

7.3 Dynamic Cache Reconfiguration and Partitioning

In this section, we first formulate our energy optimization problem based on performing dynamic reconfiguration of private L1 caches and static partitioning of shared L2 cache (DCR + CP). Next, we present our static profiling strategy. Our dynamic programming based algorithm which utilizes the static profiling information is then described in detail. Next we investigate the effects of task mapping to different cores. Then we explore the effectiveness of employing dynamic cache partitioning. Finally, we study the benefit of using Gated-V_{dd} shared cache lines.

7.3.1 Problem Formulation

The multicore system we consider here can be modeled as:

- A multicore processor with m cores $\mathcal{P}\{p_1, p_2, ..., p_m\}$.
- Each core has reconfigurable IL1 and DL1 caches both of which supports h different configurations $\mathcal{C}\{c_1, c_2, ..., c_h\}$.

- A α-way associative shared L2 cache with way-based partitioning enabled.
- A set of n independent tasks $\mathcal{T}\{\tau_1, \tau_2, ..., \tau_n\}$ with a common deadline D[2] .

Suppose we are given:

- A task mapping $\boldsymbol{M} : \mathcal{T} \rightarrow \mathcal{P}$ in which tasks are mapped to each core. Let ρ_k denotes the number of tasks on p_k.
- A L1 cache configuration assignment $\boldsymbol{R} : \mathcal{C}_I, \mathcal{C}_D \rightarrow \mathcal{T}$ in which one IL1 and one DL1 configuration are assigned to each task.
- A L2 cache partitioning scheme $\boldsymbol{P}\{f_1, f_2, ..., f_m\}$ in which core $p_i \in \mathcal{P}$ is allocated f_i ways.
- Task $\tau_{k,i} \in \mathcal{T}$ (i^{th} task on core p_k) has execution time of $t_{k,i}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P})$. Let $E_{L1}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P})$ and $E_{L2}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P})$ denote the total energy consumption of all the L1 caches and the shared L2 cache, respectively.

Our goal is to find $\boldsymbol{M}$, $\boldsymbol{R}$ and $\boldsymbol{P}$ such that the overall energy consumption E of the cache subsystem:

$$E = E_{L1}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}) + E_{L2}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}) \tag{7–1}$$

is minimized subject to:

$$\max(\sum_{i=1}^{\rho_k} t_{k,i}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P})) \leqslant D \ , \ \forall k \in [1, m] \tag{7–2}$$

$$\sum_{i=1}^{m} f_i = \alpha \ \ ; \ \ f_i \geqslant 1 \ , \ \forall i \in [1, m] \tag{7–3}$$

Equation (7–2) guarantees that all the tasks in $\mathcal{T}$ are finished by the deadline D. Equation (7–3) ensures that the L2 partitioning $\boldsymbol{P}$ is valid.

[2] Our approach can be easily extended for individual deadlines.

7.3.2 Static Profiling

For the time being, we assume that the task mapping $\boldsymbol{M}$ is given (we will discuss more about it in Section 7.3.4). A reasonable task mapping would be a bin packing of the tasks using their base case execution time to all the m cores so that the total execution time in each core is similar. Here the base case execution time refers to the time one task takes in the system where L1 cache reconfiguration is not applied (using the base configuration) and L2 cache is evenly partitioned. Theoretically, we can simply profile the entire task set $\mathcal{T}$ under all possible combinations of $\boldsymbol{R}$ and $\boldsymbol{P}$. Unfortunately, this exhaustive exploration is not feasible due to its excessive simulation time requirement. Similarly as in Chapter 3, in this work, the reconfigurable L1 cache contains four banks each of which is 1 KB. Therefore, it offers total capacities of 1 KB, 2KB and 4 KB. Line size can be tuned from 16 to 64 bytes and each set supports 1-way, 2-way and 4-way associativity. There are total $h = 18$ different configurations. Even if we have four cores with only two tasks each core and a 8-way associative L2 cache, the total number of multicore architectural simulations will be $((18^2)^2)^4 \times 35$. To be specific, 18^2 denotes the IL1 and DL1 cache configurations for each task. $(18^2)^2$ presents all possible L1 cache combinations of the two tasks in each core. Upon that, $(18^2)^2)^4$ denotes all the combinations across four cores. According to Equation (7–3), the size[3] of $\boldsymbol{P}$ ($|\boldsymbol{P}|$) equals 35. Obviously, this simulation time is even longer than the age of the universe if each simulation takes only 1 minute.

This problem can be greatly relieved by exploiting the independence of the design space's each dimension. We observe that each task can actually be profiled individually. It is because the tasks that we consider do not have application-specific interactions (e.g., data sharing) except the contention for the shared cache resource. Essentially, using L2 cache partitioning, each core p_i is running equivalently on a uniprocessor with f_i-way

[3] The size of $\boldsymbol{P}$ can be calculated as $C^m_{\alpha-1}$.

associative L2 cache (i.e., the capacity is f_i/α of the original). L1 cache activities are private at each core while L2 activities happen independently in each core's partition. Therefore, we simulate each task in $\mathcal{T}$ independently under all combinations of L1 cache configurations and L2 cache partition factors (from 1 to $\alpha - 1$). In other words, the total number of single-core simulations equals to $h^2 \cdot (\alpha - 1) \cdot n$. Using the same example above, with 8 tasks, it is $(18^2) \times 7 \times 8$. Note that this profiling process is independent of the task mapping $\boldsymbol{M}$ and the number of cores m. Apparently, it will take only reasonable profiling time (e.g., at most three days).

7.3.3 DCR + CP Algorithm

Static profiling results are used to generate profile tables for each task. Each entry in the profile table records the cache energy consumption, for both L1 and the L2 partition, as well as the execution time of that task. The dynamic energy of L2 cache is computed using Equation (2–2) based on the statistics (accesses) from the core to which the task is assigned. The static energy, however, is estimated by treating the allocated ways as a standalone cache. There are $h^2 \cdot (\alpha - 1)$ entries in every profile table. For task $\tau_{k,i} \in \mathcal{T}$ (i^{th} task on core p_k), let $e_{k,i}(h_1, h_2, f_k)$ denote the total cache energy consumption if task $\tau_{k,i}$ is executed with (IL1,DL1) configurations (c_{h_1}, c_{h_2}) and L2 partition factor f_k. Similarly, let $t_{k,j}(h_1, h_2, f_k)$ denote the execution time. Our problem now can be presented as to minimize:

$$E = \sum_{k=1}^{m} \sum_{i=1}^{\rho_k} e_{k,i}(h_1, h_2, f_k) \tag{7–4}$$

subject to:

$$\max(\sum_{i=1}^{\rho_k} t_{k,i}(h_1, h_2, f_k)) \leqslant D \ , \ \forall k \in [1, m] \tag{7–5}$$

$$\sum_{i=1}^{m} f_i = \alpha \ \ ; \ \ f_i \geqslant 1 \ , \ \forall i \in [1, m] \tag{7–6}$$

Our algorithm consists of two steps. Since static partitioning is used, all the tasks on each core share the same L2 partition factor f_k. This fact gives us an opportunity to simplify our algorithm without losing any precision. In the **first step**, we find the optimal L1 cache assignments for the tasks on each core separately under all L2 partition factors. Specifically, we find $\boldsymbol{R}$ to minimize $E_k(f_k) = \sum_{i=1}^{\rho_k} e_{k,i}(c_1, c_2, f_k)$ constrained by $\sum_{i=1}^{\rho_k} t_{k,i}(c_1, c_2, f_k) \leqslant D$ with k and f_k fixed for $\forall p_k \in \mathcal{P}$ and $\forall f_k \in [1, \alpha - 1]$. This step (sub-problem) is illustrated in Figure 7-4 for p_m with $f_m = 2$. Similar to the uniprocessor DVS problem in Section 4.2.2, each instance of this sub-problem can be reduced from the multiple-choice knapsack problem (MCKP) and thus is NP-hard.

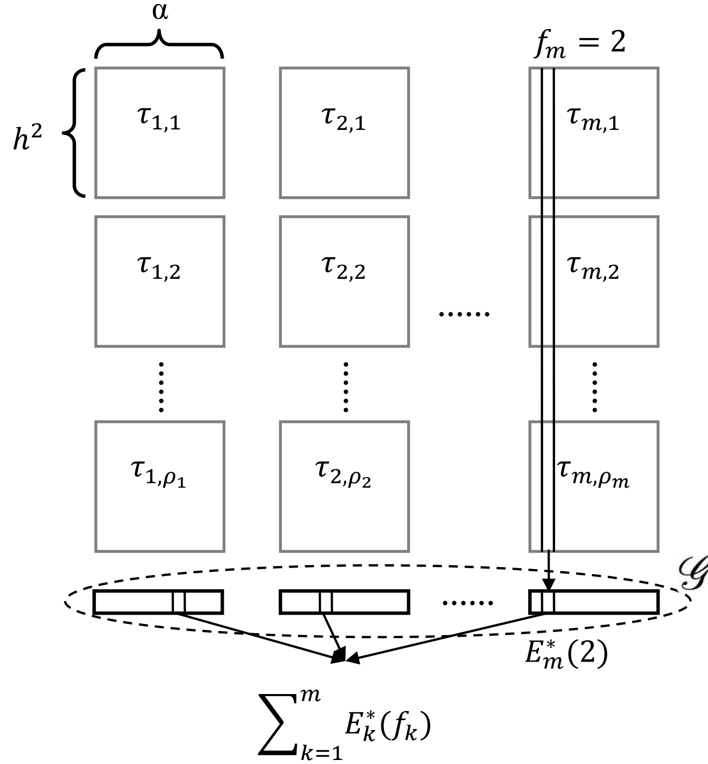


Figure 7-4. Illustration of our algorithm.

Since the subproblem size (measured by h^2, ρ_k) and the embedded application size (measured by energy value) are typically small, a dynamic programming algorithm can find the optimal solution quite efficiently as follows. Let $e_k^{max}(f_k)$ and $e_k^{min}(f_k)$ be defined as $\sum_{i=1}^{\rho_k} max\{e_{k,i}(h_1, h_2, f_k)\}$ and $\sum_{i=1}^{\rho_k} min\{e_{k,i}(h_1, h_2, f_k)\}$, respectively. Hence, $E_k(f_k)$

is bounded by $[e_k^{min}(f_k), e_k^{max}(f_k)]$. In order to guarantee the timing constraint, the energy value is discretized in our dynamic programming algorithm. Let $S_j^{E_k}$ denote the partial solution for the first j tasks which has an accumulative energy consumption equal to E_k while the execution time is minimized. We create a two-dimensional table T in which each element $T[j][E_k]$ stores the execution time of $S_j^{E_k}$. The recursive relation for dynamic programming thus is:

$$T[j][E_k] = \min_{h_1,h_2 \in [1,h]} \{T[j-1][E_k - e_{k,i}(h_1,h_2,f_k)] + t_{k,i}(h_1,h_2,f_k)\} \tag{7–7}$$

Initially, all entries in T store some value larger than D. Based on the above recursion, we fill up the table $T[j][E_k]$ in a row by row manner for all energy values in $[e_k^{min}(f_k), e_k^{max}(f_k)]$. During the process, all previous $i-1$ rows are filled when the ith row is being calculated. Finally, the optimal energy consumption $E_k^*(f_k)$ is found by:

$$E_k^*(f_k) = \{\min \ E_k \mid T[\rho_k][E_k] \leqslant D\} \tag{7–8}$$

Our algorithm iterates over all tasks in core p_k (1 to ρ_k). During each iteration, all discretized E_k values and L1 cache configurations (1 to h^2) for current task are examined. Therefore, the time complexity is $O(\rho_k \cdot h^2 \cdot (e_k^{max}(f_k) - e_k^{min}(f_k)))$. Note that energy values (reflected in the last term of the complexity) can always be measured in certain unit so that they are numerically small to make the dynamic programming efficient. The size of table T decides the memory requirement, which is $\rho_k \cdot (e_k^{max}(f_k) - e_k^{min}(f_k)) \cdot sizeof(element)$ bytes. In each entry of T, we can use minimum number of bits to remember the L1 configuration index instead of real execution time values. For calculation purposes, two two-dimensional arrays are used for temporarily storing time values for current and previous iterations. The above process is repeated for $\forall k \in [1,m]$ and $\forall f_k \in [1, \alpha-1]$[4] .

[4] If each core has at least one task, this scope can be reduced to $\forall f_k \in [1, \alpha-m+1]$ since the minimum partition factor for each core is 1.

It is possible that, for some f_k, there is no feasible solution for core p_k satisfying the deadline. We mark them as invalid. The results form a new profile table $\mathscr{G}$ for each core in which there are $[1, \alpha - 1]$ entries and each entry stores the corresponding optimal solution $E_k^*(f_k)$, as shown in Figure 7-4.

In the **second step**, the global optimal solution E^* can be found by calculating the overall energy consumption for all L2 partitioning schemes in $\boldsymbol{P}$ which complies with Equation (7–6). Given a partition factor f_k for core p_k, the optimal energy consumption $E_k^*(f_k)$ observing D has been calculated in the first step. Invalid partitioning schemes are discarded. We have $E^* = \min\{\sum_{k=1}^{m} E_k^*(f_k)\}$ for $\{f_1, f_2, ..., f_m\} \in \boldsymbol{P}$. Therefore, for each L2 partitioning scheme, the corresponding solution can be found in $O(m)$ time. Since the size of $\boldsymbol{P}$ is small (e.g., 455 and 4495 for 4 cores with 16-way and 32-way associative L2 cache, respectively), an exhaustive exploration is efficient enough for this step to find the minimum cache hierarchy energy consumption in $O(m \cdot |\boldsymbol{P}|)$ time. Otherwise, a dynamic programming algorithm can be used. Note that E^* is not strictly equal to the actual energy dissipation since the L2 cache still consumes static power in its entirety after some cores finish their tasks. Therefore, in our experimental results, we have added this portion of static energy to make it accurate. If L2 cache lines are powered off in those partitions using techniques such as cache decay [57] to save static power dissipation, E^* is already accurate. Each core along with its private caches are assumed to be turned off after it finishes execution. Algorithm 11 outlines the major steps in our DCR + CP approach.

7.3.4 Task Mapping

Since static cache partitioning is used in our approach, it is beneficial to map tasks with similar shared cache demand instead of the simple bin packing method described in Section 7.3.2 so that the shared cache is partitioned in an advantageous way for most of the time during execution. In order to characterize this demand, we define *optimal partition factor* (f_{opt}), for each benchmark, as the one larger than which the improvement of overall performance is less than a pre-defined threshold (e.g., 5%). For example, as

Algorithm 11: DCR + CP Algorithm.

```
1: for k = 1 to m do
2:   for f_k = 1 to α − 1 do
3:     for l = e_k^min(f_k) to e_k^max(f_k) do
4:       for h_1, h_2 ∈ [1, h] do
5:         if e_{k,1}(h_1, h_2, f_k) == l then
6:           if t_{k,1}(h_1, h_2, f_k) < T[1][l] then
7:             T[1][l] = t_{k,1}(h_1, h_2, f_k)
8:           end if
9:         end if
10:      end for
11:    end for
12:    for i = 2 to ρ_k do
13:      for l = e_k^min(f_k) to e_k^max(f_k) do
14:        for h_1, h_2 ∈ [1, h] do
15:          last = l − e_{k,i}(h_1, h_2, f_k)
16:          if T[i − 1][last] + t_{k,i}(h_1, h_2, f_k) < T[i][l] then
17:            T[i][l] = T[i − 1][last] + t_{k,i}(h_1, h_2, f_k)
18:          end if
19:        end for
20:      end for
21:    end for
22:    E_k^*(f_k) = min{E_k | T[ρ_k][E_k] ⩽ D}
23:  end for
24: end for
25: for all P_i{f_1, f_2, ..., f_m} ∈ P do
26:   E_i^* = Σ_{k=1}^{m} E_k^*(f_k)
27: end for
28: return min{E_i^*}
```

shown in Figure 7-5(a), we can see that f_{opt} for *swim* benchmark is around 3 since further increase in partition factor achieves very little performance improvement. Similarly, for *parser* benchmark, assigning 5 shared cache ways seems to be adequate thus $f_{opt} = 5$ for *parser*. However, *gcc* (Figure 7-5(b)) requires much larger shared cache resource since the performance keeps increasing along with the partition factor (i.e., $f_{opt} = 7$). As shown in Figure 7-5(d), *bitcount* is an extreme case in which the performance is not affected by increasing partition factor. Therefore, its f_{opt} is 1. Our study shows that L1 configuration has minor impact on each benchmark's optimal partition factor with only

minor exceptions. In other words, the performance trend normally remains the same for different partition factors when DCR is applied. Table 7-1 lists the f_{opt} values for various benchmarks.

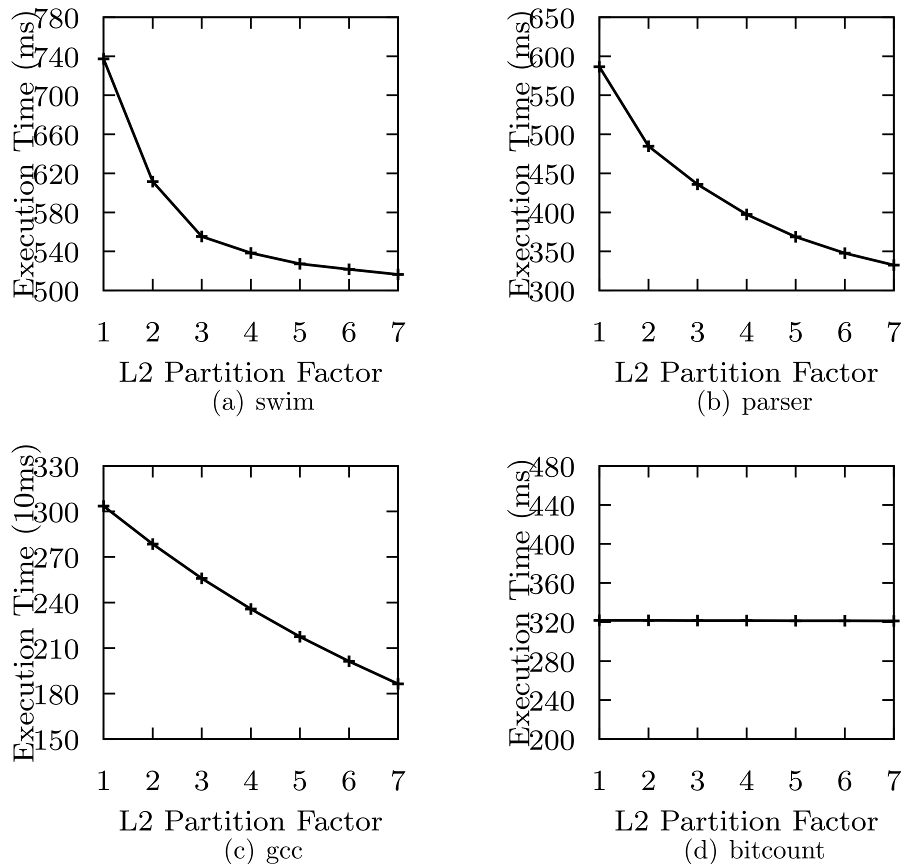


Figure 7-5. Optimal partition factor variation with L1 caches of 4KB with 2-way associativity and 32-byte line size.

To illustrate the effect of task mapping in our DCR + CP algorithm, we exhaustively examine all the task mappings for the first four task sets in Table 7-2 with two tasks per core (totally 105 possible mappings). Figure 7-6 compares the energy consumption of the worst and best task mapping schemes for each task set. For task set 2 and 3, there are a number of task mappings which cannot lead to a valid solution for the given deadline. We observe 12% - 18% differences between the two scenarios which suggest that task mapping has non-negligible impact on our DCR + CP approach.

Table 7-1. Optimal partition factors for selected benchmarks

Benchmark	f_{opt}
basicmath	7
qsort	4
ammp	4
applu	2
vpr	6
bitcount	1
sha	2
CRC32	2
dijkstra	4
toast	1
FFT	7
untoast	2
mgrid	2
lucas	3
mcf	2
gcc	7
parser	5
patricia	2
stringsearch	3
swim	3

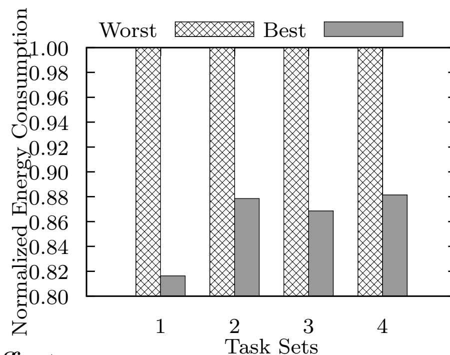


Figure 7-6. Task mapping effect.

Although it does not precisely indicate which partitioning scheme is the best, the optimal partition factor f_{opt} reflects task's shared cache requirement which can heuristically guide our task mapping scheme. Ideally, we should make the total execution time of all cores (each running multiple tasks) as close as possible. Similarly, we should also ensure that the optimal partition factors of different tasks (assigned to each core) should be as close as possible. Experimental results in Section 7.4.2.3 shows that more energy savings can be achieved by wisely grouping tasks.

7.3.5 Varying Cache Partitioning Scheme

Until now, as described in Section 7.3.1, we assume a static L2 cache partitioning scheme $\boldsymbol{P}\{f_1, f_2, ..., f_m\}$ which assigns a fixed number of ways to each core (f_i for core p_i). As shown in Table 7-1, benchmarks have different preferred partition factor (f_{opt}). Intuitively, since there are multiple tasks per core, the assigned partition factor may not be beneficial for every task of that core. For example, as shown in Figure 7-7, the first task of each core has its f_{opt} value of 2, 1, 6 and 3, respectively. A reasonable partitioning scheme for them with 8-way L2 cache would be $\boldsymbol{P}\{1, 1, 4, 2\}$. However, if the next task in each core has f_{opt} values of 7, 5, 1 and 2, respectively, $\boldsymbol{P}\{1, 1, 4, 2\}$ would obviously be inferior.

We can potentially alleviate this problem by changing the partitioning scheme at runtime for better energy efficiency and performance. For example in Figure 7-7, we can change the partitioning scheme to $\boldsymbol{P}\{3, 3, 1, 1\}$ at some point (i.e., *CP point*) which better reflects the requirement of the remaining tasks. It is a major challenge to find when and how to vary the partitioning scheme at design time.

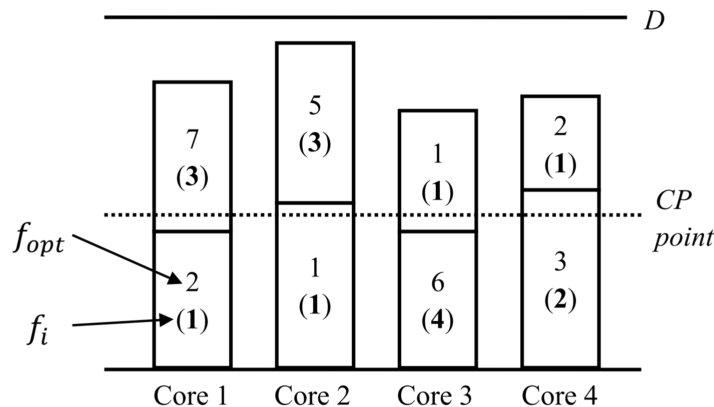


Figure 7-7. Varying partitioning scheme.

We can solve the problem in a similar way as described in Section 7.3.2 and 7.3.3 with the following modifications. Since the change of shared cache partitioning scheme happens simultaneously for all cores, it is possible that the partition factor changes during one task's execution. In order to compute the energy consumption and execution time,

we evenly divide each task into a number of checkpoints based on a fixed time interval. In other words, the distance between two neighboring checkpoints is fixed, say, one million clock cycles. During static profiling, we record the amount of energy consumed and execution progress (in dynamic instructions) up to each checkpoint. As a result, in each task's profile table, for every L1 cache configuration and L2 partition factor, there are multiple entries for all checkpoints.

We follow a two-step algorithm, namely DCR + VCP (Varying CP), similar to our DCR + CP algorithm in Section 7.3.3. At present, we assume a set of CP points is given. In the first step, we find the optimal L1 cache configurations for the tasks on each core separately under all local partitioning schemes for that core. Each local partitioning scheme F_k here consists of one partition factor for every *phase* defined by the CP points (e.g., $f_k = 3$ for the first phase from time 0 to the only CP point and $f_k = 1$ from the CP point to the end). Note that we only change the partition factor at those CP points. There are $(\alpha - 1)^{\lambda+1}$ possible local partitioning schemes[5] where α is the L2 cache associativity and λ is the number of CP points. The dynamic programming algorithm for this step is modified as follows:

1. We need to compute the actual energy consumption and execution time for each task given the varying partition factor instead of simply fetching from the profile table.
2. Task's energy consumption and execution time now depends on its start time since it determines the moment when the partition factor changes during the task's execution. Therefore, for each task $\tau_{k,i}$ with $i \in [2, \rho_k]$, we iterate all energy values of its previous task $\tau_{k,i-1}$ which gives the end time of $\tau_{k,i-1}$ (from row $i-1$ of table T) thus the start time of $\tau_{k,i}$. Specifically, line 12 to 21 of Algorithm 11 needs to be replaced with Algorithm 12. Here, $e_{k,i}(h_1, h_2, F_k, start)$ and $t_{k,i}(h_1, h_2, F_k, start)$

[5] Similarly, it could be reduced to $(\alpha - m + 1)^{\lambda+1}$.

denote the cache energy consumption and execution time for executing task $\tau_{k,i}$ using the L1 configurations, local partitioning scheme and start time.

Algorithm 12: DCR + VCP Algorithm (replaces line 12-21 of Algorithm 11).

1: **for** $i = 2$ to ρ_k **do**
2: **for** $l = e_k^{min}(F_k)$ to $e_k^{max}(F_k)$ **do**
3: **for** $h_1, h_2 \in [1, h]$ **do**
4: $start = T[i-1][l]$
5: **if** $start > e_k^{max}(F_k)$ **then**
6: continue;
7: **end if**
8: $this = l + e_{k,i}(h_1, h_2, F_k, start)$
9: **if** $T[i-1][l] + t_{k,i}(h_1, h_2, F_k, start) < T[i][this]$ **then**
10: $T[i][this] = T[i-1][l] + t_{k,i}(h_1, h_2, F_k, start)$
11: **end if**
12: **end for**
13: **end for**
14: **end for**

The second step remains the same as our DCR + CP algorithm except all possible global varying partitioning schemes are evaluated. Each phase in a global partitioning scheme defines a partition factor for each core and complies with Equation (7–6). Clearly, there are a total of $|\boldsymbol{P}|^{\lambda+1}$ such partitioning schemes and the minimum cache hierarchy energy consumption can be found in $O(m \cdot |\boldsymbol{P}|^{\lambda+1})$ time.

Note that there is certain error when we compute $t_{k,i}(h_1, h_2, F_k, start)$ since the CP points may not always align with tasks's checkpoints. As a result, the period between that pair of checkpoints observes two partition factors: one from the first checkpoint to the actual CP point and the other is from the actual CP point to the next checkpoint. We do not have this partial static profiling information since, as described above, we only record exactly at checkpoints. Therefore, we actually estimate the length of that particular time period assuming a uniform distribution of execution time and energy consumption based on the execution progress. However, since the length of the time interval (denoted by φ) between two checkpoints is significantly short compared with the tasks's length (i.e., there are hundreds of checkpoints in each task), the error introduced here can simply be

eliminated by setting the deadline $D - \lambda \cdot \varphi$. In other words, we make the deadline slightly more stringent to ensure the solution generated by DCR + VCP actually satisfies the deadline. This modification is expected to perform at par with the original version since $\lambda \cdot \varphi$ is negligible compared with D.

7.3.6 Gated-V_{dd} Shared Cache Lines

Powell et al. [85] showed that cache leakage power dissipation can be reduced by gating the supply voltage in unused portions. Cache decay exploits the sleep transistors at a granularity of individual cache lines [57]. Using this technique, we can switch off some L2 cache lines in each set whenever it is beneficial. It is especially helpful when the L2 cache has larger total capacity and associativity than what all the cores actually need (i.e., the sum of f_{opt} values of concurrent tasks is less than α) given the deadline. Specifically, the constraint ensuring the validity of $\boldsymbol{P}$ is changed to:

$$\sum_{i=1}^{m} f_i \leqslant \alpha \;\; ; \;\; f_i \geqslant 1 \; , \; \forall i \in [1, m] \tag{7–9}$$

The DCR + CP algorithm remains the same except that the number of all L2 partitioning schemes (the size of $\boldsymbol{P}$) is increased. For example, $|\boldsymbol{P}|$ becomes 1820 instead of 455 for 16-way associative L2 cache on a 4-core processor. In other words, the second step of our algorithm may take longer time but the complexity is still $O(m \cdot |\boldsymbol{P}|)$.

7.4 Experiments

7.4.1 Experimental Setup

To evaluate our approach's effectiveness, we use 20 benchmarks selected from MiBench [35] – *basicmath*, *bitcount*, *CRC32*, *dijkstra*, *FFT*, *patricia*, *qsort*, *sha*, *stringsearch*, *toast* and *untoast* – and SPEC CPU 2000 [107] – *ammp*, *applu*, *gcc*, *lucas*, *mcf*, *parser*, *swim*, *vpr* and *mgrid*. In order to make the size of SPEC benchmarks comparable with MiBench, we use reduced (but well verified) input sets from MinneSPEC [63]. Table 7-2 lists the task sets used in our experiments which are combinations of the selected benchmarks. We choose 4 task sets where each core contains 2 benchmarks, 3 task

sets where each core contains 3 benchmarks and 2 task sets where each core contains 4 benchmarks. As mentioned in Section 7.3.2, the task mapping is based on the role that the total task execution time of each core is comparable. We evaluate different task mapping strategies in Section 7.4.2.3. The deadline D is set in a way that there is a feasible L1 cache assignment for every partition factor in every core. In other words, all possible L2 partitioning schemes can be used. We will examine the effect from deadlines variation in Section 7.4.2.2.

M5 [10], a widely used architectural simulator, is adopted in our experiments. We enhanced M5 to make it support shared cache partitioning and different line sizes in different caches (IL1, DL1 and L2) to support L1 cache reconfiguration in CMP mode. We configure the simulated system with a four-core processor each of which runs at 500MHz. The TimingSimpleCPU model [10] in M5 is used which represents an in-order core which stalls during cache accesses and memory response handling. The L2 cache configuration is assumed to be 32KB, 8-way associative with 64-byte lines. The memory size is set to 256MB. The L1 cache, L2 cache and memory access latency are set to 2ns, 20ns and 200ns, respectively.

Table 7-2. Multi-task benchmark sets.

	Core 1	**Core 2**	**Core 3**	**Core 4**
Set 1	qsort, vpr	parser, toast	untoast, swim	dijkstra, sha
Set 2	mcf, sha	gcc, bitcount	patricia, lucas	basicmath, swim
Set 3	applu, lucas	dijkstra, swim	ammp, FFT	basicmath, stringsearch
Set 4	mgrid, FFT	dijkstra, parser	CRC32, swim	applu, bitcount
Set 5	mcf, toast, sha	gcc, parser, stringsearch	patricia, qsort, vpr	basicmath, CRC32, ammp
Set 6	mgrid, parser, gcc	toast, FFT, mcf	bitcount, ammp, patricia	applu, dijkstra, qsort
Set 7	vpr, sha, untoast	CRC32, lucas, qsort	mgrid, bitcount, FFT	applu, parser, stringsearch
Set 8	sha, mcf, untoast, basicmath	toast, gcc, bitcount, patricia	lucas, FFT, CRC32, ammp	vpr, applu, mgrid, swim
Set 9	gcc, stringsearch, parser, dijkstra	untoast, mcf, ammp, bitcount	lucas, patricia, qsort, vpr	basicmath, toast, applu, CRC32

7.4.2 Results

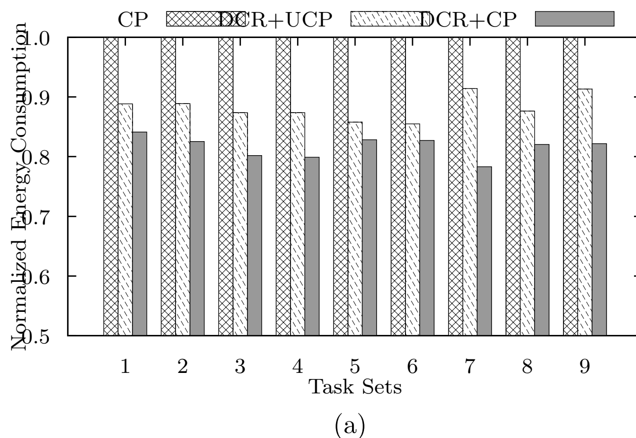
7.4.2.1 Energy Savings

We compare the following three approaches.

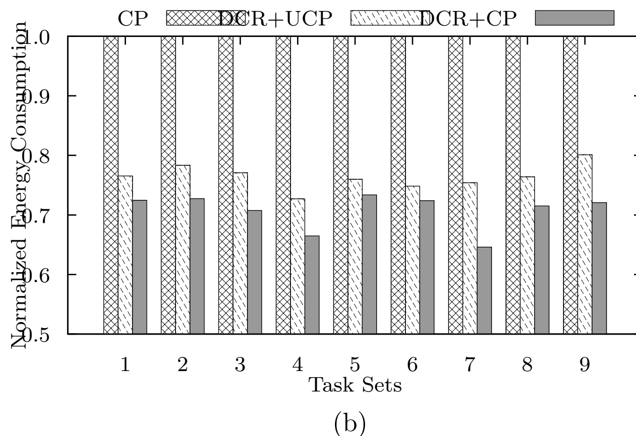
- **CP**: L2 cache partitioning only (optimal).
- **DCR + UCP**: L1 cache reconfiguration with an uniform L2 cache partitioning (our approach).
- **DCR + CP**: L1 cache reconfiguration with judicious L2 cache partitioning (our approach).

Here CP only approach uses optimal L2 partitioning scheme with all L1s in base configuration. It can be achieved using our algorithm in Section 7.3.3 without the first step. Figure 7-8 illustrates this comparison in energy consumption for all task sets in Table 7-2. Energy values are normalized to CP. As discussed in Section 7.3.2, our reconfigurable L1 cache has a base size of 4KB. Here, we examine two kinds of L1 base configurations: 4KB with 2-way associativity and 32-byte line size (4KB_2W_32B), and 4KB with 4-way associativity and 64-byte line size (4KB_4W_64B). In the former case, DCR + CP can save 18.35% of cache energy on average compared with CP. In the latter case, up to 33.51% energy saving (e.g., for task set 4) can be achieved and averagely 29.29%. Compared with DCR + UCP, our approach is able to achieve up to 14% more energy savings by carefully select cache partitioning scheme $\boldsymbol{P}$. Note that although results for only two L1 base configurations are shown here, we observe similar amount of improvements can be achieved for other base configurations (e.g., 19.30% for 2KB_2W_32B).

It is valuable to disclose the energy reduction ability of our approach. Using task set 4 in Figure 7-8 (b) as an example, CP selects the best $\boldsymbol{P}\{1,5,1,1\}$ with L1 configuration of 4KB_4W_64B. It consumes total energy of 125.8 mJ and finishes all tasks in 1600 ms. With DCR + CP, the optimal $\boldsymbol{P}\{2,4,1,1\}$ and the L1 caches are configured differently for each task. For example, *FFT* on Core 1 uses 1KB_1W_64B and 4KB_4W_16B of IL1 and



(a)



(b)

Figure 7-8. Cache hierarchy energy reduction with L1 base configuration of: (a) 4KB_2W_32B; (b) 4KB_4W_64B.

DL1, respectively, while *swim* on Core 3 uses 4KB_4W_16B and 2KB_2W_32B. Using DCR + CP, the energy requirement is reduced to 83.6 mJ and all tasks finishes in 1788 ms.

7.4.2.2 Deadline Effect

It is also meaningful to see how deadline constraint can affect the effectiveness of our approach. Using the same example above, for task set 4, we vary the deadline from 1800 ms to 1520 ms in step of 10 ms (there is no solution for deadlines shorter than 1520 ms). Figure 7-9 shows the result for both CP and DCR + CP. At each step, the reduced deadline negatively impacts the energy saving opportunity. In other words, the energy consumption increases since the configuration that was energy efficient turns invalid due to

the timing constraint. We can observe that our approach can find efficient solutions and outperforms CP consistently at all deadline levels.

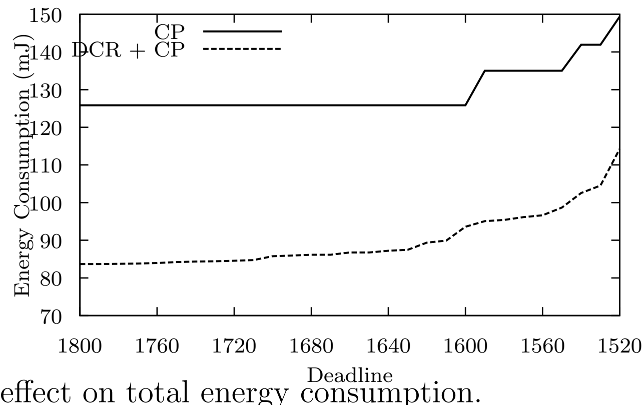


Figure 7-9. Deadline effect on total energy consumption.

7.4.2.3 Task Mapping Effect

In this section, we evaluate a simple task mapping heuristic based on optimal partition factor discussed in Section 7.3.4. For each task set in Table 7-2, benchmarks with equal or similar optimal partition factor (determined with L1 base configuration 4KB_2W_32B) are grouped together in the same core. Ideally, we would like to minimize the average variance of f_{opt} values in each core. However, it is clearly a hard problem and our heuristic does the task mapping at the best effort. We compared the solution quality of our DCR + CP approach with and without task mapping in Figure 7-10. It can be observed that partition factor aware task mapping achieves more energy saving than simple bin packing mapping. However, the improvement is not significant (up to 6.3% but on average less than 5%). The reason behind it is that, in practice, the default bin packing normally leads to a task mapping that is reasonable (note that Figure 7-6 compares the best and the worst mappings). Another reason is that when tasks are grouped based on f_{opt} values without considering their total execution time, the imbalanced workloads on each core will lead to more idle L2 cache ways and thus more static energy consumptions.

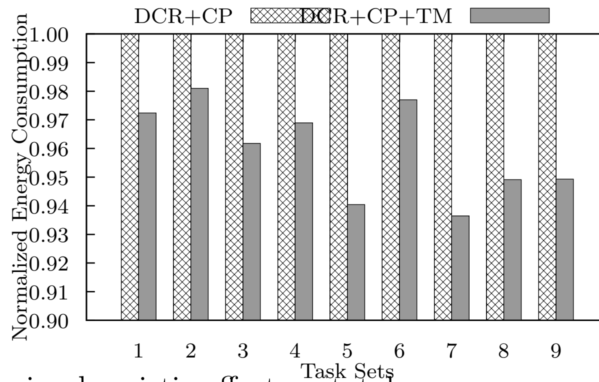


Figure 7-10. Task mapping heuristic effect on total energy consumption.

7.4.2.4 Effect of Varying Cache Partitioning

We compare DCR + CP and DCR + VCP in their energy saving ability to evaluate the varying L2 cache partitioning scheme. We make the number of CP points equal to the number of tasks per core minus one. In other words, we try to change the partition factor when a new task starts execution. However, since the partitioning scheme can only be altered simultaneously for all cores, the CP points are set based on the average start time of tasks. Specifically, for example, the first CP point is at the average start time of all the second tasks on each core under the base configuration. Figure 7-11 illustrates the results. It is worth to note that more energy savings can be achieved in scenarios with more tasks per core (12.1% versus 6.4% on average). The reason is that tasks sets with diverse f_{opt} values make varying partition factor more advantageous.

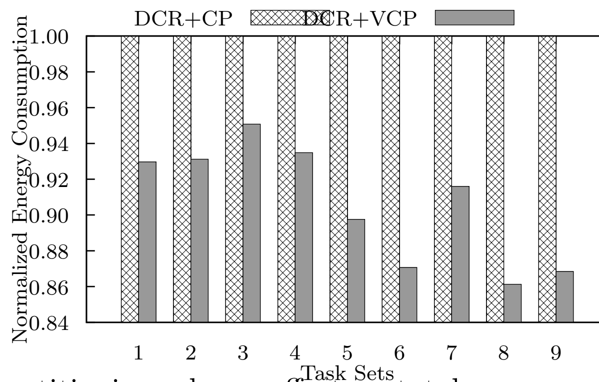


Figure 7-11. Varying partitioning scheme effect on total energy consumption.

The task mapping heuristic and varying partitioning scheme actually compete with each other. As described in Section 7.3.4, a good task mapping scheme makes f_{opt} values of tasks in each core as close as possible. This fact compromises the advantage of changing the partition factor at runtime. In our study, we observe less additional energy savings applying DCR + VCP on systems with ideal task mappings. On the other hand, if DCR + VCP is employed, the task mapping heuristic shows limited benefit. Therefore, in practice, one of the two alternatives can be selected for further improvements.

7.4.2.5 Gated-V_{dd} Cache Lines Effect

Switching-off unnecessary cache lines in our original L2 configuration (32KB with 8-way associativity) leads to limited additional energy saving since we can hardly shut down any cache line given the timing constraint. Therefore, we evaluate the effect from gated-V_{dd} shared cache lines using a larger L2 configuration (64KB with 16-way associativity). Figure 7-12 shows the result assuming a fixed partitioning scheme. It can be observed that 12% cache hierarchy energy consumption can be saved on average.

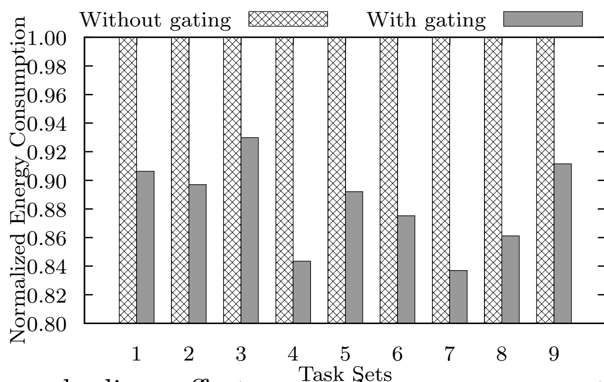


Figure 7-12. Gated-V_{dd} cache lines effect on total energy consumption.

In our study, we also observe that the energy saving achieved using power gating technique reduces if the deadline becomes more stringent. It is expected because a tight timing constraint requires utilizing more shared cache resources to maximize performance.

7.5 Summary

In this chapter, we presented an efficient approach to integrate dynamic cache reconfiguration and partitioning for real-time multicore systems. We discovered that there is a strong correlation between L1 DCR and L2 CP in CMPs. While CP is effective in reducing inter-task interferences, DCR can further improve the energy efficiency without violating timing constraints (i.e., deadlines). Our static profiling technique drastically reduces the exploration space without losing any precision. Our DCR + CP algorithm, which can find the optimal L1 configurations for each task and L2 partition factors for each core, is based on dynamic programming with discretization of energy values. Moreover, we explored varying partitioning scheme and shared cache line power gating for more energy savings. We also studied the effect of deadline variation and task mapping based on shared cache resource demand of each task. Extensive experimental results demonstrate the effectiveness of our approach (29.29 - 36.01% average energy savings).

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

Energy consumption is one of the most important design issues in industry. Energy awareness and optimization techniques are critical especially for embedded systems which normally have various constraints. Real-time systems which run applications with timing constraints demand unique design considerations. This dissertation presented a set of novel tools, techniques and methodologies for energy, power and thermal optimization in real-time embedded systems. This chapter concludes this dissertation and outlines possible future research directions.

8.1 Conclusions

Dynamic reconfiguration is successful in various system optimizations. Processor and cache subsystem are the two most significant contributors in overall system energy consumption. Dynamic cache reconfiguration (DCR) and dynamic voltage scaling (DVS) are the major techniques for cache subsystem and processor energy optimization, respectively. However, due to various constraints (e.g., timing, energy, thermal) in real-time embedded system design, it is a major challenge to decide ***when*** and ***how*** to reconfigure the system so that lower power dissipation, higher performance and lower peak temperature can be achieved. Fortunately, as discussed in Chapter 1, there are various optimization opportunities that can be exploited based on dynamic reconfiguration techniques. This dissertation's contributions are summarized as follows.

Chapter 2 described general system models, energy models for different system components and thermal models that are used throughout this dissertation. Chapter 3 presented SACR – a scheduling-aware cache reconfiguration approach – for soft real-time systems in which minor deadline violations are acceptable. Both statically and dynamically scheduled systems are studied. Our approach employed a phase-based static profiling technique whose outputs are effectively utilized during runtime to guide cache reconfiguration. We also developed efficient heuristics for design space exploration of multi-level cache

hierarchy reconfiguration. Chapter 4 proposed two algorithms for DVS in hard real-time systems. To exploit static time slack, we proposed a novel DVS scheme for preemptive task sets named PreDVS. PreDVS assigns voltage levels to tasks at a finer granularity and can achieve more energy savings than existing inter-task DVS schemes without additional runtime overhead. Our approach is based on an approximation algorithm which can guarantee to give close-to-optimal solutions. In the same chapter, we also devised an efficient dynamic slack reclamation algorithm which allocates slacks more judiciously and aggressively than existing approaches at runtime.

Chapter 5 described our study in systematic integration of DVS and DCR for system-wide energy optimization. Our energy estimation framework takes all major system components – processor, cache subsystem, buses and main memory – into consideration. Previous studies have shown that a critical speed exists below which the processor voltage level cannot be reduced to avoid leakage power dominating the processor energy savings. We found that, with respect to overall system consumption, the critical speed drastically increases when other components and DCR are accounted. Therefore, DCR and DVS, along with task procrastination, could be employed together for system-wide optimizations. Based on this study, we also proposed a general and flexible algorithm for dynamic reconfiguration in real-time systems.

High temperature on chip will result in decrease in reliability, performance and energy efficiency. In Chapter 6, we proposed a novel formal method based DVS algorithm in temperature- and energy-constrained systems. The goal of our approach is to find a valid voltage scaling scheme for a real-time task set given the timing, energy and peak temperature constraints. We modeled the problem using extended timed automata, which is verified by the model checker. If there exists a solution, our approach can generate the corresponding DVS scheme as the witness trace.

Chapter 7 presented our research in energy optimization of the cache hierarchy in real-time multicore systems. We effectively integrated DCR and CP simultaneously. Our

approach is designed to find beneficial configurations for private caches in each core and the partition scheme for the shared cache so that the energy consumption is minimized while the timing constraints are satisfied. We also studied the impact of varying CP at runtime, task mapping heuristics and Gated-V_{dd} cache lines.

In conclusion, this dissertation presented a comprehensive and cohesive study of energy optimization in real-time embedded systems. We developed a set of efficient techniques and applied on a wide a variety of systems to significantly improve overall energy consumption, system performance and thermal constraints. Our research will lead to real-time systems with higher power efficiency, performance and reliability.

8.2 Future Research Directions

Energy consumption has been and will continue to be a critical design issue. The research presented in this dissertation can be extended in the following possible directions:

- For dynamically scheduled systems which run aperiodic or sporadic tasks, employing DCR will lead to a small amount of deadline violations. Therefore, in this case, the proposed technique only works for soft real-time systems. We believe it is unlikely that DCR can be enabled in dynamically scheduled hard real-time systems with preemptive workloads. However, further studies are needed to check whether it can work for nonpreemptive tasks.

- Currently, our cache reconfiguration technique for both uniprocessor and multicore systems is based on static profiling which requires extensive design time and certain assumptions (e.g., known inputs). We have discussed that existing dynamic analysis techniques do not work for real-time systems. It is challenging to design an efficient, predictable and nonintrusive online cache performance analyzer.

- We have pointed out that PreDVS can be employed together with intra-task DVS techniques for runtime slack exploitation. Although some of the techniques are relatively independent from each other, our research can be extended by comprehensively integrating PreDVS, intra-task DVS, dynamic slack reclamation and task rescheduling for overall energy optimizations.

- Our approach for temperature- and energy-constrained scheduling can be further extended to support multicore architectures. We need to take thermal transmission among on-chip cores into consideration. Task mapping and sequencing may also play an important role in this scenario. Since the TCEC scheduling in uniprocessor

systems is NP-hard and the extended problem on multicore processors is even more difficult, an approximation algorithm is the best approach.

APPENDIX A
LIST OF PUBLICATIONS

Book

1. **Weixun Wang**, Xiaoke Qin and Prabhat Mishra. *Dynamic Reconfiguration in Real-Time Systems: Energy, Performance, Reliability and Thermal Perspectives* (Tentative title). Springer, 2012 (Expected).

Book Chapter

1. **Weixun Wang**, Xiaoke Qin and Prabhat Mishra. *Energy-Aware Scheduling and Dynamic Reconfiguration in Real-Time Systems.* Handbook of Energy-Aware and Green Computing, I. Ahmad and S. Ranka, Editors, Chapman & Hall/CRC Press, 2011.

Journal Articles

1. **Weixun Wang**, Prabhat Mishra and Ann Gordon-Ross. *Dynamic Cache Reconfiguration for Soft Real-Time Systems.* ACM Transactions in Embedded Computing Systems (TECS), accepted to appear.

2. **Weixun Wang** and Prabhat Mishra. *System-Wide Leakage-Aware Energy Minimization using Dynamic Voltage Scaling and Cache Reconfiguration in Multitasking Systems.* IEEE Transactions on Very Large Scale Integration Systems (TVLSI), accepted to appear.

3. **Weixun Wang** and Prabhat Mishra. *Dynamic Reconfiguration of Two-Level Cache Hierarchy in Real-Time Embedded Systems.* ASP Journal of Low Power Electronics (JOLPE), Vol. 7, No. 1, February 2011.

4. **Weixun Wang**, Sanjay Ranka and Prabhat Mishra. *Energy-Aware Dynamic Reconfiguration Algorithms for Real-Time Multitasking Systems.* Elsevier Sustainable Computing: Informatics and Systems (SUSCOM), Issue. 1, pages 35-45, 2011 (Invited Paper).

Conference Papers

1. **Weixun Wang**, Prabhat Mishra and Sanjay Ranka. *Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-Time Multi-Core Systems.* IEEE/ACM Design Automation Conference (DAC), pages -, Jun. 2011.

2. **Weixun Wang**, Sanjay Ranka and Prabhat Mishra. *A General Algorithm for Energy-Aware Dynamic Reconfiguration in Multitasking Systems.* IEEE International Conference on VLSI Design (VLSI Design), pages 334-339, Jan. 2011.

3. **Weixun Wang**, Xiaoke Qin and Prabhat Mishra. *Temperature- and Energy-Constrained Scheduling in Multitasking Systems: A Model Checking Approach.* IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), pages 85-90, Aug. 2010.

4. **Weixun Wang** and Prabhat Mishra. *PreDVS: Preemptive Dynamic Voltage Scaling for Real-time Systems using Approximation Scheme.* IEEE/ACM Design Automation Conference (DAC), pages 705-710, Jun. 2010.

5. **Weixun Wang** and Prabhat Mishra. *Leakage-Aware Energy Minimization using Dynamic Voltage Scaling and Cache Reconfiguration in Real-Time Systems.* IEEE International Conference on VLSI Design (VLSI Design), pages 357-362, Jan. 2010.

6. **Weixun Wang** and Prabhat Mishra. *Dynamic Reconfiguration of Two-Level Caches in Soft Real-Time Embedded Systems.* IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pages 145-150, May. 2009.

7. **Weixun Wang**, Prabhat Mishra and Ann Gordon-Ross. *SACR: Scheduling-Aware Cache Reconfiguration for Real-Time Embedded Systems.* IEEE International Conference on VLSI Design (VLSI Design), pages 547-552, Jan. 2009.

Technical Reports

1. **Weixun Wang** and Prabhat Mishra. *A Partitioned Bitmask-based Technique for Lossless Seismic Data Compression.* CISE Technical Report # 08-452, University of Florida, May 07, 2008.

Under Review

1. **Weixun Wang**, Prabhat Mishra and Sanjay Ranka. *Energy Optimization of Cache Hierarchy in Real-Time Multicore Systems.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), Under review.

2. Xiaoke Qin, **Weixun Wang** and Prabhat Mishra. *TCEC: Temperature- and Energy-Constrained Scheduling in Real-Time Multitasking Systems.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), Under review.

3. **Weixun Wang** and Prabhat Mishra. *PreDVS: Preemptive Dynamic Voltage Scaling for Real-time Multitasking Systems.* ACM Transactions on Design Automation of Electronic Systems (TODAES), Under review.

4. **Weixun Wang**, Sanjay Ranka and Prabhat Mishra. *Energy-Aware Dynamic Slack Allocation for Real-Time Multitasking Systems.* Elsevier Sustainable Computing: Informatics and Systems (SUSCOM), Under review.

REFERENCES

[1] AeA (formerly American Electronics Association) Report Cybernation.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] ARM. ARM11MPCore processor.

[4] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proc. Real-Time Systems, 13th Euromicro Conference on*, pages 225–232, 13–15 June 2001.

[5] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of Real-Time Systems Symposium*, pages 95–105, 2001.

[6] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, May 2004.

[7] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[8] L. Benini, R. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8:299–316, 2000.

[9] D. Bertozzi, L. Benini, and G. de Micheli. Low power error resilient encoding for on-chip data buses. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 102, Washington, DC, USA, 2002. IEEE Computer Society.

[10] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The m5 simulator: Modeling networked systems. *Micro, IEEE*, 26(4):52 –60, 2006.

[11] S. Borkar. Design challenges of technology scaling. 19(4):23–29, July 1999.

[12] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proc. Design Automation Conference*, pages 338–342, June 2–6, 2003.

[13] B. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pages 101 –110, aug. 2008.

[14] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical report, University of Wisconsin-Madison, 1996.

[15] J. A. Butts and G. S. Sohi. A static power model for architects. In *Proc. 33rd Annual IEEE/ACM International Symposium on MICRO-33 Microarchitecture*, pages 191–201, 10–13 Dec. 2000.

[16] J. Chen, T. Kuo, and C. Shih. $1 + \varepsilon$ approximation clock rate assignment for periodic real-time tasks on a voltage-scaling processor. In *Proceedings of International Conference on Embedded Software*, pages 247–250, 2005.

[17] J.-J. Chen, C.-M. Hung, and T.-W. Kuo. On the minimization fo the instantaneous temperature for periodic real-time tasks. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. 13th IEEE*, pages 236–248, April 2007.

[18] J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Proc. 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications RTCSA 2007*, pages 28–38, 21–24 Aug. 2007.

[19] J.-J. Chen and T.-W. Kuo. Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 153–162, New York, NY, USA, 2006. ACM.

[20] J.-J. Chen and T.-W. Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *Proc. IEEE/ACM International Conference on Computer-Aided Design ICCAD 2007*, pages 289–294, 4–8 Nov. 2007.

[21] J.-H. Chern, J. Huang, L. Arledge, P.-C. Li, and P. Yang. Multilevel metal capacitance models for cad design synthesis systems. *Electron Device Letters, IEEE*, 13(1):32 –34, jan 1992.

[22] J.-W. Chi, C.-L. Yang, Y.-J. Chen, and J.-J. Chen. Cache leakage control mechanism for hard real-time systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 248–256, New York, NY, USA, 2007. ACM.

[23] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[24] B. Doyle, R. Arghavani, D. Barlage, S. Datta, M. Doczy, J. Kavalieros, A. Murthy, and R. Chau. Transistor elements for 30nm physical gate lengths and beyond. *Intel Technology Journal*, 6:42–54, 2002.

[25] EEMBC. EEMBC, The Embedded Microprocessor Benchmark Consortium, 2000.

[26] M. J. Ellsworth. Chip power density and module cooling technology projections for the current decade. In *Proc. Ninth Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems ITHERM '04*, volume 2, pages 707–708, June 1–4, 2004.

[27] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–82, London, UK, 2002. Springer-Verlag.

[28] W. Fornaciari, D. Sciuto, and C. Silvano. Power estimation for architectural exploration of hw/sw communication on system-level buses. In *Proc. Seventh International Workshop on Hardware/Software Codesign (CODES '99)*, pages 152–156, May 3–5, 1999.

[29] T. D. Givargis, F. Vahid, and J. Henkel. Fast cache and bus power estimation for parameterized system-on-a-chip design. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 333–339, New York, NY, USA, 2000. ACM.

[30] A. Gordon-Ross and F. Vahid. Automatic tuning of two-level caches to embedded applications. In *Proceedings of Design, Automation and Test Conference in Europe*, pages 208–213, 2004.

[31] A. Gordon-Ross and F. Vahid. A self-tuning configurable cache. In *Proceedings of Design Automation Conference*, pages 234–237, 2007.

[32] A. Gordon-Ross, F. Vahid, and N. Dutt. Fast configurable-cache tuning with a unified second-level cache. In *Proc. International Symposium on Low Power Electronics and Design ISLPED '05*, pages 323–326, 8–10 Aug. 2005.

[33] A. Gordon-Ross, P. Viana, F. Vahid, W. Najjar, and E. Barros. A one-shot configurable-cache tuner for improved energy and performance. In *Proceedings of Design, Automation and Test Conference in Europe*, pages 755–760, 2007.

[34] S. Gunther, F. Binns, D. Carmean, and J. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 5(1):1–9, 2001.

[35] M. Guthaus, J. Ringenberg, D.Ernest, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[36] E. Hallnor and S. Reinhardt. A unified compressed memory hierarchy. pages 201 – 212, 2005.

[37] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 2003.

[38] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Power optimization of variable-voltage core-based systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18:1702–1714, 1999.

[39] I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastavas. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proc. 19th IEEE Real-Time Systems Symposium*, pages 178–187, 2–4 Dec. 1998.

[40] S. Hong, S. Yoo, H. Jin, K. Choi, J. Kong, and S. Eo. Runtime distribution-aware dynamic voltage scaling. In *Proceedings of International Conference on Computer-Aided Design*, pages 587–594, 2006.

[41] HP. *CACTI, HP Laboratories Palo Alto, CACTI 5.3*. http://www.hpl.hp.com/, 2008.

[42] J. Hu and R. Marculescu. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Proceedings of Design, Automation and Test Conference in Europe*, pages 234–239, 2004.

[43] Intel. Intel Core i7 processor.

[44] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. *ACM Transactions on Algorithms*, 3, 2007.

[45] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 197–202, 1998.

[46] R. Jayaseelan and T. Mitra. Temperature aware task sequencing and voltage scaling. In *Proc. IEEE/ACM International Conference on Computer-Aided Design ICCAD 2008*, pages 618–623, 10–13 Nov. 2008.

[47] R. Jejurikar and R. Gupta. Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems. In *Proc. International Symposium on Low Power Electronics and Design ISLPED '04*, pages 78–81, 9–11 Aug. 2004.

[48] R. Jejurikar and R. Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings of Design Automation Conference*, pages 111–116, 2005.

[49] R. Jejurikar and R. Gupta. Energy aware non-preemptive scheduling for hard real-time systems. In *Proc. 17th Euromicro Conference on Real-Time Systems (ECRTS 2005)*, pages 21–30, 6–8 July 2005.

[50] R. Jejurikar and R. Gupta. Energy-aware task scheduling with task synchronization for embedded real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25:1024–1037, 2006.

[51] R. Jejurikar, C. Pereira, and R. K. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of Design Automation Conference*, pages 275–280, 2004.

[52] N. Jha. Low power system scheduling and synthesis. In *Proceedings of International Conference on Computer-Aided Design*, pages 259–263, 2001.

[53] Y. Joo, Y. Choi, H. Shim, H. G. Lee, K. Kim, and N. Chang. Energy exploration and reduction of sdram memory systems. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 892–897, New York, NY, USA, 2002. ACM.

[54] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49:589–604, 2005.

[55] J. Kao, S. Narendra, and A. Chandrakasan. Subthreshold leakage modeling and reduction techniques. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 141–148, New York, NY, USA, 2002. ACM.

[56] D. Kaseridis, J. Stuecheli, and L. John. Bank-aware dynamic cache partitioning for multicore architectures. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 18 –25, sep. 2009.

[57] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 240–251, New York, NY, USA, 2001. ACM.

[58] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer-Verlag, 2004.

[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. *Computer*, 36(12):68–75, 2003.

[60] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches. leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 219–230, 2002.

[61] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, pages 111 – 122, 29 2004.

[62] W. Kim, J. Kim, and S. Min. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of Design, Automation and Test Conference in Europe*, page 788, 2002.

[63] A. KleinOsowski and D. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(1):7 – 7, january-december 2002.

[64] W.-C. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM Trans. Embed. Comput. Syst.*, 4(1):211–230, 2005.

[65] K. Lahiri and A. Raghunathan. Power analysis of system-level on-chip communication architectures. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 236–241, New York, NY, USA, 2004. ACM.

[66] C. Lee, M. Potkonjak, and W. H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of International Symposium on Microarchitecture*, pages 330–335, 1997.

[67] Y.-H. Lee, K. Reddy, and C. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 105–112, July 2003.

[68] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 367 –378, 16-20 2008.

[69] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[70] Y. Liu, H. Yang, R. P. Dick, H. Wang, and L. Shang. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In *Proc. 8th International Symposium on Quality Electronic Design ISQED '07*, pages 204–209, Mar. 26–28, 2007.

[71] A. Lungu, P. Bose, D. J. Sorin, S. German, and G. Janssen. Multicore power management: Ensuring robustness via early-stage formal verification. In *Proc. 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design MEMOCODE '09*, pages 78–87, 13–15 July 2009.

[72] A. Malik, B. Moyer, and D. Cermak. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 241–243, 2000.

[73] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic

workloads. In *Proc. IEEE/ACM International Conference on Computer Aided Design ICCAD 2002*, pages 721–725, 10–14 Nov. 2002.

[74] Marvell. Marvell StrongARM 1100 processor, 1997.

[75] Marvell. Marvell XScale microarchitecture, 2000.

[76] P. Mejia-Alvarez, E. Levner, and D. Mosse. Adaptive scheduling server for power-aware real-time tasks. *ACM Transactions on Embedded Computing Systems*, 3, 2004.

[77] MIPS. MIPS32 1004K.

[78] B. Mochocki, X. Hu, and G. Quan. A unified approach to variable voltage scheduling for nonideal dvs processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23, 2004.

[79] B. Mochocki, X. Hu, and G. Quan. Practical on-line dvs scheduling for fixed-priority real-time systems. In *Proceedings of Real Time and Embedded Technology and Applications Symposium*, pages 224–233, 2005.

[80] A. C. Nacul and T. Givargis. Dynamic voltage and cache reconfiguration for low power. In *Proceedings of Design, Automation and Test Conference in Europe*, page 21376, 2004.

[81] H. Noori, M. Goudarzi, K. Inoue, and K. Murakami. The effect of temperature on cache size tuning for low energy embedded systems. In *Proceedings of Great Lakes Annual Symposium on VLSI Design (GLSVLSI)*, 2007.

[82] C. Norström, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 182, Washington, DC, USA, 1999. IEEE Computer Society.

[83] S. Oh, J. Kim, S. Kim, and C. Kyung. Task partitioning algorithm for intra-task dynamic voltage scaling. In *Proceedings of International Symposium on Circuits and Systems*, pages 1228–1231, 2008.

[84] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *ACM SIGOPS Operating Systems Review*, pages 89–102, 2001.

[85] 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 *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 90–95, New York, NY, USA, 2000. ACM.

[86] I. Puant. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proceedings of International Workshop on worst-case execution time analysis*, 2002.

[87] I. Puant and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 114–125, 2002.

[88] I. Puant and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, 2007.

[89] G. Quan and X. S. Hu. Energy efficient dvs schedule for fixed-priority real-time systems. *ACM Transactions on Design Automation of Electronic Systems*, 6:1–30, 2007.

[90] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 423 –432, dec. 2006.

[91] R. Rao, H. Deogun, D. Blaauw, and D. Sylvester. Bus encoding for total power reduction using a leakage-aware buffer configuration. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(12):1376 – 1383, dec. 2005.

[92] R. Reddy and P. Petrov. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2007.

[93] R. Reddy and P. Petrov. Cache partitioning for energy-efficient and interference-free embedded multitasking. *ACM Trans. Embed. Comput. Syst.*, 9(3):1–35, 2010.

[94] P. Rong and M. Pedram. Energy-aware task scheduling and dynamic voltage scaling in a real-time system. *Journal of Low Power Electronics*, 4:1–10, 2008.

[95] C. Rusu, R. Melhem, and D. Mosse. Maximizing the system value while satisfying time and energy constraints. In *Proc. 23rd IEEE Real-Time Systems Symposium RTSS 2002*, pages 246–255, 3–5 Dec. 2002.

[96] S. Segars. Low power design techniques for microprocessors. In *Proceedings of International Solid State Circuit Conference*, 2001.

[97] J. Seo, T. Kim, and K. Chung. Profile-based optimal intra-task voltage scheduling for hard real-time applications. In *Proceedings of Design Automation Conference*, pages 87–92, 2004.

[98] A. Settle, D. Connors, and E. Gibert. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing*, 2:221–233, 2006.

[99] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. In *Proceedings of International Symposium on Microarchitecture*, pages 84–93, 2003.

[100] D. Shin and J. Kim. Optimizing intratask voltage scheduling using profile and data-flow information. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26:369–385, 2007.

[101] D. Shin, J. Kim, and S. Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of Design Automation Conference*, pages 438–443, 2001.

[102] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of International Conference on Computer-Aided Design*, pages 365–368, 2000.

[103] S. Shukla and R. Gupta. A model checking approach to evaluating system level dynamic power management policies for embedded systems. In *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International*, pages 53–57, 2001.

[104] A. Sinha and A. P. Chandrakasan. Jouletrack: a web based tool for software energy profiling. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 220–225, New York, NY, USA, 2001. ACM.

[105] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware computer systems: Opportunities and challenges. *IEEE Micro*, 23(6):52–61, Nov.–Dec. 2003.

[106] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, 2004.

[107] SPEC. SPEC CPU2000.

[108] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings of Euromicro Conference on Real-Time Systems*, pages 41–48, 2005.

[109] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. pages 117 – 128, feb. 2002.

[110] V. Swaminathan and K. Chakrabarty. Network flow techniques for dynamic voltage scaling in hard real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(10):1385–1398, Oct. 2004.

[111] C. Talarico, J. W. Rozenblit, V. Malhotra, and A. Stritter. A new framework for power estimation of embedded systems. *Computer*, 38(2):71–78, 2005.

[112] Y. Tan and V. J. Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embedded Computing Systems*, 6(7), 2007.

[113] Transmeta. Transmeta Crusoe Processor.

[114] A. Varma, E. Debes, I. Kozintsev, and B. Jacob. Instruction-level power dissipation in the intel xscale embedded microprocessor. In *In SPIEs 17th Annual Symposium on Electronic Imaging Science & Technology*, 2005.

[115] H. J. M. Veendrick. Short-circuit dissipation of static cmos circuitry and its impact on the design of buffer circuits. *IEEE Journal of Solid-State Circuits*, 19(4):468–473, Aug 1984.

[116] L. Villa, M. Zhang, and K. Asanovic. Dynamic zero compression for cache energy reduction. pages 214 –220, 2000.

[117] R. Viswanath, V. Wakharkar, A. Watwe, and V. Lebonheur. Thermal performance challenges from silicon to systems. *Intel Technology Journal*, 4(3):1–16, 2000.

[118] S. Wang and R. Bettati. Reactive speed control in temperature-constrained real-time systems. In *Proc. 18th Euromicro Conference on Real-Time Systems*, pages 10pp.–170, 2006.

[119] W. Wang and P. Mishra. Dynamic reconfiguration of two-level caches in soft real-time embedded systems. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, pages 145–150, 2009.

[120] W. Wang and P. Mishra. Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems. In *Proceedings of IEEE International Conference on VLSI Design*, pages 357–362, 2010.

[121] W. Wang and P. Mishra. PreDVS: Preemptive dynamic voltage scaling for real-time systems using approximation scheme. In *Proceedings of Design Automation Conference*, pages 705–710, 2010.

[122] W. Wang, P. Mishra, and A. Gordon-Ross. SACR: Scheduling-aware cache reconfiguration for real-time embedded systems. In *Proceedings of IEEE International Conference on VLSI Design*, pages 547–552, 2009.

[123] Y.-H. Wei, C.-Y. Yang, T.-W. Kuo, S.-H. Hung, and Y.-H. Chu. Energy-efficient real-time scheduling of multimedia tasks on multi-core processors. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 258–262, New York, NY, USA, 2010. ACM.

[124] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 2004.

[125] A. Wolfe. Software-based cache partitioning for real-time applications. In *Proceedings of International Workshop on Responsive Computer Systems*, 1993.

[126] F. Xie, M. Martonosi, and S. Malik. Bounds on power savings using runtime dynamic voltage scaling: an exact algorithm and a linear-time heuristic

approximation. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 287–292, 2005.

[127] R. Xu, C. Xi, R. Melhem, and D. Mosse. Practical pace for embedded systems. In *Proceedings of International Conference on Embedded Software*, pages 54–63, 2004.

[128] L. Yan, J. Luo, and N. K. Jha. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. In *Proc. ICCAD-2003 Computer Aided Design International Conference on*, pages 30–37, 9–13 Nov. 2003.

[129] C.-Y. Yang, J.-J. Chen, T.-W. Kuo, and L. Thiele. An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In *ACM/IEEE Conference of Design, Automation, and Test in Europe (DATE)*, Nice, France, 2009.

[130] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of Annual Symposium on Foundations of Computer Science*, pages 374–382, 1995.

[131] L.-T. Yeh and R. C. Chu. *Thermal Management of Microelectronic Equipment: Heat Transfer Theory, Analysis Methods, and Design Practices*. ASME Press, 2002.

[132] C. Yu and P. Petrov. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 132–137, New York, NY, USA, 2010. ACM.

[133] L. Yuan, S. Leventhal, and G. Qu. Temperature-aware leakage minimization technique for real-time systems. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 761–764, New York, NY, USA, 2006. ACM.

[134] L. Yuan and G. Qu. Alt-dvs: Dynamic voltage scaling with awareness of leakage and temperature for real-time systems. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 660–670, Aug. 2007.

[135] C. Zhang, F. Vahid, and R. Lysecky. A self-tuning cache architecture for embedded systems. In *Proceedings of Design, Automation and Test Conference in Europe*, page 10142, 2004.

[136] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache for low energy embedded systems. *ACM Transactions on Embedded Computing Systems*, 6:362–387, 2005.

[137] S. Zhang, K. Chatha, and G. Konjevod. Approximation algorithms for power minimization of earliest deadline first and rate monotonic schedules. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 225–230, 2007.

[138] S. Zhang and K. S. Chatha. Approximation algorithm for the temperature aware scheduling problem. In *Proceedings of International Conference on Computer-Aided Design*, pages 281–288, 2007.

[139] X. Zhong and C. Xu. Energy-aware modeling and scheduling of real-time tasks for dynamic voltage scaling. In *Proceedings of International Real-Time Systems Symposium*, pages 366–375, 2005.

[140] X. Zhong and C. Xu. System-wide energy minimization for real-time tasks: Lower bound and approximation. In *Proceedings of International Conference on Computer-Aided Design*, pages 516–521, 2006.

[141] Y. Zhu and F. Mueller. Feedback edf scheduling exploiting dynamic voltage scaling. In *Proceedings of Real Time and Embedded Technology and Applications Symposium*, pages 84–93, 2004.

[142] J. Zhuo and C. Chakrabarti. System-level energy-efficient dynamic task scheduling. In *Proceedings of Design Automation Conference*, pages 628–631, 2005.

BIOGRAPHICAL SKETCH

Weixun Wang received his B.E. degree from the Software Institute of Nanjing University, China in 2007. His research interests include the area of design automation of embedded systems with focuses on dynamic cache reconfiguration, energy optimization, temperature management, real-time scheduling and lossless data compression.

Since 2007, he started pursuing his Ph.D. degree in CISE department of University of Florida. He joined the Embedded Systems Group in 2008 under the supervision of Prof. Prabhat Mihsra. He participated the research project titled "Dynamic Reconfiguration in Real-Time Embedded Systems" which was funded by U.S. National Science Foundation and Semiconductor Research Corporation.

Mr. Wang currently serves as a reviewer of several ACM and IEEE conferences including DAC, DATE, CODES+ISSS, ASP-DAC, GLSVLSI, VLSI Design, and ISVLSI. He also serves as a reviewer for journals including IEEE TCAD and ASP JOLPE. He is a student member of IEEE.