Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.

2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.

3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

Pao Yue-kong Library, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

http://www.lib.polyu.edu.hk
Compiler-Assisted High Performance and Low Power Optimizations for Embedded Systems

By

WANG MENG

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

July 2009
CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____________________________________(Signature)

_____________________________________(Name of Student)
Embedded systems are used in a wide spectrum of applications, ranging from mobile consumer electronics to vehicle controllers. These systems are application specific, and have strict timing and power constraints. Designing high performance and low power embedded systems with various constraints and limited resources has become an important research problem. In this thesis, we investigate the challenging issues in designing compiler-assisted techniques for solving high performance and low power optimization problems in embedded systems.

In high performance optimization, we focus on reducing the number of memory accesses for embedded systems. The memory access significantly limits the performance of embedded systems due to the widening processor-memory gap. Besides performance, memory accesses consume a large fraction of overall power consumption. With the emergence of memory-intensive embedded applications, effective optimization techniques are required to reduce the number of memory accesses. In this thesis, we make the following original contributions in this field.

- First, we develop a general compiler optimization technique called “REALM” to reduce the number of memory accesses for Digital Signal Processing (DSP) applications with loops. In the loop kernels of DSP applications, one important characteristic is that the same memory location is repeatedly accessed by different memory operations over multiple loop iterations. For DSP applications, therefore, an important problem is how to explore redundant memory accesses and eliminate them by exploiting the desired value across iterations. We solve this problem by replacing redundant memory operations with register operations. The results show that our technique can effectively
reduce the number of memory accesses and improve performance compared with previous approaches.

- Second, as embedded systems have a limited number of registers, we propose a register allocation and instruction scheduling technique to improve the “REALM” technique with register constraints. For the register operations generated by the “REALM” technique, we analyze their data dependencies for instruction scheduling, and build up a register-matching graph model to find available physical registers that can be allocated to the operands of the register operations. The register allocation problem is solved by finding a simple path of fixed length between two specified vertices in the register-matching graph. We perform instruction scheduling based on the results of the allocation.

In low power optimization, we address two challenging issues, leakage and temperature, for embedded systems. Leakage power has become an issue comparable in importance to dynamic power as semi-conductor technologies move down to the nanometer scale. Besides leakage power, temperature issues are also important because both on-chip power density and temperature are rising exponentially with decreasing feature sizes. The increase in on-chip temperature can lead to severe problems with reliability, performance, and cooling costs for embedded systems. To address these issues, we make the following contributions.

- The first contribution is to reduce the leakage power consumption of VLIW (Very Long Instruction Word) processors. We propose a novel leakage-aware modulo scheduling technique that helps hardware-based leakage control schemes to achieve leakage power savings for embedded VLIW processors. We also consider transition time and power overhead in our technique, and discuss the trade-off between leakage savings and performance penalties.

- The second contribution is to reduce the peak temperature of the on-chip memory subsystem. Most embedded systems adopt a hybrid memory architecture, which contains
both hardware-managed cache and software-managed scratchpad memory (SPM). However, both cache and SPM have become hot spots, as they are the most frequently accessed on-chip components. We propose a temperature-aware data allocation technique to explore such a hybrid architecture to jointly optimize performance and peak temperature. Our technique can greatly alleviate the temperature hot spots of the memory subsystem by adaptively distributing the workload between cache and SPM.

**Keywords:** Embedded Systems; High Performance Optimization; Low Power Optimization; Leakage; Temperature; Compiler.
PUBLICATIONS

Journal Papers


**Conference Papers**


national Conference on Embedded and Ubiquitous Computing (EUC 2009), August 2009, Vancouver, Canada.


Acknowledgements

First, I would like to thank my supervisor, Dr. Zili Shao, for his rigorous supervision of my research. I thank him for his support, patience and encouragement during my Ph.D. study. He unremittingly trained me to be a good researcher. He taught me how to find research issues and how to solve problems. Time after time, he showed me how to express my ideas and write academic papers. His vision, passion, and attitude towards the research deeply affected me. Without his help and support, this body of work would not have been possible. What I have learned and experienced during the time I spent in his research group will benefit me much in the future.

Next, I thank the other members of Dr. Shao’s research group – Yi Wang, Duo Liu, BaoKang Zhao, Hui Liu, Miao Liu, Bin Wang, Yunan Cao, Kai Sun – for their contributions to certain aspects of my research. Also, I wish to acknowledge my appreciation to Dr. Yuan Zheng, Lei Ye, Jin Xie, Yang Liu, Guobin Liu and Qin Li, who shared with me the pleasure of the Ph.D. study at the Hong Kong Polytechnic University. Furthermore, I would like to thank all my teachers from whom I learned so much in my long journey of formal education. They are Dr. Jiannong Cao, Dr. Yan Liu, Dr. Bin Xiao, Dr. Alvin Chan, and Dr. Zhijun Wang at the Hong Kong Polytechnic University, Prof. Zhijing Liu at the Xidian University of China, and many others.

Finally, but most significantly, I thank my parents for their continuous love, support, trust, and encouragement through the whole trip of my life. Without them, none of this would have happened.
# TABLE OF CONTENTS

CERTIFICATE OF ORIGINALITY .............................................................. ii  
ABSTRACT .............................................................................................. iii  
PUBLICATIONS .......................................................... v i  
ACKNOWLEDGEMENTS ........................................................................ x  
LIST OF FIGURES ................................................................. x i v  
LIST OF TABLES ..................................................................................... x v i i  

CHAPTER 1. INTRODUCTION .................................................. 1  
1.1 Related Work ................................................................. 4  
  1.1.1 High Performance Optimization ............................................. 4  
  1.1.2 Leakage-aware Optimization ............................................... 7  
  1.1.3 Temperature-aware Optimization ....................................... 10  
1.2 The Unified Research Framework ........................................... 13  
1.3 Contributions ................................................................. 14  
1.4 Outline ............................................................................... 17  

CHAPTER 2. ON REDUCING HIDDEN REDUNDANT MEMORY ACCESES FOR DSP APPLICATIONS ................................. 18  
2.1 Overview ............................................................................... 18  
2.2 Background ............................................................................ 22  
  2.2.1 The Characteristics of DSP Applications ............................... 22  
  2.2.2 Compiler Support ............................................................. 23  
2.3 Motivational Examples .......................................................... 23  
2.4 The Redundant Load Exploration & Migration Algorithm ........... 28  
  2.4.1 The REALM Algorithm ....................................................... 29  
  2.4.2 Function Graph_Construction() ........................................... 31  
  2.4.3 Function Code_Transformation() ....................................... 36  
  2.4.4 Complexity Analysis ......................................................... 39
4.4 Leakage-Aware Modulo Scheduling Algorithm ........................................... 96
  4.4.1 The Proposed Algorithm ................................................................. 96
  4.4.2 Function IterativeSchedule() ......................................................... 98
  4.4.3 Function FindTimeSlot() ............................................................... 102
4.5 Experiments ............................................................... 102
  4.5.1 Experimental Environment .............................................................. 103
  4.5.2 Benchmark Programs ................................................................. 104
  4.5.3 Results and Discussion ................................................................. 105
4.6 Summary ............................................................... 110

CHAPTER 5. TEMPERATURE-AWARE DATA ALLOCATION FOR EMBEDDED SYSTEMS WITH CACHE AND SCRATCHPAD MEMORY .......... 112
5.1 Overview ............................................................... 112
5.2 Basic Concepts and Models ................................................................. 116
  5.2.1 System Model ................................................................. 116
  5.2.2 Power Model ................................................................. 117
  5.2.3 Temperature Model ................................................................. 118
  5.2.4 Problem Statement ................................................................. 119
5.3 Temperature-Aware Data Allocation ........................................................... 119
  5.3.1 Mathematical Formulation ................................................................. 119
  5.3.2 Temperature-Aware Adaptive Loop Scheduling ........................................ 124
5.4 Experiments ............................................................... 127
  5.4.1 Experimental Setup ................................................................. 128
  5.4.2 Results and Discussion ................................................................. 130
5.5 Summary ............................................................... 138

CHAPTER 6. CONCLUSIONS AND FUTURE WORK ......................... 139
6.1 Conclusions ............................................................... 139
6.2 Future Work ............................................................... 142

REFERENCES ............................................................... 144
LIST OF FIGURES

1.1 The relationships between our research results and prior results in high performance optimization. ......................................................... 5
1.2 The relationships between our research results and prior results in low power optimization................................................................. 8
1.3 The relationships between our research results and prior results in temperature-aware optimization. .............................................. 11
1.4 The unified research framework. ........................................................... 13

2.1 The motivational example: (a) C source code, (b) The original intermediate code generated by the IMPACT compiler after applying classical optimizations. 24
2.2 The motivational example: (a) Memory access graph, (b) the reduced memory access graph, (c) The optimized code generated by the IMPACT compiler after applying our technique...................................................... 25
2.3 The C code and intermediate code of the loop with unrolling factor of (a) 0 (the original loop), (a) 1, (b) 2 .................................................................. 27
2.4 The schedules (a) for the original loop in Figure 2.1 (b); (b) for the optimized loop in Figure 2.2 (c) .......................................................... 28
2.5 Hcode of statements S1 and S2 of the motivational example. .......... 30
2.6 Two memory operation sets V_A and V_C for array A and C of the motivational example................................................................. 30
2.7 Memory access graph construction and reduction of array C in the motivational example: (a) edge weight calculation, (b) graph construction, (c) graph reduction................................................................. 34
2.8 Code transformations for eliminating redundant loads of array C in the motivational example......................................................... 38
2.9 The implementation and simulation framework............................... 42
2.10 The percentages of dynamic load operations over the total number of dynamic operations for benchmarks from (a) DSPstone; (b) MiBench........ 44
2.11 The reduction in the number of dynamic load operations for benchmarks from (a) DSPstone. ................................................................. 46
2.12 The improvement of ILP in benchmarks from (a) DSPstone; (b) MiBench.. 47
2.13 Performance improvement due to the REALM algorithm for benchmarks from (a) DSPstone; (b) MiBench. ......................................................... 48
2.14 The code size expansion of benchmarks from (a) DSPstone; (b) MiBench. ... 49
2.15 Average performance speedup of REALM and RPMS_CP_REALM with 16 registers, 32 registers and 64 registers for benchmarks from (a) DSPstone; (b) MiBench. ......................................................... 51
3.1 The target VLIW architecture. ................................................. 57
3.2 (a) The C source code of the IIR filter (b) the assembly code of the loop
generated by TI DSP CCS (Code Composer Studio). ....................... 58
3.3 (a) The nodes and their corresponding operations; (b) The DAG that repre-
sents the loop body in Figure 3.2-(b); (c) The schedule generated by the list
scheduling. ............................................................................. 59
3.4 The register usage map of the schedule in Figure 3.3-(c). ............... 60
3.5 (a) The memory access graph of the example (Figure 3.2) in which nodes
A and C are hidden redundant load operations; (b) the DAG with register
operations that replace node A. ................................................. 61
3.6 (a) The schedule, (b) the corresponding assembly code, and (c) the register
usage map generated by our LSMAR algorithm. .......................... 62
3.7 An example of the memory access graph..................................... 64
3.8 The register operation chain. .................................................. 66
3.9 (a) The edge u → v in the memory access graph; (b) The original DAG; (b)
The new dependencies generated by replacing v with register operations; (c)
Read-write data dependencies. .................................................. 68
3.10 Schedule boundaries of the (k + 1) register operations. .................. 69
3.11 The lifetime requirements for the virtual register operands of the chain (r_0
has been assigned to phy_{u}, and r_{k+1} has been allocated to phy_{v}). .... 70
3.12 An example of the register-matching graph.................................. 72
3.13 The path finding method. ..................................................... 73
3.14 (a) The modified adjacency matrix A that represents the register-matching
graph in Figure 3.12-(b); (b) A^2; (c) A^3 and the entry A^3(S, T ) that contains
all paths from S to T with a length of 3. ....................................... 74
4.1 The FIR program and its corresponding data flow graph. ............... 90
4.2 The schedule generated by performance-oriented modulo scheduling and our
technique. ................................................................................. 91
4.3 The rough relationship between total energy consumption and the timing
constraint for different benchmarks. ............................................. 107
4.4 Leakage energy reduction without performance loss. ...................... 108
4.5 Leakage energy reduction for IALUs due to our algorithm compared with
the previous work. .................................................................. 109
4.6 Performance penalty. ......................................................... 110
5.1 (a) Block diagram of embedded systems with cache and SPM; (b) Division
of data address space between SRAM and DRAM. ......................... 117
5.2 The NLP formulation. ........................................................... 120
5.3 The flow of the code for the revised loop. ................................... 127
5.4 The implementation and simulation framework (the shaded blocks are the modified parts) ........................................... 128
5.5 The simulated floor plan ....................................................... 129
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The configurations of Trimaran</td>
<td>43</td>
</tr>
<tr>
<td>2.2</td>
<td>The benchmarks</td>
<td>43</td>
</tr>
<tr>
<td>3.1</td>
<td>The VLIW configurations</td>
<td>76</td>
</tr>
<tr>
<td>3.2</td>
<td>Benchmark descriptions and characteristics</td>
<td>78</td>
</tr>
<tr>
<td>3.3</td>
<td>The number of dynamic load operations under various register constraints for benchmarks</td>
<td>80</td>
</tr>
<tr>
<td>3.4</td>
<td>The improvement in performance under various register constraints for benchmarks</td>
<td>82</td>
</tr>
<tr>
<td>3.5</td>
<td>The expansion in code size under various register constraints for benchmarks</td>
<td>83</td>
</tr>
<tr>
<td>4.1</td>
<td>The configurations of Trimaran</td>
<td>104</td>
</tr>
<tr>
<td>4.2</td>
<td>The benchmarks</td>
<td>104</td>
</tr>
<tr>
<td>5.1</td>
<td>Variables and constants used in the NLP formulation</td>
<td>120</td>
</tr>
<tr>
<td>5.2</td>
<td>The configurations for the VLIW simulator of Trimaran</td>
<td>130</td>
</tr>
<tr>
<td>5.3</td>
<td>The parameters for HotSpot</td>
<td>130</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison of execution time and time performance for our NLP-based approach and TALS algorithm</td>
<td>131</td>
</tr>
<tr>
<td>5.5</td>
<td>The reduction in peak temperature for the benchmarks</td>
<td>133</td>
</tr>
<tr>
<td>5.6</td>
<td>The improvement in performance for the benchmarks</td>
<td>135</td>
</tr>
<tr>
<td>5.7</td>
<td>The size of the executable files</td>
<td>136</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

In today’s world, embedded systems are everywhere in our lives, ranging from consumer electronics such as cell phones, MP3 players, and personal digital assistants (PDAs), to vehicle controllers. Billions of embedded systems are sold every year, and their market share has been continuously increasing in the last couple of decades. The advances in embedded system technologies and their large-scale deployment, not only in industries and services but in all areas of human activity, will drive an information evolution of our society in which all systems, machines, and objects will become digital, communicating, self-managed resources. For example, intelligent support for people could be embedded in everyday objects, such as clothes, vehicles, buildings, roads, and smart materials; and networks with thousands or millions of embedded sensor systems could monitor the environment, the battlefield, or the factory.

Embedded systems are application specific, and have strict constraints in terms of performance, memory, register, power, temperature, and other resources. As embedded systems become more complex and incorporate more functionality, these constraints are getting tighter. The various constraints and limited resources pose a host of technical challenges for the design of high performance and low power embedded systems. In this thesis, we investigate the challenging issues involved in designing compiler-assisted techniques for solving high performance and low power optimization problems in embedded systems.

High performance has always been the most critical issue in the design of embedded systems. Memory is a key performance bottleneck in embedded systems. Many embedded computing applications spend a lot of time accessing memory. As the processor-memory gap increases, the memory access is significantly limiting the performance of embedded
systems. Therefore, reducing memory accesses to improve performance has become an important problem. This is particularly important for memory-intensive DSP (Digital Signal Processing) applications, which are widely used in image processing, multimedia, wireless security, and so on. In this thesis, we study and address the problem of reducing memory accesses for DSP applications, which is vital for achieving high performance.

- Loops are usually the most critical sections and consume a significant amount of time and power in DSP applications. A significant fraction of computational overhead involved in DSP applications often arises from the extensive memory interactions that are usually on the critical paths of loops. In loop kernels of DSP applications, one important characteristic is that the same memory location is repeatedly accessed by different memory operations during the execution of the loop. For DSP applications, therefore, an important problem is how to explore redundant memory accesses of loops and eliminate them by exploiting the reusable value across iterations. To solve this problem, we develop a compiler-assisted optimization technique called REALM to identify and eliminate redundant memory accesses by exploring the loop-carried data dependencies among memory operations.

- Register pressure issues have to be addressed in memory access optimization as embedded systems normally have a limited number of registers. Basically, register pressure occurs when there are more variables to allocate than the number of available registers. This typically results in register spilling, which will greatly degrade performance. On the other hand, for achieving a reduction in memory accesses, our REALM technique eliminates redundant memory accesses by exploiting register operations. Therefore, how to allocate the limited number of available registers to these newly generated register operations becomes an important problem. To solve this problem, we propose a loop scheduling algorithm called LSMAR to improve the REALM technique considering register constraints. We solve the register allocation problem by building up a register-matching graph and finding a fixed-length simple path between two specified vertices in this graph. Based on the register allocation results, we gener-
ate a schedule in such a way that all constraints can be fulfilled and the total number of memory accesses can be reduced.

Low power has emerged as an important design issue and performance metric, particularly for battery-powered embedded systems. In low power optimization, we address two challenging issues for embedded systems: leakage and temperature. Leakage power has become the primary contributor to total power consumption as semi-conductor technologies move down to the nanometer scale. In addition to leakage issues, temperature issues are also important because an increase in on-chip temperature can lead to severe problems with performance, reliability, and cooling costs for embedded systems. In this thesis, we have addressed the above issues from two aspects including leakage-aware scheduling and temperature-aware data allocation.

- Leakage power has been shown to account for 40% of the overall power consumption of today’s high-performance microprocessors [82, 94], and it is expected to dominate overall power consumption in future technologies [15, 59, 113, 124]. Our research in this field focuses on reducing leakage power for embedded processors. To satisfy ever-growing performance requirements, a high-performance VLIW (Very Long Instruction Word) architecture is now widely used in embedded processors. A VLIW processor has multiple functional units and can process several instructions simultaneously. While this multiple-FUs architecture can be exploited to increase instruction-level parallelism and improve time performance, it causes more leakage power consumption. Therefore, reducing leakage power consumption on embedded VLIW processors has become an important problem. To solve this problem, we propose a novel compiler-assisted leakage-aware software pipelining technique called LMS that is a tremendous improvement on current techniques.

- Temperature has become a dominant design constraint for embedded systems in the deep sub-micron era. High on-chip temperatures adversely affect performance by decreasing the transistor switching speed, power and energy by increasing leakage
power, reliability by causing electrical failures, and system cost by increasing cooling and packaging costs. Our research in this area focuses on reducing the temperature of the hybrid memory architecture, which contains both cache and scratchpad memory (SPM). Previous studies in this field have concentrated on optimizing the performance of cache and SPM without considering temperature issues. However, both cache and SPM will become temperature hot spots as they are the most frequently accessed on-chip components. In this thesis, we propose a technique called TALS to explore the hybrid memory architecture for jointly optimizing performance and temperature.

In this thesis, we have focused on developing compiler-assisted models, methodologies, and algorithms for high performance and low power embedded systems. The rest of this chapter is organized as follows: In Section 1.1, the related work is presented. In Section 1.2, the unified research framework is presented. The contributions of this thesis are summarized in Section 1.3. Section 1.4 presents the outline of the thesis.

1.1 Related Work

There have been extensive studies in the fields of high performance optimization, low leakage optimization, and temperature-aware optimization for embedded systems. The relationships between our research results and the prior results are summarized as follows.

1.1.1 High Performance Optimization

Our research on high performance optimization focuses on reducing the number of memory accesses for embedded DSP (Digital Signal Processing) applications. We focus on optimizing loops, as they consume the most time and power in DSP applications. There have been many studies in this field. The road map shown in Figure 1.1 illustrates the relationships between our research results and prior results in high performance optimization.

As shown in Figure 1.1, many loop transformation techniques, such as loop partitioning [111, 142, 149, 176], array contraction [42, 110, 117], and loop pipelining [141, 148],
have been extensively studied to improve performance. Wang et al. [142] proposed a multiple loop partition scheduling technique that combines loop partitioning and array padding to exploit data locality. In [42, 110], Gao et al. facilitated loop fusion by performing array contraction as much as possible in order to reduce the memory-register traffic of loops. Wang et al. [141] proposed an optimal loop scheduling technique for hiding memory latency based on a two-level partitioning and prefetching scheme. Different from the above techniques, our technique improves performance by reducing the number of memory accesses. In addition, our technique can be integrated with the above techniques by providing more opportunities for loop transformations.

For array-based applications, a source-code-level technique called *scalar replacement* [4, 18, 20, 91] has been developed to reduce memory accesses. This technique is a source-to-source transformation approach that reduces memory accesses by replacing subscripted variables with temporary scalar variables. However, this technique cannot be effectively applied to DSP applications in which pointer arithmetic is extensively adopted [177]. Our approach can cope with both array-based and pointer-based code by obtaining the corresponding information from compilers.
Over the last decade, memory-related issues have benefited from advances made in the fields of data reuse analysis [1, 41, 50, 55, 57, 58, 63, 72, 73, 80, 93, 109, 118, 153] and high-level synthesis [13, 40, 51, 52, 61, 64, 73, 99, 112, 137, 138, 143, 144, 169]. Jha et al. [52] proposed a high-level synthesis methodology for designing multi-partitioned architectures. This work is applicable to memory-intensive applications with complex array access patterns. Dutt et al. [64] proposed a local scheduler transformation that optimizes the accessing of a secondary memory, thereby reducing memory traffic. In [50], Hu et al. proposed a compiler-directed cache polymorphism for optimizing the data locality of array-based embedded applications. However, the above techniques focus on exploiting scratchpad memory and cache to store reusable data for embedded applications. Our technique can be combined with the above techniques in order to provide more opportunities for keeping important data within on-chip memories.

Many compiler-assisted optimization techniques have been proposed to reduce memory accesses [2, 6, 17, 21, 35, 51, 52, 61, 80, 98, 102, 118, 143, 150, 151]. Two compile-time optimizations, redundant load/store elimination and loop-invariant load/store migration, can reduce the amount of memory traffic by expediting the issue of instructions that use the loaded value [2, 6, 21, 98]. In [17, 150, 151], Xue et al. proposed effective techniques to eliminate partial redundancies in order to optimize memory accesses. Their techniques allow a value that normally resides in memory to reside in a register for some portions of the code. Li et al. [80] introduced the notion of memory access intensity to facilitate the quantitative analysis of a program’s memory behavior on multi-core architectures. However, these approaches only consider removing explicit redundant memory accesses within one iteration of a loop. Our technique can explore and eliminate hidden redundant memory accesses across multiple loop iterations based on the loop-carried dependence analysis.

Our work is closely related to Loop Unrolling. Loop unrolling [24, 67, 68] can be used to unfold a loop a few times to expose loop-carried data dependences among memory operations. However, with loop unrolling, how to determine the optimal unrolling factor is not known, and expanding code size after unrolling is undesirable for embedded systems. In Chapter 2, we propose a compiler-assisted technique to explore and eliminate hidden re-
dundant load operations for DSP applications with loops. In our technique, we propose a memory access graph model to describe the loop-carried data dependencies of memory operations. Based on this graph, our technique can achieve an optimal solution by eliminating all of the hidden redundant memory accesses. Moreover, our technique outperforms loop unrolling, since it introduces little expansion of code size.

On the other hand, embedded systems have a limited number of registers. Taking issues of register pressure into consideration, a great deal of research has been done on register-aware loop scheduling [25, 44, 84, 85, 106, 135, 140, 167]. Gao et al. [44] proposed a software pipelining technique to improve performance while minimizing register requirements. In [25], Chen et al. presented a framework for scheduling DSP applications with multi-dimensional loops subject to register constraints and other resource constraints. In Chapter 3, we further develop a loop scheduling technique to reduce memory accesses under register constraints based on the work in Chapter 2. Our technique combines register allocation and instruction scheduling, and it can greatly reduce the number of memory accesses and improve performance under register constraints.

1.1.2 Leakage-aware Optimization

Our low leakage optimization research focuses on reducing leakage power for VLIW (Very Long Instruction Word) processors. The road map shown in Figure 1.2 illustrates the relationships between our research results and prior results in low power optimization.

As shown in Figure 1.2, low power design has been an important issue for embedded systems, due to its significant impact on battery life, system density, cooling cost, and system reliability. Traditionally, dynamic power is the primary contributor to total power consumption. Dynamic Voltage Scaling (DVS) [26, 27, 54, 86, 87, 115, 139, 154, 167, 172] is an effective technique to reduce dynamic power by dynamically scaling down the supply voltage, $V_{DD}$, and the operational frequency. However, DVS often requires a reduction in the threshold voltage $V_{TH}$, which leads to an exponential rise in the sub-threshold leakage current, and hence in the leakage power consumption. Leakage power accounts for 40% of the
power consumption of today’s high-performance microprocessors [82, 94], and it is expected to dominate overall power consumption in future technologies [15, 59, 113, 124]. Thus, new research directions in low power optimization begun to address the issue of leakage power.

Many architecture-level techniques, including clock gating, power gating, and dual-threshold domino logic, have been proposed and evaluated to achieve a reduction in leakage power [16, 39, 79, 122, 127–129]. Clock gating is an effective technique for reducing leakage and has been widely used in practice. For example, the IBM Power PC 405LP [19] implements clock gating by aggressively shutting down elements of the processor that are idle. The Intel PXA [45] family processors support fine granularity clock gating to exploit the fact that not all system transistors are used simultaneously. In [128], Tsai et al. provided a comparison of three runtime techniques to reduce leakage power, input vector control, body bias control, and power gating in terms of leakage savings, feasibility and scaling trend. Liao et al. [79] applied power gating to two circuit types, memory-based units and data path components, in the forms of VRC (Virtual power/ground Rails Clamp) and MTCMOS (Multi-threshold CMOS). Dropsho et al. [39] proposed a hardware-based scheme to manage leakage power
for short idle periods. In their scheme, the dual-threshold domino logic with sleep mode is utilized, which can transit between active mode and sleep mode without any performance penalty. Our technique uses the power model in [39] to calculate both leakage and total power consumption of an application on VLIW processors.

Various system-level scheduling techniques taking leakage power into consideration have been explored in [28, 86, 154, 171–174]. Jha et al. [86] addressed the problem of the variable-voltage scheduling of multi-rate periodic task graphs in heterogeneous distributed real-time embedded systems. Kuo et al. [28] developed an on-line simulated scheduling strategy and a virtual blocking time strategy for reducing leakage power on a uniprocessor DVS system. Their algorithms derived a feasible schedule for real-time tasks with worst-case guarantees for any input instance. Xu et al. [172, 174] proposed a dynamic programming algorithm for periodic tasks on processors with practical discrete speed levels. Their algorithm determined the lower bound of energy expenditures in pseudo-polynomial time.

In recent work, compiler-assisted instruction-level scheduling techniques [34, 70, 100, 105, 158] have been investigated to reduce leakage energy. Gupta et al. [105] introduced a power-aware flow graph to determine program regions that offer opportunities to save leakage in the context of super-scalar processors. In [158], You et al. provided compiler solutions for the analysis and scheduling with power gating control at the component level. They presented a data flow analysis framework for estimating the component activities at fixed points of programs.

For VLIW architecture, a great deal of research effort has been put to reduce the leakage power consumption of functional units [62, 95, 159, 163–166]. By exploiting scheduling slacks of instructions, Nagpal et al. [95] proposed a leakage-aware scheduling algorithm for VLIW and clustered VLIW architectures. This algorithm is based on traditional list scheduling. Zhang et al. [166] proposed a compiler-based technique for optimizing leakage power on VLIW functional units. Their strategy is built upon a data-flow analysis that detects idle functional units along control-flow-graph paths, and inserts activation/deactivation instructions in the code. However, the above scheduling techniques are based on DAG (Directed A
Acyclic Graph) scheduling in which only intra-iteration dependencies are considered. In this thesis, we show that we can further reduce leakage power by exploiting the inter-iteration dependencies of a loop using software pipelining.

Software pipelining [23,38,53,69,104,108] is a technique that exploits the instruction-level parallelism among the loop iterations by overlapping the execution of consecutive iterations of a loop. Performance-oriented modulo scheduling [103] is a software pipelining technique that results in a high performance code. In previous work, power-aware resource constrained software pipelining [155, 161] was studied to find a way to reduce power consumption without degrading performance. In [155], Yang et al. formulated the power-aware software pipelining problem as an integer linear programming problem. Kim et al. [161] proposed a modulo scheduling algorithm that produces a more balanced schedule to reduce peak power and step power. However, the problem of leakage power is not considered in the above work. In Chapter 4, we propose a novel leakage-aware modulo scheduling technique that can greatly reduce leakage power on VLIW processors.

1.1.3 Temperature-aware Optimization

Our study on temperature-aware optimization focuses on exploring the hybrid memory architecture that contains both cache and SPM (scratchpad memory) to jointly optimize performance and temperature for embedded systems. There have been a lot of studies in this field. A road map shown in Figure 1.1 illustrates the relationships between our research results and prior results in temperature-aware optimization.

As shown in Figure 1.3, a great deal of research has been put into modeling and managing on-chip temperatures [3,14,32,65,66,74,81,114,116,119,134,146]. Temperature has adverse effects on multiple aspects of high-performance processors. First, it shortens the lifetime of the integrated circuit by accelerating the chemical process inside the chip. Recent studies have shown that most hardware failures are exponentially related to temperature [119, 134]. For instance, every 10-15 degree rise in temperature can reduce the average lifetime of an IC by half [134]. Second, due to the exponential relationship between leakage power and
temperature, processors with high leakage power can fall into the thermal runaway situation. Cooling costs are also rising exponentially with power density. For instance, when the power consumption is above 30-40 Watts(W), the total cooling cost of a chip is increased by more than $1/W [14].

To effectively manage on-chip temperature, Jha et al. [65] proposed and evaluated the HybDTM (Hybrid Dynamic Thermal Management) approach to manage overall temperature and improve the performance. Wu et al. [146] developed a dynamic cache sub-array permutation scheme using crossbars in the address predecoder to alleviate the thermal stress on areas of high leakage. Skadron et al. developed a thermal simulator, HotSpot [116], to calculate the transient temperature based on the given physical characteristics and power consumption of units on the die. Their model is based on an equivalent circuit of thermal resistances and capacitances.

Based on the above temperature models, various temperature-aware scheduling techniques have been proposed [8, 22, 33, 56, 82, 83, 86, 89, 92, 101, 160, 175]. In [101], the authors presented necessary and sufficient conditions for real-time schedules to guarantee the maximal temperature constraint. Qu et al. [160] studied the relationship between
temperature and leakage in real-time systems. They also proposed an online temperature-aware leakage minimization algorithm that adjusts the processor modes at runtime based on the chip temperature. Hu et al. [22] developed a temperature-aware real-time scheduling and assignment approach for hard real-time applications on MPSOC (Multiprocessor-System-on-Chip) architectures. On the other hand, as temperature is highly dependent on power consumption, in particular leakage power, existing power-aware optimization techniques [27, 28, 136, 139, 167, 172–174] have a positive effect on reducing temperature. However, the hybrid memory architecture with both on-chip cache and SPM is not considered in the above approaches. Our technique can be combined with the above temperature-aware and power-aware scheduling techniques to further reduce on-chip temperatures.

For cache and SPM, researchers have investigated a number of compiler-assisted optimization approaches to improve performance [5, 7, 36, 121, 130, 131, 133], data locality [75–78, 132, 152, 157], and power consumption [97, 123, 147, 162, 168, 170]. In [170], a low power data cache design scheme was proposed by exploiting the frequent value locality. Vera et al. [132] explored the use of cache partitioning and dynamic cache locking to provide worst-case performance estimates for multi-task systems. SPM has been widely used to improve performance and reduce power consumption on embedded processors by avoiding cache misses. To fully utilize SPM, Barua et al. [130] proposed a dynamic allocation method for global and stack data on embedded systems with SPM. Xue et al. [75] presented a general-purpose compiler approach, called memory coloring, to automatically allocate the arrays in a program to the SPM. However, temperature is not considered in the above techniques. In embedded systems, with the large amount of heat that is generated by extensive memory accesses, both on-chip cache and SPM have become the on-chip hot spots, and this will eventually lead to a significant degradation in performance and severe reliability problems. In Chapter 5, we propose an adaptive temperature-aware data allocation technique that can explore the hybrid memory architecture for achieving joint performance and temperature optimization.
1.2 The Unified Research Framework

In this section, we present the unified research framework for the proposed techniques. In this thesis, we have integrated our proposed high performance and low power optimization techniques into the open-source compiler Trimaran [47], which is the state-of-the-art compiler infrastructure for VLIW architectures. Figure 1.4 shows the sketch of our research framework.

As shown in Figure 1.4, Trimaran has three components: the front end (IMPACT), the back end (ELCOR), and the cycle-accurate VLIW simulator. IMPACT consists of various machine-independent optimization modules, and ELCOR contains a number of machine-dependent optimization modules, such as instruction scheduling, register allocation, and modulo scheduling. Our first proposed technique, REALM, is a machine-independent optimization technique, and it has been integrated into the loop optimization module of IM-
PACT for reducing memory accesses. In order to reduce memory accesses under register constraints, we have integrated our LSMAR algorithm into the post-pass scheduling module of ELCOR, where the instruction scheduling and register allocation techniques have been performed. Our LMS approach has been integrated into the modulo scheduling module for reducing leakage energy consumption of embedded VLIW processors using software pipelining. Our TALS technique has been integrated into the post-pass scheduling module of ELCOR for jointly optimizing peak temperature and performance for embedded systems with both on-chip cache and scratchpad memory (SPM). Finally, in the unified research framework, the cycle-accurate VLIW simulator has been used as the simulation and test platform for the proposed techniques.

1.3 Contributions

In this thesis, our research focuses on understanding fundamental properties and developing compiler-assisted high performance and low power optimization techniques for embedded systems. We have performed our research from various aspects including high performance optimization, low leakage optimization, and temperature-aware optimization. A lot of promising results in these fields have been obtained, and these results have tremendously improved current techniques. Our contributions are summarized as follows:

1. For high performance optimization on embedded systems, we identify that the number of memory accesses is the most important factor influencing time performance. We study and address the memory access reduction problem for DSP applications, which is vital for improving performance.

   • For DSP applications with loops, we propose a technique, REALM (REdundant Load Exploration & Migration), to reduce hidden redundant memory accesses within the loop nests. Different from previous work, our technique can optimize both array-based and pointer-based code, which is very important as pointer arithmetic is widely used in DSP applications.
We propose a novel memory access graph model to describe the loop-carried data dependencies of memory operations. Based on this graph, our REALM technique can achieve an optimal solution in which all hidden redundant memory accesses can be explored and eliminated.

We have implemented REALM into IMPACT [21], which is one of the most popular open-source compilers with full support from the compiler research community.

To the best of our knowledge, this is the first work to implement the memory access reduction with loop-carried data reuse in real world compilers.

2. To solve the memory access reduction problem under register constraints, we propose a two-phase loop scheduling algorithm, LSMAR (Loop Scheduling with Memory Access Reduction). In the first phase, we use the REALM technique to optimally explore all hidden redundant load operations; and in the second phase, we iteratively perform loop scheduling to eliminate these operations by exploiting register operations.

We solve the register allocation problem by building up a register-matching graph and finding a fixed-length simple path between two specified vertices in this graph. Based on the register allocation results, we generate a schedule in such a way that all constraints can be fulfilled and the total number of memory accesses can be reduced.

We have implemented our LSMAR algorithm into Trimaran [47], which is the most popular state-of-the-art compiler infrastructure for embedded VLIW architectures. The experimental results show that our algorithm can significantly reduce memory accesses and improve performance with little expansion of code size.

3. For low leakage optimization, we study the fundamental properties of leakage power, and identify the relationship between leakage savings and schedule length. We pro-
pose a novel leakage-aware modulo scheduling technique to reduce the leakage power consumption of an application executed on an embedded VLIW processor. Different from prior studies, our technique combines leakage-aware optimization with software pipelining. In this way, we can maximize the idle time of a functional unit to achieve leakage savings. The experimental results show that our technique can greatly reduce leakage power for VLIW processors with less performance overhead compared with previous work.

4. For temperature-aware optimization, we identify that on-chip memories are the temperature hot spots. We address both performance and temperature issues for the hybrid embedded memory architecture, which contains both cache and scratchpad memory (SPM).

- We propose both static and dynamic temperature-aware strategies for allocating data for embedded systems with on-chip cache and SPM. For problems in which the workload can be estimated \textit{a priori}, we present a mathematical formulation to optimally minimize the execution time of a loop under the constraints of SPM size and temperature. To solve problems in which the workload is not known \textit{a priori}, we propose an adaptive temperature-aware data allocation algorithm, TALS, to dynamically adjust the workload of on-chip memories based on the current temperature at runtime.

- We experiment with our proposed techniques using a set of benchmarks from DSPstone [177] and Mibench [49], which are the most popular embedded benchmark suites consisting of extensive digital filters and real-world embedded applications. The results demonstrate that our techniques can effectively achieve joint performance and temperature optimization for embedded systems with cache and SPM.

To the best of our knowledge, this is the first work to explore the hybrid architecture with both cache and SPM to jointly optimize performance and temperature.
1.4 Outline

The rest of this thesis is organized as follows. In Chapter 2, we propose the REALM technique which reduces memory accesses for DSP applications. In Chapter 3, we propose the loop scheduling algorithm with memory access reduction under register constraints. In Chapter 4, we present the leakage-aware modulo scheduling algorithm for embedded VLIW processors. In Chapter 5, we present the temperature-aware data allocation algorithm for embedded systems with cache and SPM. Chapter 6 gives the concluding remarks and a brief discussion on future work.
CHAPTER 2
ON REDUCING HIDDEN REDUNDANT MEMORY ACCESSES FOR DSP APPLICATIONS

Abstract
Reducing memory accesses is particularly important for DSP applications since they are widely used in embedded systems and need to be executed with high performance and low power consumption. In this chapter, we propose a machine-independent loop memory access optimization technique, REALM (REdundAnt Load Exploration & Migation), to explore hidden redundant load operations and migrate them outside loops based on loop-carried data dependence analysis. We implement REALM into IMPACT and Trimaran. To the best of our knowledge, this is the first work to implement the memory access reduction with loop-carried data reuse in real world compilers. We conduct experiments using a set of benchmarks from DSPstone and MiBench on the cycle-accurate VLIW simulator of Trimaran. The experimental results show that our technique significantly reduces the number of memory accesses.

2.1 Overview
The widening performance gap between processor and memory requires effective compiler optimization techniques for reducing memory accesses. This is particularly important for DSP (Digital Signal Processing) applications since they are widely used in embedded systems and need to be executed with high performance and low power consumption. On the other hand, loops are the most critical sections and consume most time and power for DSP applications. Therefore, memory access optimization for loops is vital for improving DSP performance. Computationally intensive loop kernels of DSP applications usually have a simple control-flow structure with a single-entry-single-exit and a single loop back edge. In this chapter, thus, we develop a memory-access-graph-based loop optimization technique
with loop-carried data dependence analysis to explore and eliminate hidden redundant memory accesses for loops of DSP applications.

Various techniques for reducing memory accesses have been investigated in previous work. Two classical compile-time optimizations, redundant load/store elimination and loop-invariant load/store migration [2, 6, 21, 98], can reduce the amount of memory traffic by expediting the issue of instructions that use the loaded value. Register promotion [31, 90] is recognized as an effective memory access optimization technique by allowing a value that normally resides in memory to reside in a register for some portions of the code. Most of the above optimization methods only consider removing existing explicit redundant load/store. Our technique can explore and eliminate hidden redundant loads across multiple loop iterations based on loop-carried data dependence analysis.

Over the last decade, memory-related issues have benefited from advances made in the fields of compilers [35, 55, 57, 58, 63, 72, 73, 80, 109, 118, 153] and high-level synthesis [51, 52, 61, 64, 99, 112, 137, 143]. Jha et al. [52] proposed a high-level synthesis methodology for designing multi-partitioned architectures. This work is applicable to memory-intensive applications with complex array access patterns. Dutt et al. [64] proposed a local scheduler transformation which optimizes the accessing of a secondary memory, thereby reducing memory traffic. The above techniques focus on exploiting scratch-pad memory and cache to store reusable data for optimizing buffers of embedded applications. Our technique can replace redundant memory operations with register operations; thus, it can be integrated with the above techniques by providing more opportunities to store important data in the scratchpad memory and cache.

In loop optimization, a lot of compiler transformation techniques have been applied to reduce memory accesses, such as loop partitioning [35, 117], array padding [142], partial redundancy elimination [9–12, 17, 150, 151], scalar replacement [18] and array contraction [42, 110]. Wang et al. [142] proposed a multiple loop partition scheduling technique which combines loop partitioning and array padding to exploit data locality. Ding et al [35] discussed a compiler scheme to identify code segments which are good candidates for com-
putation reuse. In this chapter, we propose a memory-access-graph-based approach in which the code replacement pattern for eliminating the corresponding redundant load can be easily determined. Different from the previous work, our memory-access-graph-based approach is more suitable for DSP applications which have loops with simple control-flow structures.

Our work is closely related to Loop Unrolling. Loop unrolling [24, 67, 68] can be used to unfold a loop a few times to expose loop-carried data dependences among memory operations. However, with loop unrolling, how to determine the optimal unrolling factor is not known, and the code size expansion after unrolling is undesirable for embedded systems. Our technique can automatically exploit the loop-carried data dependencies of memory operations using a graph, and achieve an optimal solution by removing all possible redundant memory accesses based on the graph. Moreover, our technique outperforms loop unrolling since it introduces little code size expansion.

In this chapter, we propose a machine-independent loop memory access optimization technique, REALM (REdundAnt Load Exploration & Migration), to explore hidden redundant loads and migrate them outside loops. Our basic idea is to explore loop-carried data dependencies among memory operations. In our technique, hidden redundant loads are found and replaced with registers, and by using registers in such a way that we do not need prior memory accesses which are unchanged or unnecessary to be fetched again from memory over multiple loop iterations. In REALM, we first build up a memory access graph to describe the inter-iteration data dependencies among memory operations. Then we perform code transformation by exploiting these dependencies with registers to hold the values of redundant loads and migrating these loads outside loops. The main contributions of our approach are summarized as follows.

- We study and address the memory access optimization problem for DSP applications which is vital both for improving performance and reducing memory power. Different from the previous work, our technique can optimize both array-based and pointer-based code, which is very important as pointer arithmetic is widely used in DSP applications.
- We propose a memory access graph model to analyze loop-carried data dependencies among memory operations, and develop a technique called REALM (REdundAnt Load Exploration & Migration) for reducing memory accesses within the loop nests of programs. This approach is suitable for DSP applications which typically consist of simple loop structure.

- We propose a practical algorithm called RPMS_CP_REALM (register-pressure-aware modulo scheduling with critical-path-based REALM) to minimize register pressure and improve performance by combining REALM with modulo scheduling [103] that is widely used in DSP applications as a back-end software pipelining technique.

We implemented our techniques into IMPACT [21] and Trimaran [47]. To the best of our knowledge, this is the first work to implement the memory access reduction with loop-carried data reuse in real world compilers. This function is not implemented in various open-source compilers such as IMPACT [21], Trimaran [47], GCC, Open64, and SUIF [46], and the TI DSP compiler (Code Composer Studio V3.3). In particular, REALM is implemented in IMPACT [21], and RPMS_CP_REALM, which combines REALM with modulo scheduling, is implemented in Trimaran [47] as modulo scheduling is part of Trimaran suite.

We conducted experiments using a set of benchmarks from DSPstone [177] and MiBench [49] on the cycle-accurate VLIW simulator of Trimaran [47]. The experimental results show that our REALM technique achieves significant memory access reduction and performance improvement with little code size expansion compared with classical optimizations [2, 6, 21, 98].

- For DSPstone [177], with the configuration of 64 registers, our REALM technique contributes to 22.52% reduction in the number of memory accesses, 12.61% improvement on ILP (instruction level parallelism), and a speedup of 1.13 on overall time performance with 1.43% increment in code size on average. For MiBench [49], with the same configuration, our REALM technique achieves an average of 8.3% reduction
in the number of memory accesses, 4.43% improvement on ILP (instruction level parallelism) and a speedup of 1.06 on overall time performance with 0.77% increment in code size.

- RPMS_CP_REALM, our practical algorithm that combines REALM and modulo scheduling considering register pressure, can improve performance compared with the REALM technique. For DSPstone [177], with the configurations of 16, 32 and 64 registers, our RPMS_CP_REALM technique contributes to 2%, 4% and 6% improvement on average performance, respectively. For MiBench [49], with the configurations of 16, 32 and 64 registers, our RPMS_CP_REALM technique leads to 2%, 3% and 4% improvement on average performance, respectively. Based on the experimental results, we conclude that our RPMS_CP_REALM technique is the best with limited register resources.

The rest of this chapter is organized as follows. The background is introduced in Section 2.2. Motivational examples are shown in Section 2.3. The REALM technique is presented in Section 2.4. The RPMS_CP_REALM technique is proposed in Section 2.5. The experimental results and analysis are provided in Section 2.6. The summary is given in Section 2.7.

2.2 Background

In this section, we first summarize the DSP benchmark characteristics in Section 2.2.1 and then introduce the compiler support for performing our technique in Section 2.2.2.

2.2.1 The Characteristics of DSP Applications

DSP applications are generally characterized as computationally intensive with a large data set, loop-dominant control flow behavior, and accumulation-based operations [71]. Inside loop kernels of DSP applications, the control-flow structure is simple, and repetitive memory accesses to array elements have great impact on overall performance. Among all of the DSP benchmarks from DSPstone [177], almost every loop is in the form of a simple for-loop with a single-entry-single-exit and a single loop back edge. Except loop-back branches, there are
no other conditional or unconditional branches inside these loops. Moreover, array-based and pointer-based computations are widely used in the loop kernel, and the array indexes in each loop are affine expressions of the loop index that is incremented unconditionally.

In loop kernels of DSP applications, one important characteristic is that the same memory location is repeatedly accessed by different memory operations across multiple loop iterations. By analyzing the inter-iteration relations among these memory operations, we can detect hidden redundant load operations in the innermost loop, and replace them by register operations as shown in Section 2.3.

2.2.2 Compiler Support

Pointer alias analysis is the preliminary step in our work based on points-to analysis algorithms [29, 30, 120]. When two program variables refer to the same memory address, one is called the alias of the other. Once the alias information is achieved, it will be fed into the liveness analysis pass and used to identify array accesses via pointers. In addition, our loop optimization technique is performed on the intermediate code in the form of SSA (Static Single Assignment) after the existing classical optimizations [2, 21, 29] have been applied. In SSA form, the same virtual register is never used as a destination more than once in the entire program.

2.3 Motivational Examples

In order to show how our approach works, we present an example in this section. We use the IMPACT compiler [21] to generate intermediate code for this example, and test it on the cycle-accurate VLIW (Very Long Instruction Word) simulator of Trimaran [47]. The C source code of the example is shown in Figure 2.1(a).

The intermediate code generated by the IMPACT compiler [21] with classical optimizations is presented in Figure 2.1(b). The code is in the form of Lcode which is a low-level machine-independent intermediate representation. Note that the "op" numbers represent the identifiers of operations, and they are not in sequence due to different optimization stages of
the compiler. As shown in Figure 2.1(b), two different integer arrays $A$ and $C$ are stored in the memory with consecutive locations. The base pointer for array references in the loop is initialized using the address of $C[2]$ and assigned to register $r37$ at the end of basic block 1 (Instruction: \texttt{op59 add r37, mac $LV$, -792}). In the loop segment, the second array reference for $C[i-1]$ in statement $S2$ in Figure 2.1(a) has been removed by performing classical optimizations. However, hidden redundant loads still exist in the intermediate code by analyzing inter-iteration data dependencies among memory operations. For example, as shown in Figure 2.1(b), \texttt{op30} (Instruction: \texttt{op30 ld_i r20, r37, 392; load A[i-2]}) is redundant since it always loads data from the memory location in which \texttt{op26} (Instruction: \texttt{op26 st_i r37, 400, r14; store A[i]}) writes to before two iterations in the loop. All in all, there are 5 memory operations and 300 dynamic memory accesses in this example as the loop will be executed 60 times.

Motivated by this, we have developed the loop optimization technique, REALM, that further detects and eliminates redundant load operations across iterations. As shown in Figure 2.2(a) and Figure 2.2(b), we first build up a memory access graph to describe

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{motivational_example.png}
\caption{The motivational example: (a) C source code, (b) The original intermediate code generated by the IMPACT compiler after applying classical optimizations.}
\end{figure}

\begin{verbatim}
main()
{
    int i;
    int A[100], C[100];
    C[0]=C[1]=3;

    for( i = 2; i < 62; i++)
    {
        S1: A[i] = C[i-2] + C[i-1];
        S2: C[i] = A[i-2] + C[i-1];
    }
}
\end{verbatim}
Figure 2.2. The motivational example: (a) Memory access graph, (b) the reduced memory access graph, (c) The optimized code generated by the IMPACT compiler after applying our technique.
the inter-iteration dependencies among memory operations for each array. For example, in the memory access graph of array $A$ as shown in Figure 2.2(a), the edge between the store $op_{26}$ and the load $op_{30}$ with two delays denotes that $op_{30}$ always loads data from the memory location which was written by $op_{26}$ two iterations ago. Thus, this redundant load $op_{30}$ can be eliminated by exploiting register $r_{14}$ which holds the value loaded from the memory location by $op_{26}$ across two iterations. The data-flow graph is constructed using address-related operands of load/store operations. We reduce the graph in order to keep the correctness of computation and determine a definite code replacement pattern to eliminate the detected redundant load as shown in Figure 2.2(b).

Based on the constructed memory access graph in Figure 2.2(b), our technique explores hidden redundant load operations and performs code transformation to eliminate them. The resultant code is presented in Figure 2.2(c). As shown in Figure 2.2(c), after our optimization, the redundant load operations are converted to register operations which are placed at the end of the loop body before the loop-back branch. With our technique, several iterations of the eliminated load operations are promoted into the prologue in order to provide the initial values of the registers used to replace the load in the loop. As shown in Figure 2.2, by promoting load operations into the prologue, it provides more opportunities for compilers to do further optimizations for the basic block that contains the prologue. Totally, all of the three hidden redundant load operations, $op_{18}$, $op_{22}$ and $op_{30}$ are completely removed by our technique. As the loop body runs for 60 times for this example, 180 dynamic memory accesses are eliminated. It shows that our approach can effectively reduce the number of memory accesses.

Here, we compare our technique against loop unrolling, which is widely used to optimize the loop performance at the expense of code size. The source and intermediate code generated by unrolling the original loop of Figure 2.3(a) with factor of 1 and 2 are shown in Figure 2.3(b) and (c), respectively. In the unrolled loop, loop-carried data dependences are changed to intra-iteration data dependences, and as a result a few load operations are removed by classical optimization. However, as shown in Figure 2.3, the load operations can not be fully removed with the unrolling factor of 1 and 2, and the best unrolling factor...
Figure 2.3. The C code and intermediate code of the loop with unrolling factor of (a) 0 (the original loop), (a) 1, (b) 2.

is hard to be decided. For example, there are 6 loads and 6 stores in Figure 2.3-(c) with the unrolling factor of 2, and there are 240 dynamic memory accesses as the unrolled loop will be executed 20 times. In other words, it only reduces the total dynamic memory accesses of the example by 60 which shows that our technique outperforms loop unrolling. And, this example also shows that our technique has a much smaller code size expansion compared with that of loop unrolling.

Besides the significant memory accesses reduction in this example, our technique reduces the schedule length of the loop body as well. The schedule of the original loop in Figure 2.1 (b) and that of the transformed loop in Figure 2.2 (c) are shown in Figure 2.4 (a) and Figure 2.4 (b), respectively. The schedules are generated on Trimaran [47] simulator, a VLIW based simulator that has multiple function units and can process several instructions.
simultaneously. The configurations of the simulator are as follows: 2 integer ALUs, 2 memory units and 1 branch unit (the detailed configurations are presented in Section 2.6). The reason of the reduction in schedule length is that data dependencies in the loop body are changed due to the elimination of redundant loads which are formerly on the critical path. And the register operations used to replace hidden redundant loads can be put into the empty slots with multiple function units of the VLIW architecture. From this example, we can see that our technique can effectively reduce memory accesses and schedule length. Next, we will present our proposed technique.

2.4 The Redundant Load Exploration & Migration Algorithm

In this section, we first propose the REALM (REdundant Load Exploration & Migration) algorithm in Section 2.4.1, and discuss its two key functions in Section 2.4.2 and Section 2.4.3, respectively. Then we perform complexity analysis in Section 2.4.4.
2.4.1 The REALM Algorithm

Algorithm 2.4.1 Algorithm REALM.

**Input:** Intermediate code after applying all classical optimizations [2, 6, 21, 98].

**Output:** Intermediate code with hidden redundant loads eliminated.

1: Identify different arrays in the loop. For each array, put all of the load/store operations into the node set $V = \{v_1, v_2, \ldots, v_N\}$ with their original order in the intermediate code, where $N$ is the total number;

2: for each node set $V$ do
   
   // Step 1: Construct the memory access graph of this set:
   
   3: Call function $\text{Graph\_Construction}(V)$ to build up the memory access graph $G = \langle V, E, d \rangle$ of node set $V$ in order to determine the inter-iteration dependencies among memory operations (discussed in Section 2.4.2);

   // Step 2: Perform code transformation:
   
   4: Call function $\text{Code\_Transformation}(V, G)$ to eliminate hidden redundant loads of set $V$ based on the memory access graph $G$ (discussed in Section 2.4.3).

5: end for

The REALM algorithm is designed to reduce memory accesses in loops of DSP applications by exploiting inter-iteration dependencies among memory operations based on the memory access graph model. Our basic idea is to explore loop-carried data dependencies among memory operations to find hidden redundant loads and replace these loads with registers to hold the values of prior memory accesses which are unchanged or unnecessary to be fetched again from memory over multiple loop iterations. The REALM algorithm is shown in Algorithm 2.4.1.

The input of our algorithm is the intermediate code after classical optimizations. In this chapter, we select Lcode, the low-level intermediate code of IMPACT compiler [21], as the input. We choose IMPACT because it is an open-source compiler infrastructure with full support from the open-source community. Note that our technique is general enough and can be applied in different compilers.

The REALM algorithm consists of two steps. The first step is to obtain the memory operation sets for different arrays, and the second step is to perform optimizations on each
In step one, we first identify different arrays. In the following, we introduce our implementation. In IMPACT, the explicit information of array references is maintained in the high-level intermediate code, Hcode. Figure 2.5 shows the Hcode of statements $S_1$ and $S_2$ of the motivational example in Figure 2.1(a). In this code, arrays $A$ and $C$ are represented using pointer references “var P_p.A” and “var P_q.C”, respectively. In our technique, we keep such information as the annotation of the operation data structure during the stage of intermediate code generation. For pointer-based code which is widely used in DSP applications, we adopt the alias analysis techniques to identify which array is pointed by which pointer, and pass the information to Lcode. Alias analysis is the preliminary step in our work based on points-to analysis algorithms [120]. We obtain the alias information to identify array accesses via pointers, and fed them into the annotations. Thus, we can identify memory operations of the same array by comparing the annotations of them. Based on this information, we put all memory operations of the same array into a node set $V = \{v_1, v_2, ... v_N\}$ with their original order in the intermediate code. Figure 2.6 shows two memory operation sets $V_A$ and $V_C$ for array $A$ and $C$ of the motivational example (Figure 2.1).

Figure 2.5. Hcode of statements $S_1$ and $S_2$ of the motivational example.

| S1: (assign (var P_p.A) (var P_j)) (add (var P_q.C) sub (var P_i) (2)) ((var P_q.C) sub (var P_i) (1)) |
| S2: (assign (var P_q.C) (var P_j)) (add (var P_p.A) sub (var P_i) (2)) ((var P_q.C) sub (var P_j) (1)) |

Memory operation set $V_A$ of array $A$:


Memory operation set $V_C$ of array $C$:


Figure 2.6. Two memory operation sets $V_A$ and $V_C$ for array $A$ and $C$ of the motivational example.
In step two, we perform optimizations on each memory operation set. We first call function `Graph Construction()` to build up the memory access graph of the node set that describes the inter-iteration dependencies among memory operations. Then, we invoke the function `Code Transformation()` to perform code transformation on the intermediate code based on the memory access graph. The details of these two key functions are shown in Section 2.4.2 and Section 2.4.3 below.

### 2.4.2 Function `Graph Construction()`

**Algorithm 2.4.2 Function `Graph Construction()`**

**Input:** A memory operation set \( V = \{v_1, v_2, ..., v_N\} \).

**Output:** A memory access graph \( G = (V, E, d) \).

1. Get the node set of \( G \):
   
   1. Let the memory operation set \( V \) be the node set of \( G \);

2. Step 1. Memory access graph construction: \((N \text{ is the number of nodes in } V)\)
   
   2. for \( i = 1 \) to \( N \) do
      
      3. for \( j = 1 \) to \( N \) do
          
          4. Calculate the weight for each node pair \( (v_i, v_j) \): Calculate the weight for the node pair \( (v_i, v_j) \) as \( d(v_i \rightarrow v_j) = \frac{\text{distance}}{\text{step}} \), in which \( \text{step} \) is the value of base pointer for array references changing in every iteration and \( \text{distance} = \) address operand value of \( v_i \) - address operand value of \( v_j \);
      
      5. Add an edge \( v_i \rightarrow v_j \) with delay \( d(v_i \rightarrow v_j) \) into the edge set \( E \) when the destination node is a load node and the source node is either a store node or a load node;

6. end for

7. end for

3. Step 2. Memory access graph Reduction:
   
   8. Reduce the memory access graph \( G \) by calling `Graph Reduction(G)`.

**Graph Construction()** is used to build a memory access graph for each memory operation set with loop-carried data dependence analysis. Our basic idea is that inter-iteration dependencies among memory operations remain invariant if two memory operations access the same memory location among different iterations. Such relation can be exploited to eliminate the unnecessary memory accesses. And, register values which have been loaded
from memory or newly generated to be stored can be exploited in the next iterations without loading them from memory again. Therefore, we construct the memory access graph for each memory operation set to describe the inter-iteration dependencies among load/store operations using GraphConstruction() as shown in Algorithm 2.4.2.

The input of GraphConstruction() is the memory operation set of an array, and the output is a weighted data-flow graph $G$. A memory access graph $G=\langle V, E, d,t \rangle$ is an edge-weighted directed graph, where $V=\{v_1,v_1\ldots v_N\}$ is the node set including all memory operations of the same array, $t(u)$ is a function to represent the operation type (either load or store) for any node $u \in V$, $E$ is the edge set, and $d(e)$ is a function to represent the number of delays for any edge $e \in E$. The edge without delay represents the intra-iteration data dependency, while the edge with delays represents the inter-iteration data dependency. In this chapter, the inter-iteration dependency between two memory operations denotes that the source node and the destination node operate on the same memory location among different iterations. The number of delays represents the number of iterations involved.

In GraphConstruction(), we first get the node set of graph $G$ using the input memory operation set. Then, two steps, graph construction and graph reduction, are performed to build up the memory access graph as shown below.

1) Memory Access Graph Construction

We first calculate the weight for each node pair $(v_i, v_j)$ in the first step of graph construction. It involves two parts of computation.

- The first part is the distance calculation between two nodes, $v_i$ and $v_j$. In the intermediate code, memory operations consist of two operands: one is for memory address calculation and the other is to specify the register that the operation will use to load or store data. We obtain the distance between two nodes by comparing the differences of their address-related operands. For example, the distance between op22 and 18 equals to 4 as shown in Figure 2.7(a).

- The second part is to acquire the step value of the base pointer for array references
changes in every iteration. We obtain this value directly from the operands of the corresponding base pointer calculation operations for each array in the loop. For example, as shown in Figure 2.7(a), the \textit{step} value equals to the third operand of operation \textit{op60} in which the base pointer \textit{r37} changes.

After finishing all of the required computations, we calculate the number of delays for each node pair using the memory \textit{distance} between two nodes to divide the \textit{step} value of that array (\textit{distance} / \textit{step}). Thus, we can determine across how many iterations the source node and the destination node will operate on the same memory location.

After the weight calculation, we add an edge, \(v_i \rightarrow v_j\), with the number of delays \(d(v_i \rightarrow v_j)\) into the edge set \(E\) when the weight between them is greater than zero. The positive value denotes that node \(v_j\) operates on the memory location where node \(v_i\) has operated on several iterations before. Thus, node \(v_j\) can be replaced by exploiting the register value of node \(v_i\) which is loaded from memory or stored in several iterations before.

Note that we focus on eliminating redundant loads by exploiting reusable register values across different iterations. So we only add one edge into the edge set when the destination node is a load node and the source node is either a store node or a load node. Therefore, the memory access graph has the following properties:

- The store nodes can only have outgoing edges;
- There are no edges between two store nodes.

An example of memory access graph construction is shown in Figure 2.7. For the memory operation set \(V_C\) of array \(C\), after calculating the number of delays for each edge in Figure 2.7(b), we build up the graph as shown in Figure 2.7(c). For example, the edge \((\textit{op22} \rightarrow \textit{op18})\) with one delay denotes that \textit{op18} always loads data from the same memory location as \textit{op22} writes to in the previous iteration. Thus, \textit{op18} can be replaced with the register that holds the value of \textit{op22} one iteration before.
2) Memory Access Graph Reduction

After the graph is built in the first step of Graph\_Construction(), in order to determine definite code replacement patterns to eliminate redundancy, we call function Graph\_Reduction() to delete redundant edges for all loads with more than one incoming edges. Graph\_Reduction() is shown in Algorithm 2.4.3.

In Graph\_Reduction(), for each load, we keep the edge from the closest preceding memory operation, which has the latest produced values. We use two rules as shown below.

**Rule 1:**

- For each node, if it is a load node and has more than one incoming edges, keep the incoming edges with the minimum delays and delete all other incoming edges.

**Rule 2:**
Algorithm 2.4.3 Function GraphReduction().

Input: Memory access graph \( G = \langle V, E, d \rangle \).

Output: The reduced graph \( G \).

1: for \( i = 1 \) to \( N \) do
2: if ((\( v_i \) is load) && (\( v_i \) has more than one incoming edges)) then
3: Keep the incoming edges with the minimum delays and delete all other incoming edges for \( v_i \);
4: if (\( v_i \) still has more than one incoming edges) then
5: for each incoming edge of \( v_i \) do
6: if (the source node is store) then
7: Keep this edge and delete all other edges; Break;
8: end if
9: end for
10: end if
11: end if
12: end for

- After applying rule 1, if a load node still has more than one incoming edges, check the types of the source nodes of all incoming edges. If the source node of one edge is a store node, keep this edge and delete all other edges.

The reason of choosing Rule 1 is that we can use the least registers to replace redundant loads by keeping the edges with the minimum delays. Rule 2 needs to be applied because a store node updates the data of the specific memory location so we should use its register value to replace the redundant load. An example of how to reduce the data-flow graph is shown in Figure 2.7(c). According to rule 1, the edge \( (op38 \rightarrow op18) \) is deleted as it has more delays than the edge \( (op22 \rightarrow op18) \) for load \( op18 \).

After applying the above rules, we have the following properties for the reduced graph. The properties and their proofs are shown as follows.

Property 2.4.1. For each load node in the reduced memory access graph, there is only one incoming edge.
Proof. After we reduce the memory access graph according to Rule 1, only the incoming edges with the same weight remain. Along these edges, there are at most two edges in which one source node is a load and the other is a store. In other words, if there is more than one load (or store), these loads (or stores) must load (or store) data from (to) the same memory location. Therefore, these explicit redundant loads (stores) should have been eliminated by classical optimizations [2, 6, 21, 98]. Then by applying Rule 2, only the incoming edge with store as its source node will be kept. So the property is proved.

Property 2.4.2. There is no cycle in the reduced memory access graph.

Proof. There are only two types of nodes in the reduced memory access graph, store and load. As discussed in Section 2.4.2, store nodes have no incoming edges. So the cycle can only exist among load operations. For each load node, a path starting from it cannot return based on the graph construction rules. When we build up the graph, edge direction denotes that the source node always accesses the same memory location earlier than the destination node in the loop. So no cycle will exist in the graph.

2.4.3 Function Code_Transformation()

Based on the graph constructed in Section 2.4.2, Code_Transformation() is used to perform code transformation on the original intermediate code as shown in Algorithm 2.4.4.

In Code_Transformation(), we traverse the memory access graph and eliminate redundant loads by replacing them with register operations. We use a bottom-up method to perform code replacement for each redundant load node and a node is only processed after all of its child nodes have been eliminated by our technique.

Our basic idea of code replacement is to replace redundant loads with register operations. Each redundant load is removed from the loop through two steps. First, we use register operations to replace a load and put them at the end of the loop before the loop-back branch. New registers are used to be operands of these register operations which shift the register value from the source node to the destination node across multiple loop iterations. Second,
Algorithm 2.4.4 Function Code Transformation().

**Input:** Intermediate code after performing classical optimizations, the memory operation set \( V = \{ v_1, v_2 \ldots v_N \} \) and the reduced memory access graph \( G = (V, E, d) \).

**Output:** Intermediate code with hidden redundant load operations eliminated.

1: for each node \( v_i \in V \) \( (i=1 \) to \( N) \) do 
2: Associate a boolean variable, \( \text{Mark}(v_i) \), and set \( \text{Mark}(v_i) = \text{False} \); 
3: Associate an integer variable \( \text{Dep}(v_i) \), and set \( \text{Dep}(v_i) \leftarrow \) The number of children of \( v_i \); 
4: if \( ((v_i \text{ is load}) \&\& (v_i \text{ has one incoming edge})) \) then 
5: set \( \text{Mark}(v_i) = \text{True} \); 
6: end if 
7: end for 
8: while there exists a node \( v \in V \) whose \( (\text{Dep}(v) == 0 \&\& \text{Mark}(v) == \text{True}) \) do 
9: Let \( u \) be the parent node of \( v \) for edge "\( u \rightarrow v \)" with \( m \) delays. \( u \) uses \( r_u \) to load/store data from memory and \( v \) uses \( r_v \) to load data. Generate code with the following two steps: 
10: **Step 1:** In the loop body, replace redundant load \( v \) with \( m \) register move operations and put them at the end of the loop body before the loop-back branch. 
   - When \( m = 1 \), convert load \( v \) to: move \( r_u \rightarrow r_v \); 
   - When \( m > 1 \), convert load \( v \) to \( m \) register operations with the following order: move \( r_1 \rightarrow r_u \), move \( r_2 \rightarrow r_1 \), \ldots , move \( r_u \rightarrow r_{m-1} \) in which "\( r_1, r_2, \ldots , r_{m-1} \)" are newly generated registers. 
11: **Step 2:** Promote the first \( m \) iterations of \( v \) into prologue which is at the end of the previous block of the loop with the following order: 1st iteration of \( v \rightarrow r_v \), 2nd iteration of \( v \rightarrow r_1 \), \ldots , \( m \)th iteration of \( v \rightarrow r_{m-1} \); 
12: Set \( \text{Mark}(v) = \text{False} \) and calculate \( \text{Dep}(u) = \text{Dep}(u) - 1 \) for \( v \)'s parent \( u \). 
13: end while
we put several iterations of the redundant load into the prologue. The purpose of promoting load into the prologue is to initialize the register values that will be used in the loop. In the intermediate code, the prologue is put to the end of the previous basic block of the loop. For each redundant load, both the number of iterations to be promoted into prologue and the number of move operations to be generated in the new loop body are determined by the number of delays of its incoming edge.

**Figure 2.8.** Code transformations for eliminating redundant loads of array C in the motivational example.

An example of how to perform code transformations based on the constructed memory access graph is shown in Figure 2.8. Two load nodes, \(op_{18}\) and \(op_{22}\), are redundant according to the reduced graph in Figure 2.7(c). As shown in Figure 2.8(a), in the code replacement, the load node \(op_{18}\) is the node without outgoing edges. So it is first processed...
and replaced by a move operation in which the value of register \( r24 \) (the operand of \( op22 \)) is moved to \( r9 \). And the first iteration of \( op18 \) which accesses array element \( C[0] \) in the original loop is promoted into the prologue in order to initialize the value of \( r9 \) for the transformed loop body. Similarly, \( op22 \) is processed next as shown in Figure 2.8 (b).

### 2.4.4 Complexity Analysis

In the REALM technique, let \( M \) be the number of arrays and \( N \) be the number of load/store operations for each array in the loop. In the first step of the REALM algorithm, it takes at most \( O(MN) \) to obtain the node sets. In function Graph\_Construction(), for the node set of an array, it takes at most \( O(N^2) \) to construct the memory access graph among \( N \) nodes, and it takes at most \( O(N^2) \) to traverse the graph and delete the redundant edges. In function Code\_Transformation(), we can find the number of children for \( N \) nodes in \( O(N^2) \), and it takes at most \( O(N) \) to finish code replacement. Totally, for \( M \) arrays, the REALM technique can be finished in \( O(MN^2 + MN) \).

### 2.5 A Practical Register-Pressure-Aware REALM Technique

In practice, the REALM technique requires a large number of registers that may degrade the performance. Modulo scheduling [103], a back-end software pipelining technique that is widely used in DSP applications, also increases register pressure. In this section, we combine software pipelining with the REALM technique, and propose a practical algorithm called RPMS\_CP\_REALM (register-pressure-aware modulo scheduling algorithm with critical-path-based REALM) to minimize register pressure and improve performance.

Software pipelining technique has been proposed for exploiting the instruction level parallelism of loops by overlapping the execution of successive iterations. It imposes high register requirements as the lifetime of loop variables may cross the boundary of iterations. Gao et al. [44] proposed a software pipelining technique to improve performance while minimizing register requirements. Llosa et al. [85] presented a heuristic approach for resource-constrained software pipelining with reduced register pressure. The scheduling part of our
technique is based on the heuristic approach in [85]. The \texttt{RPMS\_CP\_REALM} algorithm is shown in Algorithm 2.5.1.

The objective of modulo scheduling [103] is to compute a schedule for one iteration of the loop such that when this same schedule is repeated at regular intervals, no intra- or inter-iteration dependence is violated, and no resource usage conflict arises between operations of either the same or distinct iterations. This constant interval between the start of successive iterations is termed the initiation interval (II). The minimum initiation interval (MII) is a lower bound on the smallest possible value of II for which a modulo schedule exists. The MII must be equal to or greater than both the resource-constrained MII (ResMII) and the recurrence-constrained MII (RecMII).

In algorithm \texttt{RPMS\_CP\_REALM}, \texttt{TC} is the given upper bound of schedule length, and \texttt{Budget} denotes how many schedule attempts we will try to get a legal schedule before giving up the current II. We first perform the critical-path-based REALM technique to eliminate hidden redundant load operations along the critical path with minimum cost. Then we use the register-pressure-aware modulo scheduling to perform scheduling with minimized register lifetime. Our basic idea of the first step is to reduce memory accesses along the critical path while minimizing the number of registers used to eliminate them. The cost of each redundant load operation is calculated using the number of registers required to replace it. When the critical path contains no redundant load operations, this step finishes and the generated data dependence graph becomes the input of modulo scheduling.

In register-pressure-aware modulo scheduling, we order the node list and schedule nodes using as early/late as possible schemes depending on their previously scheduled predecessors/successors in the partial schedule. The register lifetime is thus reduced compared with the conventional top-down scheduling approach [103]. Different from the technique in [103], we first put all nodes along the critical path of \texttt{DDG} into the node list with their height-based priority, and then we put other nodes into the list with their height-based order. When an operation is to be scheduled, it is scheduled using different schemes depending on its predecessors and successors in the partial schedule. If an operation has only predecessors
Algorithm 2.5.1 Algorithm RPMS_CP_REALM.

Input: Data Dependence Graph, DDG, of the input loop, the timing constraint TC, BudgetRatio.

Output: The modulo schedule with reduced register pressure.

// A practical critical-path-based REALM technique:

1: Find CP, a critical path of DDG.
2: Perform REALM to detect hidden redundant load operations of the DDG.
3: while there exists hidden redundant load operations along CP do
4: Calculate the cost of each redundant load along CP.
5: Use REALM to replace the redundant load operation with the minimum cost.
6: DDG = The changed Data Dependence Graph.
7: CP = Find the critical path of DDG.
8: end while

// register-pressure-aware modulo scheduling:

9: Initialize the value of II to the Minimum Initiation Interval II := MII().
10: while II < TC do
11: Budget := BudgetRatio * NumberofOperations;
12: Compute priorities for each node in DDG and put them into the list.
13: while (the list of unscheduled operations is not empty) & (Budget > 0) do
14: Pick up the node with highest priority and find valid time slot for it;
15: Budget := Budget - 1;
16: end while
17: II := II+1;
18: end while
in the partial schedule, then it is scheduled as early as possible (ASAP). If an operation has only successors, then it is scheduled as late as possible (ALAP). For other cases, we employ the same schedule schemes as in [85]. As the critical-path-based REALM will take at most $O(N)$ to finish assuming that there are $N$ load operations, the complexity of algorithm RPMS_CP_REALM is bounded by modulo scheduling [103].

2.6 Experiments

We have implemented our technique into the IMPACT compiler [21] and conducted experiments using a set of benchmarks from DSPstone [177] and MiBench [49] on the cycle-accurate VLIW simulator of Trimaran [47]. In this section, we first discuss our implementation and simulation environment in section 2.6.1, and then introduce our benchmark programs in section 2.6.2. The experimental results and discussion are presented in section 2.6.3.

2.6.1 The Implementation and Simulation Platform

Our experimental platform is shown in Figure 2.9. The back-end of the IMPACT infrastructure has a machine-independent optimization component called Lopti [88] which performs classical optimizations. Our optimization technique is applied on Lcode, a low-level machine-independent intermediate code. We have implemented the REALM algorithm into IMPACT for code generation. Major modifications are performed to integrate our technique into the loop optimization module of IMPACT.

![Figure 2.9. The implementation and simulation framework.](image-url)
### Functional Units
- 2 integer ALU, 2 floating point ALU, 2 load-store units
- 1 branch unit, 5 issue slots

### Instruction Latency
- 1 cycle for integer ALU, 1 cycle for floating point ALU
- 2 cycles for load in cache, 1 cycle for store, 1 cycle for branch

### Register file
- 64 registers

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Units</td>
<td>2 integer ALU, 2 floating point ALU, 2 load-store units</td>
</tr>
<tr>
<td></td>
<td>1 branch unit, 5 issue slots</td>
</tr>
<tr>
<td>Instruction Latency</td>
<td>1 cycle for integer ALU, 1 cycle for floating point ALU</td>
</tr>
<tr>
<td></td>
<td>2 cycles for load in cache, 1 cycle for store, 1 cycle for branch</td>
</tr>
<tr>
<td>Register file</td>
<td>64 registers</td>
</tr>
</tbody>
</table>

Table 2.1. The configurations of Trimaran.

To compare our technique with classical optimizations [2, 6, 21, 98], we use the Trimaran [47] infrastructure as our test platform. The configuration for the Trimaran simulator is shown in Table 2.1. The memory system consists of a 32K 4-way associative instruction cache and a 32K 4-way associative data cache, both with 64 byte block size. In the system, there are 64 registers.

#### 2.6.2 Benchmark Programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Benchmark Description</th>
<th>Benchmark</th>
<th>Source</th>
<th>Bench. Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>DSPstone</td>
<td>Convolution</td>
<td>fft-stage</td>
<td>DSPstone</td>
<td>stage scaling FFT</td>
</tr>
<tr>
<td>dot product</td>
<td>DSPstone</td>
<td>dot product</td>
<td>fft_input</td>
<td>DSPstone</td>
<td>input scaled FFT</td>
</tr>
<tr>
<td>IIR</td>
<td>DSPstone</td>
<td>IIR filter</td>
<td>bfencrypt</td>
<td>Mibench</td>
<td>blowfish encrypt</td>
</tr>
<tr>
<td>fir</td>
<td>DSPstone</td>
<td>finite response filter</td>
<td>bfdecrypt</td>
<td>Mibench</td>
<td>blowfish decrypt</td>
</tr>
<tr>
<td>fir2dim</td>
<td>DSPstone</td>
<td>2D finite response filter</td>
<td>cjpege</td>
<td>Mibench</td>
<td>JPEG compress</td>
</tr>
<tr>
<td>lms</td>
<td>DSPstone</td>
<td>least mean square</td>
<td>djpeg</td>
<td>Mibench</td>
<td>JPEG decompress</td>
</tr>
<tr>
<td>matrix1*3</td>
<td>DSPstone</td>
<td>matrix 1*3</td>
<td>gsmencode</td>
<td>Mibench</td>
<td>GSM encode</td>
</tr>
<tr>
<td>matrix</td>
<td>DSPstone</td>
<td>product of two matrices</td>
<td>gsmdecode</td>
<td>Mibench</td>
<td>GSM decode</td>
</tr>
<tr>
<td>matrix2</td>
<td>DSPstone</td>
<td>revised matrix</td>
<td>rawcaudio</td>
<td>Mibench</td>
<td>ADPCM encode</td>
</tr>
<tr>
<td>n_complex</td>
<td>DSPstone</td>
<td>n complex updates</td>
<td>rawaudio</td>
<td>Mibench</td>
<td>ADPCM decode</td>
</tr>
<tr>
<td>n_real</td>
<td>DSPstone</td>
<td>n real updates</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2. The benchmarks.

To evaluate the effectiveness of our algorithm, we choose a suite of 21 benchmarks.
which are the only ones with loops and hidden redundant load operations from DSPstone [177] and MiBench [49]. We test both the fixed-point and the floating-point versions of benchmarks from DSPstone [177]. (Two benchmarks, $fft_{stage}$ and $fft_{input}$ only have the fixed-point version.) The details of benchmarks are show in Table 2.2.

![Bar chart](chart1.png)

**Figure 2.10.** The percentages of dynamic load operations over the total number of dynamic operations for benchmarks from (a) DSPstone; (b) MiBench.

For each benchmark, we first generate code using the IMPACT compiler, and test the code on the simulator of Trimaran. The percentages of the number of dynamic load operations over the total number of dynamic operations for benchmarks from DSPstone and MiBench are shown in Figure 2.10 (a) and (b), respectively. It can be observed that, a large fraction of the dynamic operations, on average, 16.52% and 14.16% are dynamic load operations for DSPstone and MiBench, respectively. This shows that further memory access
optimizations are necessary for improving performance.

2.6.3 Results and Discussion

In this section, we first present the results obtained by our REALM technique, and compare them with the baseline scheme of the IMPACT compiler [21]. Then we compare our RPMS\_CP\_REALM with REALM in terms of average performance improvement with various register resources.

1) REALM vs. Baseline

In this section, we compare our REALM approach with the baseline scheme of IMPACT. In the experiments, we set up the maximum number of delays adopted to determine the code replacement pattern as 4 to avoid big code expansion. In the following, we present and analyze the results in terms of memory access reduction, ILP improvement, overall performance improvement and code size expansion.

Memory Access Reduction. The percentages of memory access reduction for benchmarks from DSP stone and MiBench are shown in Figure 2.11 (a) and (b), respectively. In Figure 2.11 (a), the results for fixed and floating point benchmarks from DSPstone [177] are presented in bars with different color, and the right-most bar ”AVG.” is the average result.

Our REALM algorithm reduces memory accesses by exploring hidden redundant loads with loop-carried data dependence analysis and eliminating them with register operations. Moreover, more redundant load operations in the prologue can be further eliminated by performing classical optimizations with the output of our algorithm. The experimental results show that our algorithm significantly reduces the number of memory accesses. Compared with classical optimizations, on average, our algorithm achieves 22.52% and 8.3% reduction for the benchmarks from DSPstone and MiBench, respectively.

ILP Improvement. Our technique improves ILP for each benchmark. In our experiments, ILP refers to the average number of issued operations per cycle. As shown in Figure 2.12 (a) and (b), on average, the results show that our technique achieves 12.61% and 4.43%
Figure 2.11. The reduction in the number of dynamic load operations for benchmarks from (a) DSPstone.

improvement for the benchmarks from DSPstone and MiBench, respectively. The reason of the improvement on ILP is that our technique replaces redundant load operations with register operations, the data dependence graph is thus changed and these operations can be put into the available empty slots of the multiple functional units on the VLIW architecture. Thus, the number of executed operations per cycle is increased.

**Overall Performance Improvement.** The overall performance improvement for benchmarks from DSPstone and MiBench are shown in Figure 2.13 (a) and (b), respectively. The speedup numbers are normalized based on the result of the original code generated by IMPACT. On average, the results show that our algorithm achieves a speedup of 1.13 and 1.06
for benchmarks of DSPstone and MiBench, respectively.

From the results of the benchmarks in DSPstone, we can observe that our technique leads to 13% performance improvement. The percentage of dynamic memory operations over total operations is 16.52% (shown in section 2.6.2), and our technique can reduce 22.52% of dynamic memory operations. Thus, we can get the percentage of reduction in dynamic memory operations over total operations using the product of the above two values, which is 3.72%. In the following, we present the reasons for this 13% performance improvement caused by only 3.72% reduction in memory operations. First, our technique eliminates load operations within loops that are the most time consuming part of DSP applications. Sec-
ond, load operations are usually on the critical path of the execution for loop kernels of DSP applications. After performing our algorithm, the redundant load operations across different iterations are replaced by register operations with less execution time. Thus, the data dependencies in the loop are changed accordingly. As fewer redundant load operations remain in the loop, more operations which depend on them previously can be scheduled earlier in the new loop. This leads to the reduction of the schedule length of the loop.

The results also show that the performance gain for benchmarks in MiBench (6%) is smaller than that of DSPstone. The reason is that we eliminate about 1% of dynamic operations which is relatively small compared with the big code size of benchmarks in MiBench.
**Code Size Expansion.** The percentages of code size expansion for DSPstone and MiBench are shown in Figure 2.14 (a) and (b), respectively. On average, the results show that our technique leads to 1.43% and 0.77% expansion for the code size of benchmarks in DSPstone and MiBench, respectively.

![Bar graph showing code size expansion for DSPstone benchmarks](a)

![Bar graph showing code size expansion for MiBench benchmarks](b)

**Figure 2.14.** The code size expansion of benchmarks from (a) DSPstone; (b) MiBench.

The reason of the expansion is that our technique may use more than one register operations to replace the redundant load and promote several iterations of it into the prologue. However, in our technique, the code size expansion is controlled by the maximum number of delays that determines the maximum number of register operations used to replace one redundant load operation. In the experiments, the maximum number of delays is set as 4.
Therefore, the code size expansion is very small. With such small code size expansion, our technique is suitable for embedded systems.

2) RPMS_CP_REALM vs. REALM

In this section, we compare the average performance speedup results achieved by the baseline scheme of IMPACT, our REALM with conventional modulo scheduling (REALM_MS) and the RPMS_CP_REALM technique using configurations of 16, 32 and 64 registers. The speedup numbers are normalized based on the results of the baseline scheme in IMPACT. The results for benchmarks of DSPstone and MiBench are shown in Figure 2.15 (a) and (b), respectively.

The results show that our REALM technique achieves bigger performance improvements with more register resources. And our practical RPMS_CP_REALM technique, that combines software pipelining and REALM, performs better than using REALM as it reduces register requirements. Compared with the REALM technique, for DSPstone, with the configurations of 16, 32 and 64 registers, our RPMS_CP_REALM technique contributes to an average of 2%, 4% and 6% performance improvement, respectively. For MiBench, with the configurations of 16, 32 and 64 registers, on average, our RPMS_CP_REALM technique achieves 2%, 3% and 4% performance improvement, respectively. Based on the experimental results, we conclude that our RPMS_CP_REALM technique is the best with limited register resources.

The reasons of the performance improvement achieved by RPMS_CP_REALM over REALM are as follows. First, the critical path of the loop is changed as redundant load operations along critical path with minimum cost are replaced. Second, with lower increment of register replace requirements and reduced register lifetime, the extra code spilling that may degrade performance during register allocation is alleviated.
Figure 2.15. Average performance speedup of REALM and RPMS_CP_REALM with 16 registers, 32 registers and 64 registers for benchmarks from (a) DSPstone; (b) MiBench.

2.7 Summary

In this chapter, we proposed the machine-independent loop optimization technique REALM to eliminate redundant load operations of loops for DSP applications. In our approach, we built up the memory access graph by exploiting the loop-carried dependencies among memory operations. Based on the constructed graph, we performed code transformation to eliminate redundant loads. Finally, we proposed a practical algorithm to reduce register pressure and improve performance by combining software pipelining and REALM. We implemented our techniques into IMPACT and Trimaran, and conducted experiments using a set of benchmarks from DSPstone and MiBench based on the cycle-accurate simulator of Trimaran. The
experimental results show that our technique significantly reduces the number of memory accesses compared with classical optimizations.
CHAPTER 3
LOOP SCHEDULING WITH MEMORY ACCESS REDUCTION UNDER REGISTER CONSTRAINTS FOR DSP APPLICATIONS

Abstract
In this chapter, we propose a scheduling algorithm, LSMAR (Loop Scheduling with Memory Access Reduction), to reduce hidden redundant memory accesses for DSP applications. In LSMAR, we first use the REALM technique (Chapter 2) to build up a memory access graph in order to identify all hidden redundant load operations. Then, following the dependencies, we iteratively perform loop scheduling to eliminate these redundant operations under register constraints. To the best of our knowledge, this is the first work to reduce hidden redundant memory accesses by loop scheduling for DSP applications. We implement our technique into the Trimaran compiler, and conduct experiments using a set of benchmarks from DSPstone and MiBench on the cycle-accurate VLIW simulator of Trimaran. The experimental results show that our technique can significantly reduce the number of memory accesses.

3.1 Overview
Reducing memory accesses is very important for DSP (Digital Signal Processing) applications as this can effectively improve timing performance and reduce power consumption. Loops are usually the most critical sections and consume a significant amount of time and power in DSP applications. Therefore, it becomes an important problem to reduce memory accesses for loops in DSP applications. As typical embedded systems have a limited number of physical registers, in this chapter, we perform loop scheduling to reduce hidden redundant memory accesses under register constraints.

Many memory-related issues have been addressed for DSP applications on various memory architectures [13,40,57,63,73,93,109,138,153,169]. In [40], the authors presented
a system of rewriting that manipulates the structure of linear transform algorithms to achieve load balancing and avoid false sharing. They evaluated their approach by optimizing FFT programs for a variety of shared memory platforms. Taking memory constraints into consideration, Ko et al. [63] developed a block processing algorithm to achieve efficient block processing. Zhang et al. [153] proposed an architecture to create virtual registers that do not have physical storage locations in the register file, and proposed a region-based register allocation algorithm to exploit the virtual registers for the short-lived variables. Their experimental results indicated that virtual registers are very effective at reducing register spills. Different from the above techniques, in this chapter, we focus on eliminating redundant memory accesses for DSP applications.

In the previous work, various approaches for reducing memory accesses have been investigated [17, 21, 51, 52, 61, 80, 118, 143, 150, 151]. Two compile-time optimizations, redundant load/store elimination and loop-invariant load/store migration, can reduce the amount of memory traffic by expediting the issue of instructions that use the loaded value [21]. In [17, 150, 151], Xue et al. proposed effective partial redundancy elimination techniques for optimizing memory access. Their techniques allow a value that normally resides in memory to reside in a register for some portions of the code. Li et al. [80] introduced the notion of memory access intensity to facilitate quantitative analysis of a program’s memory behavior on multi-core architectures. Most of the above approaches only consider removing explicit redundant memory accesses within one iteration of a loop. Our technique can explore and eliminate hidden redundant load operations across multiple loop iterations based on the loop-carried dependence analysis.

Over the last decade, many loop transformations, such as loop partitioning [111, 142, 176], array contraction [42, 110, 117], and loop pipelining [141, 148], have been studied extensively to reduce memory accesses. Wang et al. [142] proposed a multiple loop partition scheduling technique that combines loop partitioning and array padding to exploit data locality. In [42, 110], Gao et al. facilitated loop fusion by performing array contraction as much as possible in order to reduce the memory-register traffic of loops. Wang et al. [141] proposed an optimal loop scheduling technique for hiding memory latency based on a two-level
partitioning and prefetching scheme. Different from the above techniques, our technique replaces redundant memory operations with register operations. In addition, our technique can be integrated with the above techniques by providing more opportunities for loop transformations.

Taking register pressure issues into consideration, a great deal of research has been done on register-aware loop scheduling [25, 44, 106, 135, 140, 167]. Gao et al. [44] proposed a software pipelining technique to improve performance while minimizing register requirements. In [25], Chen et al. presented a framework for scheduling DSP applications with multi-dimensional loops subject to register constraints and other resource constraints. In our technique, we integrate register allocation and instruction scheduling to reduce memory accesses under register constraints.

In this chapter, we propose a loop scheduling algorithm, LSMAR (Loop Scheduling with Memory Access Reduction), to eliminate hidden redundant memory accesses for DSP applications with loops. Different from previous work, our basic idea is to replace hidden redundant load operations with register operations, and schedule these operations under register constraints. LSMAR consists of two phases.

- In the first phase, we use the REALM technique (Chapter 2) to build up a memory access graph in order to identify all hidden redundant load operations.

- In the second phase, we iteratively perform loop scheduling to eliminate hidden redundant load operations. Based on the memory access graph, each redundant load is first replaced by register operations with virtual register operands. In order to allocate available physical registers to the operands, a register-matching graph is then built up to describe the timing relations among the available slacks of physical registers. We solve the register allocation problem by finding a fixed length simple path between two specified vertices in the register-matching graph. Finally, based on the results of the register allocation, we perform partial scheduling in such a way that the register operations can be put into the available slots of the given schedule.
We have implemented our algorithm into the Trimaran [47] compiler. To the best of our knowledge, this is the first work to reduce hidden redundant memory accesses by loop scheduling for DSP applications.

We conduct experiments using a set of benchmarks from DSPstone [177] and MiBench [49] on the cycle-accurate VLIW simulator of Trimaran [47]. The experimental results show that our technique achieves significant memory access reduction and performance improvement with little code size expansion compared with the baseline scheme of Trimaran. For DSPstone, with the configurations of 16, 32 and 64 registers, our LSMAR algorithm contributes to an average reduction of 24.38%, 24.67% and 24.70% in the number of memory accesses, respectively. For MiBench, with the same configurations, our LSMAR algorithm achieves an average reduction of 10.56%, 10.94%, and 11.36% in memory access, respectively.

The rest of this chapter is organized as follows. The basic concepts and models are introduced in Section 3.2. Motivational examples are shown in Section 3.3. The overview of the LSMAR algorithm is given in Section 3.4. We present the details of our register allocation and instruction scheduling algorithm in Section 3.5. The experimental results and analysis are provided in Section 3.6. The summary is given in Section 3.7.

3.2 Basic Concepts and Models

In this section, we first introduce the target VLIW architecture model. Then, we show how to use a directed acyclic graph (DAG) to model loops. Next, we introduce the static schedule and the register usage map model. Finally, we define the problem to be solved in this chapter.

3.2.1 The Target VLIW Architecture

In this chapter, we use a VLIW architecture that is similar to TI C6000, as shown in Figure 3.1. This architecture has multiple functional units that can process multiple operations at the same cycle. All functional units are fully pipelined. In this VLIW architecture, a long instruction word consists of $K$ instructions and each instruction is 32 bits long. In each
Figure 3.1. The target VLIW architecture.

clock cycle, a long instruction word is fetched to the instruction decoder through a $32 \times K$-bit instruction bus and correspondingly executed by $K$ FUs. There are different types of functional units, including the integer ALU, floating-point ALU, memory unit, and branch unit. Each type of operation can only be executed on the functional unit with the same type. In the experiments, we use this VLIW architecture by extending the VLIW simulator of Trimaran [47].

3.2.2 Loops and Directed Acyclic Graph (DAG)

We use a Directed Acyclic Graph (DAG) to model a loop. A DAG $G = \langle V, E, t \rangle$ is a node-weighted directed graph, where $V$ is a set of nodes and each node denotes an operation in the loop, $E = \{ (a, b) : a \rightarrow b \in V \}$ is the edge set that defines the dependence relations for all nodes in $V$ with $(a, b)$ denoting the edge from node $a$ to node $b$, and $t(a)$ is a function to represent the computation cycles for a node $a \in V$.

We use a real DSP application, IIR filter [126], to show how to use DAG to model a loop. The C source code of the IIR filter is shown in Figure 3.2 (a), and the corresponding assembly code of its loop body generated by TI DSP CCS (Code Composer Studio) is shown in Figure 3.2 (b).

The corresponding DAG that models the loop in Figure 3.2(b) is shown in Figure 3.3 (b). In the DAG, each node is an operation of the loop, and the mapping between the node
void iir( short x[], short y[]) {
    short c1, c2, c3;
    int i;
    for (i = 0; i < 100; i++) {
        y[i+1] = (c1*x[i] + c2*x[i+1] + c3*y[i]) >> 15;
    }
}

Figure 3.2. (a) The C source code of the IIR filter (b) the assembly code of the loop generated by TI DSP CCS (Code Composer Studio).

and operation is shown in Figure 3.3 (a). Note that, in Figure 3.3 (a), we revise the register names of the original operations because we focus on the VLIW architecture with a single cluster. In Figure 3.3 (b), each edge represents the data dependency between two nodes. For example, node $D$ is a multiplication operation and it requires the data loaded by the load operation, node $A$. This dependency is represented by an edge from node $A$ to $D$ in the DAG.

### 3.2.3 Static Schedule

From the DAG of an application, we can obtain a static schedule which is a repeated pattern of an execution of the corresponding loop on the given VLIW architecture. In our work, a schedule implies both control step assignment and allocation. A static schedule must obey the dependency relations of the DAG. Assume that we want to schedule the DAG in to the target VLIW architecture with 7 FUs (discussed in Section 3.2.1). Also, let functional units $FU1$ and $FU2$ be integer ALUs (denoted by IALU), $FU3$ and $FU4$ be floating-point ALUs (denoted by FALU), $FU5$ and $FU6$ be memory units, and $FU7$ be the branch unit. The static schedule generated by the list scheduling is shown in Figure 3.3-(c).

### 3.2.4 Register Usage Map Model

We use a register usage map to represent the register reservation of a schedule. For example, the register usage map of the schedule in Figure 3.3-(c) is shown in Figure 3.4, where we
Figure 3.3. (a) The nodes and their corresponding operations; (b) The DAG that represents the loop body in Figure 3.2-(b); (c) The schedule generated by the list scheduling.

Assume that there are 16 registers in the VLIW architecture. In Figure 3.4, the columns of the map are physical registers, and the rows represent the schedule cycles. Each bold line in the map represents the life segment of a value that is kept in a physical register. The characters beside the beginning of each line denote the computation node that generates the value. The life segment of a value starts when the value is generated, and terminates when the value is consumed by the last computation node that requires this value. For example, in Figure 3.4, the physical register $A_0$ is reserved during the time interval $[0,7]$ which includes 3 life segments and 1 idle segment. The values that are kept in $A_0$ are generated by the computation nodes $A$, $D$, and $G$, respectively. Note that the overlapping life segments of
the same register are legal since a register can be used as a source operand and a destination operand at the same schedule cycle, which is supported by our VLIW architecture.

3.2.5 Problem Statement

The problem to be solved in this chapter is defined as follows:

*Given a directed acyclic graph $G = (V, E, t)$ of a loop, a schedule, and the corresponding register usage map, generate a new schedule such that the number of memory accesses is reduced while the physical register constraint is satisfied.*

3.3 Motivational Examples

In this section, we show how our technique works by applying our LSMAR algorithm on the DAG, the schedule, and the register usage map of the example shown in Figures 3.2-3.4.

From the loop in Figure 3.2-(b), we can observe that there exists two memory operations that access the same memory location across multiple iterations. In addition, the second memory operation is redundant, since the required value has been generated/fetched by the first operation and kept in physical registers. Such redundant memory operations are called *hidden redundant memory operations*. For example, in Figure 3.2-(b), node $A$ (load $x[i]$) always accesses the same array of elements that had been loaded by node $B$ (load $x[i+1]$) one iteration before. Thus, node $A$ is a hidden redundant load and can be replaced by exploiting the register $A5$, which holds the value generated by node $B$. In our technique, we build up a memory access graph as shown in Figure 3.5-(a) to model the inter-iteration relations among
Figure 3.5. (a) The memory access graph of the example (Figure 3.2) in which nodes $A$ and $C$ are hidden redundant load operations; (b) the DAG with register operations that replace node $A$.

memory operations. With this graph, we can identify two hidden redundant load operations, nodes $A$ and $C$, for this example.

Our loop scheduling algorithm iteratively eliminates the hidden redundant load operations that have been detected. For example, we generate two register operations, $MV_1$ and $MV_2$, to replace node $A$ as shown in Figure 3.5-(b). Based on the register usage map in Figure 3.4, we find an available physical register $A2$ and allocate it to the operands of the register operations. With these register operations, the value fetched by node $B$ that is kept in register $A5$ can be transferred to register $A0$ in the next iteration, and we can reuse this value for the computation of node $D$, which is the child of node $A$. As a result, the load node $A$ is removed from the loop and migrated into the prologue to initialize the register value for the first iteration.

The final schedule, the corresponding assembly code and register usage map generated by our technique are shown in Figures 3.6 (a)-(c). When we perform partial scheduling on the generated register operations, we consider the new data dependencies introduced by
them as highlighted in the dashed area of the DAG in Figure 3.5-(c). Following the dependencies, \( MV_2 \) is scheduled first at cycle 0. As another register operation \( MV_1 \) requires the data that is generated by node \( B \), it is scheduled at cycle 1 at which node \( B \) finishes its computation. Note that node \( C \) is completely removed from the schedule. The reasons for this are as follows. According to Figure 3.5-(a), the data to be stored by node \( J \) can be exploited to replace the redundant load \( C \). For nodes \( J \) and \( F \) (the child of node \( C \)), register \( A8 \) is their operands. This register can keep the value, which is stored by node \( J \) and required by node \( F \) in the next iteration, by itself. In this way, we can remove node \( C \) by directly exploiting the register value of \( A8 \).

In summary, compared with the original schedule shown in Figure 3.3-(c), our LS-MAR algorithm eliminates two hidden redundant load operations for this example. As the loop will run for 100 iterations, 200 memory accesses are reduced at run time. This example
shows that our technique can significantly reduce hidden memory accesses under register constraints.

### 3.4 Overview of the Proposed Algorithm

**Algorithm 3.4.1 Algorithm LSMAR.**

**Input:** DAG $G = (V, E, t)$ of the loop, an initial schedule $S$, and the initial register usage map.

**Output:** The changed $G$ and $S$ with redundant load operations eliminated.

// **Phase 1. Memory Access Graph Construction:**

1: Call function `GraphConstruction()` to build up the memory access graph (Algorithm 2.4.2, Chapter 2);

2: Put all hidden redundant load operations into set $R$, and associate a cost with each load in the set;

// **Phase 2. Register Allocation & Instruction scheduling (Section 3.5):**

3: while set $R$ is not empty do

4: Let $v$ be the redundant load operation with the minimum cost; Generate a register move operation chain with virtual register operands to replace $v$ based on the memory access graph;

5: Run algorithm $SCH\_REG\_ALLOC()$ (Algorithm 3.5.1) to perform register allocation and instruction scheduling on the chain with physical register constraints;

6: Remove the operation $v$ from set $R$;

7: end while

The LSMAR algorithm (Loop Scheduling with Memory Access Reduction) is designed to reduce memory accesses for DSP applications with loops. Our basic idea is to replace hidden redundant load operations with register operations, and schedule these register operations under register constraints. The overview of our LSMAR algorithm is shown in Algorithm 3.4.1.

LSMAR takes the DAG of the loop, an initial schedule and the corresponding register usage map as the input. It mainly consists of two phases.

- The **first phase** is to build up the memory access graph (Algorithm 2.4.2, Chapter 2) to describe the loop-carried dependencies among memory operations. With this graph, we can detect hidden redundant load operations.
- The second phase is to iteratively perform register allocation and instruction scheduling to eliminate redundant load operations.

In this chapter, we use the REALM technique (Chapter 2) to build up the memory access graph in the first phase of LSMAR. Here, we use an example (Figure 3.7) to show how to build up the memory access graph for the motivational example in Section 3.3. In this example, we calculate the number of delays for node pairs (B,A) of array X and (J,C) of array y. The step for each array is calculated using the value for the address increment of this array in each iteration. For example, for array X, its base address for array references is changed by the operation $K$ and the step equals to 2. After the edge weight calculation, we add edges and obtain the graph.

![Memory Access Graph](image)

**Figure 3.7. An example of the memory access graph.**

With the memory access graph, we identify all hidden redundant load operations that are nodes with incoming edges, and put these operations into a set $R$. To process these operations, we associate a cost with each of them. The cost refers to the number of registers required to replace the redundant load, and it is calculated using the incoming edge weight of the redundant load in the memory access graph. For example, as shown in Figure 3.7, nodes $A$ and $C$ are hidden redundant load operations, each with a cost of one. In the following, we present the details of our register allocation and instruction algorithm in Section 3.5.
3.5 Register Allocation & Instruction Scheduling

In the second phase of the LSMAR algorithm, we iteratively eliminate all of the redundant load operations that are identified by the memory access graph. In addition, at each iteration of this phase, we pick up the redundant load with the minimum cost for optimization. To process each redundant load, the following two stages are applied.

- In the first stage, we generate a register operation chain to replace the redundant load based on the memory access graph. The operands of this chain are virtual registers without allocating to physical registers.
- In the second stage, algorithm $SCH\_REG\_ALLOC()$ (Algorithm 3.5.1) is invoked to perform register allocation and instruction scheduling on the generated chain under register constraints.

Next, we present the details of generating register operations and algorithm $SCH\_REG\_ALLOC()$ in Section 3.5.1 and Section 3.5.2, respectively.

3.5.1 Generating a Register Operation Chain

In this section, we show how to replace a redundant load with register operations. Let “$v$” be the target redundant load operation that is with the minimum cost. And, let “$u \xrightarrow{k} v$” be the edge from $u$ to $v$ with $k$ delays in the memory access graph. Moreover, we assume that the parent node $u$ uses physical register $phy_u$ as its data operand to load/store data, and node $v$ uses physical register $phy_v$ to hold the value of the loaded data.

According to the memory access graph model, the edge “$u \xrightarrow{k} v$” denotes that nodes $u$ and $v$ access the same memory location across $k$ iterations. In other words, the register value of $phy_u$ in the $i$th iteration is the same as the value of $phy_v$ in the $(i + k)$th iteration. Thus, we can replace the redundant load $v$ by exploiting register operations to transfer the value directly from $phy_u$ to $phy_v$ across $k$ iterations. The pattern for generating a register operation chain is shown in Figure 3.8.
\( MV_{k+1} r_k \rightarrow r_{k+1}; \)
\( MV_k r_{k-1} \rightarrow r_k; \)
\( \ldots \ldots \ldots \ldots ; \)
\( MV_2 r_1 \rightarrow r_2; \)
\( MV_1 r_0 \rightarrow r_1; \)

Figure 3.8. The register operation chain.

The generated chain consists of \((k+1)\) register operations, \(MV_1, \ldots, MV_{k+1}\), and \((k+2)\) virtual register operands, \(r_0, \ldots, r_{k+1}\). Each register operation is used to transfer the reusable value of node \(u\) to the next iteration, and each operand (except \(r_0\) and \(r_{k+1}\)) is used to keep the desired value for one iteration.

Our LSMAR algorithm uses \(MV_1\) to get the reusable value of node \(u\) in each loop iteration, and uses \(MV_{k+1}\) to define the register value consumed by \(v\)'s children \(k\) iterations later. Thus, the physical register \(phy_u\) is allocated to the operand \(r_0\), and \(phy_v\) is allocated to \(r_{k+1}\). Using this chain, we can transfer the value that is loaded or stored by node \(u\) across \(k\) iterations. In such a way, node \(v\) is eliminated as the data required by its children (kept in \(phy_v\)) is defined by the move operation \(MV_{k+1}\).

### 3.5.2 Algorithm SCH_REG_ALLOC()

Algorithm \(SCH\_REG\_ALLOC()\) is designed to schedule the register operation chain, and it solves two interrelated problems. One is how to allocate physical registers to the \(k\) intermediate virtual register operands, \(r_1, \ldots, r_k\). The other one is how to schedule the \((k+1)\) register operations. Algorithm \(SCH\_REG\_ALLOC()\) is shown in Algorithm 3.5.1. It consists of the following four steps:

- First, in order to schedule the register operations, we analyze the new data dependencies introduced by incorporating these operations into the DAG of the loop. The requirements for matching virtual register operands with physical registers are obtained accordingly.
Algorithm 3.5.1 Algorithm $SCH\_REG\_ALLOC()$.  

**Input:** $G$, $S$, the register usage map $rmap$, the memory access graph $MAG$, and the register operation chain $MV_1, ..., MV_{k+1}$.  

**Output:** The changed $G$, $S$ and $rmap$.  

1: Perform a schedule analysis of the chain, and obtain the register requirements for its virtual register operands (Step 1);  
2: Build up the register-matching graph in order to find candidate physical registers that fulfill the register requirements of the chain (Step 2);  
3: Find all simple paths of length $k + 1$ between the source and sink nodes in the register-matching graph using a path finding method (Step 3);  
4: Repeat performing allocation and scheduling on the register operations based on the paths found by the above step until a legal schedule is found or every path has been checked (Step 4).  

- Second, we build up a register-matching graph in order to find candidate physical registers that can fulfill the requirements.  
- Third, we adopt a path finding method to find all simple paths of a fixed length between two specified vertices in the register-matching graph.  
- Fourth, based on the paths that are found by the above step, we allocate the physical registers to the virtual register operands. In addition, we perform partial scheduling in such a way that the register operations can be put into the available slots of the given schedule.  

In later sections, we present details of the above four steps.  

**Step 1. Schedule Analysis and Register Requirements of the Chain**  

In this section, we first analyze the data dependencies of the register operations, and give the schedule boundaries of the chain. Then, we analyze the register requirements of the chain for register allocation.  

**Schedule Analysis.** As shown in the memory access graph in Figure 3.9-(a), there is an edge from $u$ to $v$. However, this edge is not included in the DAG as shown in Figure 3.9-(b). With
this edge, node \( v \) can be replaced by a register operation chain. The new dependencies in the DAG that are introduced by the register operations are illustrated in Figures 3.9 (c)-(d).

Figure 3.9. (a) The edge \( u \rightarrow v \) in the memory access graph; (b) The original DAG; (b) The new dependencies generated by replacing \( v \) with register operations; (c) Read-write data dependencies.

In Figure 3.9, the dashed edges are added by our LSMAR algorithm. The number over each edge denotes the number of iterations of the data dependence, and the symbol below each edge represents the register (either physical or virtual) that holds the value generated by the source node and consumed by the destination node. As shown in Figure 3.9-(c), there is an inter-iteration data dependence with one delay between every two consecutive register operations. The sum of the delays associated with the edges of the chain equals to \( k \). By exploiting this chain, we can transfer the desired value from the physical register \( \text{phy}_u \) to \( \text{phy}_v \) across \( k \) iterations.

When we schedule register operations, inter-iteration data dependences are modeled as the intra-iteration read-write data dependencies as shown in Figure 3.9-(d). For example, the dashed edge, “\( MV_1 \leftarrow \frac{0}{r_1} MV_2 \)”, denotes that operation \( MV_2 \) must be scheduled no later than \( MV_1 \). Following read-write dependencies, the chain should be scheduled with the order of “\( MV_{k+1}, MV_k, ..., MV_1 \)”. In addition, the schedule range of the chain is bounded by the schedule times of the first operation, \( MV_1 \), and the last operation, \( MV_{k+1} \). In the DAG,
there is an edge from $u$ to $MV_1$ with one delay, which means that $MV_1$ consumes the data generated by node $u$ in each iteration. Thus, we schedule $MV_1$ into the live range of the value generated by $u$. Similarly, $MV_{k+1}$ has to be scheduled into the live range of the value generated by node $v$. The schedule boundaries of the chain are illustrated in Figure 3.10.

**Register Requirements.** Here, we analyze the lifetime requirements for the virtual register operands of the chain. For each register operation, $MV_i$, of the chain, its two operands, $r_{i+1}$ and $r_i$, must have an intersection of their lifetime. Also, each virtual register operand, $r_i$, is used to keep the desired value for one iteration, which means that the lifetime of $r_i$ ($i \neq 0, k + 1$) must span two iterations. With the above analysis, we obtain the lifetime requirements of the virtual register operands, $r_1, ..., r_k$, as shown in Figure 3.11.

In Figure 3.11, the beginning of the lifetime for $r_i$ is represented as “$r_i.st$”, and the end of the lifetime for it is represented as “$r_i.et$”. The first set of requirements is obtained based on the fact that a register operation is only legal when there exists an overlap between the lifetime of its source and destination operands. In other words, an overlap is required between the end time of the virtual register $r_{i-1}$ and the start time of $r_i$. The second set of requirements shows that the lifetime of each virtual register operand is required to span two iterations.
Step 2. Constructing a Register-Matching Graph

With the above analysis, in this section, we build up a Register-Matching Graph (RMG) to find available physical registers that fulfill the register requirement of the chain. Our basic idea is to match the lifetime of the virtual register operands with the idle (available) time of the physical registers. We use register slack to refer to the idle time of a physical register. Each register slack is associated with a start time, st, an end time, et, and the physical register to which the slack belongs. The register-matching graph model is defined as follows.

Definition 3.5.1. A Register-Matching Graph, $\text{RMG}=(\text{RV}, \text{RE})$, is a cyclic directed graph, where $\text{RV}$ is the set of physical register slacks whose idle intervals span two iterations, and $\text{RE}$ is the set of connection edges that are added based on the first set of register requirements of the chain.

We distinguish two vertices in an $\text{RMG}$: a source $S$ and a sink $T$. The source node $S$ is the time interval during which the value generated by node $u$ is live in physical register $\text{phy}_u$. The sink node $T$ is the time interval during which the value generated by node $v$ is live in physical register $\text{phy}_v$ before reaching $v$’s earliest child. The $\text{RMG}$ construction algorithm is shown in Algorithm 3.5.2.

In Algorithm 3.5.2, we first obtain the set of nodes based on the input register usage map (lines 1 - 3). Besides $S$ and $T$, a register slack is selected only if its idle interval spans
Algorithm 3.5.2 Constructing a Register-Matching Graph.

**Input:** The register usage map, the redundant load \( v \), and its predecessor \( u \) of the memory access graph.

**Output:** A Register-Matching Graph, \( \text{RMG} = (\text{RV}, \text{RE}) \).

// Select physical registers to be the nodes:
1: Let \( S \) be the lifetime of the value generated by node \( u \) (kept in \( \text{phy}_u \)) within one iteration;
2: Let \( T \) be the lifetime of the value generated by node \( v \) (kept in \( \text{phy}_v \)) within one iteration;
3: Let \( S, T \), and all register slacks whose idle intervals span two iterations (Requirement 2) be the nodes;

// Add edges among the selected candidates based on Requirement 1:
4: for each node \( s_i \in \text{RV} - S - T \) do
5: if \( S.\text{et} \geq s_i.\text{st} \) then add an edge from node \( S \) to node \( s_i \);
6: if \( s_i.\text{et} \geq T.\text{st} \) then add an edge from node \( s_i \) to node \( T \);
7: for each node \( s_j \in \text{RV} - S - T \ (j \neq i) \) do
8: if \( s_i.\text{et} \geq s_j.\text{st} \) then add an edge from node \( s_i \) to node \( s_j \);
9: end for
10: end for

two iterations. Then, we add edges among the nodes according to Requirement 1 (lines 4 - 10). An edge is added from \( s_i \) to \( s_j \) if the end time of \( s_i \) is greater than or equals to the start time of \( s_j \), which means that there is an overlap between the time intervals of nodes \( s_i \) and \( s_j \). The edge direction shows that \( s_i \) can be allocated to the source operand and \( s_j \) can be allocated to the destination operand.

Here, we give an example to illustrate the register-matching graph as shown in Figure 3.12. In this example, we assume that there is an edge, “\( A \xrightarrow{2} B \)”, in the memory access graph. With our technique, we can eliminate the load operation \( B \) by exploiting \( A \)’s value across two iterations.

According to the register usage map shown in Figure 3.12-(a), we obtain the nodes as follows. The source \( S \) is the time range, \([1,3]\), of \( A \)’s value residing in physical register \( r_1 \) within one iteration, and the sink \( T \) is the time range, \([0,1]\), of \( B \)’s value residing in physical register \( r_4 \) within one iteration. There are 3 register slacks that satisfy Requirement 2. They
are \(s_1:\{r2,3,2\}, \ s_2:\{r3,2,0\}\) and \(s_3:\{r5,\text{any,any}\}\) of the unassigned physical register \(r_5\).

The register-matching graph of this example is shown in Figure 3.12-(b). In this graph, \(S\) has only outgoing edges while \(T\) has only incoming edges, and there is no edge between \(S\) and \(T\) since they are time intervals within one iteration and cannot be connected without using cross-iteration register slacks. As node \(S_3\) is the slack of an unassigned register, it has incoming edges from every node except \(T\), and has outgoing edges to every node except \(S\). All other edges are added according to **Requirement 1**. For example, as the equation “\(S_1.et = 2 = S_2.st\)” holds, we add an edge from \(S_1\) to \(S_2\).

**Step 3. Finding a Fixed Length Simple Path**

Based on the register-matching graph, we solve the register allocation problem by finding a simple path \(p\) from \(S\) to \(T\) with length \((k + 1)\) in the graph. The length of path \(p\) is the number of edges along \(p\), and a path is simple if no vertex is traversed more than once. Our basic idea is as follows.

The intermediate nodes of \(p\) are register slacks that satisfy the register requirements of the chain, and we can allocate them to the virtual register operands. The length \((k + 1)\) denotes that we should have \(k\) intermediate candidate register slacks between \(S\) and \(T\), and the simple path avoids the case of assigning a register slack more than once. By using the
Input: A register-matching graph $G$ with $n$ vertices $v_1, v_2, \ldots, v_n$, two specified vertices $v_1$ (the source node $S$) and $v_n$ (the sink node $T$) of $G$, and the number of delays $k$.

Output: All simple paths from $v_1$ to $v_n$ with the length $(k + 1)$ of $G$.

Method:

1. Construct the $n \times n$ matrix $\alpha$ of $G$ in which $(\alpha(i, j) = v_j)$ ($v_j$ is the name of node $v_j$) if there is an edge from node $v_i$ to $v_j$, and is 0 otherwise;

2. Raise the matrix $\alpha$ to power $(k + 1)$, and get its entry $\alpha^{k+1}(1, n)$ which is the sum of all of the products of nodes containing $(k + 1)$ edges from node $v_1$ to $v_n$;

3. Remove all products (non-simple paths) that contain node $v_i$ or any repeated node from the entry $\alpha^{k+1}(1, n)$;

4. Get all simple paths from $v_1$ to $v_n$ with length $(k + 1)$ by concatenating $v_1$ with each of the remaining node products in entry $\alpha^{k+1}(1, n)$.

Figure 3.13. The path finding method.

$k$ physical registers whose idle intervals are connected, we can transfer the desired value across $k$ iterations.

The fixed-length simple path finding problem is NP-Complete [96]. We adopt a path searching method as shown in Figure 3.13. The method is based on the following graph property.

Property: Given an $n \times n$ adjacency matrix $A$ of a graph $G$, $A^k(i, j)$ is the number of (simple and non-simple) paths from $i$ to $j$, containing $k$ edges.

In our approach, to enumerate all paths with length $(k + 1)$ of the register-matching graph, we use a modified adjacency matrix $\alpha$ in which $\alpha(i, j) = v_j$ if there is an edge from node $v_i$ to $v_j$, and is 0 otherwise. Here, $v_j$ is the name of the $j$th node of the register-matching graph. When we calculate the product of two entries, $\alpha(i, k)$ and $\alpha(k, j)$, the result is 0 if either of them is 0, and is the concatenation $(v_kv_j)$ otherwise. Based on the above graph
property, the sum of the node products in the entry $\alpha^{k+1}(i, j)$ is the sum of paths with a length $(k + 1)$ from node $v_i$ to $v_j$. We remove the node products that contain node $v_i$ or any repeated node from the entry $\alpha^{k+1}(i, j)$, as these products represent non-simple paths. The simple paths are then obtained by concatenating $v_i$ with each of the remaining node products in entry $\alpha^{k+1}(i, j)$. An example is given in Figure 3.14 to illustrate our path finding method.

![Figure 3.14](image)

Figure 3.14. (a) The modified adjacency matrix $A$ that represents the register-matching graph in Figure 3.12-(b); (b) $A^2$; (c) $A^3$ and the entry $A^3(S, T)$ that contains all paths from $S$ to $T$ with a length of 3.

In this example, the modified adjacency matrix $A$ (shown in Figure 3.14-(a)) represents the register-matching graph in Figure 3.12-(b). For example, the entry $A(S, S1)$ is $S1$ as there is an edge from $S$ to $S1$. When we raise the matrix $A$ to power 3, the entry $A^3(S, T)$ contains all paths from $S$ to $T$ of length 3, and the paths are represented by the sum of the node products. For this example, we can get 5 paths with a length of 3 from $S$ to $T$. They are “$P_1 : S \rightarrow S_1 \rightarrow S_2 \rightarrow T$”, “$P_2 : S \rightarrow S_1 \rightarrow S_3 \rightarrow T$”, “$P_3 : S \rightarrow S_2 \rightarrow S_4 \rightarrow T$”, “$P_4 : S \rightarrow S_3 \rightarrow S_1 \rightarrow T$” and “$P_5 : S \rightarrow S_3 \rightarrow S_2 \rightarrow T$”. All of them are simple paths,
and the intermediate nodes of each path can be used to allocate registers.

The time complexity of the path finding method is $O(N^{k+1})$. Here, $N$ is the number of nodes (register slacks) in the register-matching graph, and $k$ is the number of delays for eliminating a hidden redundant load operation. Theoretically, the method is exponential. However, in practice, we fix “$k$” as a small constant in order to control the expansion of code sizes. For example, in our experiments, $k$ is set as 4. In addition, the number of nodes in the register-matching graph, $N$, is bounded by the number of physical registers for the target architecture. Therefore, in practice, the method is polynomial-time solvable with a complexity of $O(N^\beta)$, where $\beta$ is a small constant.

**Step 4. Instruction Scheduling**

We perform instruction scheduling based on the results of register allocation. If multiple simple paths exist, we sort the simple paths with the number of nodes (register slacks) that belong to unassigned registers of the path increasing in order. The motivation for this is that we try to use the register slacks of the assigned registers first, and leave more unassigned registers for replacing other redundant load operations. For example, as shown in Figure 3.14, we have 5 simple paths with a length of 3 from $S$ to $T$. In our technique, when we perform register allocation, we first choose path $P_1$ as it includes the least number of register slacks that belong to unassigned registers among the three paths.

Following the increase in order, we process the simple paths with a length of $(k + 1)$ one by one. For each such path, we perform register allocation, and attempt to schedule each register operation into the intersection time interval of its source and destination register operands as early as possible. Note that we schedule the move operations with the order of “$MV_{k+1}, MV_k, ..., MV_1$” to follow the read-write data dependencies. Once we obtain a legal schedule of the chain, the input DAG, schedule and register usage map are changed accordingly. We also migrate the redundant load operations into the prologue in order to initialize the intermediate registers of the chain. If we cannot have a legal schedule after checking every path, the input DAG, schedule and register usage map remain unchanged.
3.6 Experiments

We have implemented our technique into the Trimaran compiler [47] and conducted experiments using a set of benchmarks from DSPstone [177] and MiBench [49] on the cycle-accurate VLIW simulator of Trimaran. In this section, we first discuss our implementation and simulation environment in section 3.6.1, and then introduce our benchmark programs in section 3.6.2. The experimental results and discussions are presented in section 3.6.3.

3.6.1 Experimental Setup

We have implemented the LSMAR algorithm into Trimaran for code generation. Major modifications are performed to integrate our technique into the instruction scheduling and register allocation modules of Trimaran.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Units</td>
<td>2 integer ALUs, 2 floating point ALUs, 2 load-store units</td>
</tr>
<tr>
<td></td>
<td>1 branch unit, 5 issue slots</td>
</tr>
<tr>
<td>Instruction Latency</td>
<td>1 cycle for integer ALU, 1 cycle for floating point ALU</td>
</tr>
<tr>
<td></td>
<td>2 cycles for load in cache, 7 cycles for load in level 2 cache</td>
</tr>
<tr>
<td></td>
<td>35 cycles for load in main memory</td>
</tr>
<tr>
<td></td>
<td>1 cycle for store, 1 cycle for branch</td>
</tr>
<tr>
<td>Register file</td>
<td>64 registers</td>
</tr>
</tbody>
</table>

Table 3.1. The VLIW configurations.

To compare our technique with the baseline scheme of the compiler, as our test platform, we use the VLIW simulator of Trimaran, which has the same architecture as our system model in Section 3.2.1. The configuration for the VLIW architecture is shown in Table 3.1. The memory system consists of a 32K 4-way associative instruction cache and a 32 K 4-way associative data cache, both with a block size of 64 bytes. In the system, there are 64 registers. In the experiments, we change the number of registers to test our technique with various register constraints.
3.6.2 Benchmark Programs

To evaluate the effectiveness of our algorithm, we choose a suite of 21 benchmarks that are the only ones with loops and hidden redundant load operations from DSPstone [177] and MiBench [49]. For each benchmark, we generate code using Trimaran, and test the code on our simulator with the VLIW configurations in Table 3.1. We test both the fixed-point and the floating-point versions of benchmarks from DSPstone. Two benchmarks, fft_staged_scaled and fft_input_scaled, only have the fixed-point version. For each benchmark from Mibench, we use the set of large data as the input for execution. The details of the benchmarks are show in Table 3.2.

In Table 3.2, the columns “Total”, “Load”, “IALU”, and “FALU” represent the number of total dynamic operations, dynamic load operations, dynamic integer operations, and dynamic floating-point operations, respectively. In this chapter, dynamic operations refer to the operations at run time when executing the application with the input. The performance of each benchmark is affected by the number and types of dynamic operations. The percentages of the number of dynamic load operations over the total number of dynamic operations for benchmarks from DSPstone and MiBench are shown in column “Load over Total(%)” of Table 3.2. It can be observed that a large fraction of the dynamic operations, on average 16.52% and 14.16%, are dynamic load operations for DSPstone and MiBench, respectively. This shows that further memory access optimization is necessary to improve DSP performance and memory power.

3.6.3 Results and Discussion

In the experiments, we obtain the results of reducing memory access, speeding up performance and expanding code size with various register constraints on the code generated by our LSMAR algorithm. We compare these results with that of the baseline scheme generated by Trimaran [47]. In the experiments, as our register allocation results are based on the path finding method, we set up 4 to be the maximum number of delays adopted for eliminating redundant load operations. By doing this, we can avoid a big expansion of code, as the number
<table>
<thead>
<tr>
<th>Bench.</th>
<th>Input</th>
<th>No. of dynamic instructions</th>
<th>Load over Total(%)</th>
<th>IALU over Total(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>Load</td>
<td>IALU</td>
</tr>
<tr>
<td>DSPstone</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolution</td>
<td>-</td>
<td>305</td>
<td>51</td>
<td>115</td>
</tr>
<tr>
<td>Convolution-fix</td>
<td>-</td>
<td>271</td>
<td>36</td>
<td>165</td>
</tr>
<tr>
<td>dot_product</td>
<td>-</td>
<td>757</td>
<td>109</td>
<td>125</td>
</tr>
<tr>
<td>dot_product-fix</td>
<td>-</td>
<td>1,020</td>
<td>300</td>
<td>515</td>
</tr>
<tr>
<td>IIR</td>
<td>-</td>
<td>597</td>
<td>84</td>
<td>331</td>
</tr>
<tr>
<td>IIR-fix</td>
<td>-</td>
<td>478</td>
<td>39</td>
<td>295</td>
</tr>
<tr>
<td>fir</td>
<td>-</td>
<td>364</td>
<td>32</td>
<td>111</td>
</tr>
<tr>
<td>fir-fix</td>
<td>-</td>
<td>438</td>
<td>54</td>
<td>243</td>
</tr>
<tr>
<td>fir2dim</td>
<td>-</td>
<td>3,377</td>
<td>622</td>
<td>1,391</td>
</tr>
<tr>
<td>fir2dim-fix</td>
<td>-</td>
<td>2,658</td>
<td>451</td>
<td>1,606</td>
</tr>
<tr>
<td>lms</td>
<td>-</td>
<td>647</td>
<td>92</td>
<td>315</td>
</tr>
<tr>
<td>lms-fix</td>
<td>-</td>
<td>657</td>
<td>87</td>
<td>373</td>
</tr>
<tr>
<td>matrix 1x3</td>
<td>-</td>
<td>263</td>
<td>28</td>
<td>131</td>
</tr>
<tr>
<td>matrix 1x3-fix</td>
<td>-</td>
<td>152</td>
<td>28</td>
<td>91</td>
</tr>
<tr>
<td>matrix</td>
<td>-</td>
<td>13,960</td>
<td>3,003</td>
<td>6,210</td>
</tr>
<tr>
<td>matrix-fix</td>
<td>-</td>
<td>14,558</td>
<td>3,003</td>
<td>7,810</td>
</tr>
<tr>
<td>matrix2</td>
<td>-</td>
<td>13,270</td>
<td>2,713</td>
<td>5,910</td>
</tr>
<tr>
<td>matrix2-fix</td>
<td>-</td>
<td>13,868</td>
<td>2,713</td>
<td>7,710</td>
</tr>
<tr>
<td>n_complex_updates</td>
<td>-</td>
<td>1,498</td>
<td>327</td>
<td>718</td>
</tr>
<tr>
<td>n_complex_updates-fix</td>
<td>-</td>
<td>1,291</td>
<td>172</td>
<td>735</td>
</tr>
<tr>
<td>n_real_updates</td>
<td>-</td>
<td>788</td>
<td>155</td>
<td>400</td>
</tr>
<tr>
<td>n_real_updates-fix</td>
<td>-</td>
<td>590</td>
<td>53</td>
<td>332</td>
</tr>
<tr>
<td>fft_stage_scaled</td>
<td>-</td>
<td>470,447</td>
<td>67,525</td>
<td>276,993</td>
</tr>
<tr>
<td>fft_input_scaled</td>
<td>-</td>
<td>347,467</td>
<td>47,045</td>
<td>215,463</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mibench</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blowfishencrypt</td>
<td>large</td>
<td>1.12M</td>
<td>0.26M</td>
<td>0.64M</td>
</tr>
<tr>
<td>blowfishdecrypt</td>
<td>large</td>
<td>1.12M</td>
<td>0.26M</td>
<td>0.64M</td>
</tr>
<tr>
<td>cjpeg</td>
<td>large</td>
<td>51.18M</td>
<td>6.92M</td>
<td>25.04M</td>
</tr>
<tr>
<td>djpeg</td>
<td>large</td>
<td>29.88M</td>
<td>5.87M</td>
<td>18.30M</td>
</tr>
<tr>
<td>gsmencoder</td>
<td>large</td>
<td>387.51M</td>
<td>74.70M</td>
<td>220.87M</td>
</tr>
<tr>
<td>gsmdecoder</td>
<td>large</td>
<td>780.26M</td>
<td>40.45M</td>
<td>334.09M</td>
</tr>
<tr>
<td>rawaudio</td>
<td>large</td>
<td>885.43M</td>
<td>39.98M</td>
<td>312.61M</td>
</tr>
<tr>
<td>rawaudio-fix</td>
<td>large</td>
<td>744.68M</td>
<td>33.33M</td>
<td>265.04M</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>14.16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2. Benchmark descriptions and characteristics.
of newly generated register operations is limited accordingly.

In Tables 3.3 to 3.5, we give the results for all benchmarks tested with 16, 32, and 64 registers, respectively. In these tables, the column “Base” refers to the results generated by the baseline scheme of Trimaran [47], and column “LSMAR” represents the results obtained by our LSMAR technique. Column “IMP(%)” represents the improvement obtained by our LSMAR algorithm compared to the baseline scheme of Trimaran. In the following, we present and analyze the results in terms of reducing memory access, improving overall performance and expanding code size in Section 3.6.3, Section 4.5.3, and Section 3.6.3, respectively.

1) Reducing Memory Access

Comparing our algorithm with the baseline scheme of Trimaran, the percentages of reduction in the number of dynamic load operations for benchmarks from DSPstone and MiBench are shown in Table 3.3. For DSPstone, with the configurations of 16, 32 and 64 registers, our LSMAR technique contributes to an average reduction of 24.38%, 24.67% and 24.70% in the number of dynamic load operations, respectively. For MiBench, with the configurations of 16, 32, and 64 registers, our LSMAR technique achieves an average reduction in memory access of 10.56%, 10.94%, and 11.36%, respectively.

From these experimental results, we can see that our scheduling algorithm can effectively reduce the number of memory accesses under various register constraints. The reasons are as follows. Our LSMAR algorithm reduces memory accesses by exploring hidden redundant loads with loop-carried data dependence analysis and eliminating them by scheduling register operations. In our technique, we iteratively process the load operation with the minimum cost that requires the minimum number of registers for replacement. Furthermore, when scheduling the register operations, we use the register slacks of used registers for allocation first to save more registers. In such a way, even with tight register constraints, we have more opportunities to eliminate all possible redundant load operations.
<table>
<thead>
<tr>
<th>Bench.</th>
<th>No. of dynamic load operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of registers = 16</td>
</tr>
<tr>
<td></td>
<td>Base [47]</td>
</tr>
<tr>
<td>DSPstone</td>
<td></td>
</tr>
<tr>
<td>Conv.</td>
<td>51</td>
</tr>
<tr>
<td>Conv.-fix</td>
<td>36</td>
</tr>
<tr>
<td>product</td>
<td>109</td>
</tr>
<tr>
<td>IIR</td>
<td>84</td>
</tr>
<tr>
<td>IIR-fix</td>
<td>39</td>
</tr>
<tr>
<td>fir</td>
<td>38</td>
</tr>
<tr>
<td>fir-fix</td>
<td>57</td>
</tr>
<tr>
<td>fir2dim</td>
<td>622</td>
</tr>
<tr>
<td>fir2dim-fix</td>
<td>468</td>
</tr>
<tr>
<td>lms</td>
<td>92</td>
</tr>
<tr>
<td>lms-fix</td>
<td>104</td>
</tr>
<tr>
<td>mat.1x3</td>
<td>28</td>
</tr>
<tr>
<td>mat.1x3-fix</td>
<td>28</td>
</tr>
<tr>
<td>mat.</td>
<td>3,003</td>
</tr>
<tr>
<td>mat.-fix</td>
<td>3,003</td>
</tr>
<tr>
<td>mat.2</td>
<td>2,713</td>
</tr>
<tr>
<td>mat.2-fix</td>
<td>2,718</td>
</tr>
<tr>
<td>n_complex</td>
<td>342</td>
</tr>
<tr>
<td>n_complex-fix</td>
<td>173</td>
</tr>
<tr>
<td>n_real/f</td>
<td>x</td>
</tr>
<tr>
<td>n_real/f</td>
<td>53</td>
</tr>
<tr>
<td>ff_input/fix</td>
<td>53,127</td>
</tr>
<tr>
<td>Average Improvement (%)</td>
<td>24.38</td>
</tr>
<tr>
<td>Mibench</td>
<td></td>
</tr>
<tr>
<td>bfencrypt</td>
<td>0.30M</td>
</tr>
<tr>
<td>bfdecrypt</td>
<td>0.30M</td>
</tr>
<tr>
<td>cjpeg</td>
<td>7.81M</td>
</tr>
<tr>
<td>gsmencode</td>
<td>80.48M</td>
</tr>
<tr>
<td>gsmdecode</td>
<td>48.93M</td>
</tr>
<tr>
<td>rawcaudio</td>
<td>40.00M</td>
</tr>
<tr>
<td>rawaudio</td>
<td>33.33M</td>
</tr>
<tr>
<td>Average Improvement (%)</td>
<td>10.56</td>
</tr>
</tbody>
</table>

Table 3.3. The number of dynamic load operations under various register constraints for benchmarks.
2) Improving Performance

The overall improvement in performance for benchmarks from DSPstone and MiBench is shown in Table 3.4. On average, for DSPstone, with the configurations of 16, 32, and 64 registers, our LSMAR technique contributes to a respective 13.24%, 13.48%, and 13.65% improvement in performance. For MiBench, with the configurations of 16, 32, and 64 registers, on average, our LSMAR technique achieves a performance improvement of 8.26%, 8.73%, and 9.16%, respectively. The reasons for the improvement are shown as follows.

First, our technique eliminates load operations within loops, which are the most time-consuming part of DSP applications. With the reduction in the number of memory accesses, the number of cache misses is reduced accordingly. Second, load operations are usually on the critical path of the execution of the loop kernels of DSP applications. After performing our algorithm, the redundant load operations across different iterations are replaced by register operations with less execution time. Thus, the data dependencies in the loop are changed accordingly. As fewer redundant load operations remain in the loop, more operations that previously depended on them can be scheduled earlier in the new loop. Instruction-level parallelism improves accordingly, with higher utilization of multiple functional units. This leads to a reduction in the length of the schedule of the loop. Third, when scheduling register operations, we consider register constraints and functional unit constraints. We only remove a redundant load when we have enough registers for replacement. This alleviates the spilling of registers that has a great impact on performance.

3) Expanding Code Size

The percentages of code size expansion for DSPstone and MiBench are shown in Table 3.5. In this table, static instructions refer to the instructions at compile time, and the code size of each benchmark is evaluated by the number of static instructions. For DSPstone, with the configurations of 16, 32, and 64 registers, our LSMAR technique leads to an average expansion in code size of 1.90%, 2.00%, and 2.02%, respectively. For MiBench, with the configurations of 16, 32, and 64 registers, on average, our LSMAR technique leads to an expansion in code size of 0.81%, 0.87%, and 0.90%, respectively.
<table>
<thead>
<tr>
<th>Bench.</th>
<th>No. of execution cycles</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of registers = 16</td>
<td>Base [47]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv.</td>
<td>461</td>
<td>402</td>
</tr>
<tr>
<td>Conv.-fix</td>
<td>482</td>
<td>448</td>
</tr>
<tr>
<td>product</td>
<td>1,262</td>
<td>1,134</td>
</tr>
<tr>
<td>product-fix</td>
<td>2,235</td>
<td>2,046</td>
</tr>
<tr>
<td>IIR</td>
<td>968</td>
<td>770</td>
</tr>
<tr>
<td>IIR-fix</td>
<td>876</td>
<td>717</td>
</tr>
<tr>
<td>fir</td>
<td>627</td>
<td>581</td>
</tr>
<tr>
<td>fir-fix</td>
<td>860</td>
<td>718</td>
</tr>
<tr>
<td>fir2dim</td>
<td>4,172</td>
<td>4,044</td>
</tr>
<tr>
<td>fir2dim-fix</td>
<td>3,247</td>
<td>3,318</td>
</tr>
<tr>
<td>lms</td>
<td>992</td>
<td>767</td>
</tr>
<tr>
<td>lms-fix</td>
<td>1,060</td>
<td>833</td>
</tr>
<tr>
<td>mat.1x3-fix</td>
<td>444</td>
<td>363</td>
</tr>
<tr>
<td>mat.</td>
<td>10,611</td>
<td>9,973</td>
</tr>
<tr>
<td>mat.-fix</td>
<td>13,384</td>
<td>12,066</td>
</tr>
<tr>
<td>mat.2</td>
<td>9,770</td>
<td>7,525</td>
</tr>
<tr>
<td>mat.2-fix</td>
<td>9,672</td>
<td>8,930</td>
</tr>
<tr>
<td>n_complex</td>
<td>1,831</td>
<td>1,571</td>
</tr>
<tr>
<td>n_complex-fix</td>
<td>1,722</td>
<td>1,470</td>
</tr>
<tr>
<td>n_real</td>
<td>881</td>
<td>758</td>
</tr>
<tr>
<td>n_real-fix</td>
<td>663</td>
<td>574</td>
</tr>
<tr>
<td>fft_stage/fix</td>
<td>332,363</td>
<td>279,617</td>
</tr>
<tr>
<td>fft_input/fix</td>
<td>261,398</td>
<td>208,935</td>
</tr>
<tr>
<td>Average Improvement (%)</td>
<td>13.24</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4. The improvement in performance under various register constraints for benchmarks.
<table>
<thead>
<tr>
<th>Bench.</th>
<th>No. of registers = 16</th>
<th>No. of static instructions</th>
<th>No. of registers = 32</th>
<th>No. of static instructions</th>
<th>No. of registers = 64</th>
<th>No. of static instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Base [47]</td>
<td>LSMAR</td>
<td>EXP(%)</td>
<td>Base [47]</td>
<td>LSMAR</td>
<td>EXP(%)</td>
</tr>
<tr>
<td>Conv.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>43</td>
<td>44</td>
<td>2.33</td>
<td>43</td>
<td>44</td>
<td>2.33</td>
</tr>
<tr>
<td>Conv.-fix</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>product</td>
<td>64</td>
<td>65</td>
<td>1.56</td>
<td>64</td>
<td>65</td>
<td>1.56</td>
</tr>
<tr>
<td>product-fix</td>
<td>30</td>
<td>31</td>
<td>3.33</td>
<td>30</td>
<td>31</td>
<td>3.33</td>
</tr>
<tr>
<td>IIR</td>
<td>137</td>
<td>139</td>
<td>1.46</td>
<td>137</td>
<td>139</td>
<td>1.46</td>
</tr>
<tr>
<td>IIR-fix</td>
<td>140</td>
<td>142</td>
<td>1.43</td>
<td>140</td>
<td>142</td>
<td>1.43</td>
</tr>
<tr>
<td>fir</td>
<td>81</td>
<td>82</td>
<td>1.23</td>
<td>81</td>
<td>82</td>
<td>1.23</td>
</tr>
<tr>
<td>fir-fix</td>
<td>93</td>
<td>94</td>
<td>1.08</td>
<td>93</td>
<td>94</td>
<td>1.08</td>
</tr>
<tr>
<td>fir2dim</td>
<td>278</td>
<td>280</td>
<td>0.72</td>
<td>266</td>
<td>268</td>
<td>0.75</td>
</tr>
<tr>
<td>fir2dim-fix</td>
<td>323</td>
<td>325</td>
<td>0.62</td>
<td>311</td>
<td>313</td>
<td>0.64</td>
</tr>
<tr>
<td>lms</td>
<td>160</td>
<td>162</td>
<td>1.25</td>
<td>140</td>
<td>142</td>
<td>1.43</td>
</tr>
<tr>
<td>lms-fix</td>
<td>194</td>
<td>195</td>
<td>1.04</td>
<td>161</td>
<td>163</td>
<td>1.24</td>
</tr>
<tr>
<td>mat.1x3</td>
<td>72</td>
<td>74</td>
<td>2.78</td>
<td>72</td>
<td>74</td>
<td>2.78</td>
</tr>
<tr>
<td>mat.1x3-fix</td>
<td>38</td>
<td>40</td>
<td>5.26</td>
<td>38</td>
<td>40</td>
<td>5.26</td>
</tr>
<tr>
<td>mat.</td>
<td>154</td>
<td>156</td>
<td>1.30</td>
<td>154</td>
<td>156</td>
<td>1.30</td>
</tr>
<tr>
<td>mat.-fix</td>
<td>149</td>
<td>151</td>
<td>1.34</td>
<td>149</td>
<td>151</td>
<td>1.34</td>
</tr>
<tr>
<td>mat.2</td>
<td>166</td>
<td>168</td>
<td>3.03</td>
<td>166</td>
<td>168</td>
<td>3.03</td>
</tr>
<tr>
<td>mat.2-fx</td>
<td>183</td>
<td>185</td>
<td>1.09</td>
<td>183</td>
<td>185</td>
<td>1.09</td>
</tr>
<tr>
<td>n-complex</td>
<td>189</td>
<td>193</td>
<td>2.12</td>
<td>165</td>
<td>169</td>
<td>2.42</td>
</tr>
<tr>
<td>n-complex-fx</td>
<td>203</td>
<td>207</td>
<td>1.97</td>
<td>163</td>
<td>167</td>
<td>2.45</td>
</tr>
<tr>
<td>n-real/fx</td>
<td>100</td>
<td>102</td>
<td>2.00</td>
<td>100</td>
<td>102</td>
<td>2.00</td>
</tr>
<tr>
<td>n-real-fx</td>
<td>98</td>
<td>100</td>
<td>2.04</td>
<td>98</td>
<td>100</td>
<td>2.04</td>
</tr>
<tr>
<td>fft-stagefix</td>
<td>364</td>
<td>372</td>
<td>2.18</td>
<td>316</td>
<td>323</td>
<td>2.22</td>
</tr>
<tr>
<td>fft-input/fx</td>
<td>348</td>
<td>359</td>
<td>3.19</td>
<td>332</td>
<td>343</td>
<td>3.32</td>
</tr>
<tr>
<td>Average Expansion (%)</td>
<td>1.90</td>
<td>2.00</td>
<td></td>
<td></td>
<td>2.00</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5. The expansion in code size under various register constraints for benchmarks.
The reason of the expansion is that our technique may use more than one register operation to replace the redundant load and promote it into the prologue. However, in our technique, the expansion in code size is controlled by the maximum number of delays that determines the maximum number of register operations used to replace one redundant load operation. In the experiments, the maximum number of delays is set as 4. Therefore, the code size expansion is very small. With such a small expansion in code size, our technique is suitable for embedded systems.

3.7 Summary

In this chapter, we proposed a two-phase loop scheduling algorithm, LSMAR, to reduce memory accesses for DSP applications with loops. In our approach, we built up the memory access graph to describe the inter-iteration dependencies among memory operations. Based on this graph, we performed register allocation and instruction scheduling to eliminate redundant load operations under register constraints. We implemented our technique into the Trimaran compiler, and conducted experiments using a set of benchmarks from DSPstone and MiBench based on the cycle-accurate VLIW simulator of Trimaran. The experimental results show that our technique significantly reduces the number of memory accesses.
CHAPTER 4  
LEAKAGE-AWARE MODULO SCHEDULING FOR EMBEDDED VLIW PROCESSORS

Abstract
As semi-conductor technologies move down to the nanometer scale, leakage power has become a significant component of the total power consumption. In this chapter, we develop a novel leakage-aware modulo scheduling algorithm to achieve leakage energy savings for applications with loops on VLIW architecture. The proposed algorithm is designed to maximize the idleness of function units integrating with the dual-threshold domino logic, and reduce the number of transitions between active and sleep modes. We have implemented our technique into the Trimaran compiler and conducted experiments using a set of embedded benchmarks from DSPstone and Mibench on the cycle-accurate VLIW simulator of Trimaran. The results show that our technique achieves significant leakage energy savings compared with the DAG based leakage-aware scheduling algorithm.

4.1 Overview
Low-power design has been an important issue for embedded systems, due to its significant impact on battery life, system density, cooling cost, and system reliability. Traditionally, dynamic power is the primary contributor to total power consumption. Dynamic Voltage Scaling (DVS) is an effective technique to reduce the dynamic power by dynamically scaling down the supply voltage $V_{DD}$ and operational frequency [26, 27, 54, 60, 87]. However, supply voltage scaling often requires a reduction in the threshold voltage $V_{TH}$, which leads to an exponential rise in the sub-threshold leakage current, and hence the leakage power consumption. Leakage power constitutes approximately 50% of the processor’s power for sub-65nm technologies [59, 113], and it is expected to dominate the overall energy consumption
in future technologies [124].

VLIW (Very Long Instruction Word) architecture that has multiple functional units (FUs) and can process several instructions simultaneously is widely adopted in embedded processors. While this multiple-FUs architecture can be exploited to increase instruction-level parallelism and improve time performance, it causes more leakage power consumption. Therefore, it becomes an important problem to reduce leakage energy of applications on VLIW architecture. Since loops are usually the most critical parts in an application, we develop a loop scheduling technique to reduce the overall leakage energy of applications on VLIW architecture.

Many techniques, including clock gating, power gating and dual-threshold domino logic with sleeping transistors, have been proposed and evaluated to achieve leakage power reduction on circuit-level [16, 39, 128, 129]. Clock gating is an effective technique for leakage reduction and has been widely used in practices. For example, the IBM Power PC 405LP [19] implements clock gating by aggressively shutting down elements of the processor which are idle. The Intel PXA [45] family processors support fine granularity clock gating to exploit the fact that not all system transistors are used simultaneously. In [128], Tsai et al. provided a comparison of three run-time leakage power reduction techniques, input vector control, body bias control and power gating in terms of leakage savings, feasibility and scaling trend. Dropsho et al. [39] proposed a hardware based leakage energy management scheme for short idle periods. In their scheme, the dual-threshold domino logic with sleep mode that can transit between active mode and sleep mode without any performance penalty is utilized. In this chapter, we use the energy model in [39] to calculate the leakage energy and total energy.

Energy-efficient system-level scheduling techniques with considerations of leakage power dissipation have been explored in [28, 34, 70, 86, 100, 154, 163, 165, 171–174]. Jha et al. [86] addressed the problem of variable-voltage scheduling of multi-rate periodic task graphs in heterogeneous distributed real-time embedded systems. Kuo et al. [28] developed an on-line simulated scheduling strategy and a virtually blocking time strategy for reducing leakage power consumption on a uniprocessor DVS system. Their algorithms derived a
feasible schedule for real-time tasks with worst-case guarantees for any input instance. Xu et al. [172,174] proposed a dynamic programming algorithm for periodic tasks on processors with practical discrete speed levels. Their algorithm determined the lower bound of energy expenditure in pseudo-polynomial time.

In recent work, compiler-directed instruction-level scheduling techniques [62, 95, 105, 158] have been investigated to reduce leakage energy. Gupta et al. [105] introduced a power-aware flow graph to determine program regions which offer opportunities to save leakage in the context of super-scalar processors. By exploiting scheduling slacks of instructions, Nagpal et al. [95] proposed a leakage-aware instruction-level scheduling algorithm for VLIW and clustered VLIW architectures to reduce leakage energy. In [158], You et al. provided compiler solutions for the analysis and scheduling with power gating control at the component level. They presented a data flow analysis framework for estimating the component activities at fixed points of programs whilst considering pipeline architectures. In most of the above work, the instruction scheduling for reducing leakage power is based on DAG (Directed Acyclic Graph) scheduling in which only intra-iteration dependencies are considered. In this chapter, we show that we can reduce the leakage energy by carefully exploiting inter-iteration dependencies of a loop using software pipelining.

Software pipelining [37,38,106–108] is a technique that exploits the instruction level parallelism among the iterations of a loop by overlapping the execution of consecutive loop iterations. Modulo scheduling [103] is a software pipelining technique that results in high performance code. In the previous work, power-aware resource constrained software pipelining [155, 156, 161] has been studied for reducing power consumption without degrading performance. In [155], Yang et al. formulated the power-aware software pipelining problem as an integer linear programming problem. Kim et al. [161] proposed a modulo scheduling algorithm that produces a more balanced schedule to reduce the peak power and step power. Different from the above work, in this chapter, we focus on reducing overall leakage energy of embedded VLIW processors.

In this chapter, we propose a leakage-aware modulo scheduling technique that assists
the hardware based leakage control scheme [39] to achieve leakage savings for DSP applications in the context of VLIW architecture. Our basic idea is to schedule operations into better locations in order to maximize the idleness of function units integrating with leakage reduction control, and reduce the number of transitions between active and sleep modes. In our technique, we first obtain the data flow graph generated by the compiler, and begin modulo scheduling with the calculated minimum initiation interval. Then we perform leakage-aware scheduling within the timing constraint and record the minimum leakage energy achieved.

The main contributions of our approach can be summarized as follows.

- We study and address the leakage-aware scheduling problem which is important for embedded VLIW DSP processors. Different from the previous work, our technique exploits inter-iteration dependencies of a loop to reduce leakage by maximizing the idle slacks of functional units.

- We propose a novel leakage-aware modulo scheduling technique, and discuss the trade-off between leakage energy reduction and performance penalty.

- We have implemented our technique into the Trimaran compiler [47] and conducted experiments on a set of benchmarks from DSPstone [177] and Mibench [49] based on the power model in [39]. The results show that our algorithm achieves significant leakage energy savings compared with the previous work. On average, our technique contributes to 13.53% reduction in leakage energy consumption with only 1.74% performance penalty compared with the leakage-aware scheduling algorithm [95].

The rest of the chapter is organized as follows. The basic concepts are introduced in Section 4.2. Motivational examples are shown in Section 4.3. The leakage-aware modulo scheduling algorithm is proposed in Section 4.4. The experimental results and analysis are provided in Section 4.5, and the summary is given in Section 4.6.
4.2 Basic Concepts

In this section, we first introduce the Data Flow Graph in Section 4.2.1, and present the system model and static schedules in Section 4.2.2. Then we give the energy model in Section 4.2.3. Finally, we provide the overview of modulo scheduling in Section 4.2.4.

4.2.1 Cyclic Data-Flow-Graph (DFG)

We use a cyclic Data Flow Graph (DFG) to denote a loop in our works. A cyclic DFG \( G=\langle V, E, d, t \rangle \) is a node-weighted and edge-weighted directed graph, where \( V \) is a set of nodes and each node denotes a computation task in the loop, \( E \) is the edge set, \( d(e) \) is a function to represent the number of delays for any edge \( e \in E \), and \( t(u) \) is a function to represent the computation cycles for any node \( u \in V \).

The delay represents the number of iterations of the dependency. The edge without delay represents the intra-iteration data dependency, and the edge with delays represents the inter-iteration data dependency. For example, in the cyclic DFG shown in Figure 4.1, the edge \( A \rightarrow D \) with no delay represents the intra-iteration data dependency, which denotes that the data generated by node \( A \) is needed to process node \( D \) in the same iteration. And the edge \( L \) to itself with one delay represents the inter-iteration data dependency, which denotes that the data generated by node \( L \) one iteration ago is needed for processing node \( L \) in the current iteration.

4.2.2 System Model and Static Schedule

In this chapter, we use a VLIW architecture which has multiple functional units that can process multiple operations at the same cycle. All functional units are fully pipelined. In the VLIW architecture, there are different types of functional units including integer ALU, floating-point ALU, memory unit and branch unit. Each type of operations can only be executed on the functional unit with the same type.

An exemplary VLIW architecture is shown in Figure 4.2. In Figure 4.2, in the VLIW architecture, there are 7 functional units in which \( FU1 \) and \( FU2 \) are integer ALUs (de-
Figure 4.1. The FIR program and its corresponding data flow graph.

From the DFG of an application, we can obtain a static schedule which is a repeated pattern of an execution of the corresponding loop on the architecture. In our work, a schedule implies both control step assignment and allocation. A static schedule must obey the dependency relations of the Directed Acyclic Graph (DAG) portion of the DFG. The DAG is obtained by removing all edges with delays in the DFG. From the DFG shown in Figure 4.1, we obtain a static schedule as shown in Figure 4.2-(a) generated by mod-
ulo scheduling. A set that contains consecutive idle time in a column of a schedule is defined as an empty block. For example, for the integer ALUs of the schedule in Figure 4.2-(a) (Section 5.3), there are 2 empty blocks: Block$_1$ = \{4, FU1\}, (5, FU1\} and Block$_2$ = \{(3, FU2), (4, FU2), (5, FU2)\}.

### 4.2.3 Energy Model

The energy model used in this chapter is based on [39]. According to this model, circuits have three operating modes: (1) Active$_{mode}$, in which a circuit dissipates both dynamic and leakage power; (2) Standby$_{mode}$, in which a circuit is waiting to execute an operation, dissipating only leakage power; (3) Sleep$_{mode}$, in which a circuit is deactivated by the leakage control mechanism, dissipating a reduced leakage power. The total energy in a functional unit in this model is determined as follows:
\[ E_{total} = E_{Dyn} + E_{Leak} + E_{Tran} + E_{Sleep} \] (4.1)

\[ E_{total} = n_A(\alpha E_A + (1-D)E_{S1}) + (n_A D + n_{UI})(\alpha E_{S0} + (1-\alpha)E_{S1}) + \\
M_Z((1-\alpha)E_A + E_{Sleep}) + n_Z E_{S0} \] (4.2)

Here, \( E_{Dyn} \) is the dynamic energy, \( E_{Leak} \) is the total leakage energy, \( E_{Tran} \) is the transition energy overhead and \( E_{Sleep} \) is the leakage energy consumption for the sleep mode. In equation 4.2, \( n_A \) is the number of active cycles, \( n_{UI} \) is the number of idle cycles which cannot be put into sleep mode, \( n_Z \) is the number of sleep mode cycles and \( M_Z \) is the number of transitions between different modes. The activity factor \( \alpha \) is the fraction of the circuits that are actually discharged during the evaluation phase. The duty cycle \( D \) is the fraction of time in which the clock signal is high. \( E_{S0} \) and \( E_{S1} \) are low leakage and high leakage energy, and are related by the following equations, \( E_{S0} = s \times E_{S1} \) and \( E_{S1} = p \times E_A \).

Here, \( s \) is the ratio of the leakage energy per cycle in the low leakage state to that in the high leakage state, and \( p \) is the ratio of the maximum leakage energy expended to the maximum energy for evaluation per cycle. After simplifying and normalizing the equations with respect to the active energy, the model is as follows,

\[ E_{total} = n_A(\alpha + (1-D)p) + (n_A D + n_{UI})(\alpha sp + (1-\alpha p) + \\
M_Z((1-\alpha)E_A + E_{Sleep}) + n_Z sp \] (4.3)

In our technique, after we obtain a fix schedule, we get the number of cycles at which the nodes are executed as the number of active cycles, \( n_A \), of the energy model. For these cycles, as we can not achieve leakage saving, the consumed leakage energy per cycle is in the high leakage state. In the leakage energy model, the leakage management scheme utilizes the characteristics of dual-threshold domino logic with sleep mode without any performance penalty, and it puts any functional units into low leakage mode after one cycle of idleness [39]. Based on this, we apply the leakage management scheme to all of the empty blocks of the fixed schedule, and get the number of cycles which work in the low
leakage state and the number of transitions. The transition overhead is thus calculated by using the number of transitions and the energy consumption per transition. In such a way, we calculate the overall leakage energy of the schedule.

### 4.2.4 Modulo Scheduling Overview

The objective of modulo scheduling [103] is to compute a schedule for one iteration of the loop such that when this same schedule is repeated at regular intervals, no intra- or inter-iteration dependence is violated, and no resource usage conflict arises between operations of either the same or distinct iterations. This constant interval between the start of successive iterations is termed the initiation interval (II). The repetitive portion can be re-rolled to yield a new loop which is termed the kernel. The prologue is the code that precedes the repetitive part and the epilogue is the code following the repetitive part.

The minimum initiation interval (MII) is a lower bound on the smallest possible value of II for which a modulo schedule exists. The MII must be equal to or greater than both the resource-constrained MII (ResMII) and the recurrence-constrained MII (RecMII).

The recurrence-constrained MII, denoted by RecMII, is defined to be the maximum-time-to-delay ratio over all cycles of the DFG, i.e.,

\[
RecMII = \max_{\forall \text{ cycle } l \in G} \frac{\text{Time}(l)}{\text{Delay}(l)},
\]

where \(\text{Time}(l)\) is the sum of computation time in cycle \(l\), and \(\text{Delay}(l)\) is the sum of delay counts in cycle \(l\).

The resource-constrained MII, denoted by ResMII, is defined as the maximum ratio of number of operations to number of FUs over all FU types, i.e.,

\[
ResMII = \max_{\forall \text{ FU type } A} \frac{N(A)}{F(A)},
\]

where \(N(A)\) is the number of operations using type-A FUs in the DFG, and \(F(A)\) is the number of type-A FUs available. After RecMII and ResMII are obtained, then the minimum initiation interval, denoted by MII, can be obtained by taking the maximum value of RecMII.
and ResMII, i.e. 

$$MII = \max \{ \text{RecMII}, \text{ResMII} \}.$$ 

In the iterative modulo scheduling, the candidate II is initially set to the MII and increased until a legal modulo schedule is found.

### 4.3 Motivational Examples

In order to show how our approach works, we present motivational examples in this section. We use Trimaran compiler [47] to generate the Data Flow Graph for this example and perform modulo scheduling on it. We compare the scheduling generated by the traditional modulo scheduling and our technique. The energy model to calculate the leakage energy is introduced in Section 4.2.3. The parameters are set as same as in [39]. Particularly, we assume that the energy consumption in different modes is normalized with respect to the active mode as 

$$E_S = 0.5 \times E_A, E_{\text{sleep}} = 0.01 \times E_A \text{ and } E_{\text{overhead}} = 0.5 \times E_A.$$ 

A real DSP (Digital Signal Processing) application, the FIR (Finite Impulse Response) program, and the corresponding data flow graph are shown in Figure 4.1. The Data Flow Graph of the innermost loop of the application is generated by the Trimaran compiler [47]. In the graph, each node denotes a computation task in the loop, and there are 7 integer ALU operations, 4 memory operations, 1 floating ALU operation and 1 branch for this example. The edge without delay represents the intra-iteration data dependency (e.g. \(A \rightarrow D\)), and the edge with delays represents the inter-iteration data dependency (e.g. \(J \rightarrow A\) has an edge with one delay which is denoted by one bar), in which the number of delays represents the number of iterations involved. In this chapter, intra-iteration dependencies mean the dependencies inside an iteration while inter-iteration dependencies mean the dependencies among different iterations.

Assume that we want to schedule the graph in Figure 4.1 to the VLIW architecture with 7 FUs which is mentioned in section 4.2.2. And let \(FU1\) and \(FU2\) be integer ALUs, \(FU3\) and \(FU4\) be floating-point ALUs, \(FU5\) and \(FU6\) be memory units and \(FU7\) be the branch unit. Note that we assume the integer operations \(A, B, C, H, J, K, L\) take 1 cycle
for execution, the load operations $D$, $E$, $F$ take 2 cycles, the store operation $I$ takes 1 cycle, and the floating multiplication node $G$ takes 2 cycles to finish execution in this example.

The schedule generated by the traditional modulo scheduling is shown in Figure 4.2-(a). Based on the power model in [39], for integer ALUs which are most heavily utilized, the dual-threshold domino logic with sleep mode can transit between active mode and sleep mode after one cycle of idleness. The circuit expends very little leakage energy in the sleep mode. However, the energy savings are severely affected by frequent transitions from active mode to sleep mode and vice-versa because of many short idle periods. The leakage energy of integer ALUs for this schedule can be calculated as:

$$\begin{align*}
E_{\text{leakage}} &= 7 * E_S + 1 * E_{\text{sleep}} + 2 * E_{\text{overhead}} = 4.51 * E_A
\end{align*}$$

The schedule generated by our technique without performance loss is shown in Figure 4.2-(b). In this schedule, we enlarge the consecutive idle cycle in functional unit $FU2$ by scheduling more operations in the first integer functional unit. Compared with the performance-oriented schedule in Figure 4.2-(a), we can transit more time slots into the sleeping mode using this schedule. It shows that we have chance to achieve leakage optimization with such tight timing constraint. The leakage energy of integer ALUs for this schedule can be calculated as

$$\begin{align*}
E_{\text{leakage}} &= 7 * E_S + 3 * E_{\text{sleep}} + 1 * E_{\text{overhead}} = 4.03 * E_A
\end{align*}$$

The procedure of our scheduling technique is shown in Figure 4.2-(c)-(f). In the DFG of the motivational example as shown in Figure 4.1, nodes $A$, $B$ and $C$ have the highest priority. $B$ and $C$ are scheduled at the same cycle by the performance-oriented modulo scheduling as shown in Figure 4.2-(a). In the first step of our algorithm, when we schedule operation $B$, a time slot $(1, FU1)$ is selected since it is the earliest available slot in the first integer ALU $FU1$. Accordingly, $C$ is scheduled at cycle 2 since the time slot $(2, FU1)$ is the first available slot of the first integer functional unit. The partial schedule of operations $A$, $B$ and $C$ is shown in Figure 4.2-(c).
The final schedule generated by our technique is shown in Figure 4.2-(f). Our basic idea is to exploit idleness of FUs as much as possible so that they can be transited from the active mode to the sleep mode with the consideration of the transition dynamic power overhead. By carefully exploiting the inter-iteration data dependencies, we have more chance to schedule nodes into better locations in order to maximize the idleness of function units integrating with leakage reduction control, and reduce the number of transitions between active and sleep modes. In our approach, we set a timing constraint and record the schedule with the minimum leakage energy. For the motivational example, the schedule generated by our technique has little performance loss than the traditional modulo scheduling algorithm. In this schedule, \(FU2\) is totally unused, so we can put \(FU2\) into the sleep mode before entering the loop body and no transition energy overhead is needed to be calculated again inside the loop body. The leakage energy of integer ALUs for this schedule can be calculated as \(E_{\text{leakage}} = 7*E_S + 7*E_{\text{sleep}} = 3.57*E_A\). Thus, our technique achieves big leakage savings compared with the performance-oriented modulo scheduling.

4.4 Leakage-Aware Modulo Scheduling Algorithm

In this section, we first propose the leakage-aware modulo scheduling algorithm in Sections 4.4.1. Then we discuss its two key functions in Section 4.4.2 and Section 4.4.3, respectively.

4.4.1 The Proposed Algorithm

In the proposed algorithm, our basic idea is to schedule nodes of a loop to better locations in order to enlarge the idleness in FUs which can be exploited to apply leakage energy control mechanism. In most of the previous work, loop is modeled as the DAG part of a DFG in which only intra-iteration dependencies are considered. As shown in Section 2, by exploring inter-iteration dependencies, we can get more opportunities to schedule nodes of DFG to better locations in a schedule to achieve more leakage energy saving. The leakage-aware modulo scheduling algorithm is shown in Algorithm 4.4.1.
Algorithm 4.4.1 The leakage-aware modulo scheduling algorithm.

**Input:** The data flow graph $G=(V, E, d, t)$, the timing constraint $TC$, BudgetRatio.

**Output:** A schedule $S$ with minimum leakage energy $Min\_Energy$.

// Initialize the value of II to the Minimum Initiation Interval

1: $II := MII()$;
2: $Min\_Energy \leftarrow \infty$;
3: while $II < TC$ do
4:   Budget := BudgetRatio * NumberofOperations;
5:   while IterativeSchedule(II, Budget) != SUCCESS do
6:     $II := II + 1$;
   // Calculate the leakage energy based on the power model [39]
7:   $S' :=$ The generated legal schedule;
8:   $E_{S'} := $ CalculateEnergy($S'$);
9:   if $Min\_Energy > E_{S'}$ then
10:     $S \leftarrow S'$ and $Min\_Energy \leftarrow E_{S'}$
11: end if
12: end while
13: end while
In the algorithm, $G$ is the data flow graph of the loop, $TC$ is the timing constraint, and $BudgetRatio$ is the ratio of the maximum number of operation scheduling steps attempted before giving up the number of operations in the loop. This parameter determines how hard the function IterativeSchedule() tries to find a legal schedule for a candidate II before giving up. The output of the algorithm is a legal schedule $S$ with reduced leakage energy $Min_{Energy}$.

In the algorithm, we first initialize the value of II to the minimum initiation interval using function MII(). The value of MII is calculated by analyzing the input data flow graph. One lower bound, the resource-constrained MII (ResII), is derived from the critical resource usage requirements of the computation. The other one, the recurrence-constrained MII (RecII) is derived from latency calculations around elementary circuits in the data flow graph. The minimum initiation value equals to the minimum value of ResII and RecII.

Then, function IterativeSchedule() is used to perform the actual leakage-aware scheduling as shown in Algorithm 4.4.2. This function contains a height-based priority function which defines a topological sort of the operations so that each operation is scheduled before any of its successors. The basic idea of function IterativeSchedule() is to search for a legal partial schedule each time avoiding the dead-end state. When the scheduler finds that there is no available slot for scheduling the currently selected operation, it displaces it with one or more previously scheduled, conflicted operations. This rescheduling policy guarantees the forward progress of the scheduling algorithm.

After all operations have been scheduled and a legal schedule is generated, we record the leakage energy of it and compare it with $Min_{Energy}$. We apply the leakage reduction technique in the energy model [39] to every available idle slack considering the timing and energy overhead. The algorithm terminates when the timing constraint is achieved.

### 4.4.2 Function IterativeSchedule()

In function IterativeSchedule(), we first calculate the priority for each operation based on the height-based priority function ComputePriority() of modulo scheduling, and pick up
Algorithm 4.4.2 Function IterativeSchedule().

Input: Graph \( G \), the initiation interval \( II \) and the Budget.

Output: A schedule \( S \) or failure information.

// Calculate the schedule time bounds for the current operation
1: ComputePriority();

2: while (the list of unscheduled operations is not empty) & (Budget > 0) do

3: CurrOper = HighestPriority();

   // Calculate the schedule time bounds for the current operation
4: (MinTime,MaxTime) = ComputeSlack(CurrOper);

   // Select the time slot for leakage energy optimization
5: SchedSlot = FindTimeSlot(CurrOper,MinTime,MaxTime);

   // Perform the actual scheduling
6: Schedule(CurrOper,SchedSlot);
7: Budget–;

8: end while

9: if all operations are scheduled then
10: return SUCCESS;
11: else
12: return FAILURE
13: end if
the operation with highest priority to be scheduled. In function ComputePriority(), we calculate the longest path from the node to the end of the data flow graph. This function gives higher priority to operations on the critical path in order to achieve a good schedule.

Then, the schedule time bounds for the current operation are calculated according to the data dependence constraint. We use function FindTimeSlot() to find a legal time slot for the current operation within the range \((\text{MinTime}, \text{MaxTime})\). \text{MinTime} is the earliest start time for an operation as constrained by its dependences on its predecessors. \text{MaxTime} equals to \((\text{MinTime} + \text{II} - 1)\) since each iteration in modulo scheduling begins exactly \text{II} cycles after the previous one. Function FindTimeSlot() is shown in Algorithm 4.4.3.

It is possible that we can not find any empty block to put the operation in. In this case, we employ the same backtracking method as that of Rau’s modulo scheduling algorithm [103]. In the context of recurrences and iterative modulo scheduling, it is impossible to guarantee that all of an operation’s predecessors have been scheduled [103]. In the backtracking, the operations which are conflict with the currently scheduled operation, either because of resource usage or due to dependence conflicts, are unscheduled and put into the unscheduled node list. The backtracking procedure can be shown as follows.

- Assume that there is a time slot between MinTime and MaxTime for the current operation with no resource conflict. The scheduled immediate successors of the current operations are unscheduled.

- Assume that every time slot is with resource conflict. To ensure the forward progress, if the current operation is scheduled previously, it can not be scheduled at the same time slot this time. This avoids the situation where two operations keep on displacing each other endlessly. If MinTime is less than the previous schedule time, the current operation is scheduled at MinTime; if not, it is scheduled one cycle later than it was scheduled previously.
Algorithm 4.4.3 Function FindTimeSlot().

Input: CurrOper, MinTime, MaxTime, the number of functional units $n$, functional unit set $FUS = (FU_1, FU_2, ..., FU_n)$.

Output: The schedule time $SchedSlot$.

1: CurrTime := MinTime;
2: SchedSlot := NULL;
3: while (SchedSlot == NULL) & (CurrTime <= MaxTime) do
4:   for j=1 to $n$ do
5:     Calculate the start time $ST_B$ and the end time $ET_B$ of the empty block $B$ on FU[j].
6:     if node $CurrOper$ can be scheduled at $ST_B$ then
7:       the schedule time $SchedSlot = ST_B$;
8:     else
9:       $SchedSlot = ET_B - t(u)$;
10:    end if
11:   end for
12:  CurrTime++;
13: end while
14: if SchedSlot == NULL then
15:   if (NeverScheduled(CurrOper)) or (MinTime > PrevScheduleTime(CurrOper)) then
16:     SchedSlot := MinTime;
17:   else
18:     SchedSlot := PrevScheduleTime(CurrOper) + 1;
19:   end if
20: end if
21: return SchedSlot;
4.4.3 Function FindTimeSlot()

In function FindTimeSlot(), the goal is to find an empty block to put the operation \( CurrOper \) in. We first calculate the start time and the end time of each empty block in each functional unit.

In order to enlarge the idleness in FUs, we always start to search from \( FU_1 \) and try to find the earliest empty block on it. If we can not find an empty block in \( FU_1 \), \( FU_2 \) will be tried next time; then we try \( FU_3, \ldots, FU_n \), until we can find such an empty block. In this way, we can schedule operations onto one functional unit as much as possible. And thus, we can enlarge the idleness of FUs and increase the number of unused FUs. The benefit is that we have more chance to put the total unused functional units into low leakage mode before entering the loop body.

After finding the suitable empty block, we compare the earliest schedule time of the operation and the start time of the empty block in order to determine whether the operation should be scheduled at the beginning or at the end of this empty block. By scheduling the nodes into locations close to each other, we can maximize the consecutive idle period in functional units.

Here, we perform complexity analysis on our leakage-aware modulo scheduling algorithm. In the iterative scheduler (Algorithm 4.4.2), the \( Budget \) which denotes how many attempts we will try to get a legal schedule before giving up the current II, equals to a constant times the number of nodes in the given DFG. Thus, Algorithm 4.4.2 has a complexity factor of \( O(N^2) \), where \( N \) is the number of nodes of the input DFG. In Algorithm 4.4.1, the termination of the outer loop (line 3) depends on the legal II that we can get. Therefore, the time complexity of the proposed algorithm is \( O(II \times N^2) \).

4.5 Experiments

We have implemented our technique into the Trimaran compiler [47] and conducted experiments using a set of benchmarks from DSPstone [177] and MiBench [49] on the cycle-accurate VLIW simulator of Trimaran. In this section, we first discuss the setup of our
experiments in Section 4.5.1, and then introduce our benchmark programs in section 4.5.2. The experimental results and discussion are presented in section 4.5.3.

4.5.1 Experimental Environment

Trimaran is developed to conduct state-of-the-art research in compilation techniques for ILP architectures with a specific focus on VLIW class of architectures. The back-end of the Trimaran infrastructure, ELCOR, has a modulo scheduler designed and implemented for flat VLIW architectures. We have modified the Trimaran suite to generate code. The result is simulated using a variety of VLIW configurations on the VLIW simulator of Trimaran. The modulo scheduler has been modified to implement the algorithm described in the last section.

To compare our technique with the performance-oriented modulo scheduling [103] and the leakage-aware scheduling algorithm [95], we use the VLIW simulator of Trimaran [47] as our test platform. The leakage-aware scheduling algorithm [95] is a DAG-based technique which only considers intra-iteration data dependencies. It consists of two main steps. In the first step, it uses a priority function to sort the instruction list. Then it maintains a functional unit map which explicitly keeps track of the status of each functional unit, and decides the binding of a chosen instruction to a functional unit. We implement this algorithm for the comparison.

In the experiment, the technology parameters (s=0.01 and $\frac{E_{	ext{dissipate}}}{E_A}$ =0.01) are set as same as in [39]. Considering the 65nm fabrication technology, we assume that the leakage energy is 50% of the total energy consumption of the VLIW processor ($\rho$=0.5), and use a typical value of 0.5 for both $\alpha$ and $D$ in our simulation.

The configuration for the VLIW Trimaran simulator is shown in Table 4.1. In the configuration, there are 2 integer ALUs, 2 floating-point ALUs, 2 memory units, 1 branch unit and 5 VLIW issue slots. The memory system consists of a 32K 4-way associative instruction cache and a 32K 4-way associative data cache, both with 64 byte block size. And the second-level cache is a unified 32K 4-way associative cache. In the system, there are 64 registers.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Units</td>
<td>2 integer ALU, 2 floating point ALU, 2 load-store units</td>
</tr>
<tr>
<td></td>
<td>1 branch unit, 5 issue slots</td>
</tr>
<tr>
<td>Instruction Latency</td>
<td>1 cycle for integer ALU, 1 cycle for floating point ALU</td>
</tr>
<tr>
<td></td>
<td>4 cycles for integer multiplication, 4 cycles for float multiplication</td>
</tr>
<tr>
<td></td>
<td>2 cycles for load in cache, 1 cycle for store, 1 cycle for branch</td>
</tr>
<tr>
<td>Register file</td>
<td>64 registers</td>
</tr>
</tbody>
</table>

Table 4.1. The configurations of Trimaran.

### 4.5.2 Benchmark Programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Benchmark Description</th>
<th>Benchmark</th>
<th>Source</th>
<th>Benchmark Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>DSPstone</td>
<td>Convolution</td>
<td>fft-stage</td>
<td>DSPstone</td>
<td>stage scaling FFT</td>
</tr>
<tr>
<td>dot_product</td>
<td>DSPstone</td>
<td>dot product</td>
<td>fft_input</td>
<td>DSPstone</td>
<td>input scaled FFT</td>
</tr>
<tr>
<td>IIR</td>
<td>DSPstone</td>
<td>IIR filter</td>
<td>bfencrypt</td>
<td>Mibench</td>
<td>blowfish encrypt</td>
</tr>
<tr>
<td>fir</td>
<td>DSPstone</td>
<td>finite response filter</td>
<td>bfdecrypt</td>
<td>Mibench</td>
<td>blowfish decrypt</td>
</tr>
<tr>
<td>fir2dim</td>
<td>DSPstone</td>
<td>2D finite response filter</td>
<td>cjpeg</td>
<td>Mibench</td>
<td>JPEG compress</td>
</tr>
<tr>
<td>lms</td>
<td>DSPstone</td>
<td>least mean square</td>
<td>djpeg</td>
<td>Mibench</td>
<td>JPEG decompress</td>
</tr>
<tr>
<td>matrix1*3</td>
<td>DSPstone</td>
<td>matrix 1*3</td>
<td>gsmencode</td>
<td>Mibench</td>
<td>GSM encode</td>
</tr>
<tr>
<td>matrix</td>
<td>DSPstone</td>
<td>product of two matrices</td>
<td>gsmdecode</td>
<td>Mibench</td>
<td>GSM decode</td>
</tr>
<tr>
<td>matrix2</td>
<td>DSPstone</td>
<td>revised matrix</td>
<td>rawaudio</td>
<td>Mibench</td>
<td>ADPCM encode</td>
</tr>
<tr>
<td>n_complex</td>
<td>DSPstone</td>
<td>n complex updates</td>
<td>rawaudio</td>
<td>Mibench</td>
<td>ADPCM decode</td>
</tr>
<tr>
<td>n_real</td>
<td>DSPstone</td>
<td>n real updates</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2. The benchmarks.

To evaluate the effectiveness of our algorithm, we use a suite of thirteen benchmarks from DSPstone [177], and eight benchmarks from MiBench [49]. The details of benchmarks are show in Table 4.2. For each benchmark, we generate code using Trimaran for modulo scheduling, the leakage-aware scheduling technique [95] and our technique. We test the code on the simulator of Trimaran.
4.5.3 Results and Discussion

In the experiments, we obtain the results of the leakage energy reduction of Integer ALUs and performance penalty on the code generated by our technique. We compare the energy results with that of the code generated by the leakage-aware scheduling algorithm in [95]. We compare the performance penalty results with that of the code generated by the performance-oriented modulo scheduling [103]. In our implementation, we apply the technique in [95] to reduce leakage for non-loop basic blocks, and apply our technique to loops. When we compare the energy results of the two schemes, the timing constraint is set as the schedule length generated by the technique in [95].

In this section, we report the experimental results for integer ALUs which are heavily used for the benchmarks from DSPStone and MiBench. As shown in Table 3.2, on average, 57.61% and 49.38% of total operations are IALU operations for the benchmarks from DSPStone and MiBench, respectively. Thus, the leakage power gains reported here have not been magnified by the leakage energy saving of the memory, floating-point, and branch units which are mostly idle.

In the experiments, we assume that the technology is 65nm and 50% of the total energy of the VLIW processor is the leakage energy. In the following, we first discuss our timing constraint selection policy. Then, we present and analyze the results in terms of zero performance penalty, leakage energy reduction for IALUs, and performance penalty.

1) Timing Constraint Selection

In this section, we discuss the trade-off between the total energy reduction and performance penalty. We choose the timing constraint for our scheduling technique based on the analysis of the following energy equation.

\[
E_{total} = (E_A + E_S) * n_A + E_S * n_S + E_{sleep} * n_{sleep} + M_Z * E_{overhead} \tag{4.4}
\]

Here, \(E_A\), \(E_S\) and \(E_{sleep}\) are energy consumption for the active mode, the unco-
trolled idle mode and the sleep mode. $n_A$, $n_S$ and $n_{sleep}$ are the number of cycles for the active mode, the uncontrolled idle mode and the sleep mode. $M_Z$ is the number of transitions between different modes, and $E_{\text{overhead}}$ is the transition energy overhead.

Using our technique, we can enlarge idle slacks for each functional unit and transit many cycles in the uncontrolled idle mode $n_U$ into the sleep mode $n_S$. The leakage energy is reduced accordingly since $E_S$ is 50 times larger than $E_{sleep}$. However, with the increment of the schedule length, the circuits in extra cycles consume more leakage energy even though we can put them into the sleep mode. Frequent transitions can also affect the energy reduction results.

The experimental results show the similar trend as discussed above. In the experiment, we normalize the energy consumption in different modes with respect to the active mode as $E_S = 0.5 \times E_A$ and $E_{sleep} = 0.01 \times E_A$, which is the same as that of the parameters in [39]. Based on our experimental results, the rough relationship between total energy consumption and the timing constraint for the benchmarks can be shown in Figure 4.3.

In Figure 4.3, the X-axis represents the increment of TC and the Y-axis represents the increment of the total energy. It can be observed that the total energy increases when the timing constraint is greater than the optimal point. Thus, in the experiment, we set the timing constraint as $1.3 \times MII$ (Minimum Initiation Interval) since legal schedules and minimum leakage energy can be achieved within this TC for all benchmarks.

2) Zero Performance Penalty

As performance and energy consumption are two major concerns for embedded systems. In the experiment, we restrict the timing constraint to be the schedule length achieved by performance-oriented modulo schedule. We compare our algorithm with modulo scheduling integrated with leakage management scheme [39], and the percentage of leakage energy reduction for integer ALUs under this constraint is shown in Figure 4.4. On average, our algorithm achieves 1.04% leakage energy reduction for integer ALUs without performance loss.
Figure 4.3. The rough relationship between total energy consumption and the timing constraint for different benchmarks.
Figure 4.4. Leakage energy reduction without performance loss.

In Figure 4.4, the results for modulo scheduling with/without leakage control and our technique are presented in bars with different color, and the right-most bar “AVG.” is the average result. We can observe that our technique achieves more leakage energy reduction than modulo scheduling for some large benchmarks. The reason is that our technique is able to put some operations into better locations and enlarge idle slacks of functional units within the timing constraint. For other benchmarks with limited schedule length, our technique achieves the same leakage energy reduction as modulo scheduling.

3) Leakage Energy Reduction

We compare our algorithm with the leakage-aware scheduling [95], and the percentage of reduction in the leakage energy consumption for integer ALUs is shown in Figure 4.5. The experimental results show that our algorithm significantly reduces the leakage energy of the processor. Compared with the leakage-aware scheduling technique [95], on average, our algorithm achieves 13.53% reduction for the benchmarks.

The reason for the reduction is described as follows. Our algorithm reduces leakage consumption in the functional units by scheduling operations using less functional units to
maximize the idleness of the functional units. Moreover, in the loop-level granularity, our technique minimizes the number of transition time between low level and high level leakage mode by turning off the totally unused functional units before entering the loop body. The leakage-aware scheduling [95] is a DAG-based technique which only take intra-iteration data dependencies into consideration. By exploiting inter-iteration data dependencies, our technique has more chance to schedule nodes into better locations.

4) Performance Penalty

We compare our technique with the performance-oriented modulo scheduling, and the performance penalty of our technique is shown in Figure 4.6. In Figure 4.6, the performance of our technique is normalized using the performance of traditional modulo scheduling. On average, the results show that our technique leads to a 1.74% performance penalty for the benchmarks.

The reason of the performance loss is that our technique may use less functional units to schedule the operations and try to schedule them close to each other. Thus, it may enlarge the schedule length to achieve the goal of maximizing the idleness of functional units. How-
ever, in our technique, the performance penalty is controlled by the timing constraint that determines whether employing our technique or not. In the experiment, the maximum initiation interval (timing constraint) is set as 1.3 * MII (Minimum Initiation Interval). Therefore, the performance penalty is very small. With such small performance loss, our technique is suitable for embedded systems.

4.6 Summary

In this chapter, we have studied the scheduling problem that minimizes leakage energy for applications with loops on VLIW architectures. A leakage-aware modulo scheduling algorithm has been proposed to reduce overall leakage energy for VLIW architectures. We have introduced the leakage power management model, and presented the relationship between this model and the fixed schedule. The basic idea of the proposed algorithm is to maximize the idleness of function units, and reduce the number of transitions between active and sleep modes. The overall leakage energy is thus reduced by exploiting the schedule slacks with leakage saving schemes. We have implemented our technique into the Trimaran compiler and conducted experiments using a set of embedded benchmarks from DSPstone and Mibench.
on the cycle-accurate VLIW simulator of Trimaran. The results show that our technique achieves significant leakage energy savings compared with the DAG based leakage-aware scheduling technique.
CHAPTER 5
TEMPERATURE-AWARE DATA ALLOCATION FOR EMBEDDED SYSTEMS WITH CACHE AND SCRATCHPAD MEMORY

Abstract
The hybrid memory architecture that contains both on-chip cache and scratchpad memory (SPM) has been widely used in embedded systems. In this chapter, we explore the benefits of both cache and SPM in jointly optimizing performance and temperature for embedded applications with loops. Our basic idea is to adaptively adjust the workload distribution of on-chip memories based on the current temperature. For a problem in which the workload can be estimated a priori, we present a nonlinear programming formulation to optimally minimize the total execution time of a loop under the constraints of SPM size and temperature. To solve a problem in which the workload is not known a priori, we propose a temperature-aware adaptive loop scheduling algorithm, TALS, to dynamically allocate data to cache and SPM at runtime. The experimental results show that our algorithms can effectively achieve joint performance and temperature optimization for embedded systems with cache and SPM.

5.1 Overview
As processors and memories are packed in ever smaller areas with reduced feature sizes, on-chip temperature has become a dominant design constraint in embedded systems. High on-chip temperatures adversely affect system performance by decreasing transistor switching speed, circuit reliability by causing electrical failures, power and energy by increasing power consumption, and system cost by increasing cooling and packaging costs. In embedded systems, hot spots caused by high temperatures are often generated by the on-chip memory subsystem, which is the the most frequently accessed component. Therefore, the problem of how to manage the temperature of the on-chip memory subsystem has become an important
Traditionally, on-chip cache is the most common on-chip memory that has been extensively used to improve performance. Cache is completely controlled by hardware, and it consumes a large amount of power due to its complex tag-decoding logic. Scratchpad memory (SPM), a small fast software-managed on-chip SRAM (Static Random Access Memory), has been widely used in embedded systems, as its advantages lie in power and area [7]. A recent study [7] has shown that SPM has area that is 34% smaller and power consumption that is 40% lower than hardware-managed cache of the same capacity. As cache typically accounts for 25%-50% of total chip power, SPM can help to significantly reduce the power consumption of embedded processors. However, unlike the cache, SPM requires complex program analysis and explicit support from software. To strike a balance, most modern embedded processors such as ARM10E, ARM11, and ColdFire MCF5, adopt a hybrid memory architecture that contains both on-chip cache and SPM. Previous studies in this field have focused on utilizing cache and SPM to improve performance and power consumption. However, without taking into account issues involving temperature, both cache and SPM will become hot spots due to the large number of memory accesses. Different from previous work, in this chapter, we focus on exploring a hybrid architecture to jointly optimize performance and peak temperature.

A lot of research effort has been put into the field of architecture-level temperature modeling and management [65, 66, 74, 81, 114, 116, 146]. Jha et al. [65] proposed and evaluated the HybDTM (Hybrid Dynamic Thermal Management) approach to model and manage the overall temperature of processors. Wu et al. [146] developed a dynamic cache sub-array permutation scheme using crossbars in the address predecoder to alleviate the thermal stress on areas of high leakage. Skadron et al. developed a thermal simulator, HotSpot [116], to calculate the transient temperature based on the given physical characteristics and power consumption of units on the die. Their model is based on an equivalent circuit of thermal resistances and capacitances.

Based on the above temperature models, various temperature-aware scheduling tech-
niques have been proposed [22, 82, 86, 89, 101, 160, 175]. In [101], the authors presented necessary and sufficient conditions for real-time schedules to guarantee the maximal temperature constraint. Qu et al. [160] studied the relation between the temperature and leakage in real-time systems. They also proposed an online temperature-aware leakage minimization algorithm that adjusts the processor modes at runtime based on the chip temperature. Hu et al. [22] developed a temperature-aware real-time scheduling and assignment approach for hard real-time applications on MPSOC (Multiprocessor-System-on-Chip) architectures. Moreover, as temperature is highly dependent on power consumption, in particular leakage power, existing power-aware optimization techniques [27, 28, 136, 139, 167, 172–174] have positive effects for reducing temperature. Our technique can be combined with the above system-level and instruction-level scheduling techniques to further reduce on-chip temperatures.

In the research fields of cache-aware and SPM-aware optimizations, existing investigations have mainly focused on improving performance, data locality, and power consumption [75, 76, 132, 162, 170]. In [170], a low-power data cache design scheme has been proposed to reduce power by exploiting the frequent value locality. Vera et al. [132] explored the use of cache partitioning and dynamic cache locking to provide worst-case performance estimates for multi-task systems. SPM has been widely used in embedded systems, given its advantages in power and area compared to the hardware-managed cache. Xue et al. [75] presented a general-purpose approach, called memory coloring, to automatically allocate the arrays in a program to the scratchpad memory. However, temperature is not considered in the above techniques. In embedded systems, with the large amount of heat that is generated by extensive memory accesses, both on-chip cache and SPM have become the on-chip hot spots. This will eventually lead to a significant degradation in performance and severe problems with reliability. Different from these techniques, we not only fully explore SPM to improve time performance and reduce the temperature of cache, but also reassign the workload of SPM to cache in order to balance the distribution of the on-chip temperature.

In this chapter, we jointly optimize performance and temperature by adaptively utilizing cache and SPM for data allocation in embedded systems. In our method, we divide
the loop iterations into several segments, and alternatively allocate the data of each segment to either cache or SPM. To the best of our knowledge, this is the first work to explore the hybrid embedded memory architecture for jointly optimizing performance and peak temperature. Our main contributions are summarized as follows:

- We mathematically formulate the problem of finding an optimal data allocation scheme for embedded systems with cache and SPM to minimize the execution time of a loop using nonlinear programming (NLP). Our formulation considers the constraints of SPM size and temperature. This formulation can be utilized for solving the problem in which the workload can be estimated \textit{a priori}.

- To solve the problem in which the workload is not known \textit{a priori}, we propose a temperature-aware adaptive loop scheduling algorithm, \textit{TALS}, to dynamically allocate data to cache and SPM at runtime. Our basic idea is to adaptively adjust the workload distribution of on-chip memories based on the current on-chip temperature. In \textit{TALS}, for improving performance, SPM is used with the highest priority for data allocation. By monitoring the temperature, we make runtime decisions to reassign the workload when the temperature of any of the two memory components exceeds the threshold temperature. In such a way, both cache and SPM can be cooled down for some iterations during the execution of the loop. As a result, the on-chip temperature can be reduced.

We have implemented our algorithms into the Trimaran compiler [47]. We conduct experiments using a set of benchmarks from DSPstone [177] and MiBench [49], and evaluate our algorithms on the cycle-accurate VLIW simulator of Trimaran [47] by integrating the power model for both cache and SPM in [7, 145], the SPM cost model [75], and the thermal simulator, HotSpot [116]. In the experiments, we use LINGO 8.0 [125] to solve instances of our NLP formulation for optimal execution time under the constraints of SPM size and temperature. Experimental results show that our NLP formulation produces optimal solutions for small problem instances in which the workload is known \textit{a priori} and the number of loop
iterations is small. For the problem in which the workload is not known \textit{a priori}, the results show that our \textit{TALS} algorithm gives solutions in much less time compared with the NLP model.

The rest of this chapter is organized as follows. The basic concepts and models are introduced in Section 5.2. We present the NLP formulation and propose the \textit{TALS} algorithm in Section 5.3. The experimental results and analysis are provided in Section 5.4. The summary is given in Section 5.5.

5.2 Basic Concepts and Models

In this section, we first introduce the system model, power model, and temperature model in Section 5.2.1, Section 5.2.2, and Section 5.2.3, respectively. Then, in Section 5.2.4, we define the problem to be solved in this chapter.

5.2.1 System Model

In this section, we present the working mechanism for embedded systems with cache and SPM. Figure 5.1-(a) shows the architecture block diagram of embedded processors with cache and SPM. The address and data buses from the CPU connect to data cache, scratchpad memory, and the external memory interface (EMI) blocks. For a memory access request from the CPU, the data cache indicates a cache hit to the EMI block through the C\_HIT signal. Similarly, if the SRAM interface determines that the referenced memory address maps into the on-chip SRAM, it assumes control of the data bus and indicates this status to the EMI through signal S\_HIT. If both the cache and SRAM report misses, the EMI transfers a block of data of the appropriate size between the cache and the DRAM.

The data address space mapping is shown in Figure 5.1-(b), for a memory of size N data words. Memory addresses \(0 \ldots P - 1\) map into the Scratch-Pad memory, and have a single processor cycle access time. Thus, in Figure 5.1-(a), S\_HIT would be asserted whenever the processor attempts to access any address in the range \(0 \ldots P - 1\). Memory addresses \(P \ldots N - 1\) map into the off-chip DRAM, and are accessed by the CPU through the
Figure 5.1. (a) Block diagram of embedded systems with cache and SPM; (b) Division of data address space between SRAM and DRAM.

data cache. A cache hit for an address in the range $P \ldots N - 1$ results in a single-cycle delay, whereas a cache miss, which leads to a block transfer between off-chip and cache memory, results in a delay of 10-20 processor cycles.

### 5.2.2 Power Model

In this chapter, we use *cacti* [145] to determine the power per access for the data cache. The power per access for the scratchpad memory is determined using the model in [7]. Basically, the power, $P(v)$, consumed by accessing a variable $v$ from on-chip memories is expressed as:

$$P(v) = \begin{cases} 
P_S(v), & \text{if the variable } v \text{ is present on the SPM} \\
NCache(v), & \text{otherwise} 
\end{cases} \quad (5.1)$$

where $P_{Cache}(v)$ can be calculated as follows:
\[ P_{\text{Cache}}(v) = \begin{cases} P_{\text{Hit}}(v), \text{if there is a hit} \\ P_{\text{miss}}(v), \text{if there is a miss} \end{cases} \] (5.2)

where \( P_{\text{hit}} \) is the power of a cache hit and \( P_{\text{miss}} \) is the power of a cache miss. Since there is no miss when the variable \( v \) is fetched from the SPM, we can directly have \( P_S \) as the power per access of the SPM.

### 5.2.3 Temperature Model

In this chapter, the temperature variation on the chip is modeled using the RC equivalence [116]. As the energy generated by the memory system is converted into heat, the temperature of the system rises and will reach a state where the amount of heat generated during a period of time becomes equal to the amount of heat being removed by the heat sink. During this state, there will be no temperature variation and this state is generally called the thermal equilibrium. The variation in temperature is modeled using the following equation:

\[ T(t) = P \times R + T_{\text{amb}} - (P \times R + T_{\text{amb}} - T_{\text{init}})e^{-t/RC} \] (5.3)

Here, \( T(t) \) is the temperature at the end of the time interval \( t \), and \( P \) is the average power consumed by the system over this time interval. \( R \) is the thermal resistance, and \( C \) is the thermal capacitance. \( T_{\text{amb}} \) is the ambient temperature, and \( T_{\text{init}} \) is the initial temperature at the beginning of the time interval \( t \). Based on equation 5.3, the steady state temperature associated with an average power dissipation of \( P \) is calculated by:

\[ T_{\text{steady}}(t) = P \times R + T_{\text{amb}} \] (5.4)

By substituting equation 4 into equation 3 and rearranging the terms, we can get:

\[ T(t) = (1 - K(t)) \times T_{\text{steady}}(t) + K(t) \times T_{\text{init}} \] (5.5)

where we use \( K(t) \) to denote the expression \( e^{-t/RC} \).
5.2.4 Problem Statement

The problem to be solved in this chapter is defined as follows:

*Given a loop, an architecture with cache and SPM, the size of SPM, and the temperature constraint, determine a data allocation scheme such that the execution time of the loop is minimized under the constraints of SPM size and temperature.*

5.3 Temperature-Aware Data Allocation

In this section, we propose temperature-aware data allocation techniques to jointly optimize performance and temperature for embedded systems with cache and SPM. Our basic idea is to adaptively adjust data allocation based on the current temperature of on-chip memories. In our method, we divide the loop iterations into several segments, and alternatively allocate the data of each segment to either cache or SPM. In such a way, we balance the workload of cache and SPM by reassigning the workload when the temperature of any of the two memory components exceeds the constraint of temperature.

For solving the problem in which workload can be estimated *a priori*, in Section 5.3.1, we present a nonlinear programming formulation to optimally minimize the total execution time of a loop under the constraints of SPM size and temperature. To solve the problem in which the workload is not known *a priori*, in Section 5.3.2, we propose a temperature-aware adaptive loop scheduling algorithm, TALS, to dynamically allocate data to cache and SPM at run time.

5.3.1 Mathematical Formulation

The nonlinear programming formulation for the data allocation problem is shown in Figure 5.2. The objective of the NLP formulation is to minimize $\sum_{i=1}^{m} E(i)$, which is the total execution time of the loop. In the NLP model, we divide the loop iterations into $m$ partitions, and the expression $\sum_{i=1}^{m} E(i)$ is the summation of the execution time for each partition. The first constraint indicates that the summation of the number of iterations for each partition, $\sum_{i=1}^{m} X(i)$, must equal to the total number of iterations, $N$, for the entire loop. The second
Objective function: Minimize \( \sum_{i=1}^{m} E(i) \)
subject to constraints:
\[
\begin{align*}
\sum_{i=1}^{m} X(i) &= N \\
S_{\text{data}} \cdot X(i) \cdot b(i) &\leq SPM_{\text{Size}}(i = 1, \ldots, m) \\
T_{\text{cache}}(i) &\leq T_{\text{threshold}}(i = 1, \ldots, m) \\
T_{\text{SPM}}(i) &\leq T_{\text{threshold}}(i = 1, \ldots, m)
\end{align*}
\]

Figure 5.2. The NLP formulation.

The constraint denotes that the SPM size constraint must be satisfied when we allocate the data of the \( i \)th partition to SPM. The third and fourth constraints indicate that the temperature for both cache and SPM at the end of each partition cannot exceed the threshold temperature, \( T_{\text{threshold}} \). The variables and constants used in the NLP formulation are summarized in Table 5.1. In the following, we explain the NLP formulation in detail.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>the total number of iterations of the loop</td>
</tr>
<tr>
<td>( m )</td>
<td>the number of partitions for the loop (( 1 \leq m \leq N ))</td>
</tr>
<tr>
<td>( X(i) )</td>
<td>an integer that denotes the number of iterations in the ( i )th partition (( 1 \leq i \leq m ))</td>
</tr>
<tr>
<td>( E(i) )</td>
<td>the execution time of the ( i )th partition</td>
</tr>
<tr>
<td>( b(i) )</td>
<td>set of 0-1 variables such that ( b(i) = 1 ) if the ( i )th partition is assigned to SPM</td>
</tr>
<tr>
<td>( S_{\text{data}} )</td>
<td>the total size of all variables for each iteration of the loop</td>
</tr>
<tr>
<td>( SPM_{\text{Size}} )</td>
<td>the constraint of SPM size</td>
</tr>
<tr>
<td>( T_{\text{cache}}(i) )</td>
<td>the temperature of cache at the end of the ( i )th partition</td>
</tr>
<tr>
<td>( T_{\text{SPM}}(i) )</td>
<td>the temperature of SPM at the end of the ( i )th partition</td>
</tr>
<tr>
<td>( T_{\text{threshold}} )</td>
<td>the temperature constraint of the memory subsystem</td>
</tr>
</tbody>
</table>

Table 5.1. Variables and constants used in the NLP formulation.

In the formulation, \( X(i) \) is an integer variable that denotes the number of iterations in the \( i \)th partition of the loop. \( S_{\text{data}} \) is the total size of all variables for one iteration of the loop, and it is calculated as \( S_{\text{data}} = \sum_{v \in \text{allvars}} size(v) \). The binary variable \( b(i) \) denotes the location of the data of the \( i \)th partition in the memory hierarchy:

\[
b(i) = \begin{cases} 
1, & \text{if the data of the } i \text{th partition is allocated to SPM} \\
0, & \text{otherwise}
\end{cases}
\]
We use $E(i)$ to represent the execution time of the $i$th partition which is assigned to either cache or SPM. More specifically, the total execution time of the loop with $m$ partitions, $\sum_{i=1}^{m} E(i)$, is given by:

$$\sum_{i=1}^{m} E(i) = \sum_{i=1}^{m} ((1 - b(i)) \times t_{\text{cache}}(i) + b(i) \times t_{\text{SPM}}(i)) \quad (5.7)$$

where $t_{\text{cache}}(i)$ and $t_{\text{SPM}}(i)$ represent the execution time of the $i$th partition with cache and SPM, respectively. And, they are calculated by the following equations:

$$t_{\text{cache}}(i) = \sum_{v \in \text{allvars}} Miss(v, i) \times N_{\text{stall}} + SL \times X(i) \quad (5.8)$$

$$t_{\text{SPM}}(i) = C_{\text{startup}} + C_{\text{transfer}} \times S_{\text{data}} \times X(i) + SL \times X(i) \quad (5.9)$$

Here, $SL$ is the schedule length of the loop that is independent of the data allocation scheme, and $N_{\text{stall}}$ denotes the stall cycles caused by a cache miss. The functions, $Miss(v, i)$ and $Hit(v, i)$, return the number of cache misses and hits for the variable $v$ in the $i$th partition of the loop, respectively. In addition, in the $i$th partition of the loop, the number of misses and hits for a variable $v$ is equal to the total number of memory references for this variable, which is expressed by $Miss(v, i) + Hit(v, i) = X(i) \times ref(v)$. Here, the number of memory references for the variable $v$ in each iteration of the loop, $ref(v)$, is a constant and is independent of the memory hierarchy.

Equation 5.9 is obtained based on the cost model [75] that takes into account the data transfer cost between the SPM and off-chip memory. In this model, the cost of communicating $n$ bytes between the SPM and off-chip memory is typically approximated by $C_{\text{startup}} + C_{\text{transfer}} \times n \times (\text{cycles})$, where $C_{\text{startup}}$ is the startup cost and $C_{\text{transfer}}$ is the transfer cost per byte. Thus, in Equation 5.9, $t_{\text{SPM}}(i)$ is calculated using the summation of the startup time, the time spent on transferring $(S_{\text{data}} \times X(i))$ bytes, and the execution time of repeating the schedule for $X(i)$ iterations.
Next, we present how to calculate the temperature for both cache and SPM at the end of the $i$th partition according to the temperature model discussed in Section 5.2.3.

The average power consumption equations for cache and SPM during the execution of the $i$th partition are given by:

$$P_{\text{cache}}(i) = \sum_{v \in \text{allvars}} (P_{\text{miss}} \times \text{Miss}(v, i) + P_{\text{Hit}} \times \text{Hit}(v, i)) / t_{\text{cache}}(i)$$  \hspace{1cm} (5.10)

$$P_{\text{SPM}}(i) = \sum_{v \in \text{allvars}} P_s \times \text{ref}(v) \times X(i) / t_{\text{SPM}}(i)$$  \hspace{1cm} (5.11)

where the functions, $P_{\text{cache}}(i)$ and $P_{\text{SPM}}(i)$, return the average power consumption for executing the $i$th partition with cache and SPM, respectively; $P_{\text{miss}}$ is the power of a cache miss and $P_{\text{Hit}}$ is the power of a cache hit; $P_s$ is the power consumption per access of the scratchpad memory.

Based on Equation 5.4, the steady state temperature of cache and SPM when executing the $i$th partition of the loop can be expressed as:

$$T_{\text{cache,steady}}(i) = P_{\text{cache}}(i) \times (1 - b(i)) \times R_C + T_{\text{amb}}$$  \hspace{1cm} (5.12)

$$T_{\text{SPM,steady}}(i) = P_{\text{SPM}}(i) \times b(i) \times R_S + T_{\text{amb}}$$  \hspace{1cm} (5.13)

where $R_C$ and $R_S$ are the thermal resistance of cache and SPM, respectively; and $T_{\text{amb}}$ is a constant that represents the ambient temperature. From this equation, we can observe that the steady state temperature of each memory component (either cache or SPM) is equal to the ambient temperature if the component is not selected for data allocation. This shows that the temperature of both the cache and SPM can be reduced by alternately allocating data to them.
We substitute the power equations for $P_{\text{cache}}(i)$ and $P_{\text{SPM}}(i)$ from Equations 5.10 and 5.11, respectively, into Equations 5.12 and 5.13. By rearranging the terms, we transform Equations 5.12 and 5.13 into the following forms:

$$T_{\text{cache, steady}}(i) = \sum_{v \in \text{allvars}} \left( (P_{\text{miss}} - P_{\text{Hit}}) \times \text{Miss}(v, i) + P_{\text{Hit}} \times X(i) \times \text{ref}(v) \right) \times \frac{(1 - b(i)) \times R_C + T_{\text{amb}}}{C_{\text{startup}} + C_{\text{transfer}} \times S_{\text{data}} \times X(i) + SL \times X(i)}$$  

(5.14)

$$T_{\text{SPM, steady}}(i) = \sum_{v \in \text{allvars}} \frac{P_S \times \text{ref}(v) \times X(i)}{C_{\text{startup}} + C_{\text{transfer}} \times S_{\text{data}} \times X(i) + SL \times X(i)} \times b(i) \times R_S + T_{\text{amb}}$$  

(5.15)

According to the temperature model in Equation 5.5, the temperature for cache and SPM at the end of the $i$th partition is calculated as follows:

$$T_{\text{cache}}(i) = (1 - K_{C}(i)) \times T_{\text{cache, steady}}(i) + K_{C}(i) \times T_{\text{cache}}(i - 1)$$  

(5.16)

$$T_{\text{SPM}}(i) = (1 - K_{S}(i)) \times T_{\text{SPM, steady}}(i) + K_{S}(i) \times T_{\text{SPM}}(i - 1)$$  

(5.17)

where the functions, $T_{\text{cache}}(i)$ and $T_{\text{SPM}}(i)$, return the temperature at the end of the $i$th partition for cache and SPM, respectively. Based on Equation 5.5, the functions $K_{C}(i)$ and $K_{S}(i)$ are calculated by $K_{C}(i) = e^{-\frac{E(i)}{C_{C} R_C}}$ and $K_{S}(i) = e^{-\frac{E(i)}{C_{S} R_S}}$, respectively. Here, $C_{C}$ is the thermal capacitance of cache, and $C_{S}$ is the thermal capacitance of the SPM. Note that the initial temperature at the beginning of the $i$th partition is the temperature at the end of the $(i - 1)$th partition. In other words, the current temperature is calculated as an accumulated temperature. Basically, before we execute the first partition, we have $T_{\text{cache}}(0) = T_{\text{SPM}}(0) = T_{\text{amb}}$. Thus, we can express $T_{\text{cache}}(i)$ using $T_{\text{cache}}(1), ..., T_{\text{cache}}(i - 1)$ as follows:

$$T_{\text{cache}}(i) = (1 - K_{C}(i)) \times T_{\text{cache, steady}}(i) + K_{C}(i) \times (1 - K_{C}(i - 1)) \times T_{\text{cache, steady}}(i - 1) +$$

$$... + K_{C}(i) \times K_{C}(i - 1) \times ... \times K_{C}(1) \times T_{\text{amb}}$$  

(5.18)

Similarly, $T_{\text{SPM}}(i)$ can be expressed as follows:

$$T_{\text{SPM}}(i) = (1 - K_{S}(i)) \times T_{\text{SPM, steady}}(i) + K_{S}(i) \times (1 - K_{S}(i - 1)) \times T_{\text{SPM, steady}}(i - 1) +$$
In our NLP formulation, we set up a threshold temperature, $T_{\text{threshold}}$. The respective temperature at the end of the $i$th partition ($i = 1, \ldots, m$) for cache and SPM, $T_{\text{cache}}(i)$ and $T_{\text{SPM}}(i)$, cannot exceed the threshold. In such a way, we can minimize the total execution time of the loop, while the temperature and SPM constraints are satisfied.

There are three limitations to the NLP-based approach: (i) The NLP formulation cannot be used to efficiently solve large problem instances (the problem is NP-hard); (ii) It cannot be applied to solve the problem in which the workload is not known a priori (i.e., the number of iterations is not known at compile time); (iii) It may generate irregular partitions of loop iterations, which will introduce difficulties for code generation. Next, we will introduce our $TALS$ algorithm to overcome these limitations.

### 5.3.2 Temperature-Aware Adaptive Loop Scheduling

In this section, we propose a temperature-aware adaptive loop scheduling algorithm, $TALS$, to dynamically allocate data to cache and SPM at runtime. Our basic idea is to adaptively adjust the data allocation scheme based on the current temperature of on-chip memories. In $TALS$, we balance the workload of cache and SPM by reassigning the workload when the temperature of any of the two memory components exceeds the temperature constraint. The proposed $TALS$ algorithm is shown in Algorithm 5.3.1.

The input of the $TALS$ algorithm includes the target loop, the power profile and data size for each iteration of the loop, the total number of iterations, the size of the scratchpad memory, and the threshold temperature. We obtain the number of iterations for the loop at runtime as it typically depends on the input data set. The power profile and data size for each iteration can be obtained by performing a static analysis of the loop at compile-time.

In the $TALS$ algorithm, we use a 0-1 variable “$FLAG$” to control the loop execution behaviors. The loop is executed with its data allocated to SPM if the “$FLAG$” is set as 0, and is executed with cache otherwise. SPM is assigned with a higher priority for data allocation due to its advantages in performance and power. As shown in lines 3 - 12 of Algorithm 5.3.1,
Algorithm 5.3.1 Algorithm TALS.

**Input:** A loop \( L \) with \( \eta \) iterations, the power profile, \( \phi(L) \), and data size, \( S_{data} \), for each iteration of \( L \), the size of SPM, \( SPM_{Size} \), and the threshold temperature, \( T_{th} \).

**Output:** The revised loop \( L \) with its data allocated to cache and SPM.

1: \( FLAG \leftarrow 0; \ EXE\_COUNT \leftarrow 1; \)
2: \( \textbf{while} \ EXE\_COUNT < \eta \ \textbf{do} \)
3: \( \quad \textbf{while} \ FLAG == 0 \ \textbf{do} \)
4: \( \quad \quad \text{Calculate the number of iterations, } \theta, \text{ that is to be allocated to the SPM as:} \)
5: \( \quad \quad \quad \theta = \min\{\text{Iter\_Power\_Temp}(\phi(L), SPM\_Temp, T_{th}), \frac{SPM\_Size}{S_{data}}, \eta - EXE\_COUNT\} \)
6: \( \quad \quad \textbf{for each of the } \theta \text{ iterations do} \)
7: \( \quad \quad \quad \text{Execute the current iteration with SPM; } EXE\_COUNT \leftarrow EXE\_COUNT + 1; \)
8: \( \quad \quad \quad SPM\_Temp \leftarrow \text{the current temperature of SPM;} \)
9: \( \quad \quad \quad \textbf{if} (SPM\_Temp \geq T_{th}) \text{ then } FLAG \leftarrow 1; \text{ break;} \)
10: \( \quad \quad \textbf{end for} \)
11: \( \quad \text{MEM\_WRITEBACK\_FROM\_SPM}(\theta, L); \)
12: \( \quad \textbf{end while} \)
13: \( \textbf{while} \ FLAG == 1 \ \textbf{do} \)
14: \( \quad \text{Execute the current iteration with cache; } EXE\_COUNT \leftarrow EXE\_COUNT + 1; \)
15: \( \quad Cache\_Temp \leftarrow \text{the current temperature of cache;} \)
16: \( \quad \textbf{if} (Cache\_Temp \geq T_{th}) \text{ then } FLAG \leftarrow 0; \text{ break;} \)
17: \( \quad \textbf{end while} \)
18: \( \textbf{end while} \)

we begin the loop execution by grouping \( \theta \) iterations and executing them with SPM. The number of iterations, \( \theta \), to be allocated to the SPM is calculated by:

\[
\theta = \min\{\text{Iter\_Power\_Temp}(\phi(L), SPM\_Temp, T_{th}), \frac{SPM\_Size}{S_{data}}, \eta - EXE\_COUNT\}
\]

The above calculation takes both the temperature and SPM constraints into consideration for prediction. It consists of three parts. In the first part, based on the thermal model discussed in Section 5.2.3, we estimate the maximum number of iterations that can be allocated to the SPM. The average power consumption, \( \phi(L) \), for each iteration of the loop, the current temperature of the SPM, \( SPM\_Temp \), and the threshold temperature, \( T_{th} \), are used in the calculation. The second part is to calculate how many iterations can be put into the SPM under the SPM size constraint, and the third part is to obtain how many iterations of the loop
is left for execution.

With the estimated number, we first copy the data that will be used in the following $\theta$ iterations from main memory to SPM. Then, we process these iterations one by one. At the end of each loop iteration, we get the current temperature of SPM from the on-chip temperature sensors, and compare it with the temperature constraint in order to determine whether or not we should adjust the data allocation scheme. In our implementation, a thermal simulation is performed to measure the transient temperature of each memory component at runtime. If the temperature of SPM exceeds the threshold temperature, we reassign the workload to cache. Finally, we update the main memory with the data presented in SPM to ensure that the data is consistent.

Similarly, we process the loop iterations with cache as shown in lines 13 - 17 of Algorithm 5.3.1. Note that our TALS algorithm fully utilizes each memory component (either cache or SPM) before it reaches the threshold temperature. In other words, the behavior of the program only changes when the temperature constraint is violated. As a result, the performance can be improved by executing the loop iterations with SPM as much as possible. At the same time, a low overhead will be incurred due to a small number of context switches between cache and SPM. On the other hand, our algorithm divides the loop iterations into several segments, and assigns each segment to one memory component. Thus, during the execution of the loop, either cache or SPM is idle for some iterations, and it can be cooled down to reduce its temperature. Also, as we always check the temperature at the end of each loop iteration, our algorithm can guarantee that the maximum time interval spent over the threshold temperature for each memory component (either cache or SPM) is at most one iteration.

Next, we present the revisions on the program of the loop in order to implement the TALS algorithm. The flow for the code of the revised loop is illustrated in Figure 5.3.

As shown in Figure 5.3, the original program of the loop is divided into two code blocks, cache block and SPM block. At the end of each iteration for both blocks, we add instructions to obtain the temperature and compare it with the threshold temperature. In par-
In particular, when we enter the SPM code block, we calculate the number of iterations to be allocated to the SPM. We use two functions, `MEM_COPY()` and `MEM_WRITEBACK()`, to transfer data between the scratchpad memory and off-chip memory. The former function is used to copy data from the main memory to the SPM, and the latter one is adopted to copy data from the SPM to the main memory. In addition, we use the `JUMP` instruction to achieve the transition between two code blocks.

**5.4 Experiments**

In this section, we first discuss our implementation and simulation environment in section 5.4.1. Then, we present the experimental results in section 5.4.2.
### 5.4.1 Experimental Setup

We have implemented both the NLP-based approach and TALS algorithm into the Trimaran [47] compiler for code generation. To solve instances of our NLP formulation, we use LINGO 8.0 [125] to get the optimal partitions of loop iterations, and transform the loop based on these partitions using Trimaran. The Trimaran compiler includes two compilation modules, Impact [21] and Elcor, as well as a cycle-accurate VLIW simulator. The Elcor module implements machine-dependent optimizations such as instruction scheduling and register allocation. Our algorithms are added in the instruction scheduling part of the Elcor.

![Diagram](image)

**Figure 5.4.** The implementation and simulation framework (the shaded blocks are the modified parts).

Figure 5.4 depicts our implementation/simulation platform. The simulation framework consists of four parts: the cycle-accurate VLIW simulator of Trimaran [47], the power models for both cache and SPM [7, 145], the SPM cost model [75], and the thermal simulator, HotSpot [116]. As shown in Figure 5.4, at runtime, the power data is read by HotSpot and the simulated temperature data is returned to the VLIW simulator when we finish executing each loop iteration. The feedback is used to determine the data allocation scheme for the following iterations of the loop.

In the experiments, we configure the simulator to simulate the floor plan as shown in Figure 5.5. This floor plan is taken from the TI’s data sheet [48] as an approximation of the on-chip SPM. In the simulation, we model a 65 nm processor with a supply voltage
$V_{dd}$ of 1.2V, and a base frequency of 4 GHz. According to the SPM model in [75], the cost of communicating $n$ bytes between the SPM and off-chip memory is approximated by $C_s + C_t \times n$ in cycles, where $C_s$ is the startup cost and $C_t$ is the cost per byte transfer. The values of the two parameters are $C_s = 20$, and $C_t = 1$. A detailed configurations of the Trimaran simulator is shown in Table 5.2.

In order to track the thermal behavior of the on-chip memory system, we incorporate the power models in [7, 145] into the Trimaran simulator to capture the power per access of memory operations. With the provided power data, we use HotSpot [116], an architecture-level thermal model, to measure the temperature of the on-chip cache and SPM. In the experiments, the size of the scratchpad memory is configured as 4 KB, as shown in Table 5.2, and we set up $75 \degree C$ to be the threshold temperature for both cache and SPM. The parameters for HotSpot are shown in Table 5.3.

To evaluate the effectiveness of our algorithm, we use a set of 21 benchmarks from DSPstone [177] and MiBench [49]. We test both the fixed-point and the floating-point versions of benchmarks from DSPstone [177]. (Two benchmarks, fft.stage_scaled and fft.input_scaled only have the fixed-point version.) As several benchmarks from DSP-
<table>
<thead>
<tr>
<th>Processor core size</th>
<th>20.2 $mm^2$ (4.5mm × 4.5mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPM</td>
<td>4KB</td>
</tr>
<tr>
<td>The startup cost $C_s$</td>
<td>20 cycle</td>
</tr>
<tr>
<td>The transfer cost per byte $C_t$</td>
<td>1 cycle</td>
</tr>
<tr>
<td>SPM access time</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>32 KB, 2 cycles</td>
</tr>
<tr>
<td>L1 instruction cache</td>
<td>32 KB, 2 cycles</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64 KB, 16 cycles</td>
</tr>
<tr>
<td>Off-chip Memory access time</td>
<td>First access: 100 cycles</td>
</tr>
<tr>
<td></td>
<td>Subsequently: 6 cycles</td>
</tr>
</tbody>
</table>

Table 5.2. The configurations for the VLIW simulator of Trimaran.

<table>
<thead>
<tr>
<th>Ambient temperature</th>
<th>55 °C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Die</td>
<td>0.5mm thick, 7.52mm*7.52mm</td>
</tr>
<tr>
<td></td>
<td>1mm thick, 1cm*1cm</td>
</tr>
<tr>
<td>Heat sink</td>
<td>7mm thick, 6cm*6cm</td>
</tr>
</tbody>
</table>

Table 5.3. The parameters for HotSpot.

stone [177] (including dot_product, IIR, fir, fir2dim, lms, matrix 1*3, $n_{\text{complex updates}}$, and $n_{\text{real updates}}$) have a small number of iterations, in the experiments, we reset the number of iterations for each loop of these benchmarks as 1000 unless otherwise specified. The benchmarks from Mibench [49] are evaluated using the data sets that come with their source files.

5.4.2 Results and Discussion

In this section, we first compare our NLP-Based approach with TALS in terms of execution time and time performance with various numbers of loop iterations. Then, we present the results generated by our TALS algorithm in terms of reducing peak temperature, improving
performance, and expanding code size, and compare them with the baseline scheme of the Trimaran compiler [47].

1) NLP-Based Approach vs. TALS

We use LINGO 8.0 [125] to solve our proposed NLP formulation for achieving optimal loop performance under the constraints of SPM size and temperature. Although the NLP-based approach can obtain an optimal solution for the data allocation problem, it is an NP-hard problem to solve the NLP model [43]. Therefore, the NLP model may take a very long time to get results even when the given number of loop iterations is not very large. In addition, the optimal solutions may produce irregular partitions of loop iterations, and this will lead to a big expansion in code size. In this section, we use the _lms_ benchmark and perform experiments to compare execution time and time performance for the NLP-based approach and _TALS_ algorithm. The experimental results are shown in Table 5.4, in which the number of iterations is set as 100, 150, 200, 250, 300, 350, and 400, respectively.

<table>
<thead>
<tr>
<th>Number of Iterations</th>
<th>Our NLP-based Approach</th>
<th>Our <em>TALS</em> Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Execution Time (s)</td>
<td>Time Performance (cycles)</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>2764</td>
</tr>
<tr>
<td>150</td>
<td>22</td>
<td>3862</td>
</tr>
<tr>
<td>200</td>
<td>101</td>
<td>4653</td>
</tr>
<tr>
<td>250</td>
<td>1308</td>
<td>5630</td>
</tr>
<tr>
<td>300</td>
<td>4910</td>
<td>6498</td>
</tr>
<tr>
<td>350</td>
<td>72496</td>
<td>7568</td>
</tr>
<tr>
<td>400</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.4. Comparison of execution time and time performance for our NLP-based approach and _TALS_ algorithm.

Through the experimental results from Table 5.4, we can observe that the NLP-based approach takes much more time to get results compared with the _TALS_ algorithm. When
the number of iterations for the input loop is 400, it cannot get the result even in five days. The results also show that our TALS algorithm achieves near-optimal results compared with the optimal results obtained by the NLP-based approach. In the following, we present the results of our TALS algorithm which can work effectively in practice.

2) TALS vs. Baseline

In Tables 5.5 to 5.7, we present the results for all benchmarks. In these tables, the column “Base” refers to the results generated by the baseline scheme of Trimaran [47], and the column “TALS” represents the results obtained by our TALS technique. Column “IMP(%)” represents the improvement obtained by our TALS algorithm compared to the baseline scheme of Trimaran. In the following, we present and analyze the results in terms of reducing peak temperature, improving performance, and expanding code size, respectively.

Reducing Peak Temperature. The reduction in peak temperature of the memory subsystem including cache and SPM is shown in Table 5.5. In this table, column “REDUCTION (°C)” represents the peak temperature reduction obtained by our TALS algorithm compared to the baseline scheme of Trimaran. For the benchmarks from DSPstone and Mibench, our TALS algorithm achieves an average reduction in peak temperature of 4.8 degrees and 9.8 degrees, respectively.

From the results, we can observe that (1) our technique can effectively reduce the peak temperature of the memory subsystem under the dual constraints of SPM and temperature; and (2) our technique performs better for the data-intensive benchmarks from Mibench than those from DSPStone. The reasons for these are as follows.

First, our TALS technique reduces the peak temperature by dynamically adjusting the workload distribution of on-chip memories. In such a way, either cache or SPM can be cooled down for some loop iterations. Second, at runtime, our algorithm checks whether or not the threshold temperature is exceeded at the end of each loop iteration. Thus, the temperature constraint can be strictly satisfied. In addition, as our algorithm guarantees that the time spent over the threshold temperature is at most one iteration, the reliability
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Peak Temperature °C</th>
<th>Base (°C) [47]</th>
<th>TALS (°C)</th>
<th>REDUCTION (°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DSPstone</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolution</td>
<td>65.3</td>
<td>62.1</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>Convolution-fix</td>
<td>62.4</td>
<td>59.1</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>dot_product</td>
<td>65.1</td>
<td>60.7</td>
<td>4.4</td>
<td></td>
</tr>
<tr>
<td>dot_product-fix</td>
<td>66.2</td>
<td>61.7</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>IIR</td>
<td>68.5</td>
<td>63.7</td>
<td>4.8</td>
<td></td>
</tr>
<tr>
<td>IIR-fix</td>
<td>68.2</td>
<td>62.4</td>
<td>5.8</td>
<td></td>
</tr>
<tr>
<td>fir</td>
<td>67.6</td>
<td>63.4</td>
<td>4.2</td>
<td></td>
</tr>
<tr>
<td>fir-fix</td>
<td>66.7</td>
<td>62.9</td>
<td>3.8</td>
<td></td>
</tr>
<tr>
<td>fir2dim</td>
<td>64.9</td>
<td>61.3</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>fir2dim-fix</td>
<td>64.1</td>
<td>59.9</td>
<td>4.2</td>
<td></td>
</tr>
<tr>
<td>lms</td>
<td>65.3</td>
<td>62.4</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>lms-fix</td>
<td>65.5</td>
<td>62.5</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>matrix 1x3</td>
<td>64.4</td>
<td>61.2</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>matrix 1x3-fix</td>
<td>63.1</td>
<td>61.8</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>matrix</td>
<td>63.7</td>
<td>58.9</td>
<td>4.8</td>
<td></td>
</tr>
<tr>
<td>matrix-fix</td>
<td>63.5</td>
<td>57.7</td>
<td>5.8</td>
<td></td>
</tr>
<tr>
<td>matrix 2</td>
<td>62.9</td>
<td>57.3</td>
<td>5.6</td>
<td></td>
</tr>
<tr>
<td>matrix 2-fix</td>
<td>61.3</td>
<td>55.1</td>
<td>6.2</td>
<td></td>
</tr>
<tr>
<td>n_complex_updates</td>
<td>69.3</td>
<td>63.8</td>
<td>5.5</td>
<td></td>
</tr>
<tr>
<td>n_complex_updates-fix</td>
<td>68.5</td>
<td>63.2</td>
<td>5.3</td>
<td></td>
</tr>
<tr>
<td>n_complex_updates</td>
<td>66.6</td>
<td>62.7</td>
<td>3.9</td>
<td></td>
</tr>
<tr>
<td>n_complex_updates-fix</td>
<td>65.4</td>
<td>61.8</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>fft-stageScaled</td>
<td>79.5</td>
<td>69.1</td>
<td>10.4</td>
<td></td>
</tr>
<tr>
<td>fft-input_scaled</td>
<td>79.1</td>
<td>68.8</td>
<td>10.3</td>
<td></td>
</tr>
<tr>
<td><strong>Average Reduction (°C)</strong></td>
<td></td>
<td></td>
<td>4.8</td>
<td></td>
</tr>
<tr>
<td><strong>Mibench</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfencrypt</td>
<td>82.6</td>
<td>75.4</td>
<td>7.2</td>
<td></td>
</tr>
<tr>
<td>bfdecrypt</td>
<td>82.6</td>
<td>75.4</td>
<td>7.2</td>
<td></td>
</tr>
<tr>
<td>cjpeg</td>
<td>93.1</td>
<td>76.2</td>
<td>16.9</td>
<td></td>
</tr>
<tr>
<td>djpeg</td>
<td>91.5</td>
<td>75.8</td>
<td>15.7</td>
<td></td>
</tr>
<tr>
<td>gsmencode</td>
<td>83.2</td>
<td>75.6</td>
<td>7.6</td>
<td></td>
</tr>
<tr>
<td>gsmdecode</td>
<td>82.5</td>
<td>75.3</td>
<td>7.2</td>
<td></td>
</tr>
<tr>
<td>rawcaudio</td>
<td>82.8</td>
<td>74.6</td>
<td>8.2</td>
<td></td>
</tr>
<tr>
<td>rawdaudio</td>
<td>81.9</td>
<td>74.2</td>
<td>8.7</td>
<td></td>
</tr>
<tr>
<td><strong>Average Reduction (°C)</strong></td>
<td></td>
<td></td>
<td>9.8</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5. The reduction in peak temperature for the benchmarks.
of embedded systems can be greatly improved. Third, as shown in the column "Base" of Table 5.5, the previous peak temperature is with cache as there is no SPM allocation support in the baseline scheme of Trimaran. Our technique fully utilizes the SPM which consumes less power than cache and accordingly generates less heat.

For the loop kernels extracted from DSPStone, the original temperature is low due to the small number of memory accesses. For these benchmarks with a small data size, our technique can almost allocate the data of the entire loop to SPM before its temperature reaches the threshold. As a result, the temperature decreases. On the other hand, the original peak temperature for the data-intensive benchmarks from Mibench is high. This implicitly indicates that many hot spots may exist in these benchmarks. For this case, our TALS algorithm can divide the workload into several segments, and execute each segment with either cache or SPM. Note that some of the results generated by our TALS algorithm for the benchmarks from Mibench exceed the threshold temperature. This can be explained from two aspects. First, our technique only checks the temperature at the end of each loop iteration. Peak temperature is likely to appear between two checks. On the other hand, these benchmarks are real-world applications with a number of basic blocks besides loops. Before executing the current loop, the on-chip temperature will increase due to the power generated by the previous blocks. As the temperature accumulates with time, the reported peak temperature possibly exceeds the threshold.

**Improving Performance.** The overall improvement in performance for benchmarks from DSPstone and MiBench is shown in Table 5.6. In this table, the performance of each benchmark is evaluated by the number of execution cycles. On average, for DSPstone and Mibench, our TALS technique contributes to a respective improvement in performance of 9.55% and 3.86%.

The reasons for the improvement are as follows. First, our technique focuses on improving the performance of loops, which are normally the most time-consuming part of embedded applications. Second, different from the baseline compilation scheme of Trimaran, SPM is of the highest priority for data allocation. Our TALS technique aggressively assigns...
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base [47]</th>
<th>TALS</th>
<th>IMP(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DSPstone</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolution</td>
<td>11,370</td>
<td>10,377</td>
<td>8.73</td>
</tr>
<tr>
<td>Convolution-fix</td>
<td>7,126</td>
<td>6,536</td>
<td>8.28</td>
</tr>
<tr>
<td>dot_product</td>
<td>7,462</td>
<td>6,939</td>
<td>7.01</td>
</tr>
<tr>
<td>dot_product-fix</td>
<td>20,442</td>
<td>19,015</td>
<td>6.98</td>
</tr>
<tr>
<td>IIR</td>
<td>56,512</td>
<td>49,668</td>
<td>12.11</td>
</tr>
<tr>
<td>IIR-fix</td>
<td>46,929</td>
<td>41,185</td>
<td>12.24</td>
</tr>
<tr>
<td>fir</td>
<td>38,436</td>
<td>35,803</td>
<td>6.85</td>
</tr>
<tr>
<td>fir-fix</td>
<td>42,572</td>
<td>40,035</td>
<td>5.96</td>
</tr>
<tr>
<td>fir2dim</td>
<td>21,377</td>
<td>20,725</td>
<td>3.05</td>
</tr>
<tr>
<td>fir2dim-fix</td>
<td>20,324</td>
<td>19,741</td>
<td>2.87</td>
</tr>
<tr>
<td>lms</td>
<td>23,702</td>
<td>20,073</td>
<td>15.31</td>
</tr>
<tr>
<td>lms-fix</td>
<td>22,842</td>
<td>19,345</td>
<td>15.31</td>
</tr>
<tr>
<td>matrix 1x3</td>
<td>14,076</td>
<td>12,491</td>
<td>11.26</td>
</tr>
<tr>
<td>matrix 1x3-fix</td>
<td>12,654</td>
<td>11,028</td>
<td>12.85</td>
</tr>
<tr>
<td>matrix</td>
<td>10,611</td>
<td>9,549</td>
<td>8.48</td>
</tr>
<tr>
<td>matrix-fix</td>
<td>13,384</td>
<td>12,028</td>
<td>10.50</td>
</tr>
<tr>
<td>matrix 2</td>
<td>9,770</td>
<td>7,978</td>
<td>12.67</td>
</tr>
<tr>
<td>matrix 2-fix</td>
<td>9,672</td>
<td>8,035</td>
<td>11.54</td>
</tr>
<tr>
<td>n_complex_updates</td>
<td>58,958</td>
<td>53,463</td>
<td>9.32</td>
</tr>
<tr>
<td>n_complex_updates-fix</td>
<td>51,168</td>
<td>46,338</td>
<td>9.44</td>
</tr>
<tr>
<td>n_real_updates</td>
<td>20,454</td>
<td>18,065</td>
<td>11.68</td>
</tr>
<tr>
<td>n_real_updates-fix</td>
<td>19,867</td>
<td>18,732</td>
<td>11.50</td>
</tr>
<tr>
<td>fft_inputScaled</td>
<td>315,305</td>
<td>276,775</td>
<td>10.03</td>
</tr>
<tr>
<td>fft_input_scaled</td>
<td>241,299</td>
<td>212,657</td>
<td>9.94</td>
</tr>
<tr>
<td><strong>Average Improvement (%)</strong></td>
<td></td>
<td></td>
<td>9.55</td>
</tr>
<tr>
<td><strong>Mibench</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfencrypt</td>
<td>1.13M</td>
<td>1.08M</td>
<td>4.31</td>
</tr>
<tr>
<td>bfdecrypt</td>
<td>1.13M</td>
<td>1.08M</td>
<td>4.31</td>
</tr>
<tr>
<td>cjjpeg</td>
<td>20.61M</td>
<td>19.75M</td>
<td>4.19</td>
</tr>
<tr>
<td>djpeg</td>
<td>18.79M</td>
<td>18.00M</td>
<td>4.22</td>
</tr>
<tr>
<td>gsmencode</td>
<td>371.80M</td>
<td>359.68M</td>
<td>3.26</td>
</tr>
<tr>
<td>gsmdecode</td>
<td>546.47M</td>
<td>527.68M</td>
<td>3.44</td>
</tr>
<tr>
<td>rawaudio</td>
<td>711.00M</td>
<td>683.63M</td>
<td>3.85</td>
</tr>
<tr>
<td>rawaudio</td>
<td>585.31M</td>
<td>565.99M</td>
<td>3.33</td>
</tr>
<tr>
<td><strong>Average Improvement (%)</strong></td>
<td></td>
<td></td>
<td>3.86</td>
</tr>
</tbody>
</table>

Table 5.6. The improvement in performance for the benchmarks.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Executable File Size (MB)</th>
<th>Base [47]</th>
<th>TALS</th>
<th>EXP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DSPstone</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolution</td>
<td>0.58</td>
<td>0.95</td>
<td></td>
<td>63.79</td>
</tr>
<tr>
<td>Convolution-fix</td>
<td>0.53</td>
<td>0.89</td>
<td></td>
<td>67.92</td>
</tr>
<tr>
<td>dot_product</td>
<td>0.55</td>
<td>0.87</td>
<td></td>
<td>58.18</td>
</tr>
<tr>
<td>dot_product-fix</td>
<td>0.51</td>
<td>0.83</td>
<td></td>
<td>62.74</td>
</tr>
<tr>
<td>IIR</td>
<td>0.58</td>
<td>0.93</td>
<td></td>
<td>60.34</td>
</tr>
<tr>
<td>IIR-fix</td>
<td>0.56</td>
<td>0.91</td>
<td></td>
<td>62.50</td>
</tr>
<tr>
<td>fir</td>
<td>0.56</td>
<td>0.90</td>
<td></td>
<td>60.71</td>
</tr>
<tr>
<td>fir-fix</td>
<td>0.54</td>
<td>0.87</td>
<td></td>
<td>61.11</td>
</tr>
<tr>
<td>fir2dim</td>
<td>0.81</td>
<td>1.20</td>
<td></td>
<td>48.15</td>
</tr>
<tr>
<td>fir2dim-fix</td>
<td>0.78</td>
<td>1.15</td>
<td></td>
<td>47.44</td>
</tr>
<tr>
<td>lms</td>
<td>0.58</td>
<td>0.92</td>
<td></td>
<td>58.62</td>
</tr>
<tr>
<td>lms-fix</td>
<td>0.57</td>
<td>0.91</td>
<td></td>
<td>59.65</td>
</tr>
<tr>
<td>matrix 1x3</td>
<td>0.67</td>
<td>1.04</td>
<td></td>
<td>55.22</td>
</tr>
<tr>
<td>matrix 1x3-fix</td>
<td>0.59</td>
<td>0.94</td>
<td></td>
<td>59.32</td>
</tr>
<tr>
<td>matrix</td>
<td>0.69</td>
<td>1.07</td>
<td></td>
<td>55.07</td>
</tr>
<tr>
<td>matrix-fix</td>
<td>0.67</td>
<td>1.05</td>
<td></td>
<td>56.72</td>
</tr>
<tr>
<td>matrix 2</td>
<td>0.70</td>
<td>1.09</td>
<td></td>
<td>55.71</td>
</tr>
<tr>
<td>matrix 2-fix</td>
<td>0.68</td>
<td>1.06</td>
<td></td>
<td>55.88</td>
</tr>
<tr>
<td>n_complex_updates</td>
<td>0.59</td>
<td>0.87</td>
<td></td>
<td>47.46</td>
</tr>
<tr>
<td>n_complex_updates-fix</td>
<td>0.58</td>
<td>0.84</td>
<td></td>
<td>44.83</td>
</tr>
<tr>
<td>n_real_updates</td>
<td>0.56</td>
<td>0.78</td>
<td></td>
<td>39.28</td>
</tr>
<tr>
<td>n_real_updates-fix</td>
<td>0.54</td>
<td>0.75</td>
<td></td>
<td>38.89</td>
</tr>
<tr>
<td>fft_stage_scaled</td>
<td>1.32</td>
<td>1.87</td>
<td></td>
<td>41.67</td>
</tr>
<tr>
<td>fft_input_scaled</td>
<td>1.30</td>
<td>1.84</td>
<td></td>
<td>41.54</td>
</tr>
<tr>
<td><strong>Average Expansion (%)</strong></td>
<td></td>
<td></td>
<td></td>
<td>52.19</td>
</tr>
<tr>
<td><strong>Mibench</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfencrypt</td>
<td>6.14</td>
<td>6.54</td>
<td></td>
<td>6.51</td>
</tr>
<tr>
<td>bfdecrypt</td>
<td>6.14</td>
<td>6.54</td>
<td></td>
<td>6.51</td>
</tr>
<tr>
<td>cjpeg</td>
<td>38.57</td>
<td>42.38</td>
<td></td>
<td>9.88</td>
</tr>
<tr>
<td>djpeg</td>
<td>38.61</td>
<td>42.54</td>
<td></td>
<td>10.18</td>
</tr>
<tr>
<td>gsmencode</td>
<td>15.99</td>
<td>16.79</td>
<td></td>
<td>5.01</td>
</tr>
<tr>
<td>gsmdecode</td>
<td>15.98</td>
<td>16.82</td>
<td></td>
<td>5.26</td>
</tr>
<tr>
<td>rawcaudio</td>
<td>1.23</td>
<td>1.32</td>
<td></td>
<td>7.32</td>
</tr>
<tr>
<td>rawaudio</td>
<td>1.29</td>
<td>1.38</td>
<td></td>
<td>6.98</td>
</tr>
<tr>
<td><strong>Average Expansion (%)</strong></td>
<td></td>
<td></td>
<td></td>
<td>7.21</td>
</tr>
</tbody>
</table>

Table 5.7. The size of the executable files.
the workload to SPM before it reaches the threshold temperature. In particular, for the loop kernels from DSPStone that are small in terms of data size, our technique can allocate the data of almost the entire loop to SPM before its temperature reaches the threshold. As a result, when the loop is executed with SPM, the cache misses are avoided, and the performance can be improved due to the guaranteed fast memory accesses to SPM. Third, we limit the number of context switches between SPM and cache by taking full advantage of each memory component (either cache or SPM) before it reaches the threshold temperature. In other words, the behavior of the program only changes when the temperature constraint is violated. In addition, in our implementation, we reduce the number of additional instructions that are added into the loop, such as the instructions for reading temperature from the thermal simulator. Thus, a low overhead is invoked. This also contributes to the improvement in performance.

**Expanding Code Size.** To explore the hybrid memory architecture, with our technique, the original cache-based code block of a loop is divided into two blocks. In addition, the instructions for reading the temperature value from the thermal simulator will be inserted into the original code. As a result, our technique leads to an expansion in code size. The expansion in code size for the benchmarks from DSPStone and MiBench is shown in Table 5.7.

In Table 5.7, we show the size of the executable files that is generated by the baseline scheme of Trimaran and our TALS algorithm. Through the results from Table 5.7, we can observe that the original size for each benchmark from DSPStone is very small. Compared with the baseline scheme, for the benchmarks from DSPStone, our technique leads to an average expansion in code size of 52.19%, which is suitable for embedded systems. Different from DSPStone, the benchmarks from MiBench are big applications with a number of basic blocks, and loops are only a small part of the programs. Therefore, the expansion ratio for these benchmarks is relatively small. On average, for the benchmarks from MiBench, our technique leads to an expansion of 7.21%.
5.5 Summary

In this chapter, we proposed to explore both cache and scratchpad memory (SPM) to jointly optimize performance and temperature. We first mathematically presented a nonlinear programming (NLP) formulation. Our NLP optimally minimizes the total execution time of a loop under the constraints of SPM size and temperature. We then developed an adaptive data allocation technique, which adjusts the workload distribution of on-chip memories based on the current on-chip temperature at runtime. We implemented our algorithms into the Tri-Maran compiler. The experimental results demonstrate that our technique can effectively achieve joint performance and temperature optimization for embedded systems with cache and SPM.
CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Embedded systems are application specific, and have strict timing and power constraints. As embedded systems become more complex and incorporate more functionality, these constraints are getting tighter. Therefore, designing high performance and low power embedded systems with various constraints and limited resources has become an important research problem. In our research, we investigated three challenging issues, including memory optimization, leakage power optimization, and temperature-aware optimization. Many promising results have been obtained in these fields, and these results have tremendously improved current techniques. The results show that, when achieving high performance is the primary goal, our LSMAR and REALM technique should be performed for reducing memory accesses. For battery-driven applications in which low power is highly needed, our LMS technique should be employed for reducing leakage power consumption. With the widening deployment of high-reliable embedded systems, such as the systems used in military and medical domains, our TALS technique should be applied as the primary optimization function for improving the reliability of embedded systems.

Our research focuses on understanding fundamental properties and developing compiler-assisted models, methodologies, and algorithms for high performance and low power embedded systems. In summary, our contributions are:

1. For high performance optimization on embedded systems, we found that the number of memory accesses is the most important factor that influences the time performance.
We studied and addressed the problem of reducing memory accesses for DSP applications, which is vital both for improving performance and reducing memory power.

- For DSP applications with loops, we proposed a technique, REALM (REdundAnt Load Exploration & Migration), to reduce hidden redundant memory accesses within the loop nests. Different from previous work, our technique can optimize both array-based and pointer-based code, which is very important as pointer arithmetic is widely used in DSP applications.

- We proposed a novel memory access graph model to describe the loop-carried data dependencies of memory operations. Based on this graph, our REALM technique can achieve an optimal solution in which all hidden redundant memory accesses can be explored and eliminated.

- We implemented the REALM technique into IMPACT [21], which is one of the most popular open-source compilers with full support from the compiler research community.

To the best of our knowledge, this is the first work to implement the memory access reduction with loop-carried data reuse in real world compilers.

2. To solve the memory access reduction problem under register constraints, we proposed a two-phase loop scheduling algorithm, LSMAR (Loop Scheduling with Memory Access Reduction). In the first phase, we used the REALM technique to optimally explore all hidden redundant load operations; and in the second phase, we iteratively performed loop scheduling to eliminate these operations by exploiting register operations.

- We solved the register allocation problem by building up a register-matching graph and finding a fixed-length simple path between two specified vertices in this graph. Based on the register allocation results, we generated a schedule in such a way that all constraints can be fulfilled and the total number of memory accesses can be reduced.
We implemented our LSMAR algorithm into Trimaran [47], which is the most popular state-of-the-art compiler infrastructure for embedded VLIW architectures. The experimental results show that our algorithm can significantly reduce memory accesses and improve performance with little expansion in code size.

3. For low leakage optimization, we studied the fundamental properties of leakage power, and identified the relationship between leakage savings and schedule length. We proposed a novel leakage-aware modulo scheduling technique to reduce the leakage power consumption of an application executed on an embedded VLIW processor. Different from prior studies, our technique combines leakage-aware optimization with software pipelining. In this way, we can maximize the idle time of a functional unit to achieve leakage savings. The experimental results show that our technique can greatly reduce leakage power for VLIW processors with less performance overhead compared with previous work.

4. For temperature-aware optimization, we found that on-chip memories are the temperature hot spots. We addressed both performance and temperature issues for the hybrid embedded memory architecture which contains both cache and scratchpad memory (SPM).

- We proposed both static and dynamic temperature-aware strategies for allocating data for embedded systems with on-chip cache and SPM. For problems in which the workload can be estimated accurately \textit{a priori}, we presented a mathematical formulation to optimally minimize the execution time of a loop under the constraints of SPM size and temperature. To solve problems in which the workload is not known \textit{a priori}, we proposed an adaptive temperature-aware data allocation algorithm, TALS, to dynamically adjust the workload of on-chip memories based on the current temperature at run time.

- We experimented with our proposed techniques using a set of benchmarks from DSPstone [177] and Mibench [49], which are the most popular embedded benchmark suites consisting of extensive digital filters and real-world embedded appli-
cations. The results demonstrate that our techniques can effectively achieve joint performance and temperature optimization for embedded systems with cache and SPM.

To the best of our knowledge, this is the first work to explore the hybrid architecture with both cache and SPM to jointly optimize performance and temperature.

6.2 Future Work

Our research in this thesis mainly focused on developing compiler-assisted high performance and low power optimization techniques for embedded systems. In the future, we plan to investigate high performance and low power issues for clustered VLIW and multi-core architectures.

Clustered VLIW architectures have been widely used in modern embedded processors such as HP/ST’s Lx, TI’s TMS320C6x, Analog’s Tigersharc, and Equator’s MAP1000. These systems consist of a number of clusters, and each cluster has its own register file and a set of local functional units. Local communications (communications inside a cluster) are fast, while global communications (inter-cluster communications) among the clusters are slow. Inter-cluster communications are used to propagate register values when the producer and the consumer of the value are assigned to different clusters. Hence, we need to develop efficient task assignment and scheduling methods in order to minimize global communications and balance the workload among the clusters. In addition, the distributed register files in a clustered VLIW architecture make our memory access reduction problem more difficult, as we have to take into account the cost of the inter-cluster communication. In the future, we will address these issues, and investigate how to extend our research results to the clustered VLIW architectures.

Multi-core architectures have become the prevalent design style in embedded systems as they permit higher performance than uniprocessor architectures. Multi-core embedded systems impose a lot of challenges for high performance and low power optimizations.
One such challenge is to reduce the power consumption in order to extend the battery life for mobile devices with multiple cores. Another challenge concerns the management of the tremendous heat generated from the multiple cores. As a large number of cores have been packed into a single die, and the escalating power density and temperatures have made multi-core systems much more vulnerable to failures that single-core systems. To reduce the temperature of multi-core systems, we need to consider the variations in spatial temperature, since the heat generated by an active core will affect its neighboring elements (other cores or the heat sink). Furthermore, as multi-core systems often have a complex memory hierarchy, efficient memory management and scheduling methods are needed to minimize the memory requirements of embedded applications. In the future, we will study these problems, and develop new methodologies and techniques for multi-core embedded systems.
REFERENCES


dsp synthesis. In *Proceedings of International Symposium on Circuits and Systems*

In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language

instruction scheduling for low power. *ACM Trans. on Design Automation of Electronic


[72] R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. In *Pro-
ceedings of the 8th International Symposium on System Synthesis (ISSS’95)*, pages

[73] S. Leventhal, L. Yuan, N. K. Bambha, S. S. Bhattacharyya, and G. Qu. DSP address
optimization using evolutionary algorithms. In *Proceedings of the 2005 Workshop on

simulation for run-time temperature tracking and management. In *Proceedings of the

[75] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad
memory management. In *Proceedings of the 14th International Conference on Paral-

[76] L. Li, Q. H. Nguyen, and J. Xue. Scratchpad allocation for data aggregates in su-
Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’07)*,


