2013

Exploring Hybrid SPM-Cache Architectures to Improve Performance and Energy Efficiency for Real-time Computing

Lan Wu
Virginia Commonwealth University

Follow this and additional works at: http://scholarscompass.vcu.edu/etd

Part of the Engineering Commons

© The Author

Downloaded from
http://scholarscompass.vcu.edu/etd/3280

This Dissertation is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.
Exploring Hybrid SPM-Cache Architectures to Improve Performance and Energy Efficiency for Real-time Computing

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

Lan Wu
B.S., University of Science and Technology of China, China, 2004
M.S. North China Institute of Computing Technology, China, 2007

Director: Dr. Wei Zhang,
Associate Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University
Richmond, Virginia
December, 2013
ACKNOWLEDGMENTS

I would like to thank Dr. Wei Zhang for his invaluable assistance and insights leading to my Phd research and this dissertation in the last five years.

My sincere thanks also goes to the members of my graduate committee for their patience and understanding during the numerous time of effort that went into the production of this dissertation.

Last but not the least, many thanks go to my family from the bottom of my heart.
# TABLE OF CONTENTS

Acknowledgments ....................................................... ii

List of Tables ......................................................... viii

List of Figures ......................................................... xi

Abstract ................................................................. 1

1 Introduction ......................................................... 4

Introduction ............................................................. 4

2 Background ........................................................... 9

2.1 Real-Time Computing ............................................. 9

2.2 On-chip Memory .................................................... 10

3 A Model Checking Based Approach to Bounding Worst-Case Execution

Time for Multicore Processors ....................................... 12

3.1 Chapter Overview ................................................ 12

3.2 Related Work ...................................................... 14

3.2.1 Prior Work in WCET Analysis ................................ 14

3.2.2 Prior Work in Model Checking ............................. 15

3.3 Background ......................................................... 17

3.3.1 The Assumed Dual-Core Processor with a Shared L2 Instruction Cache ........................................... 17

3.3.2 Assumptions ..................................................... 18

3.3.3 SPIN and PROMELA .......................................... 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.4 Static Analysis for WCET</td>
<td>20</td>
</tr>
<tr>
<td>3.4 Model Checking Based WCET Analysis For Multicore Processors</td>
<td>21</td>
</tr>
<tr>
<td>3.4.1 Model Single-Core Systems with SPIN</td>
<td>21</td>
</tr>
<tr>
<td>3.4.2 Model the Dual-Core Processor with SPIN</td>
<td>25</td>
</tr>
<tr>
<td>3.4.3 Improvement of the Previous Model</td>
<td>27</td>
</tr>
<tr>
<td>3.4.4 Architectural Parameters Impacting the Performance of Verifi-</td>
<td>33</td>
</tr>
<tr>
<td>cation</td>
<td></td>
</tr>
<tr>
<td>3.5 Evaluation Methodology</td>
<td>34</td>
</tr>
<tr>
<td>3.6 Experimental Results</td>
<td>36</td>
</tr>
<tr>
<td>3.6.1 Comparing the Performance Before and After Model Simplification</td>
<td>36</td>
</tr>
<tr>
<td>3.6.2 Comparing the Model Checking Based Approach with Simulation</td>
<td>37</td>
</tr>
<tr>
<td>and Static Analysis</td>
<td></td>
</tr>
<tr>
<td>3.6.3 Sensitivity to Cache Configurations</td>
<td>42</td>
</tr>
<tr>
<td>3.6.4 Limitation of the Model Checking Based Method</td>
<td>43</td>
</tr>
<tr>
<td>3.7 Conclusion</td>
<td>44</td>
</tr>
<tr>
<td>4 Exploiting Hybrid SPM-Cache Architectures to Reduce Energy Consump-</td>
<td>47</td>
</tr>
<tr>
<td>tion for Embedded Computing</td>
<td></td>
</tr>
<tr>
<td>4.1 Chapter Overview</td>
<td>47</td>
</tr>
<tr>
<td>4.2 Motivation</td>
<td>49</td>
</tr>
<tr>
<td>4.3 Background on Hybrid SPM-Caches</td>
<td>53</td>
</tr>
<tr>
<td>4.3.1 Instruction Hybrid and Data Hybrid Architectures</td>
<td>53</td>
</tr>
</tbody>
</table>

iv
5.5.2 WCET Results of Different SPM Allocation Algorithms . . . 96
5.5.3 Average-Case Performance Results . . . . . . . . . . . . . . 97
5.5.4 Sensitivity Study . . . . . . . . . . . . . . . . . . . . . . . . 98
5.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
5.7 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

6 Cache-Aware SPM Allocation for Maximizing Performance on Hybrid SPM-
Cache Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.1 Chapter Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.2 Related Works . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
6.3 Basic SPM Allocation Algorithms . . . . . . . . . . . . . . . . . . 112
  6.3.1 Frequency-based SPM Allocation . . . . . . . . . . . . . . . 112
  6.3.2 Hybrid SPM-Cache Allocation . . . . . . . . . . . . . . . . . 112
6.4 Stack Distance Based SPM Allocation Algorithms . . . . . . . . 114
  6.4.1 Stack Distance . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
  6.4.2 Stack Distance Analysis for The HSC Architecture . . . . . 114
  6.4.3 Stack Distance Based SPM Allocation Algorithms . . . . . 117
  6.4.4 Side Effects of Basic Block Based Allocation . . . . . . . . . 123
  6.4.5 An Example To Compare HSA and GSDA . . . . . . . . . . . 125
6.5 Evaluation Methodology . . . . . . . . . . . . . . . . . . . . . . . . . 128
6.6 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . 131
  6.6.1 Performance of Different Algorithms . . . . . . . . . . . . . . 131
  6.6.2 Sensitivity to the Cache Size . . . . . . . . . . . . . . . . . . . 134
6.6.3 Sensitivity to the Block Size ........................................ 136
6.6.4 Running Time Under Configuration I and II .................... 138
6.7 Conclusions ................................................................. 140

7 Cache-Aware SPM Allocation for Maximizing Energy Efficiency on Hybrid
SPM-Cache Architecture .................................................... 141
7.1 Chapter Overview .......................................................... 141
7.2 Stack Distance Based SPM Allocation Algorithms for Energy ...... 143
  7.2.1 Stack Distance Analysis for HSC on Energy Consumption . 143
  7.2.2 Exploit Cache Stack Distance to Improve SPM Allocation . 144
7.3 Evaluation Methodology ................................................... 147
7.4 Experimental Results ...................................................... 149
  7.4.1 Memory Energy Consumption ....................................... 149
  7.4.2 Performance Results .................................................. 151
  7.4.3 EDP Results ............................................................ 151
7.5 Conclusions ................................................................. 152

8 Conclusion Remarks ........................................................ 154
  8.1 Future Work .............................................................. 157
References ................................................................. 159
LIST OF TABLES

3.1 Explanations for symbols used in Listing 3.3. .......................... 28
3.2 Benchmark description. ................................................... 35
3.3 Configuration of the dual-core memory hierarchy. ................ .. 35
3.4 WCET of a single-core. .................................................... 36
3.5 Comparison of the performance before and after using the simplified model. ........................................ 37
3.6 Comparing the worst-case execution time, L1 miss ratio and L2 miss ratio of \texttt{bs} in core 1 among model checking, simulation and static analysis based approaches.(W: WCET, $R_{L1}$: L1 miss ratio, $R_{L2}$: L2 miss ratio, \(\frac{W_{MC}}{W_{SIMU}}\): MC/SIMU WCET ratio, \(\frac{W_{MC}}{W_{STAT}}\): MC/STAT WCET ratio ) . . 39
3.7 Comparing the worst-case execution time, L1 miss ratio and L2 miss ratio of the benchmark in core 2 among model checking, simulation and static analysis based approaches.(W: WCET, $R_{L1}$: L1 miss ratio, $R_{L2}$: L2 miss ratio, \(\frac{W_{MC}}{W_{SIMU}}\): MC/SIMU WCET ratio, \(\frac{W_{MC}}{W_{STAT}}\): MC/STAT WCET ratio ) 40
3.8 Comparing the worst-case execution time, L1 miss ratio and L2 miss ratio of \texttt{qsort} in core 1 among model checking, simulation and static analysis based approaches.(W: WCET, $R_{L1}$: L1 miss ratio, $R_{L2}$: L2 miss ratio, \(\frac{W_{MC}}{W_{SIMU}}\): MC/SIMU WCET ratio, \(\frac{W_{MC}}{W_{STAT}}\): MC/STAT WCET ratio ) 40
3.9 Comparing the worst-case execution time, L1 miss ratio and L2 miss ratio of the benchmark in core 2 among model checking, simulation and static analysis based approaches. (\(W\): WCET, \(R_{L1}\): L1 miss ratio, \(R_{L2}\): L2 miss ratio, \(\frac{W_{MC}}{W_{SIMU}}\): MC/SIMU WCET ratio, \(\frac{W_{MC}}{W_{STAT}}\): MC/STAT WCET ratio)

3.10 Compare the timing of model checking and static analysis approach (in seconds).

3.11 Configuration I of the Dual-core Chip Memory Hierarchy.

4.1 All the hybrid on-chip memories studied.

4.2 Salient characteristics of benchmarks.

5.1 Four SPM allocation algorithms studied in this chapter.

5.2 Benchmarks used in our experiments.

6.1 The SPM and cache parameters used in the example.

6.2 The execution sequence of the basic blocks.

6.3 The memory access trace before allocation. \(M=13\). (A: instruction address, BA: block address, SI: set index, SD: stack distance, M: number of cache misses)

6.4 The number of cache misses of the line blocks before the SPM allocation.

6.5 The memory access trace and cache misses after the SPM allocation by the HSA. (A: instruction address, BA: block address, SI: set index, SD: stack distance)
6.6 Checking the cache misses for each line block to identify the first candidate by the GSDA-based SPM allocation. (A: instruction address, BA: block address, SI: set index, SD: stack distance, M: number of cache misses).

6.7 Checking the cache misses for each line block to identify the second candidate by the GSDA based SPM allocation. (A: instruction address, BA: block address, SI: set index, SD: stack distance, M: number of cache misses).

6.8 Three memory configurations in our experiments.

6.9 General information of all benchmarks.

6.10 The cache misses of all 4 allocation algorithms in default configuration.

6.11 The running time (in msec) of all 4 allocation algorithms in default configuration.

6.12 The cache misses of all 4 allocation algorithms with the Configuration I.

6.13 The cache misses of all 4 allocation algorithms with the Configuration II.

6.14 The allocation time (in msec) of all 4 allocation algorithms in Configuration I.

6.15 The allocation time (in msec) of all 4 allocation algorithms in Configuration II.

6.16 The compression ratio of OSDA model during verification.

7.1 The symbols used in the equations.
# LIST OF FIGURES

2.1 The WCET estimation .................................................. 9

3.1 (a) A normal dual-core with a shared L2 cache; (b) a dual-core with a shared L2 instruction cache where the L1 data caches (i.e., dL1*) are perfect, that is, there are no L1 data cache misses. ..................... 18

3.2 The control flow graph of the motivating example. .................. 22

3.3 Control flow graphs of four different cases. Note the last three cases shared the same control flow graph though they differ in the L2 cache access conditions. .................................................. 32

3.4 Design flow for the model checking based dual-core WCET analysis. . . 34

3.5 Comparing the normalized WCET of three different cache models. . . 43

3.6 Feasibility of WCET analysis according to the total number of conflicting instructions modeled and subset size of L2 cache used by these instructions. .................................................. 45

4.1 Baseline caches or SPMs only architectures. ......................... 49

4.2 The comparison of on-chip memory energy consumption between the IC-DC and IS-DS architectures, which is normalized to that of the IS-DS architecture. .................................................. 52

4.3 The comparison of total energy consumption between the IC-DC and IS-DS architectures, which is normalized to that of the IS-DS architecture. 52
4.4 The comparison of the performance (i.e., the total number of execution cycles) between the IC-DC and IS-DS architectures, which is normalized to that of the IS-DS architecture. ........................................ 52

4.5 Three hybrid SPM-Cache architectures. ........................................ 53

4.6 Energy evaluation framework. .................................................. 59

4.7 The comparison of on-chip memory and total energy consumption among all 9 architectures, which is normalized to the on-chip and total energy respectively of the IS-DS architecture. ........................................ 60

4.8 The comparison of performance and EDP among all 9 architectures, which are normalized to execution cycles and EDP respectively of the IS-DS architecture respectively. ........................................ 61

4.9 The comparison of on-chip memory and total energy consumption among the IS-DS, the IC-DC, and the IH-DC, IC-DH, and IH-DH architectures with two different SPM and cache partitions, which is normalized to the on-chip memory and total energy consumption respectively of the IS-DS architecture. ........................................ 67

4.10 The comparison of on-chip memory and total energy consumption of the IC-DC and IS-DS architectures with their total size varying from 8KB to 16KB, 32KB, and 64KB, which is normalized to the on-chip memory and total energy consumption respectively of the 8KB IS-DS architecture. 67
4.11 The comparison of on-chip memory and total energy consumption of the IH-DC architectures with their total size varying from 8KB to 16KB, 32KB, and 64KB, which is normalized to the on-chip memory and total energy consumption respectively of the 8KB IS-DS architecture.

4.12 The comparison of on-chip memory and total energy consumption of the IC-DH architectures with their total size varying from 8KB to 16KB, 32KB, and 64KB, which is normalized to the on-chip memory and total energy consumption respectively of the 8KB IS-DS architecture.

4.13 The comparison of on-chip memory and total energy consumption of the IH-DH architectures with their total size varying from 8KB to 16KB, 32KB, and 64KB, which is normalized to the on-chip memory and total energy consumption respectively of the 8KB IS-DS architecture.

5.1 The hybrid SPM-Cache system architecture.

5.2 High-level overview of our evaluation framework.

5.3 Comparing the estimated WCET and the simulated WCET for all the four SPM allocation algorithms, which is normalized to the simulated WCET.

5.4 Comparing the WCET of different SPM allocation algorithms with the default configuration, which is normalized to the WCET of the FSA algorithm.
5.5 Comparing the ACET of different SPM allocation algorithms with the default configuration, which is normalized to the ACET of the FSA algorithm.

5.6 Comparing the WCET of different SPM allocation algorithms for the hybrid SPM-Cache with a 96B SPM and a 32B cache, which is normalized to the WCET of the FSA algorithm.

5.7 Comparing the ACET of different SPM allocation algorithms for the hybrid SPM-Cache with a 96B SPM and a 32B cache, which is normalized to the ACET of the FSA algorithm.

5.8 Comparing the WCET of different SPM allocation algorithms for the hybrid SPM-Cache with a 32B SPM and a 128B cache, which is normalized to the WCET of the FSA algorithm.

5.9 Comparing the ACET of different SPM allocation algorithms for the hybrid SPM-Cache with a 32B SPM and a 128B cache, which is normalized to the ACET of the FSA algorithm.

6.1 The example of side effect of basic block based allocation.

6.2 The control flow graph of the example code segment.

6.3 The performance of all 4 algorithms in default configuration, which is normalized to the total number of execution cycles of the FSA.

6.4 The performance of all 4 algorithms with Configuration I, which is normalized to the total execution cycles of the FSA under Configuration I.
6.5 The performance of all 4 algorithms with Configuration II, which is normalized to the total execution cycles of the FSA under Configuration II. 

7.1 The memory energy of all the allocation algorithms in default configuration (normalized to OSDA). 

7.2 The total energy of all the allocation algorithms in default configuration (normalized to FSA). 

7.3 Compare the EDP of all the allocation algorithms in default configuration (normalized to FSA).
Abstract

Exploring Hybrid SPM-Cache Architectures to Improve Performance and Energy Efficiency for Real-time Computing

By Lan Wu, Ph.D.

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2013.

Director: Dr. Wei Zhang,
Associate Professor, Department of Electrical and Computer Engineering

Real-time computing is not just fast computing but time-predictable computing. Many tasks in safety-critical embedded real-time systems have hard real-time characteristics. Failure to meet deadlines may result in the loss of life or in large damages. Known of Worst Case Execution Time (WCET) is important for reliability or correct functional behavior of the system.

As multi-core processors are increasingly adopted in industry, it has become a great challenge to accurately bound the worst-case execution time (WCET) for real-time systems running on multi-core chips. This is particularly true because of the inter-thread interferences in accessing shared resources on multi-cores, such as shared L2 caches, which can significantly affect the performance but are very difficult to be estimate statically. We propose an approach to analyzing Worst Case Execution Time (WCET) for multi-core processors with shared L2 instruction caches by using a model checking based method. Our experiments
indicate that compared to the static analysis technique based on extended ILP (Integer Linear Programming), our approach improves the tightness of WCET estimation more than 31.1% for the benchmarks we studied. However, due to the inherent complexity of multi-core timing analysis and the state explosion problem, the model checking based approach currently can only work with small real-time kernels for dual-core processors.

At the same time, improving the average-case performance and energy efficiency has also been important for real-time systems. Recently, Hybrid SPM-Cache (HSC) architectures by combining caches and Scratch-Pad Memories (SPMs) have been increasingly used in commercial processors and research prototypes. Our research explores the HSC architectures for real-time systems to reconcile time predictability, performance, and energy consumption. We study the energy dissipation of a number of HSC architectures by combining both caches and Scratch-Pad Memories (SPM) without increasing the total on-chip memory size. Our experimental results indicate that with the equivalent total on-chip memory size, several hybrid SPM-Cache architectures are more energy-efficient than either pure software controlled SPMs or pure hardware-controlled caches. In particular, using the hybrid SPM-Cache to store both instructions and data can achieve the best energy efficiency.

However, the SPM allocation for the HSC architecture must be aware of the cache performance to harness the full potential of the HSC architecture. First, we propose and evaluate four SPM allocation strategies to reduce WCET for hybrid
SPM-Caches with different complexities. These algorithms differ by whether or not they can cooperate with the cache or be aware of the WCET. Our evaluation shows that the cache aware and WCET-oriented SPM allocation can maximally reduce the WCET with minimum or even positive impact on the average-case execution time (ACET).

Moreover, we explore four SPM allocation algorithms to maximize performance on the HSC architecture, including three heuristic-based algorithms, and an optimal algorithm based on model checking. Our experiments indicate that the Greedy Stack Distance based Allocation (GSDA) can run efficiently while achieving performance either the same as or close to the optimal results got by the Optimal Stack Distance based Allocation (OSDA).

Last but not the least, we extend the two stack distance based allocation algorithms to GSDA-E and OSDA-E to minimize the energy consumption of the HSC architecture. Our experimental results show that the GSDA-E can also reduce the energy either the same as or close to the optimal results attained by the OSDA-E, while achieving performance close to the OSDA and GSDA.

Detailed implementation and experimental results discussion are presented in this dissertation.
CHAPTER 1
INTRODUCTION

Real-time systems have been widely used in our society especially for safety-critical systems, such as automobile and aircraft controllers. Besides performance, time predictability is also critical to real-time systems, because the missing deadlines may either lead to disastrous consequences or decrease quality of services badly. The Worst-Case Execution Time (WCET) of an application must be calculated to determine if its deadline can be always met. Moreover, cache memories have been widely used in modern processors to effectively bridge the speed gap between the fast processor and the slow memory to achieve good average-case performance. However, cache performance is heavily dependent on the history of memory accesses and the cache placement and replacement algorithms, making it hard to accurately predict the worst-case execution time. When threads running concurrently on different cores in a multicore platform, the shared cache memory can significantly impact execution time of each concurrent thread and complicate WCET analysis.

Scratch-Pad Memory (SPM) is an alternative on-chip memory to the cache, which has been increasingly used in embedded processors due to its energy and area efficiency. It is time-predictable because the allocation is controlled by software and the latency to access data from the SPM is fixed. However, SPMs generally are not adaptive to runtime instruction and data access patterns, and
thus may lead to inferior average-case performance. Processors that employ caches or SPMs alone can only benefit either the average-case performance or the time predictability, not both.

A hybrid cache and SPM model has also been used in some prototype or commercial processors such as TRIPS [1], ARM1136JF-S [2], and Nvidia Fermi [3]. Recent studies show that a hybrid SPM-Cache can greatly improve the performance [4], energy efficiency [5] and time-predictability [6], all of which are potentially beneficial to embedded systems, including hard real-time systems. However, the traditional SPM allocation, including both static and dynamic allocation, mainly focuses on the SPM alone. These cache-unaware SPM allocation algorithms are unlikely to harness the full potential of the hybrid SPM and cache.

To use the aggregate SPM and cache space more efficiently, we believe the SPM allocation for the hybrid SPM-Cache architecture must be aware of the cache performance to maximally optimize the worst-case execution time, the average performance and the energy consumption.

Motivated by these challenges, the rest of this dissertation is organized as follows.

Chapter 2 provides the background knowledge.

Chapter 3 focuses on studying a model checking based approach to safely and accurately estimate inter-thread cache interferences and WCET for real-time tasks running on multicore processors. Our approach is built on top of Metzners single-core analysis method [7] which is extended to model inter-thread
interferences in a dual-core processor with a shared L2 instruction cache. We exploit program control flow information to derive basic block automation (BBA), based on which a PROMELA process [8] is generated to model each concurrent thread as well as its accesses to the shared cache. We use the SPIN model checker [8] to prove the upper bound of execution time and then use a binary search algorithm to compute the WCET for real-time tasks running on multicore processors. Our experiments demonstrate that the model checking based method indeed improves the tightness of WCET analysis, as compared to the state-of-the-art static analysis approach [9], although the state explosion problem currently limits its applicability to small real-time kernels for computers with constrained physical memories.

Chapter 4 is built upon the prior work in [6] to study the performance and time predictability of hybrid SPM-Cache architectures. We systematically study the energy consumption behaviors of 7 different hybrid SPM-Cache architectures, which is expected to provide important insights on energy-efficient on-chip memory design for embedded processors. We demonstrates that the hybrid on-chip memory architectures can also make better tradeoffs between performance and energy consumption, making it a very attractive design option for real-time and embedded systems.

Chapter 5, Chapter 6 and Chapter 7 study the SPM allocation algorithms for the hybrid SPM-Cache architectures. First, Chapter 5 explores four different SPM allocation algorithms to reduce WCET for the hybrid SPM-Cache architecture.
The first one is the Frequency based SPM Allocation (FSA), which is not aware of the cache and WCET, which is used as the baseline. The Longest Path based Allocation (LPA) is WCET aware but cache unaware. The other two algorithms are all cache-aware, but the Hybrid SPM-Cache Allocation (HSA) allocates the basic blocks according to the results of cache analysis based on Abstract Interpretation (AI) [10] and does not consider the WCET. The Enhanced Hybrid SPM-Cache Allocation (EHSA) algorithm is a WCET-oriented and cache-aware SPM allocation algorithm, and our experimental results indicate that the EHSA algorithm can outperform other three algorithms to reduce WCET for the hybrid SPM-Cache with little or even positive impact on the average-case performance.

In Chapter 6, we design and comparatively evaluate 4 different SPM allocation algorithms to maximally optimize the execution time. The baseline allocation algorithm is still the Frequency based SPM Allocation (FSA). The other three algorithms are all cache-aware, but exploit cache information in different ways. The Hybrid SPM-Cache Allocation (HSA) is different from that of Chapter 5 and it exploits cache profiling information. It tries to allocate the memory objects with the largest cache misses into the SPM. The remaining two algorithms are both based on the Stack Distance Analysis (SDA) [11], [12]. The Greedy Stack Distance based Allocation (GSDA) is a greedy algorithm, whereas the Optimal Stack Distance based Allocation (OSDA) is an optimal algorithm by using model checking. By experiments we find that all the three cache-aware algorithms attain superior performance than the FSA algorithm. In particular, the HSA and the
GSDA improve the performance by 9% and 11% respectively as compared to the FSA. The OSDA always achieves the best performance, but requires significantly more memory space and longer running time and may not be scalable for larger benchmarks. The GSDA can achieve performance either the same as or very close to that of the OSDA.

Moreover, we extend the GSDA and OSDA to reduce the energy consumption in Chapter 7, which are called the Greedy Stack Distance based Allocation for Energy (GSDA-E) and the Optimal Stack Distance based Allocation for Energy (OSDA-E). We evaluate them together with the four different SPM allocation algorithms from Chapter 6, and find that GSDA-E can reduce the energy either the same as or close to the optimal results attained by the OSDA-E, while achieving performance close to the OSDA and the GSDA.

Finally, Chapter 8 concludes this dissertation.
CHAPTER 2
BACKGROUND

In this chapter, background information is provided for the topics covered in this dissertation.

2.1 REAL-TIME COMPUTING

Real-time computing is the study of hardware and software systems that are subject to a real-time constraint. The tasks running on the real-time system usually have strict time constraints which are referred to as “deadlines”. There are two kind of deadlines: hard deadline and soft deadline. Missing a hard deadline can causes a total system failure or disastrous consequences, while missing a soft deadline may degrade the quality of service of the system because the degradation of the usefulness of the result. Therefore, the timing correctness of hard realtime systems should be guaranteed by the safe and accurate estimation of worst-case execution time (WCET) The WCET estimation is demonstrated in Figure 2.1, which is defined as the upper bounds for the execution times of real-time tasks.

![Figure 2.1. The WCET estimation](image-url)
2.2 ON-CHIP MEMORY

In order to boost the performance of modern processors, cache memories have been widely used in modern processors to effectively bridge the speed gap between the fast processor and the slow memory to achieve better average-case performance and to reduce the energy consumption of accessing the main memory. It stores the data used before to save the access time to the same data requested in the future. If the data requested are found in the cache, there will be a cache hit, otherwise a cache miss happens. The performance of the cache memory is heavily dependent on the history of memory accesses, as well as the cache placement and replacement algorithms.

The Scratch-Pad Memory (SPM) [13] are also on-chip memories based on SRAM, which can be used to store instructions, data, or both. Unlike caches that are controlled by hardware, the mapping of program and data elements into the SPM is usually performed either by the user or the compiler. This leads to statically predictable memory access time, which is desirable to real-time systems. Moreover, since an SPM does not need to use tag arrays, it is generally more energy- and area-efficient than a cache with the same size. On the other hand, since SPMs are totally controlled by software, they are generally less adaptable to various instruction/data access patterns that are dependent on runtime inputs. Also, because SPM allocation is done statically, the SPMs generally cannot dynamically reuse the limited on-chip SPM space as efficiently as the caches. A number of commercial processors employing scratch-pad memory are already

10
available in the market such as Motorola MPC500 [14], ARMv6 [15].
CHAPTER 3
A MODEL CHECKING BASED APPROACH TO BOUNDING
WORST-CASE EXECUTION TIME FOR MULTICORE PROCESSORS

3.1 CHAPTER OVERVIEW

In the last two decades, worst-case execution time (WCET) analysis has been extensively studied, primarily for single-core processors [16]. However, due to technology advancement and concerns about power and heat dissipation, the wide use of multicore systems introduces new challenges to WCET analysis. Specifically, in a multicore platform, threads running concurrently on different cores may interfere with each other in accessing shared resources, such as shared buses, cache or memory modules, which can significantly impact execution time of each concurrent thread and complicate WCET analysis. Nevertheless, to safely employ multicore chips to benefit real-time cyber-physical systems, especially for hard real-time systems, it is a necessity to derive safe and precise timing guarantees through WCET analysis, which must take into account the impact of inter-thread interferences.

In this chapter, we focus on studying a model checking based approach to safely and accurately estimate inter-thread cache interferences and WCET for real-time tasks running on multicore processors. In addition, while our approach can be applied generally to any multicore processor, as a first step towards using
model checking for multicore timing analysis, we focus on analyzing a dual-core processor. Our approach is built on top of Metzners single-core analysis method [7] which is extended to model inter-thread interferences in a dual-core processor with a shared L2 instruction cache. We exploit program control flow information to derive basic block automation (BBA), based on which a PROMELA process [8] is generated to model each concurrent thread as well as its accesses to the shared cache. We use the SPIN model checker [8] to prove the upper bound of execution time and then use a binary search algorithm to compute the WCET for real-time tasks running on multicore processors.

While recently there have been a few studies on bounding the worst-case interthread cache interferences for multicore processors [17, 18, 19, 20, 9], the novelty of our proposed approach lies in that we propose to analyze WCET for multicore systems by using model checking technology. To the best of our knowledge, this work is the first effort to apply model checking technology to multicore WCET analysis which has become increasingly important in this multicore era, considering the benefits of multicore computing, such as high throughput and better energy efficiency, etc. We have also introduced several techniques to reduce the memory consumption of the model checker without compromising the quality of analysis by intelligently exploiting domain-specific information. Our experiments demonstrate that the model checking based method indeed improves the tightness of WCET analysis, as compared to the state-of-the-art static analysis approach [9], although the state explosion problem
currently limits its applicability to small real-time kernels for computers with constrained physical memories.

The rest of the chapter is organized as follows. Section 3.2 discusses the related work. Section 3.3 describes background information, and Section 3.4 introduces the model checking based WCET analysis for multicore processors. The evaluation methodology is given in Section 3.5, and the experimental results are presented in Section 3.6. Finally, we draw conclusions in Section 3.7.

3.2 RELATED WORK

3.2.1 Prior Work in WCET Analysis

WCET analysis has been studied intensively in the last two decades. A good review of the state of the art can be found in Wilhelm et al. [16]. Most of the research efforts on WCET analysis have been focused on single-core processors [21, 22, 23, 24, 25]. However, these techniques cannot be applied to estimate the WCET for multicore processors, because they do not consider the possible inter-core interferences caused by concurrent threads accessing resources shared among different cores.

In contrast, there are relatively few efforts to study WCET analysis for multicore processors. Stohr et al. [19] proposed a measurement-based approach to bounding worst-case access time for multicore platforms. However, this approach may be unsafe due to the fact that it is generally impossible to exhaust all the possible paths with various inputs for concurrent threads. Rosen et al. [17] studied the implicit bus traffic due to cache misses by different processors. However, they
did not investigate the challenging problem of inter-thread cache interferences on a multicore chip, which is crucial for accurately bounding the worst-case performance for multicore processors. Recently, Yan and Zhang proposed a control flow based approach [20] and an enhanced approach [9] to deriving WCET for multicore processors with shared instruction caches. However, due to the conservative nature of static timing analysis and the lack of actual runtime information, the tightness (or accuracy) of the static analysis is not guaranteed. In contrast to all these existing studies, we propose a model checking based method to safely and accurately bound the WCET for multicore processors.

3.2.2 Prior Work in Model Checking

Pioneering work in the model checking was done by Clarke et al. in the early 1980s [26]. They developed a model checking method for checking models of system designs where the specification is given by a temporal logic formula. The initial temporal logic is proposed by Pnueli [27] called linear-time propositional temporal logic to specify and compute the behaviors of computer systems. Temporal logic has proved to be useful for specifying concurrent systems. There have been many variants of temporal logic proposed in the literature [28]. Alur and Dill first proposed the model of timed automata [29]. Both logic formulas and automata can be used to specify the model of a system. Model checking techniques have been applied not only to finite state systems but also to real-time systems [30]. In practice, a real-time system is usually described as a set of process-timed automata, each representing the behavior of an autonomous process [30].
In this chapter, we do not intend to make a comprehensive survey of all the related work in model checking. Instead, our focus is to discuss related work on using model checking technology to cope with the WCET analysis problem. In this area, Metzner [7] firstly showed that model checking could be used to compute WCET for single-core systems. Lv et al. [31] compared the performance of the WCET analysis techniques using static path analysis and model checking for uniprocessors. Wilhelm [32] compared model checking, integer linear programming (ILP) and a combination of abstract interpretation (AI) with ILP to determine the WCET and argued that AI+ILP is a better approach. Recently, Huber and Schoeberl [33] compared ILP and model checking based WCET analysis for Java uniprocessors and found that model checking is fast enough for local analysis and small applications, leading them to suggest combining model checking with ILP for attaining tight WCET results with reasonable analysis time. Mohalik et al. [34] applied UPPAAL model checker to bound the end-to-end latency in real-time systems with clock drifts. However, due to the inherent high complexity of WCET analysis for multicore systems, to the best of our knowledge, there is no prior work to study applying model checking technology to solve the WCET analysis problem for multicore systems.
3.3 BACKGROUND

3.3.1 The Assumed Dual-Core Processor with a Shared L2 Instruction Cache

In a multicore processor, each core typically has private L1 instruction and data caches. The L2 (and/or L3) caches can be shared or private. While private L2 caches are more time predictable in the sense that there are no inter-core L2 cache conflicts, they suffer from other deficiencies. First, each core with a private L2 cache can only exploit separated and limited cache space. Due to the great impact of the L2 cache hit rate on the performance of multicore processors [35], private L2 caches may have worse performance than a shared L2 cache with the same total size, because each core with a shared L2 cache can make use of the aggregate L2 cache space more effectively. Second, separated L2 caches will increase the cache synchronization and coherency cost. Moreover, a shared L2 cache architecture makes it easier for multiple cooperative threads to share instructions and data, which becomes more expensive in separated L2 caches.

Therefore, we will study the WCET analysis of multicore processors with shared L2 caches (by contrast, the WCET analysis for multicore chips with private L2 caches is a less challenging problem).

Although our study focuses on examining the WCET analysis for a dual-core processor with a shared L2 cache, our approach could also be generally applied to multicore processor with a higher number of cores (e.g., a quad-core chip). Figure 3.1(a) shows a typical dual-core processor where each core has private L1
instruction and data caches and shares a unified L2 cache.

Figure 3.1. (a) A normal dual-core with a shared L2 cache; (b) a dual-core with a shared L2 instruction cache where the L1 data caches (i.e., dL1*) are perfect, that is, there are no L1 data cache misses.

3.3.2 Assumptions

We assume that the worst-case execution time of a single core can be handled by existing WCET analysis techniques [16]. We also assume that the shared bus and memory of the multicore processor are time predictable, which actually can be supported by recently proposed techniques, such as the interference-aware bus arbiter [36] and the predictable SDRAM memory controller [37], respectively. Furthermore, we assume a perfect data cache so that we can concentrate on examining the inter-thread interferences of instruction accesses. It should be noted that this last assumption is consistent with the recent multicore WCET analysis work [9], which will be compared with the proposed model checking based approach. Specifically, the assumed dual-core architecture is depicted in Figure 3.1(b), where each core has its own L1 instruction cache and a perfect L1 data cache (i.e., dL1*) and shares the L2 cache.

Also, we assume that either two real-time threads (RTs) or a real-time
thread and a non-real-time thread (NRT) are running simultaneously on these two cores. Our goal is to safely and accurately estimate the maximum inter-core L2 cache interferences, based on which the WCET of each RT can be calculated. Moreover, we assume these two concurrent running threads are totally independent with each other, that is, they do not share any data or instructions and they do not communicate with each other or need to be synchronized. Consequently, cache accesses from each thread can interleave with another thread in any order, making it challenging to compute the worst-case inter-thread cache interferences and WCET. To focus on WCET analysis, our study does not consider real-time scheduling for multicore processors. However, we believe the WCET results obtained for each real-time task running on a multicore platform can provide a basis for schedulability analysis and are crucial to designing multicore real-time schedulers for reducing inter-thread cache interferences.

3.3.3 SPIN and PROMELA

In this chapter, we use the SPIN model checker [8] to perform WCET analysis on the assume dual-core architecture. Spin is a tool developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. It can be used for the formal verification of distributed software systems. SPIN verification models can simulate the interactions of different processes using asynchronous message rendezvousing at buffered channels, shared variables, or with any combination of these, and its internal sequential computations are abstracted as much as possible to verify the correctness of temporal specification.
Specifications of the model in SPIN are written in the verification language PROMELA (Process Meta Language), and its correctness is checked by the syntax of standard linear temporal logic (LTL). Instead of synchronous control in hardware systems, SPIN focuses on asynchronous control in software systems which distinguishes it from other well-known approaches to model checking [38].

PROMELA is the verification modeling language of the SPIN system. PROMELA programs consist of processes, message channels, and variables. Processes are global objects, and message channels and variables can be declared either globally or locally within a process. Processes specify behavior; channels and global variables define the environment in which the processes run. All statements in the PROMELA processes are atomic, which are executed concurrently, interleaving and non-deterministically. It is worth noting that in SPIN, atomic means that each statement is executed without interleaving with other processes interleaving indicates that statements of different processes do not occur at the same time and non-deterministic means that each process may have several different possible actions, and only one choice is made non-deterministically.

3.3.4 Static Analysis for WCET

Yan and Zhangs previous work proposed a static analysis approach for WCET using extended ILP (EILP) [9], which we used for comparison to evaluate our model checking approach. In Yan and Zhangs article, they adopt the same assumed dual-core processor as we did in Section 3.1. The basic idea of this approach is to try to find out the maximum impact of inter-thread instruction
interference to the WCET. For two co-running threads from different cores, the shared L2 cache line accessed by one thread may be also requested by another thread, and this will cause additional L2 cache misses for the former thread. As a result, the execution time of the former thread may be longer than just running by itself. In Yan and Zhang's work, they first build a formula to describe the WCET of multicore task in EILP which is a summation of computation time of each basic block, total cache hit latency, and total cache miss latency, including intra-thread and inter-thread. Then structural constraints are constructed by looking into the CFG, and functionality constraints are derived by bounding the loops and other path information. Moreover, interthread constraints, which are the key factors to the WCET computation in multicore, are provided by considering the possible interferences from another core. All of them are put together into an ILP analyzer to finally obtain the maximum value of the original WCET computation formula. More details of this work can be found in [9].

3.4 MODEL CHECKING BASED WCET ANALYSIS FOR MULTICORE PROCESSORS

3.4.1 Model Single-Core Systems with SPIN

For single-core WCET analysis, we adopt Metzner's method [7] with several extensions. First, we analyze the input program to build its control flow graph (CFG). Second, the formal semantics of constructing an automaton from the CFG is defined, which is called basic block automaton (BBA). Third, concrete models described in PROMELA are generated based on which the upper bounds of
execution time can be calculated. Last, we use a binary search algorithm to find the WCET which is described in detail in Section 2.4.2.

Listing 3.1. The C program source code for an motivating example

```c
#include "stdafx.h"
#include <iostream>

int main(int argc, char* argv[])
{
    for(int i=1; i<=10; i++)
    {
        if(argc/2==0)
            cout <<"argc_is_even"<<endl;
        else
            cout <<"argc_is_odd"<<endl;
        i++;
    }
    return 0;
}
```

Figure 3.2. The control flow graph of the motivating example.

Listing 3.1 gives a specific example of a simple C program; Figure 3.2 shows its control flow graph; and Listing 3.2 illustrates the SPIN model for this example,
which is automatically generated given the BBA of the program. In this example, an integer variable \( lpc1 \) is used as the loop counter, and another variable \( wcet \) is used to record the time passed. The proctype \( \text{BBA()} \) implements the BBA by simulating all the state transitions and performing the corresponding actions. Each block led by line number ‘Si: ’ represents a state of the BBA. The statements wrapped in \text{atomic} implement the semantics of all possible transitions enabled in the current state. The \text{init()} proctype is mandatory in SPIN which initializes variables and starts all the other user-defined proctypes.

The never claim in Listing 3.2 implements the LTL property \( \Box \ p. \)\(^1\) If \( wcet \geq actualWCET \), \( \Box \ p \) is evaluated true, and SPIN exits with 0; otherwise, \( \Box \ p \) is violated, and SPIN writes the counterexample into the trace file and exits with 1. A successful run of SPIN proves an upper bound of the execution time based on which the WCET can be derived by using a binary search algorithm (more details can be seen in Algorithm 1).

Listing 3.2. The SPIN model of the motivating example

```c
1  int wcet, lpc1;
2  proctype BBA()
3  {
4    S1: atomic{
5      wcet=wcet+execution_time1; goto S2;
6    }
7    S2: atomic{
8      wcet=wcet+execution_time2;
9      lpc1 ++;
10     if
```

\(^1\)\( \Box \) denotes the always (or globally) operator in linear temporal logic
::goto S3;
::goto S4;
if;
}
S3: atomic{
  wcet = wcet + execution_time3; goto S5;
}
S4: atomic{
  wcet = wcet + execution_time4; goto S5;
}
S5: atomic{
  wcet = wcet + execution_time5;
  if :
lpcl < 10 -> goto S2;
  else goto S46;
  if;
}
S6: atomic{
  wcet = wcet + execution_time6;
}
}
init{
  atomic{
    wcet = 0; lpcl = 0; run BBA();
  }
}
#define p(wcet <= BOUND)
never{
  T0_init;
  if :
  !(p)) > goto accept_all;
  else > goto T0_init;
  fi;
  accept_all;

24
3.4.2 Model the Dual-Core Processor with SPIN

We propose the model checking based WCET analysis approach for the dual-core processor with the following major steps. First, our approach models each thread running on each core with a PROMELA process, as shown in Listing 3.3, which describes the cache and memory access behavior for every instruction running on each core. The symbols used in Listing 3 are defined in Table 3.1. Second, our approach models both the L1 and L2 instruction caches as two-dimensional arrays in which each row represents a cache line and each element represents an instruction. After each access to the L1 instruction cache or the L2 cache in case of an L1 miss, our approach then calculates the mapped cache line and identifies the corresponding L1 or L2 array element. Our approach then updates this particular row of the L1/L2 array that holds the mapped instruction with the instruction number associated with the core number (i.e., either 1 or 2), which uniquely identifies this cache access. Based on this instruction identification number, our approach can then determine the L1/L2 hit or miss for each instruction access, including the L2 miss caused by an inter-thread cache interference. Finally, our approach uses another instruction per basic block to update the latency of executing this block (assuming cache hits) and add the accumulated cache miss penalties (including both L1 and L2 misses) onto one of the two global variables, $wcet_1$, $wcet_2$, using Equation 3.1, as shown in Listing 3.3.
\[ wcet_n = wcet_n + \text{execution}_{bbi} + \text{num}_{bbi} \times L1\text{hit}\_latency \]

\[ + l1\text{miss}\_of\text{core}_n \times L2\text{miss}\_latency + l2\text{miss}\_of\text{core}_n \times L2\text{miss}\_latency \]  

(3.1)

In our approach, LTL formula \( \Box (wcet_1 < BOUND1) \) and

\[ \Box (wcet_2 < BOUND2) \]

are used to verify that the WECT of core 1 is less than BOUND1 and the WCET of core 2 is less than BOUND2. The actual WCET value is then calculated by changing the value of BOUND until the LTL formulas

\[ \Box (wcet_1 < BOUND1) \land \Box (wcet_1 < BOUND1 - 1) \]

and

\[ \Box (wcet_2 < BOUND2) \land \Box (wcet_2 < BOUND2 - 1) \]

hold. A binary search algorithm can be used to efficiently find this tight WCET value automatically, which is depicted in Algorithm 1.

Listing 3.3. The PROMELA model for the dual-core processor

```prome
int wcet1, wcet2;

proc core1()
{
    int l1misscore1=0; int l2misscore1=0;
    bb1:for(i=1; i<=the number of instructions in bb1; i++)
    {
        atomic{
            /* run instruction i */
            if (there is an l1 miss)
            {
                l1misscore1++;
            }
        }
    }
}
```

\(^2\)Note that BOUND1 and BOUND2 are constants that represent the possible upper bounds of execution time for thread 1 and thread 2, respectively.
update l1[row(map(i))];
if (there is an l2 miss)
{
    l2missofcore1++;
    update l2[row(map(i))];
}
}  /* end of if (there is an l1 miss) */
}  /* end of atomic */
}  /* end of for */
wcet1 = wcet1 + execution_bb1 + num_bb1 * L1hit_latency + l1missofcore1 * L1miss_latency + l2missofcore1 * L2miss_latency;
goto bb2;
bb2: ... ...
}

proctype core2()
{
/* Operations similar to those in core1 */
}
init{
atomic{
    Initial array of L1 and L2;
    wcet1 = 0; wcet2 = 0;
    run core1(); run core2();
}
}

3.4.3 Improvement of the Previous Model

Due to the inherent complexity of concurrent thread interactions on multicore platforms and the well-known state explosion problem with model checking, we try to exploit domain-specific knowledge to mitigate the state explosion without
<table>
<thead>
<tr>
<th>Symbols</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>limissofcorei</td>
<td>total number of L1 misses in corei (i=1,2)</td>
</tr>
<tr>
<td>row(map(i))</td>
<td>the cache line that instruction i mapped to</td>
</tr>
<tr>
<td>execution_bbi</td>
<td>execution time of basic block i</td>
</tr>
<tr>
<td>num_bbi</td>
<td>total instruction number in basic block i</td>
</tr>
<tr>
<td>L1hit_latency</td>
<td>latency of an L1 hit</td>
</tr>
<tr>
<td>L1miss_latency</td>
<td>latency of an L1 miss</td>
</tr>
<tr>
<td>L2miss_latency</td>
<td>latency of an L2 miss</td>
</tr>
</tbody>
</table>

Table 3.1. Explanations for symbols used in Listing 3.3.

**Algorithm 1** Finding the WCET using binary search

1: begin

2: set the upper and lower bound of binary search;

3: while lower_bound < upper_bound – 1 do

4: middle = (lower_bound + upper_bound)/2;

5: check the property □(WCET <= middle);

6: end while

7: if □(WCET <= middle) is satisfied then

8: upper_bound = middle;

9: else

10: lower_bound = middle;

11: end if

12: return upper_bound

13: end
compromising the safety or tightness of the proposed WCET analysis approach. The details of these techniques are described in Sections 2.4.3.1 and 2.4.3.2.

**Simplify the Model**

First of all, we intend to minimize the instructions modeled in each core without affecting the WCET analysis. To achieve this, the WCET of the application when a single core is used can be calculated separately by using the SPIN model of the single-core processor, as described in Section 4.1. The L1-hit, L1-miss, L2-hit, and L2-miss instructions for a single core can be easily calculated in the simulation mode of SPIN. Similar to the model in Listing 3.3, we have already modeled the calculation whether it is a cache miss or hit for every instruction, and this information can be imported when we simulate the model on the fly. We then assign the cost of each basic block as the WCET of that block in a single-core system. After categorizing cache accesses into L1/L2 hits or misses, our approach only needs to model the L2-hit instructions in each core to derive the possible inter-thread L2 cache interferences and to use one instruction to calculate the cost of each basic block. This is because the inter-thread cache interferences can only happen in the shared L2 cache (note that the L1 cache is private to each core). Also, we do not need to model L2 misses, because they cannot be further degraded by the interferences from another thread. Therefore, by only modeling the L2 hit instructions from each thread, we can significantly simplify the model and reduce the number of states checked. At the same time, the analysis of worst-case inter-thread interferences is not affected.
For example, assume a simple benchmark with seven basic blocks and 100
dynamic instructions in which only 11 instructions are identified as L2 hits. Before
the aforementioned simplification, we have to model all 100 instructions. After
simplification, however, only 11 L2-hit instructions and seven instructions for
calculating the costs of the seven basic blocks need to be modeled. As a result, the
state explosion problem can be alleviated.

Another benefit by only modeling L2-hit instructions is that we do not need
to use two arrays to represent L1 and L2 caches, respectively. Instead, we can use
a single array to model the L2 cache only, whose size only needs to be equal to the
number of L2-hit instructions. Moreover, because all these cache accesses are
guaranteed to be L2 hits, each element in this array now only needs to store a
single bit, indicating whether this L2 cache line is used by core 1 or core 2.
Compared with the original method that has to use an instruction number in
addition to another bit indicating the core number for uniquely identifying each
cache access, this optimization can greatly shrink the size of each state the verifier
generates, thus further mitigating the state explosion problem.

Additional Optimizations for Simplification

In addition to the preceding optimizations, we find that it is possible to
further simplify the model without compromising the quality of analysis by
exploiting control flow information. Specifically, by analyzing the CFG of each
real-time application, we find that more instructions do not need to be modeled,
which are discussed next based on different cases.
For loops. If the instructions in a loop are all always-hit\textsuperscript{3} or first-miss\textsuperscript{4} instructions [39], then we only need to use two PROMELA instructions to calculate the execution time of this loop, instead of using \( N \) PROMELA instructions, where \( N \) is the number of loop iterations. Specifically, we can use only one PROMELA instruction to compute the execution time of the first iteration and another instruction to compute the execution time of the second iteration multiplied by \( N1 \). This is because except for the first iteration, all the remaining loop iterations have the same cache behavior and, thus, the same execution latency. As a result of this optimization, \( N2 \) PROMELA instructions can be reduced for this loop.

For single path and multi-path blocks. There are four different cases as shown in Figure 3.3 and described next.

- Case 1. If there is no L2-hit instruction in block 1, then bb1 can be deleted from the PROMELA model. However, to maintain the correctness and accuracy of analysis, Equation 3.2 needs to be used to adjust the cost of bb2.

\textsuperscript{3}An always-hit instruction is an instruction that is guaranteed to hit in the cache. If there is an L1 miss instruction in a loop and if it is an L2 hit in a single-core case, we calculate its cache set number and the conflicting cache set due to L2 accesses from other core(s) in the dual-core case. If another core will not use this set during its execution time, then this instruction is classified as an always-hit instruction.

\textsuperscript{4}A first-miss instruction in a loop is an instruction that misses for the first access and then hits for all the remaining accesses. If there is an L1-miss instruction in a loop that is also an L2 miss at the first time but hits in the remaining loop iterations, and if the another core will not use the conflicting set during its execution time, this instruction will be classified as a first-miss instruction.
Figure 3.3. Control flow graphs of four different cases. Note the last three cases shared the same control flow graph though they differ in the L2 cache access conditions.

In other words, these two blocks can be aggregated into one unit for calculating the total execution time if bb1 does not have any L2-hit instructions.

\[ bb2_{\text{cost}} = bb2_{\text{cost}} + bb1_{\text{cost}} \]  \hspace{2cm} (3.2)

- Case 2. If there is no L2-cache-hit instruction in block 1, then bb1 can be deleted from the PROMELA. Equations 3.2 and 3.3 will be used to adjust the costs of bb2 and bb3, respectively.

\[ bb3_{\text{cost}} = bb3_{\text{cost}} + bb1_{\text{cost}} \]  \hspace{2cm} (3.3)

- Case 3. If there is no L2-hit instruction in blocks 2 and 3, then both blocks can be deleted from the PROMELA model. However, Equation 3.4 needs to be used to adjust the cost of bb4 by adding the maximum cost of bb2 and bb3.

\[ bb4_{\text{cost}} = bb4_{\text{cost}} + \max(bb2_{\text{cost}}, bb3_{\text{cost}}) \]  \hspace{2cm} (3.4)
• Case 4. If block 2 (or 3) has $n$ L2-hit instructions and it satisfies Equation 3.5, then block 2 (or 3) can be deleted from the PROMELA model without impacting the WCET calculation.

\[
n \times L2\text{-}miss\_penalty + bb2(3)\_cost < bb3(2)\_cost.
\] (3.5)

It should be noted that the aforementioned four cases can be combined and applied to optimize multiple blocks, which can further reduce the number of instructions modeled and, hence, mitigate the state explosion.

### 3.4.4 Architectural Parameters Impacting the Performance of Verification

We find that there are mainly three architectural parameters that will influence the performance of verification, including (1) the total physical memory size of the machine the SPIN runs on, (2) the number of instructions executing on each core, and (3) the size of L2 cache lines that the L2-hit instructions access (note that the interfering instructions may not access all the L2 cache lines). First, if the total physical memory of the machine on which SPIN runs is large, there will be more memory for SPIN to store states and, thus, larger problems can be solved. However, the physical memory can not be enlarged unlimitedly in practice. Therefore, we have to simplify the model of the problem to reduce the number of states to ensure that the verification problem can be reasonably solved by an actual machine. Second, the number of instructions executing on each core is also a critical parameter to affecting the performance. Fortunately, this number can be
largely reduced by deep analysis of the application, as explained in Section 2.4.3.1. Finally, the number of L2 cache lines used by the L2-hit instructions in each core can hardly be reduced, because it is determined by the cache configuration and the mapping method from memory to cache, which are often fixed for a given cache.

3.5 EVALUATION METHODOLOGY

Our approach makes use of an extended Trimaran [40] to extract useful program information for each concurrent thread. Trimaran is an integrated compilation and performance monitoring infrastructure. An overview of our design flow is illustrated in Figure 3.4. The starting point of our analysis is C programs. We use an extended Trimaran framework to generate basic block information which is stored in IR (intermediate representation) files. According to the benchmark information and the CFG, PROMELA programs are constructed and run in SPIN to either prove an upper bound of execution time or reject an underestimated value of WCET. We then use the binary search algorithm described in Algorithm 1 to compute the WCET value.

![Figure 3.4. Design flow for the model checking based dual-core WCET analysis.](image)

The benchmarks are selected from Mälardalen WCET benchmarks [41] and
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Number of Insts</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs</td>
<td>31</td>
<td>Malardalen</td>
<td>Binary search</td>
</tr>
<tr>
<td>fibcall</td>
<td>20</td>
<td>Malardalen</td>
<td>Iterative Fibonacci calculation</td>
</tr>
<tr>
<td>insertsort</td>
<td>70</td>
<td>Malardalen</td>
<td>Insertion sort on a reversed array</td>
</tr>
<tr>
<td>jfdctint</td>
<td>219</td>
<td>Malardalen</td>
<td>Discrete-cosine transformation</td>
</tr>
<tr>
<td>ludcmp</td>
<td>247</td>
<td>Malardalen</td>
<td>Read ten values, output half to LCD</td>
</tr>
<tr>
<td>matmul</td>
<td>70</td>
<td>Malardalen</td>
<td>Multiplication of two 20x20 matrices</td>
</tr>
<tr>
<td>minver</td>
<td>127</td>
<td>Malardalen</td>
<td>Inversion of floating point matrix</td>
</tr>
<tr>
<td>qsort</td>
<td>199</td>
<td>Malardalen</td>
<td>Non-recursive quick sort algorithm</td>
</tr>
<tr>
<td>qurt</td>
<td>97</td>
<td>Malardalen</td>
<td>Root computation of quadratic equations</td>
</tr>
<tr>
<td>select</td>
<td>168</td>
<td>Malardalen</td>
<td>Select the Nth largest number</td>
</tr>
<tr>
<td>cordic</td>
<td>768</td>
<td>Mediabench</td>
<td>Rotating complex numbers over the real field</td>
</tr>
</tbody>
</table>

Table 3.2. Benchmark description.

<table>
<thead>
<tr>
<th></th>
<th>Size(B)</th>
<th>Bsize(B)</th>
<th>Assoc</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>16K</td>
<td>8</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512K</td>
<td>16</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 3.3. Configuration of the dual-core memory hierarchy.

MediaBench [42], which are listed in Table 3.2. The cache configuration of the baseline dual-core processor simulated is listed in Table 3.3. It should be noted that while our approach can be generally applied to set-associate caches, this chapter focuses on evaluating direct-mapped caches for reducing the number of states modeled and alleviating the state explosion problem. All our experimental are conducted on an Intel processor with 1GB memory. The WCET of each benchmarks running on single-core system is given in Table 3.4.
Table 3.4. WCET of a single-core.

3.6 EXPERIMENTAL RESULTS

3.6.1 Comparing the Performance Before and After Model Simplification

To evaluate the possible advantages of the simplified model described in Section 2.4.3, we run bs on core 1 and another benchmark concurrently on core 2. The results of bs are given in Table 3.5. Note that the verifier in this experiment can only use a physical memory of 1GB. When bs is running simultaneously with itself or fibcall, we observe the same WCET results both with and without using the simplified model. However, without using the simplified model, the memory usage is significantly larger because more instructions need to be modeled, and the state-vector size is also much longer. When bs is running with two other larger benchmarks, that is, insertsort and ludcmp, however, the memory consumption is
Table 3.5. Comparison of the performance before and after using the simplified model.

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>instructions modeled</th>
<th>state-vector size(B)</th>
<th>memory usage(B)</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>before</td>
<td>after</td>
<td>before</td>
<td>after</td>
</tr>
<tr>
<td>bs</td>
<td>62</td>
<td>27</td>
<td>560</td>
<td>40</td>
</tr>
<tr>
<td>fibcall</td>
<td>51</td>
<td>13</td>
<td>560</td>
<td>36</td>
</tr>
<tr>
<td>insertsort</td>
<td>101</td>
<td>29</td>
<td>1028</td>
<td>40</td>
</tr>
<tr>
<td>ludcmp</td>
<td>278</td>
<td>28</td>
<td>3956</td>
<td>40</td>
</tr>
</tbody>
</table>

beyond 1GB without using the simplified model. In comparison, by using the simplified model, the memory usage is below 3MB, and the verifier can return the WCET results correctly. Therefore, in the following experiments, we will always use the simplified model to mitigate the state explosion problem.

### 3.6.2 Comparing the Model Checking Based Approach with Simulation and Static Analysis

To compare the effectiveness of the model checking based method (MC) with simulation (SM) and static analysis (SA) [9], we run bs on core 1 and another benchmark on core 2 which is selected from those 11 benchmarks shown in Table 3.2, including bs itself. Table 3.6 and Table 3.7 show the WCET, L1-miss ratio, and, L2-miss ratio of bs on core 1 and of another benchmark on core 2 for each method, respectively.

As can be seen from both Table 3.6 and Table 3.7, the model checking based approach can always get bigger WCET values than simulation (otherwise, it would be unsafe) but tighter WCET values than static analysis. On average, the model checking based approach improves the tightness of WCET estimation by at least
31.1%, as compared to the static analysis. We also find that the static analysis always reports the same WCET as well as L1- and L2-miss rates for bs, regardless of which benchmark is chosen to run on the second core. This is because bs is a very small benchmark which only contains 31 instructions, as can be seen from Table 3.2. As a result, all its L2 hits become L2 misses when running concurrently with most other benchmarks, except for a smaller benchmark, such as fibcall. We observe similar insensitivity behavior for most benchmarks for the model checking based approach. However, for fibcall, the model checking based approach returns smaller WCET, while the static analysis approach still conservatively reports the same WCET as bs is running with other larger benchmarks. This again indicates that the model checking based analysis is more accurate and thus can return tighter WCET results.

Since bs is among the smallest benchmarks, we also choose a larger benchmark — qsort — to run concurrently with another benchmark and then compare its estimated WCET and L1-miss rate and L2-miss rate among the model checking, simulation, and static analysis based methods. The results of core 1 and core 2 are shown in Table 3.8 and Table 3.9, respectively, where N indicates the experiment cannot be finished. We find that the model checking based approach can finish three out of the four pairs, except for qsort and ludcmp. The reason being that ludcmp is a relatively larger benchmark having more L2 hits and interfering instructions to be modeled and leading to state explosion exceeding the 1GB memory limit. In contrast, both the simulation and static analysis based
Table 3.6. Comparing the worst-case execution time, L1 miss ratio and L2 miss ratio of bs in core 1 among model checking, simulation and static analysis based approaches. (W: WCET, $R_{L1}$: L1 miss ratio, $R_{L2}$: L2 miss ratio, $\frac{W_{MC}}{W_{SIMU}}$: MC/SIMU WCET ratio, $\frac{W_{MC}}{W_{STAT}}$: MC/STAT WCET ratio)

<table>
<thead>
<tr>
<th></th>
<th>model checking</th>
<th>simulation</th>
<th>static analysis</th>
<th>$\frac{W_{MC}}{W_{SIMU}}$</th>
<th>$\frac{W_{MC}}{W_{STAT}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>libccall</td>
<td>1856 0.267 0.842</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.215</td>
<td>0.693</td>
</tr>
<tr>
<td>insertsort</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>jdfcint</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>ludcmp</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>jnlmul</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>minver</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>jnqsort</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>jnqurt</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>select</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>jncordic</td>
<td>2119 0.267 1.000</td>
<td>1456 0.268 0.632</td>
<td>3077 0.543 0.500</td>
<td>1.388</td>
<td>0.689</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td>1.372</td>
<td>0.681</td>
</tr>
</tbody>
</table>

approaches can finish all these experiments. However, we find that for the three pairs that the model checker can successfully complete, the model checking based approach is more sensitive to different programs and returns more accurate WCET results than the static analysis.

Moreover, We compare the analysis time of the model checking approach and the static analysis approach in Table 3.10. From this comparison, we could see that the average analysis time of static analysis is shorter than the model checking approach. However, the model checking approach for all the benchmarks we studied can be finished within 20 seconds, which is acceptable.
### Table 3.7. Comparing the worst-case execution time, L1 miss ratio and L2 miss ratio of the benchmark in core 2 among model checking, simulation and static analysis based approaches.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Model Checking</th>
<th>Simulation</th>
<th>Static Analysis</th>
<th>( \frac{W_{MC}}{W_{SIMU}} )</th>
<th>( \frac{W_{MC}}{W_{STAT}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( bs )</td>
<td>2119  0.267  1.000</td>
<td>1456  0.268  0.632</td>
<td>3077  0.543  0.500</td>
<td>1.388  0.689</td>
<td></td>
</tr>
<tr>
<td>( fbscall )</td>
<td>1553  0.075  1.000</td>
<td>1253  0.075  0.750</td>
<td>2657  1.000  0.512</td>
<td>1.239  0.584</td>
<td></td>
</tr>
<tr>
<td>( insertsort )</td>
<td>4105  0.012  1.000</td>
<td>3305  0.012  0.833</td>
<td>4619  0.084  0.523</td>
<td>1.242  0.889</td>
<td></td>
</tr>
<tr>
<td>( jfdctint )</td>
<td>8561  0.058  0.627</td>
<td>7809  0.058  0.555</td>
<td>13284  1.000  0.502</td>
<td>1.096  0.645</td>
<td></td>
</tr>
<tr>
<td>( ludcmp )</td>
<td>14280  0.053  0.672</td>
<td>11451  0.053  0.613</td>
<td>17943  0.098  0.567</td>
<td>1.247  0.796</td>
<td></td>
</tr>
<tr>
<td>( mactmul )</td>
<td>4958  0.017  0.795</td>
<td>4258  0.017  0.667</td>
<td>5886  0.078  0.686</td>
<td>1.164  0.842</td>
<td></td>
</tr>
<tr>
<td>( minver )</td>
<td>6247  0.078  0.706</td>
<td>5250  0.078  0.601</td>
<td>9778  0.253  0.549</td>
<td>1.190  0.639</td>
<td></td>
</tr>
<tr>
<td>( qsort )</td>
<td>10626  0.080  0.657</td>
<td>8680  0.080  0.602</td>
<td>12579  0.123  0.523</td>
<td>1.224  0.845</td>
<td></td>
</tr>
<tr>
<td>( qurt )</td>
<td>4266  0.505  0.735</td>
<td>3269  0.505  0.551</td>
<td>7743  0.119  0.578</td>
<td>1.305  0.551</td>
<td></td>
</tr>
<tr>
<td>( select )</td>
<td>9390  0.078  0.670</td>
<td>7283  0.078  0.574</td>
<td>11086  0.124  0.500</td>
<td>1.289  0.847</td>
<td></td>
</tr>
<tr>
<td>( cordic )</td>
<td>2442787  0.0001  0.561</td>
<td>2441987  0.0001  0.541</td>
<td>2656780  0.0002  0.521</td>
<td>1.001  0.919</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td>1.217  0.751</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.8. Comparing the worst-case execution time, L1 miss ratio and L2 miss ratio of \( qsort \) in core 1 among model checking, simulation and static analysis based approaches.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Model Checking</th>
<th>Simulation</th>
<th>Static Analysis</th>
<th>( \frac{W_{MC}}{W_{SIMU}} )</th>
<th>( \frac{W_{MC}}{W_{STAT}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( bs )</td>
<td>10626  0.080  0.657</td>
<td>8680  0.080  0.602</td>
<td>12579  0.123  0.523</td>
<td>1.224  0.845</td>
<td></td>
</tr>
<tr>
<td>( fbscall )</td>
<td>10426  0.080  0.639</td>
<td>8580  0.080  0.592</td>
<td>12579  0.123  0.523</td>
<td>1.215  0.829</td>
<td></td>
</tr>
<tr>
<td>( insertort )</td>
<td>11226  0.080  0.713</td>
<td>8780  0.080  0.611</td>
<td>12579  0.123  0.523</td>
<td>1.279  0.892</td>
<td></td>
</tr>
<tr>
<td>( ludcmp )</td>
<td>N  N  N</td>
<td>8780  0.080  0.611</td>
<td>13431  0.131  0.502</td>
<td>N  N</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td>1.239  0.855</td>
<td></td>
</tr>
<tr>
<td>benchmarks</td>
<td>bs on core1</td>
<td>bs on core2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bs</td>
<td>MC 10 SA 10</td>
<td>MC 10 SA 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fibcall</td>
<td>MC 10 SA 9</td>
<td>MC 10 SA 1.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertsort</td>
<td>MC 11 SA 10</td>
<td>MC 11 SA 1.35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jfdctint</td>
<td>MC 11 SA 2.34</td>
<td>MC 11 SA 1.92</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ludcmp</td>
<td>MC 11 SA 3.49</td>
<td>MC 11 SA 6.75</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>matmul</td>
<td>MC 11 SA 2.29</td>
<td>MC 11 SA 1.99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>minver</td>
<td>MC 11 SA 4.69</td>
<td>MC 11 SA 9.84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td>MC 11 SA 2.26</td>
<td>MC 11 SA 4.13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qurt</td>
<td>MC 11 SA 4.31</td>
<td>MC 11 SA 16.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>MC 11 SA 2.36</td>
<td>MC 11 SA 3.12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cordic</td>
<td>MC 11 SA 28.8</td>
<td>MC 11 SA 40.6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>MC 10.82 SA 4.98</td>
<td>MC 12.18 SA 8.11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.10. Compare the timing of model checking and static analysis approach (in seconds).
Table 3.11. Configuration I of the Dual-core Chip Memory Hierarchy.

3.6.3 Sensitivity to Cache Configurations

To study the effectiveness of the model checking based approach to different cache configurations in addition to the base cache configuration (i.e., Model I) shown in Table 3.3, we also choose two other cache models, as depicted in Table 3.11. For Model II, we reduce the size of the L1 cache to 8KB and the size of the L2 cache to 128KB, and the block size of each cache is kept the same as Model I. For Model III, we increase the L1 block size to 16B and the L2 block size to 64B, while keeping the L1 and L2 cache sizes the same as Model I. Figure 3.5 compares the WCETs of three different methods in these three models which are normalized with the simulation results. Each normalized WCET is listed for each benchmark in the x-axis which runs on core 1, and core 2 always runs bs. We observe that in all these three cache configurations, the model checking based approach returns tighter WCET than the static analysis based approach, indicating its superiority in terms of the accuracy of analysis.
3.6.4 Limitation of the Model Checking Based Method

Due to the state explosion problem of the model checker, even with the simplified model we find that the memory consumption can easily exceed 1GB for larger benchmarks. To explore the limit of the model checking based method, we systematically form 121 pairs from the 11 real-time benchmarks and run all these pairs on the dual-core processor. In all these experiments, we use a machine with 1GB memory. We observe that out of the 121 pairs, only 41 pairs are solvable, while for all other pairs the memory consumption exceeds 1GB, and the model checker cannot finish execution successfully.

Figure 3.6 shows the feasibility of WCET analysis according to the total number of conflicting instructions modeled on the dual-core system and the subset size of the L2 cache lines these instructions access. As can be seen, the WCET value can only be solved if the total number of L2 cache lines used by the interleaving instructions does not exceed 20 and the total number of total instructions modeled is no more than 49. In most cases, the fewer instructions a
benchmark has, the fewer accesses to the L2 cache it will make and the fewer interfering instructions we need to model. This will also result in smaller subset sizes of L2 cache lines used by the conflicting instructions, and these kinds of benchmarks are more likely to be solved by the model checking based approach. Therefore, while our experiments indicate that the model checking based method can improve the accuracy of WCET analysis, even with the simplified model we developed, the state explosion can still be the major limiting factor to the potential wide application of this method in worst-case multicore timing analysis.

By comparison, the static analysis approach can finish all the experiments and thus have wider applicability. Nevertheless, while real-time applications can be large, the segments that require hard real-time constraints are often small. Therefore, we believe that even in its current form, the model checking based approach can be used to analyze small hard real-time kernels to get tight analysis results. On the other hand, other approaches, such as static timing analysis technique [9], may be used to analyze other parts of code to strike a balance between the tightness of analysis and the computation cost.

3.7 CONCLUSION

This chapter presents a model checking based approach to bounding the worst-case performance of a multicore processor with shared L2 instruction caches. To alleviate the state explosion problem, we propose several techniques for reducing the memory consumption without compromising the quality of WCET analysis. Our experimental results show that the model checking based approach is
safe and improves the tightness of WCET estimation as compared to the static analysis approach [9]. However, due to the inherent complexity of multicore WCET analysis, the state explosion problem, and the physical memory constraint, this approach currently can only solve small benchmarks, while larger benchmarks with more interfering instructions will cause out-of-memory fault. However, it is possible to combine the model checking based method with the static analysis to benefit larger real-time applications.

In our future work, we would like to seamlessly integrate static analysis with the model checking based method to attain safe and tight WCET results with much smaller memory consumption and less computation time. Our idea is to exploit static analysis to identify possible worst-case paths and inter-thread interferences and then use the model checker to verify a subset of worst-case scenarios with much reduced state space. Also, we intend to study the applicability and scalability of this integrated approach to data caches and
set-associative caches and possibly for processors with more than two cores.
CHAPTER 4
EXPLOITING HYBRID SPM-CACHE ARCHITECTURES TO
REDUCE ENERGY CONSUMPTION FOR EMBEDDED
COMPUTING

4.1 CHAPTER OVERVIEW

Energy consumption has become the primary concern for microprocessor design, which is particularly crucial for battery-operated embedded systems. Cache memories have been widely used in modern processors to effectively bridge the speed gap between the fast processor and the slow memory to achieve better average-case performance and to reduce the energy consumption of accessing the main memory. However, the cache performance is heavily dependent on the history of memory accesses, as well as the cache placement and replacement algorithms, making it hard to accurately predict the worst-case execution time (WCET) [16]. For this reason, in many hard real-time and safety-critical systems, designers may simply choose to not use caches.

An alternative to the cache is the Scratch-Pad Memory (SPM) [13], which has been increasingly used in embedded processors such as ARMv6 and Motorola MCORE. The SPMs are also on-chip memories based on SRAM, which can be used to store instructions, data, or both. Unlike caches that are controlled by hardware, the mapping of program and data elements into the SPM is usually performed either by the user or the compiler. This leads to statically predictable memory
access time, which is desirable to real-time systems. Moreover, since an SPM does not need to use tag arrays, it is generally more energy- and area-efficient than a cache with the same size. On the other hand, since SPMs are totally controlled by software, they are generally less adaptable to various instruction/data access patterns that are dependent on runtime inputs. Also, because SPM allocation is done statically, the SPMs generally cannot dynamically reuse the limited on-chip SPM space as efficiently as the caches. Both these two factors may have negative impacts on the performance and total energy consumption of pure SPMs.

The recent work [43] studies hybrid SPM-Cache architectures by combining SPMs and caches to achieve both time predictability and high performance, which can widely benefit a variety of real-time and non-real-time applications. In a hybrid SPM-Cache, instead of using a single cache (or SPM) with size $N$, it employs an SPM with size $M$ ($M < N$) and a cache with size $N - M$ in parallel. Such a hybrid SPM-Cache architecture can be used to store either instructions or data, which is called Instruction Hybrid (IH) architecture or Data Hybrid (DH) architecture respectively. The hybrid SPM-Cache relies on the compiler to allocate a fraction of “profitable” instructions or data to the SPM until it is full, while the rest of instructions or data are stored in main memory, which can use the cache to exploit the temporal and space locality for improving performance.

While the prior work in [43] has quantitatively studied both the performance

---

1. Dynamic SPM allocation algorithms exist; however, these algorithms generally still need to statically determine which instruction or data objects need to be swapped in or out at runtime, which may not perfectly match the actual runtime instruction/data access patterns.
and time predictability of hybrid SPM-Caches and found their superiority over pure caches or pure SPMs in making better tradeoffs between performance and time predictability, it is not clear what is the implication of hybrid SPM-Caches on energy consumption. Compared to a pure SPM, the cache part of the hybrid SPM-Cache may consume more energy per access. Also, compared to a pure cache, the SPM part of the hybrid SPM-Cache may not reuse its space efficiently, potentially leading to more timing- and energy-consuming accesses to the main memory. Therefore, the energy consumption of the hybrid SPM-Caches is not guaranteed to be better than the traditional pure caches or pure SPMs. In this chapter, we will systematically study the energy consumption behaviors of 7 different hybrid SPM-Cache architectures, which is expected to provide important insights on energy-efficient on-chip memory design for embedded processors.

4.2 MOTIVATION

We first evaluate the energy consumption of two baseline architectures, including a pure cache and a pure SPM, which are shown in Figure 4.1. The first baseline architecture employs only an instruction cache (IC) and a data cache.
(DC), and thus is referred as the IC-DC architecture in this dissertation. The
other baseline architecture contains only an instruction SPM (IS) and a data SPM
(DS), and thus is called the IS-DS architecture. The benchmarks are selected from
MediaBench suite [42], and we use Trimaran [40] to simulate a Very Long
Instruction Word (VLIW) processor with an one-level cache or SPM. The
experiments are conducted by following the evaluation methodology and
configurations detailed in Section 4.5.

Figure 4.2 compares the on-chip memory energy consumption between the
IC-DC and the IS-DS architectures, which is normalized to that of the IS-DS
architecture. As expected, the pure SPMs are more energy-efficient than the pure
caches for all the benchmarks, because the SPMs do not have the tag arrays and
consume less energy per access. The IC-DC consumes at least 20% more on-chip
memory energy than the IS-DS, with an average of 28.9% more on-chip memory
energy dissipation.

However, on-chip memory energy is only part of the total energy. Figure 4.3
compares the total energy consumption between the IC-DC and the IS-DS
architectures, which is normalized to that of the IS-DS architecture. Unlike the
on-chip memory energy, we find that the total energy consumption results vary for
different benchmarks. For cjpeg, mesatexgen, and mpeg2dec, the IC-DC
architecture actually consumes less total energy than the IS-DS architecture, while
for the rest of benchmarks, the IS-DS is still more energy-efficient than the IC-DC.
The reason is because for cjpeg, mesatexgen, and mpeg2dec, the IC-DC can
significantly reduce the total execution time than the IS-DS, as can be seen in Figure 4.4. The energy reduced by improving the performance by the IC-DC is more than the energy increased by accessing the caches instead of the SPMs, thus leading to less total energy dissipation. For the other four benchmarks, while the IC-DC can still reduce the total execution time, the amount of reduction is less significant. In other words, the energy saving due to reduced execution time by the IC-DC is not large enough to compensate for the increased energy consumption caused by accessing the cache instead of the SPM. As a result, the IS-DS consumes less total energy than the IC-DC for these benchmarks.

Therefore, in terms of the total energy consumption, neither pure caches nor pure SPMs are always better. The total energy consumption depends on many factors, such as the total execution time, the number of accesses to different types of on-chip memories, and the energy efficiency of different types of on-chip memories. In a hybrid SPM-Cache, given that a pure SPM is more energy-efficient per access, whereas the cache is likely to conserve total energy consumption by reducing the total execution time, simply putting SPMs and caches together is not guaranteed to result in more energy-efficient computing than pure SPMs or pure caches. Therefore, it is worthy to quantitatively assess the energy consumption behavior of different SPM-Caches to understand their implications on energy. The hybrid SPM-Cache can become an attractive design option only if it can reduce the total energy dissipation as compared to both the pure cache and the pure SPM of the equivalent size.
Figure 4.2. The comparison of on-chip memory energy consumption between the IC-DC and IS-DS architectures, which is normalized to that of the IS-DS architecture.

Figure 4.3. The comparison of total energy consumption between the IC-DC and IS-DS architectures, which is normalized to that of the IS-DS architecture.

Figure 4.4. The comparison of the performance (i.e., the total number of execution cycles) between the IC-DC and IS-DS architectures, which is normalized to that of the IS-DS architecture.
4.3 BACKGROUND ON HYBRID SPM-CACHES

4.3.1 Instruction Hybrid and Data Hybrid Architectures

Since both caches and SPMs have their own advantages and disadvantages, it would be desirable to combine their advantages while avoiding their respective disadvantages. Recently we have witnessed an increasing number of studies [4, 5] on hybrid on-chip memory architectures by placing caches and SPMs together to cooperatively improve performance and/or energy efficiency, which are termed as the hybrid SPM-Caches in this dissertation. A hybrid SPM and cache model has also been used in some prototype or commercial processors such as TRIPS [1], ARM1136JF-S [2], and Nvidia Fermi [3]. This work is based on the hybrid SPM-Cache architectures studied in [43], in which caches and SPMs are placed on-chip in parallel to achieve both high performance and time predictability. Figure 4.5 shows three such hybrid SPM-Cache architectures. The first architecture has a hybrid SPM-Cache for storing instructions and a regular data cache, which is named as the IH-DC architecture. The second one has a regular instruction cache and a hybrid SPM-Cache for data, which is called the IC-DH.
architecture. The third one employs hybrid SPM-Caches for both instruction and data, which is referred as the IH-DH architecture.

In the hybrid architecture, the SPM is mapped into an address range disjoint from the off-chip main memory, but it is connected to the same address and data buses as the cache. We assume virtual memory system support as described in [44]. The instructions or data stored in the SPM are mapped to adjacent physical addresses. Therefore, an access is to the SPM if its physical address (PA) lies within the SPM address range by comparing its PA with the SPM base register. The instructions and/or data are assigned to the SPMs by software. Thus after SPM allocation, an instruction or data can be stored either in the SPM or in the off-chip memory. In the latter case, the instruction or data can be accessed by the processor through the small instruction or data cache within the hybrid SPM- Cache architecture, which can exploit the temporal and spatial locality dynamically for improving the average-case performance.

There have been many studies on efficient SPM allocation algorithms to improve either the average-case performance [45, 46, 47, 48, 49, 50] or WCET [51, 52, 53, 54, 55]. In this work, we implement a static SPM allocation algorithm for both instructions and data by exploiting profiling information. More advanced SPM allocation algorithms, including dynamic SPM allocation or optimal SPM allocation, may be used to exploit the SPM space more efficiently; however, these algorithms are more complex and generally are not scalable to larger benchmarks. Also, our experiments show that even a simple heuristic frequency-based SPM
Table 4.1. All the hybrid on-chip memories studied.

<table>
<thead>
<tr>
<th>I-Cache</th>
<th>D-Cache</th>
<th>D-Hybrid</th>
<th>D-SPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC-DC</td>
<td>IC-DC</td>
<td>IC-DH</td>
<td>IC-DS</td>
</tr>
<tr>
<td>I-Hybrid</td>
<td>IH-DC</td>
<td>IH-DH</td>
<td>IH-DS</td>
</tr>
<tr>
<td>I-SPM</td>
<td>IS-DC</td>
<td>IS-DH</td>
<td>IS-DS</td>
</tr>
</tbody>
</table>

allocation can already make hybrid SPM-Caches achieve very good energy and performance results. We will leave it as our future work to investigate other SPM allocation strategies to further enhance the energy efficiency of SPM-Caches.

In our SPM allocation method, the instructions are assigned into the instruction SPM in the unit of a basic block. All the basic blocks are sorted in the descending order based on their weights (i.e. the number of times each basic block is accessed). If a basic block has a larger weight and the total size of the instructions in it is less than or equal to the remaining size of the instruction SPM, its instructions will be assigned into the instruction SPM earlier. Similarly the data objects are assigned into the data SPM by the compiler in the descending order of the number of accesses, subject to the capacity of the data SPM.

Algorithm 2 describes our SPM allocation method in detail, where the memory object is a basic block in case of the instruction SPM and it is a data object for the data SPM allocation. The algorithm ends when all the memory objects have been checked or until there is no available space left in the SPM. The computational complexity is linear to the number of the memory objects to be checked.
Algorithm 2 SPM Allocation

1: input: the list of the memory objects $MOList$ and the empty $SPM$
2: output: the $SPM$ with the memory objects assigned
3: begin
4: Sort_By_Frequency_Descending_Order($MOList$)
5: $MO = MOList.head$
6: while $MO$ is not null do
7:     if $SPM.avail\_size > 0$ then
8:         if $MO.size \leq SPM.avail\_size$ then
9:             assign $MO$ into $SPM$
10:                $SPM.avail\_size = SPM.avail\_size - MO.size$
11:         end if
12:     end if
13:     $MO = MO.next$
14: else
15:     break
16: end while
17: end
4.3.2 Additional Hybrid Architectures

In addition to the proposed hybrid SPM-Caches, there are also other types of hybrid on-chip memory architectures, for example using a cache for instructions and an SPM for data. Generally, depending on the use of a cache, an SPM, or a hybrid SPM-Cache for storing either instructions or data, there are totally 9 different combinations as shown in Table 4.1. Among these 9 different architectures, two are homogeneous: IC-DC is the traditional cache only architecture, and IS-DS is the traditional SPM only architecture, which are two extremes. Besides the three hybrid SPM-Caches depicted in Figure 4.5, the other four hybrid architectures include Instruction Cache and data SPM (IC-DS), Instruction SPM and Data Cache (IS-DC), Instruction Hybrid and data SPM (IH-DS), and Instruction SPM and Data Hybrid (IS-DH). The first two use a cache or an SPM to store either instructions or data but not both. The latter two involve the hybrid SPM-Cache, in addition to a regular SPM, to store either instructions or data.

4.4 ENERGY MODELS

The main components in a cache include the decoder, the tag memory array, the tag column multiplexers, the tag sense amplifiers, the tag comparators, the tag output drivers, the data memory array, the data column multiplexers, the data sense amplifiers, and the data output drivers, while the SPM only needs the decoding and the column circuitry logic. Thus, the SPM is essentially more energy-efficient than the cache of the same size.
Based on the cache components, Kamble and Ghose proposed an analytical energy dissipation model for the low power cache in [56], which has been widely used in the research of cache energy estimation. In Equation 4.1, the total amount of energy dissipated by a cache can be expressed as the sum of four components, including bit-line dissipations, word-line dissipations, dissipations in output lines, and dissipations in input lines. The energy model of an SPM can largely reuse this equation but needs to remove the consideration of tag bits in the calculation of $E_{\text{bit}}$ and $E_{\text{word}}$. Also, the SPM energy estimation does not need to consider the address output in the calculation of $E_{\text{output}}$ due to the direct connection between SPMs and the processor. We have adopted Kamble and Ghose’s model [56] and calculated the energy consumption of both SPMs and caches by using CACTI [57]. The total energy consumption, including the on-chip memory energy, the processor energy, and the main memory energy, is computed by using the energy consumption evaluation tool EPIC-Explorer [58].

$$E_{\text{dissipation}} = E_{\text{bit}} + E_{\text{word}} + E_{\text{output}} + E_{\text{ainput}}$$  \hspace{1cm} (4.1)

4.5 EVALUATION METHODOLOGY

We use Trimaran compiler/simulator framework [40] to implement and evaluate all the 9 different on-chip memory architectures on a VLIW processor. The baseline processor has 2 integer ALUs, 2 floating-point ALUs, 1 branch predictor, 1 load/store unit, and 1-level on-chip memory. Our energy consumption is based on EPIC-Explorer [58]. The evaluation frame is shown in Figure 4.6. The
Figure 4.6. Energy evaluation framework.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Code Size (bytes)</th>
<th>Data Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cjpeg</td>
<td>jpeg image compression</td>
<td>50960</td>
<td>135565</td>
</tr>
<tr>
<td>djpeg</td>
<td>jpeg image decompression</td>
<td>46060</td>
<td>26508</td>
</tr>
<tr>
<td>epic</td>
<td>an image compression program</td>
<td>19608</td>
<td>329611</td>
</tr>
<tr>
<td>mesamipmap</td>
<td>OpenGL graphics clone: using mipmap quadrilateral</td>
<td>71240</td>
<td>39397</td>
</tr>
<tr>
<td>mesatexgen</td>
<td>OpenGL graphics clone: texture mapping</td>
<td>98792</td>
<td>45074</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>MPEG digital compressed format decoding</td>
<td>30252</td>
<td>389669</td>
</tr>
<tr>
<td>rasta</td>
<td>A program for speech recognition</td>
<td>55384</td>
<td>132369</td>
</tr>
</tbody>
</table>

Table 4.2. Salient characteristics of benchmarks.

On-chip memory energy for SPM-Caches consists of cache energy and SPM energy. The total energy consumption includes both the processor (including the on-chip memory) and the main memory energy consumption.

By default, we use a 16KB on-chip memory, which can be an SPM, a cache, or a hybrid SPM-Cache. The parameters of the cache include: 32B block size, 4-way set-associative, and LRU replacement policy. We assume both a cache hit and an SPM access take 1 cycle and a memory access takes 20 cycles. We do not use any L2 cache in the experiments.

The benchmarks are selected from MediaBench [42] (also referred as media benchmarks in this dissertation). The salient characteristics of all benchmarks are
Figure 4.7. The comparison of on-chip memory and total energy consumption among all 9 architectures, which is normalized to the on-chip and total energy respectively of the IS-DS architecture.

4.6 EXPERIMENTAL RESULTS

4.6.1 Energy Results

Figure 4.7 (a) compares the on-chip memory energy consumption among all these 9 architectures, which is normalized to that of the IS-DS. We observe that all the seven hybrid architectures consume less on-chip memory energy than that of the IC-DC. Among them, the IH-DC, the IH-DH, and the IH-DS have much smaller energy consumption than the IS-DS, while the rest of the hybrid architectures consume on-chip memory energy either more than or close to that of the IS-DS. This is because in a hybrid SPM-Cache such as an IH or DH, the small SPM is more energy-efficient to access than the larger pure SPM, i.e., the IS or DS. Similarly, the small cache in the IH or DH is more energy-efficient than the larger pure cache, i.e., the IC or DC. Since instructions are accessed every clock cycle, the instruction access energy dominates the total on-chip memory energy.
dissipation. Consequently, the hybrid architectures that employ the IH, including the IH-DC, the IH-DH, and the IH-DS all consume much less on-chip memory energy than the IS-DS. Moreover, since the DH is more energy-efficient than either the DS or the DC, the IH-DH is the best among these three. Also, the IH-DS is superior to the IH-DC because the DS is more energy-efficient than the DC. On average, the IH-DH consumes 45.7% and 74.7% less on-chip memory energy than that of the IS-DS and the IC-DC respectively.

Our next experiment compares the total energy consumption among all these 9 architectures, and the results normalized to the IS-DS are shown in Figure 4.7 (b). As we can see, on average, the IC-DC and most hybrid SPM-Caches consume less total energy than the IS-DS architecture, though it varies for different benchmarks. This trend is quite different from the on-chip memory energy consumption, because the IS-DS consumes less on-chip memory energy than the IC-DC and a few other hybrid architectures such as the IC-DH, IC-DS and IS-DC. This is because although an SPM is more energy-efficient per access than a cache
of the equivalent size, the significant performance improvement by using the cache or hybrid SPM-Caches can lead to large energy reduction. The normalized execution cycles of these nine architectures are shown in Figure 4.8 (a). As we can see, all the hybrid SPM-Caches, as well as the pure cache architecture, can achieve performance either better than or close to the IS-DS architecture, because the SPM is totally controlled by the software and cannot dynamically reuse its space as efficiently. As a result, for those benchmarks whose performance can be significantly improved by the IC-DC or other hybrid SPM-Caches, the total energy consumption can be reduced.

From Figure 4.7 (b), we also observe that the three hybrid architectures that are efficient in on-chip memory energy, including the IH-DC, the IH-DH, and the IH-DS, all consume less total energy than the IC-DC for most benchmarks. Among these three, the IH-DH is the most energy-efficient, which consumes 22% and 16% less total energy than that of the IS-DS and the IC-DC respectively. The IH-DC is the second best. It consumes less total energy than the IH-DS, because the DC in the IH-DC can reduce the execution time better than the DS in the IH-DS. We also find that two other hybrid architectures, i.e., the IC-DS and the IS-DC, actually consume 3.4% and 2.6% more total energy than that of the IC-DC. This is because the pure SPMs used in these two architectures lead to more accesses to the main

---

Enhancing the SPM allocation algorithm, for example by exploiting dynamic SPM allocation may alleviate this problem, which will be studied in our future work. However, in general, compiler-based allocation still have the fundamental limit that it does not have perfect knowledge of runtime instruction/data access patterns, which may be varied based on runtime inputs.
memory and longer execution time, thus increasing the total energy consumption.

4.6.2 Energy-Delay Product Results

Figure 4.8 (b) compares the Energy-Delay Product (EDP) of these architectures, which is normalized to the EDP of the IS-DS. Again, we find that the IH-DH architecture has lower EDP than the IS-DS, IC-DC, or any other hybrid SPM-Cache. On average, the IH-DH reduces the EDP by 38.1% and 16.4% as compared to that of the IS-DS and the IC-DC respectively, indicating that the IH-DH is superior by considering both energy consumption and performance. Among other hybrid on-chip memory architectures, both the IH-DC and the IC-DH can achieve EDP less than the IC-DC for all the benchmarks, because the IH (DH) is more energy-efficient than the IC (DC). Between the IH-DC and the IC-DH, the IH-DC can reduce the EDP by 4.7% more than the IC-DH, because the IH-DC is more efficient in reducing the instruction access energy consumption, which dominates the total energy consumption for the media benchmarks.

The other four hybrid architectures, i.e., the IC-DS, the IH-DS, the IS-DC, and the IS-DH, on average, still have much smaller EDP than the IS-DS, because compared to the pure SPM, the cache or the hybrid SPM-Cache used in those architectures can help lower the total energy consumption by reducing the execution time. However, on average, these four architectures have larger EDP than that of the IC-DC, because the pure instruction or data SPM used in these architectures lead to longer execution time and hence more total energy consumption than a pure instruction or data cache. Among these four
architectures, both the IH-DS and the IS-DH have lower EDP values than those of
the IC-DS and the IS-DC, indicating that using the IH (DH) can make better
balance of performance and energy consumption than the IC (DC) of an
equivalent size. Additionally, while the IH-DS consumes less energy than the
IC-DC, the performance of the IH-DS is worse than that of the IC-DC, leading to
larger EDP results.

In summary, by considering both energy consumption and performance, we
believe that the IH-DH, the IH-DC, and the IC-DH are the three top hybrid
on-chip memory architectures to make good tradeoffs between performance and
energy dissipation. Among them, the IH-DH is the best, and the IH-DC is better
than the IC-DH because the instruction accesses, not data accesses dominate both
the execution time and energy dissipation for these benchmarks.

4.6.3 Sensitivity Study on SPM and Cache Partitioning

In the sensitivity study experiments, we focus on studying the three hybrid
architectures that can achieve better EDP than the IC-DC, including the IH-DC,
the IC-DH, and the IH-DH architectures. In our experiments, for each of these
three hybrid SPM-Caches, we try two different partitions between the cache and
the SPM, while keeping the total hybrid SPM-Cache size fixed (i.e. 16KB by
default). Generally, for an N-byte hybrid SPM-Cache i with the partition of a
M-byte cache and an (N-M)-byte SPM, we refer it as the i-M scheme. For
example, for a 16K IH-DC architecture with a 4KB instruction cache and a 12KB
instruction SPM, it is denoted as IH-DC-4K (note that the DC, i.e. the data
cache, is the default size, which is 16KB). As the cache simulator requires that the cache size must be a power of 2, for a total SPM-Cache size of 16KB, we can only try a 4KB cache with a 12KB SPM, and an 8KB cache with an 8KB SPM.

Figure 4.9 (a) shows the on-chip memory energy consumption of the IS-DS, the IC-DC, and the IH-DC, IC-DH, and IH-DH architectures with two different partitions, which is normalized to that of the IS-DS architecture. As we can see, all these three hybrid SPM-Caches with different partitions have on-chip memory energy consumption less than that of the IC-DC, and both the IH-DC and the IH-DH consume much less on-chip memory energy than the IS-DS as well. However, we also find that different partitions between the SPM and cache can significantly impact the on-chip memory energy dissipation. In general, for all these three hybrid on-chip memory architectures with a 16KB total size, an even partition, i.e., an 8KB SPM with an 8KB cache, seems to be more energy-efficient than an uneven partition, i.e., a 12K SPM and a 4K cache. For example, on average, the IH-DC-8K consumes 21.1% less on-chip memory energy than the IH-DC-4K; the IC-DH-8K consumes 2.4% less on-chip memory energy than the IC-DH-4K; and the IH-DH-8K consumes 23.5% less on-chip memory energy than the IH-DH-4K. This is because in the hybrid SPM-Cache, we find the number of accesses to the SPM is much more than the number of accesses to the cache in both partitions, and a larger SPM consumes more energy per access than a smaller

\footnote{We did not choose a 2KB cache and a 14KB SPM because the partition is too unbalanced. While we attempted to try a 12KB cache with a 4KB SPM, our cache simulator could not simulate a 12KB cache correctly.}
SPM. However, we also observe that pure caches without any SPM, i.e. the IC-DC architecture, is not as energy-efficient as these hybrid SPM-Caches, because a larger cache has diminishing return of cache miss reduction, while significantly increasing the energy consumption per access as compared to a smaller cache or SPM.

Figure 4.9 (b) compares the total energy consumption of the IS-DS, the IC-DC, and the IH-DC, IC-DH, and IH-DH architectures with two different partitions, which is normalized to that of the IS-DS architecture. Similar to the trend of the on-chip memory energy consumption, we find an 8KB SPM with an 8KB cache is more energy-efficient than a 12KB SPM with a 4KB cache for the IH-DC, IC-DH, and IH-DH architectures. On average, the IH-DC-8K consumes 4.2% less total energy than the IH-DC-4K; the IC-DH-8K consumes 0.5% less total energy than the IC-DH-4K; and the IH-DH-8K consumes 4.7% less energy than the IH-DH-4K. Therefore, for the benchmarks we studied, an even partition of the size between the SPM and the cache is more energy-efficient than the unbalanced partition.

4.6.4 Sensitivity Study on SPM and Cache Sizes

We also study the impact of different SPM and cache sizes on the energy consumption of various on-chip memory architectures. Figure 4.10 (a) compares the on-chip memory energy consumption between the IS-DS and the IC-DC architectures with the total size varying from 8KB to 16KB, 32KB and 64KB, which is normalized to the on-chip memory energy consumption of the 8KB IS-DS
Figure 4.9. The comparison of on-chip memory and total energy consumption among the IS-DS, the IC-DC, and the IH-DC, IC-DH, and IH-DH architectures with two different SPM and cache partitions, which is normalized to the on-chip memory and total energy consumption respectively of the IS-DS architecture.

Figure 4.10. The comparison of on-chip memory and total energy consumption of the IC-DC and IS-DS architectures with their total size varying from 8KB to 16KB, 32KB, and 64KB, which is normalized to the on-chip memory and total energy consumption respectively of the 8KB IS-DS architecture.
architecture. As expected, a larger SPM or cache leads to more on-chip memory energy dissipation, and in all 4 different sizes, the IS-DS is more energy-efficient than the IC-DC of the same size.

However, the IC-DC can potentially reduce the total execution time, which may save the total energy consumption. As depicted in Figure 4.10 (b), the IC-DC consumes less total energy than the IS-DS when the total size is 8KB or 16KB. However, as the total size increases to 32KB and 64KB, the IS-DS actually consumes less total energy. This is because increasing the IC-DC leads to diminishing performance improvement, while consuming much more energy for cache accesses. Overall, we find the 8KB IC-DC is the most energy-efficient as compared to other IS-DS and IC-DC architectures with various sizes.

Figure 4.11 (a) shows the on-chip memory energy consumption of the IH-DC architectures with the total size varying from 8KB to 16KB, 32KB, and 64KB, which is normalized to that of the 8KB IS-DS architecture. The 8KB IH-DC consumes less on-chip memory energy than the 8KB IS-DS. As the size of the IH-DC increases, it consumes more on-chip memory energy.

The total energy consumption of IH-DC with different sizes is shown in Figure 4.11 (b). As the total size increases from 8KB to 16KB, the total energy consumption decreases for most benchmarks due to the reduction of the total execution time. However, when the size further increases to 32KB and 64KB, the total energy consumption becomes larger, because of the increased energy consumption per access to larger caches and SPMs and the diminishing return of
Figure 4.11. The comparison of on-chip memory and total energy consumption of the IH-DC architectures with their total size varying from 8KB to 16KB, 32KB, and 64KB, which is normalized to the on-chip memory and total energy consumption respectively of the 8KB IS-DS architecture.

performance improvement. On average, the 16KB IH-DC consumes 14%, 9.1%, and 30% less total energy than the 8KB, 32KB, and 64KB IH-DCs respectively.

The energy efficiency of the IC-DH is also heavily dependent on its size. As we can see from Figure 4.12 (a), the IC-DH consumes more on-chip energy as the size increases. The total energy consumption of the IC-DH also become larger as the size increases for most benchmarks, as can be seen from Figure 4.12 (b). On average, the 8KB IC-DH consumes 26.9%, 6.3%, 23.1%, and 59.3% less total energy than the 8KB IS-DS, and the 16KB, 32KB, and 64KB IC-DHs respectively.

Figure 4.13 (a) shows the on-chip memory energy consumption of the IH-DH architectures with different sizes, which is normalized to that of the 8KB IS-DS. As we can see, compared to the 8KB IS-DS, the 8KB IH-DH consumes much less on-chip memory energy. However, as the size of the IH-DH increases, the on-chip memory energy also increases. This is because a larger SPM and a larger cache in
Figure 4.12. The comparison of on-chip memory and total energy consumption of the IC-DH architectures with their total size varying from 8KB to 16KB, 32KB, and 64KB, which is normalized to the on-chip memory and total energy consumption respectively of the 8KB IS-DS architecture.

Figure 4.13. The comparison of on-chip memory and total energy consumption of the IH-DH architectures with their total size varying from 8KB to 16KB, 32KB, and 64KB, which is normalized to the on-chip memory and total energy consumption respectively of the 8KB IS-DS architecture.
the IH-DH consume more energy per access, both of which lead to more on-chip energy dissipation.

Figure 4.13 (b) shows the total energy consumption of the IH-DH architectures with different sizes. Although a larger IH-DH generally results in more on-chip memory energy, the total energy consumption may not increase linearly with the size, because a larger IH-DH can lead to better performance. On average, we find that the 8KB IH-DH consumes the least total energy, which is 1.3%, 8.9%, and 26% less than the total energy consumption of the 16KB, 32KB, and 64KB IH-DHs respectively. The reason is that for media benchmarks, the 8KB IH-DH can already achieve very good performance by reducing the number of accesses to main memory (i.e. cache misses), and further increasing the size leads to diminishing return of performance but much larger energy consumption per access to the on-chip SPM-Caches. Therefore, for embedded systems, it is important to profile the applications to find out the best configuration for hybrid SPM-Caches to minimize the total energy consumption.

4.7 RELATED WORK

Previous studies on SPM mainly treated it as an alternative to the cache memory for achieving time predictability or improving performance and energy efficiency. Steinke et al. [50] developed a compiler-based method to assign program and data objects into the SPM to reduce the dynamic energy consumption. Kandemir et al. [59] and Chen et al. [60] studied compiler-based approaches to reduce leakage energy of SPMs. Several SPM allocation algorithms have also been
proposed to improve the average-case performance [46, 47, 48, 49] or WCET [51, 52, 53, 54, 55]. However, all these research efforts generally focused on pure SPMs, not on hybrid SPM-Caches.

There are only a few research efforts to combine the SPM with the cache. In particular, Wang et al. [61] proposed a method to remap portions of data segments into SPM space to reduce cache conflict misses and they also introduced an SPM controller with a tightly coupled DMA to minimize the swapping overhead of dynamic SPM allocation. This work however, is limited to the IC-DH architecture, whereas in this chapter, we have explored 7 different hybrid SPM-Cache architectures and comparatively evaluated their energy consumption behaviors. Especially, our work indicates that using the hybrid SPM-Cache for instructions rather than data is more effective at reducing the total energy consumption and EDP for the mediabench we studied.

Xue et al. [62] proposed a hybrid SPM consisting of Non-Volatile Memory (NVM) and SRAM to achieve energy efficiency. In contrast, our study is focused on hybrid architectures by combining an SPM with a cache based on the same SRAM technology.

Panda et al. [45] investigated partitioning scalar and array variables into SPM and data cache to minimize the execution time for embedded applications. Verma et al. [63] studied an instruction cache behavior based SPM allocation technique to reduce the energy consumption. Recently, Cong et al. [5] proposed an adaptive hybrid cache by reconfiguring a part of the cache as software-managed
SPM to improve both performance and energy efficiency. Kang et al. [4] introduced a synergetic memory allocation method to exploit SPM to reduce data cache pollution.

Comparing to all these studies that basically use SPMs to boost the performance and/or energy efficiency of caches, the hybrid SPM-Cache architectures proposed in this dissertation treat both SPMs and caches equally, though for different objectives. More specifically, the hybrid architectures in our study rely on the SPM to ensure a basic level of time predictability [43], while using caches to improve the average-case performance or energy efficiency by exploiting the access locality for instructions and data that are not stored into the SPM. In this work, we do not change the SPM allocation used in [43] to preserve the time predictability that can be achieved by the SPM. Moreover, prior efforts only study a limited hybrid model by combining a cache and an SPM for either instructions or data only. In contrast, in this work, we have systematically and comparatively evaluated all the seven different hybrid on-chip memory architectures to understand their implication on energy consumption.

4.8 CONCLUSIONS

While cache memories are usually effective at improving the average-case performance, they are harmful to time predictability. In contrast, SPMs are time-predictable and more energy-efficient per access, but generally are less adaptive to runtime instruction/data access patterns and may result in inferior performance. Built upon the prior work in [43] to study the performance and time
predictability of hybrid SPM-Cache architectures, this chapter investigates the energy consumption of seven different SPM-Caches. We find that all these seven hybrid on-chip memory architectures consume less energy than the pure SPM based architecture. Three hybrid SPM-Cache architectures, including the IH-DC, the IH-DH, and the IH-DS, can reduce the total energy consumption than the IC-DC. By considering both energy consumption and performance, the IC-DH, IH-DC, and IH-DH can achieve energy-delay product less than both the pure cache-based and SPM-based architectures.

Among all the hybrid on-chip memory architectures, our evaluation indicates that the IH-DH architecture is the best in terms of both total energy consumption or EDP. More specifically, on average, the IH-DH architecture can reduce the total energy consumption by 22% and 16%, and reduce the EDP by 38.1% and 16.4% as compared to that of the IS-DS and the IC-DC respectively. Therefore, in addition to reconciling performance and time predictability as revealed in [43], our study demonstrates that the hybrid on-chip memory architectures, in particular the IH-DH, can also make better tradeoffs between performance and energy consumption, making it a very attractive design option for real-time and embedded systems.
CHAPTER 5

REDUCING WORST-CASE EXECUTION TIME OF HYBRID
SPM-CACHES

5.1 CHAPTER OVERVIEW

Cache memories have been widely used in modern processors to effectively bridge the speed gap between the fast processor and the slow memory to achieve good average-case performance. However, cache performance is heavily dependent on the history of memory accesses and the cache placement and replacement algorithms, making it hard to accurately predict the worst-case execution time. In contrast, Scratch-Pad Memories are time-predictable because the allocation is controlled by software and the latency to access data from the SPM is fixed. However, SPMs generally are not adaptive to runtime instruction and data access patterns, and thus may lead to inferior average-case performance.

A cache memory or an SPM alone can only benefit performance or time predictability respectively, not both. For real-time systems, it is attractive to enhance both performance and time predictability. Recently we have witnessed an increasing number of studies [4, 5] on hybrid on-chip memory architectures by placing caches and SPMs together to cooperatively improve performance and/or energy efficiency, which are termed as the hybrid SPM-Caches in this . A hybrid cache and SPM model has also been used in some prototype or commercial processors such as TRIPS [1], ARM1136JF-S [2], and Nvidia Fermi [3]. Recent
studies show that a hybrid SPM-Cache can greatly improve the performance [4] or energy efficiency [5], both of which are potentially beneficial to embedded systems, including hard real-time systems.

However, for hard real-time systems to safely and reliably exploit hybrid SPM-Cache architectures, it is crucial to be able to predict the worst-case execution time for real-time tasks running on processors with hybrid SPM-Caches. A safely and accurately estimated WCET value provides the basis for schedulability analysis. Moreover, it is attractive to optimize (i.e. reduce) the WCET for those systems. The reduced WCET of a task can give more flexibility to the real-time scheduler and hence may enable the system to meet stringent timing constraints that would otherwise be impossible. Moreover, reducing the WCET of a task may allow the embedded processor to use a lower clock rate or to place itself into a low-power mode during the idling periods (while meeting the deadlines) to save energy consumption. However, prior studies on hybrid SPM-Caches [4, 5, 45, 63] mainly focus on improving performance and/or energy efficiency, and the impacts of these techniques on predicting and reducing WCET are uncertain.

To benefit hard real-time systems, this explores four different SPM allocation algorithms to reduce WCET for the hybrid SPM-Cache architecture. Compared to existing allocation algorithms for pure SPMs [45, 46, 47, 48, 49, 50, 51, 52, 54, 55], in a hybrid SPM-Cache architecture, the existence of the parallel cache requires that an intelligent SPM allocation algorithm needs to consider the cache effects, particularly its impact on WCET, so that both the SPM and the cache can
effectively and cooperatively reduce the WCET for programs running on the hybrid SPM-Cache architecture. To this end, we have developed a WCET-oriented and cache-aware SPM allocation algorithm, called the Enhanced Hybrid SPM-Cache Allocation (EHSA) algorithm. Our experimental results indicate that the EHSA algorithm can outperform other three algorithms to reduce WCET for the hybrid SPM-Cache with little or even positive impact on the average-case performance.

The main contributions of this chapter include the following.

- We propose a WCET-oriented and cache-aware SPM allocation algorithm for the hybrid SPM-Cache architecture. To the best of our knowledge, this is the first paper to study how to exploit both cache and SPM to cooperatively minimize WCET for hard real-time systems.

- We have also explored three other SPM allocation algorithms for the hybrid SPM-Cache with different complexity and effectiveness, including the Frequency-based SPM Allocation (FSA), the Hybrid SPM-Cache Allocation (HSA), and the Longest Path based Allocation (LPA).

- We have implemented and evaluated all the four SPM allocation algorithms. Our experiments indicate that the EHSA algorithm outperforms other allocation algorithms in reducing WCET. Interestingly, we also observe that the EHSA algorithm can even achieve better ACET than other SPM allocation algorithms for many benchmarks with various SPM and cache configurations.
The rest of this chapter is organized as follows. Section 5.2 introduces the SPM-Cache architecture. Section 5.3 presents different SPM allocation algorithms designed for the hybrid SPM-Cache architecture. The evaluation methodology is given in Section 5.4, and the experimental results are shown in Section 5.5. We discuss related work in Section 5.6, and finally make conclusions in Section 5.7.

5.2 BACKGROUND ON HYBRID SPM-CACHES

The SPM is a small, high-speed on-chip SRAM memory that is mapped into an address space disjoint from the off-chip memory, as shown in Figure 5.1 (a). Both the SPM and the cache can be accessed in a single processor cycle, which is much faster than an access to the off-chip memory. However, while the SPM can guarantee a single-cycle access time, an access to the cache is dependent on whether the access is a hit or a miss. Also, caches are usually controlled by hardware, whereas SPMs are managed by software.

A hybrid SPM-Cache is an on-chip memory architecture by placing a cache and an SPM in parallel to store instructions and/or data. A hybrid SPM-Cache for storing instructions are depicted in Figure 5.1 (b). When the CPU accesses an instruction, the interface circuitry of the SPM determines whether or not the referenced memory address maps into the address space of SPM. If that is the case, it issues the signal $S_{\text{HIT}}$ and takes control of the address and data buses. If not, this instruction will be accessed through the regular memory hierarchy. Specifically, if the instruction hits in the cache, then the signal $C_{\text{HIT}}$ is generated and this instruction is directly passed to the processor; otherwise, this instruction...
needs to be fetched from the main memory.

In a hybrid SPM-Cache architecture, a certain fraction of instructions and/or data can be loaded into the SPM by software, subject to the available SPM space. After SPM allocation, instructions and/or data that are not stored in the SPM can be accessed through the traditional memory hierarchy (i.e. different levels of caches and main memory), which can exploit the temporal and spatial locality dynamically to improve the average-case performance. The existence of a small SPM in the SPM-Cache architecture can reduce the number of conflict misses in a pure cache. Also, the accesses to the instructions/data stored in the SPM can be more energy-efficient than accessing the cache. Therefore, compared to a pure cache-based architecture, the hybrid SPM-Cache architecture can potentially combine the advantages of both caches and SPMs to achieve better performance, energy efficiency, and/or time predictability to benefit a wide range of applications.

A hybrid SPM-Cache also has significant advantages over a pure SPM. Since an SPM is controlled by software, the instructions/data stored into the SPM may not match the actual instructions/data accessed at runtime, especially if the program has paths that are dependent on runtime inputs. In contrast, a hybrid SPM-Cache can leverage the cache part to dynamically reuse its space for the instructions/data that are not stored into the SPM. While there are schemes to load instructions/data dynamically into the SPM at runtime [64], the decisions are still made statically (i.e., at the compilation time), which is unlikely to match the performance of the hardware-controlled cache in terms of exploiting runtime
instruction/data locality. Actually, the recent work in [43] has shown that using the hybrid SPM-Caches for both instructions and data can lead to better time predictability than the pure cache, and better performance than the pure SPM. Therefore, the hybrid SPM-Cache architecture is an attractive on-chip memory design option for embedded processors.

However, it is still a new and challenging problem to exploit both the cache and the SPM collaboratively in the hybrid architecture to minimize WCET to benefit hard real-time systems or a mix of hard, soft, or non-real-time tasks. As the first step to exploiting the hybrid SPM-Caches deterministically for hard real-time systems, our study focuses on studying an instruction SPM-Cache as depicted in Figure 5.1(b), in which both the SPM and the cache are used for storing instructions only. We plan to explore hybrid SPM-Caches for data accesses in our future work.

5.3 SPM ALLOCATION FOR HYBRID SPM-CACHES

The SPM allocation algorithms can be either ACET or WCET-oriented. The ACET-oriented algorithms are not aware of the worst-case path (WC-path), thus their effectiveness in reducing the WCET is not ensured. In contrast, the WCET-oriented SPM allocation algorithms specifically target the worst-case path, which can reduce the WCET more effectively, though it may have negative impact on ACET. Prior works on WCET-oriented SPM allocation [51, 52, 54, 55], however, are designed for pure SPMs. Since these algorithms are not aware of the parallel cache in the hybrid SPM-Cache architecture, they may lead to suboptimal
(a) Division of address space between SRAM (i.e. SPM) and DRAM (i.e. main memory).

(b) Hybrid SPM-Cache architecture for instructions, where I-Cache denotes the instruction cache, and I-SPM denotes the instruction SPM.

Figure 5.1. The hybrid SPM-Cache system architecture.
Table 5.1. Four SPM allocation algorithms studied in this chapter.

<table>
<thead>
<tr>
<th></th>
<th>cache-unaware</th>
<th>cache-aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCET-unaware</td>
<td>FSA</td>
<td>HSA</td>
</tr>
<tr>
<td>WCET-aware</td>
<td>LPA</td>
<td>EHSA</td>
</tr>
</tbody>
</table>

results. In this chapter, we systematically explore four different SPM allocation algorithms as shown in Table 5.1. These algorithms are classified based on two criteria: 1) whether or not the algorithm is aware of the cache, and 2) whether or not the algorithm is aware of the WCET. For all the four SPM allocation algorithms, we propose to allocate SPM space at the Basic Block (BB) granularity. Our compiler will generate book-keeping instructions after SPM allocation to ensure the correct transfer of control between instructions stored in the SPM and the main memory [53].

As we can see in Table 5.1, the Frequency-based SPM Allocation (or FSA) is a traditional SPM allocation algorithm that is aware of neither the cache nor WCET. The hybrid SPM-Cache Allocation (or HSA) is aware of the cache, but is not WCET-oriented. By comparison, the Longest-Path based Allocation (or LPA) is WCET-oriented, but is not cache-aware. The Enhanced HSA (or EHSA) is both WCET-oriented and cache-aware. The details of these algorithms are presented in the following subsections respectively.
5.3.1 Frequency-based SPM Allocation

Since our study focuses on studying the hybrid SPM-Cache for instructions, we propose to allocate SPM space at the Basic Block (BB) granularity to avoid generating too many bookkeeping instructions, which may degrade instruction locality and performance.

The FSA algorithm allocates SPM space based on the access frequency of each basic block extracted from the profiling of simulated traces. Specifically, the heuristic is that the basic blocks are stored into the SPM based on the decreasing order of their frequencies, until the SPM is full or there is no enough space left to hold another basic block. This algorithm is straightforward, which is described in Algorithm 3. The FSA algorithm only needs to check each basic block once and its complexity is dominated by sorting all the basic blocks based on their access frequencies. Therefore the time complexity of FSA is $O(N \log(N))$, where $N$ is the number of basic blocks in the given program.

5.3.2 Longest-Path based Allocation

The longest-path based allocation algorithm is a greedy approach to target SPM allocation on the worst-case path. We adopt the approach used in [65]. The algorithm first constructs weighted Directed Acyclic Graphs (DAGs) from the Control Flow Graph (CFG) of the program, based on which it identifies the longest path of the DAG. The algorithm then sorts all the basic blocks on the longest path according to their execution frequencies from the highest to the lowest, which are obtained through profiling. The basic block on the longest path
Algorithm 3 Frequency-based SPM Allocation Algorithm

1: begin
2: allocations ← empty;
3: run simulation and get the frequency of each basic block;
4: sort basic blocks by frequency from the highest to lowest in f_list;
5: while isNotFull(SPM) && isEmpty(f_list) do
6:     get the first basic block B from f_list;
7:     if sizeof(B) ≤ sizeof(available SPM space) then
8:         allocate B into SPM;
9:     end if
10:    remove B from f_list;
11: end while
12: return allocations
13: end

with a larger frequency will be allocated into the SPM first. Algorithm 4 describes the LPA allocation algorithm. The time complexity of this algorithm is $O(N + E) + O(N \log N)$, where $E$ is the number of edges in the CFG and $O(N + E)$ is the complexity to find the longest path.

5.3.3 Hybrid SPM-Cache Allocation

Both the FSA and LPA do not consider the cache that is in parallel with the SPM in the hybrid SPM-Cache. In contrast, the hybrid SPM-Cache allocation algorithm is designed to take into account both the SPM and the cache to cooperatively reduce WCET. We use Abstract Interpretation (AI) to do the cache analysis [10], which can classify memory references into three classes: Always Hit(AH), Always Miss(AM) and Not Classified (NC). The basic idea of the HSA algorithm is to only allocate the basic blocks with more AM and/or NC instructions.
Algorithm 4 Longest Path based Allocation Algorithm

1: begin
2: allocations ← empty;
3: calculate the longest path $P$;
4: sort BBs on $P$ by frequency in decreasing order in $lf$ list;
5: while isNotFull($SPM$) && isNotEmpty($lf$ list) do
6: get the first basic block $B$ from $lf$ list;
7: if sizeof($B$) $\leq$ sizeof(available $SPM$ space) then
8: allocate $B$ into $SPM$;
9: end if
10: remove $B$ from $lf$ list;
11: end while
12: return allocations
13: end

into the $SPM$, while leaving basic blocks with more AH instructions into the cache. This is because for basic blocks with more AH instructions, their worst-case performance in the cache is already guaranteed to be very good. Therefore, the $SPM$ space can be saved for other basic blocks to improve the WCET or performance more efficiently.

The HSA algorithm begins by doing static cache analysis, based on which all the basic blocks can be classified and some of them can be put into two lists: the AM list and the NC list. The AM list stores the basic blocks with Always Miss instructions, and those basic blocks are sorted by the descending order of their numbers of AM instructions. For the basic blocks with the same number of AM instructions, the HSA algorithm sorts them by the ascending order of their numbers of the Always Hit instructions. If two basic blocks have the same number of AM and AH instructions, they are then sorted by the descending order of their
numbers of NC instructions (otherwise; it does not matter which basic block is
allocated first). The NC_list is used to store basic blocks with NC instructions but
no AM instructions. For the basic blocks with the same number of NC
instructions, the HSA algorithm sorts them by the ascending order of their
numbers of AH instructions.

Algorithm 5 gives the pseudo-code of the HSA allocation algorithm. After
getting the AM_list and NC_list (lines 5-6), the algorithm selects and allocates the
first available basic block from the AM_list if its block size does not exceed the
available SPM size (lines 8-13). If the block size is too large to be stored into the
SPM, the algorithm then removes it from the AM_list and marks it (lines 14-15).
After each SPM allocation, the instruction reference classification may be changed.
For example, if a conflicting instruction is stored into the SPM, an AM instruction
in the cache may become AH. Thus, after allocating SPM space to each basic
block, the algorithm re-invokes the cache timing analysis to categorize the
instruction accesses and then allocates the remaining SPM space by bypassing the
marked blocks (lines 8-16). Note that by setting b_allocate to be \texttt{TRUE} (at line
12), the innermost while loop will exit, so the cache analysis and sorting logic in
the outermost while loop will be used again. If there is no basic block with AM
instructions left, the algorithm then checks the NC_list and allocate SPM space to
blocks on that list in the same way as it does for the AM_list (lines 17-25). The
algorithm stops if the SPM is full or if there is no unmarked basic block. In
Algorithm 5, lines 26-28 are used to avoid endless loops.
The time complexity of HSA is heavily dependent on the complexity of static cache timing analysis (see line 5). The cache analysis based on abstract interpretation has been shown to run efficiently [10], assuming its complexity is $\alpha$. For each allocation iteration, the HSA needs to conduct the cache analysis, then sorts the basic blocks by either the number of AM instructions or the number of NC instructions. Therefore, the time complexity of HSA becomes $O(N_s \times (\alpha + N \log N))$, where $N_s$ is the size of the SPM.

5.3.4 Enhanced Hybrid SPM-Cache Allocation

While the HSA algorithm is aware of the cache, it is not aware of the WC-path. Therefore, it may happen that some AM instructions stored in the SPM may have no impact on the WCET at all. To further decrease the WCET specifically, we propose to enhance the HSA by considering the worst-case path during SPM allocation, which is called the EHSA algorithm. In this algorithm, after finding every candidate basic block to be allocated into the SPM, it will do WCET analysis to ensure this block is on the worst-case path. If the attempted SPM allocation leads to the WCET reduction, then this allocation is confirmed. Otherwise, the allocation is undone (because this block is not the WC-path) and the algorithm continues to find the next available allocation. Algorithm 6 describes the EHSA allocation algorithm in detail. Compared to the pseudo-code

---

It should be noted that the HSA algorithm is generally independent of any specific cache timing analysis approach such as the abstract interpretation based analysis[10]. Thus enhancing the efficiency of cache timing analysis can also improve the efficiency of the HSA algorithm.
Algorithm 5  Hybrid SPM cache Allocation Algorithm

1:  begin
2:  allocations ← empty;
3:  availableBB ← total number of basic blocks;
4:  while isNotFull(SPM) && availableBB > 0 do
5:      do cache analysis;
6:      sort basic blocks into AM list and NC list;
7:      b allocate ← FALSE;
8:      while isNotEmpty(AM list) && b allocate == FALSE do
9:         get the first available basic block Bm from AM list;
10:        if sizeof(Bm) <= sizeof(available SPM space) then
11:           allocate Bm into SPM;
12:           b allocate ← TRUE;
13:        end if
14:        remove Bm from the AM list and marked;
15:        availableBB − −;
16:      end while
17:      while isNotEmpty(NC list) && b allocate == FALSE do
18:         get the first available basic block Bn from NC list;
19:        if sizeof(Bn) <= sizeof(available SPM space) then
20:           allocate Bn into SPM;
21:           b allocate ← TRUE;
22:        end if
23:        remove Bn from the NC list and marked;
24:        availableBB − −;
25:      end while
26:      if b allocate == FALSE then
27:         use FSA to allocate one BB from the basic blocks left;
28:      end if
29:  end while
30:  return allocations
31:  end
of the HSA depicted in Algorithm 5, the EHSA algorithm inserts logic to conduct WCET analysis (line 12 and line 28 respectively) and to check the potential impact of SPM allocation on WCET (line 13 and line 29 respectively). The rest of the logic is similar to the HSA algorithm.

The EHSA algorithm used WCET analysis to make SPM allocation WCET aware. Since our WCET analysis is based on ILP, the complexity of the ILP solver based on the simplex algorithm [66] is $O(2^N)^2$.

### 5.3.5 WCET Analysis of Hybrid SPM-Caches

To conduct WCET analysis for the hybrid SPM-Cache architecture, we extend the ILP-based method proposed by Li et al. [68]. We use ILP to calculate the maximum value of the total execution time (i.e. the objective function) under three types of linear constraints: structural constraints, functional constraints, and cache constraints. The structural constraints are derived from the program’s CFG, and the functional constraints are provided by the loop bounds and other path information, both of which are the same with those in [68]. However, when we build the cache constraints from the cache conflict graph [68], we do not consider the basic blocks allocated into the SPM. Therefore, the cache timing analysis for the hybrid SPM-Cache is actually less complex than the analysis for a pure cache, \footnote{However, for small-scale problems, it can achieve expected time polynomial to $n$, $d$, and $1/\sigma$ [67], denoted as $f(n, d, 1/\sigma)$, where $n$ is the size of the problem, $d$ is the dimension, and $\sigma$ is the standard deviation. Therefore, the complexity of EHSA for small-scale problems can be $O(N_s \times (\alpha + N \log N) + N_s \times f(n, d, 1/\sigma))$.}
Algorithm 6 Enhanced Hybrid SPM cache Allocation

1: begin
2: allocations ← empty;
3: availableBB ← total number of basic blocks;
4: get the initial WCET value from WCET analysis;
5: while isNotFull(SPM) && availableBB > 0 do
6:   do cache analysis;
7:   sort basic blocks into AM list and NC list;
8:   b_allocate ← FALSE;
9:   while isNotEmpty(AM list) && b_allocate == FALSE do
10:      get the first available basic block Bm from AM list;
11:      if sizeof(Bm) <= sizeof(available SPM space) then
12:         do WCET analysis with this allocation;
13:         if WCET is reduced then
14:            allocate Bm into SPM;
15:            b_allocate ← TRUE;
16:            remove Bm from the AM list and marked;
17:            availableBB ← –;
18:            end if
19:            remove Bm from the AM list;
20:      else
21:         remove Bm from the AM list and marked;
22:         availableBB ← –;
23:      end if
24:   end while

90
while isNotEmpty(NC_list) && b_allocate == FALSE do

get the first available basic block Bn from NC_list;

if sizeof(Bn) <= sizeof(available SPM space) then

do WCET analysis with this allocation;

if WCET is reduced then

allocate Bn into SPM;

b_allocate = TRUE;

remove Bn from the NC_list and marked;

availableBB -= -;

end if

remove Bn from the NC_list;

else

remove Bn from the NC_list and marked;

availableBB -= -;

end if

end while

if b_allocate == FALSE then

use FSA to allocate one BB from the basic blocks left;

end if

end while

return allocations

end
not only because the cache in the hybrid SPM-Cache is typically smaller, but also because less instructions need to be stored into the cache and be modeled in the cache conflict graph. An ILP solver is then used to solve all the ILP equations and inequalities to compute the WCET based on the objective function. More information about the ILP-based WCET analysis can be found at [68].

5.4 EVALUATION METHODOLOGY

Our evaluation framework is depicted in Figure 5.2. In this framework, we first use Trimaran compiler [40] to compile a benchmark into the intermediate representation (IR) targeting a Very Long Instruction Word (VLIW) processor that is supported by Trimaran. Based on the IR information, we can construct the control flow graph that will be used by both the WCET analyzer and the SPM allocator. The WCET analyzer generates all the structural constraints, functional constraints, and cache constraints with the on-chip memory configuration and SPM
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Path Info</th>
<th>Description</th>
<th>Code Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>single path</td>
<td>cyclic redundancy check computation on 40 bytes of data</td>
<td>520</td>
</tr>
<tr>
<td>edn</td>
<td>single path</td>
<td>finite impulse response (FIR) filter calculations</td>
<td>3452</td>
</tr>
<tr>
<td>matmult</td>
<td>single path</td>
<td>matrix multiplication of two $20 \times 20$ matrices</td>
<td>480</td>
</tr>
<tr>
<td>ndes</td>
<td>multiple paths</td>
<td>complex embedded code</td>
<td>3452</td>
</tr>
<tr>
<td>cnt</td>
<td>multiple paths</td>
<td>Counts non-negative numbers in a matrix</td>
<td>408</td>
</tr>
<tr>
<td>fir</td>
<td>multiple paths</td>
<td>a 700 items long sample impulse response filter</td>
<td>356</td>
</tr>
</tbody>
</table>

Table 5.2. Benchmarks used in our experiments.

allocation information. A commercial ILP solver-CPLEX [69] is used to solve the ILP problem to compute the estimated WCET. All the four allocation algorithms are implemented in the SPM allocator. In case of the EHSA, the SPM allocator needs to call the WCET analyzer to make the SPM allocation WCET-aware. In addition, we use Trimaran simulator to report the average-case performance.

We select 6 real-time benchmarks from Mälardalen WCET benchmark suite [41] for the experiments. The salient characteristics of all benchmarks are shown in Table 5.2. It should be noted that Malardalen benchmark suite includes both single-path and multiple-path programs. In our evaluation, the first three benchmarks are single-path programs, while the last three benchmarks have multiple paths.

In our experiments, the baseline processor has 2 integer ALUs, 2 float ALUs, 1 branch predictor, 1 load/store unit, and 1-level on-chip memory. To focus on the instruction on-chip memory, the data cache is assumed to be perfect. Since the WCET benchmarks are typically very small, as can be seen in Table 5.2, we need
to use smaller configurations for both the cache and the SPM\(^3\). By default, we assume a 64B instruction cache and a 64B SPM. The SPM takes 1 clock cycle to access. The parameters of the cache include: 16B block size, direct-mapped, and LRU replacement policy. A cache hit takes 1 cycle and a memory access takes 20 cycles.

5.5 EXPERIMENTAL RESULTS

5.5.1 Safety and Accuracy of WCET Analysis

We first evaluate the safety and accuracy of our WCET analysis for the hybrid SPM-Cache architecture with the four different SPM allocation algorithms. Figure 5.3 shows the estimated WCET reported by the developed WCET analyzer, which is normalized to the simulated WCET through simulation on different inputs. As we can see, for all the SPM allocation algorithms, the estimated WCET is more than the simulated WCET, indicating that our WCET analysis can safely estimate the upper bound of the execution time.

For single-path benchmarks, we find that the estimated WCET is very close to the simulated WCET. The slight overestimation is mainly due to the conservative assumption used in static cache analysis. For example, the unclassified cache accesses, which are assumed to be misses in the worst case, may actually be hits during the simulation. For multiple-path programs, however, we observe larger differences between the estimated WCET and the simulated WCET. One reason is

\(^3\)It should be noted that this is not uncommon in the research of WCET analysis; otherwise, the benchmarks can easily all fit into a regular cache with several kilo bytes.
still due to the conservative cache analysis, which can become worse for programs with more paths. Another possible reason is that for a multiple-path program, it becomes much harder to simulate (or observe) the actual worst-case path by limited simulation, unless we can exhaust all possible paths for all inputs in our simulation, which is prohibitively expensive in computation. On average, the estimated WCET is 21.3%, 19.1%, 15.9%, and 16.1% larger than the corresponding simulated WCET for the FSA, HSA, LPA, and EHSA algorithms respectively.

In particular, we find our WCET analyzer is more accurate for the HSA, LPA, and EHSA algorithms as compared to the FSA. This is because all these three SPM allocation algorithms can reduce WCET by exploiting the worst-case path information and/or the worst-case cache performance information. The overestimation of the WCET analysis is also reduced by improving the worst-case cache performance and by reducing the execution time on the worst-case path. Given the conservative nature of worst-case execution time analysis, and the difficulty to obtain the actual WCET through simulation for multiple-path programs, we believe our WCET analysis approach is reasonably accurate.
Figure 5.4 compares the WCET of the four different SPM allocation algorithms, which is normalized to the WCET of the FSA algorithm. As we can see, all the three other algorithms can achieve better (i.e. smaller) WCET than the FSA. Among these three algorithms, the HSA can only reduce WCET slightly as compared to the FSA. This is because the HSA is not aware of the WC-path. While the HSA can exploit the cache analysis information, the basic blocks with large Always Miss numbers but not on the worst-case path may be selected to use the SPM, which do not help the WCET reduction.

The LPA can reduce the WCET more than the HSA. For single-path benchmarks, i.e., *crc*, *edn*, and *matmult*, the LPA achieves the same WCET as the FSA, because there is only a single path in those programs, which is also the longest path. For benchmarks with multiple paths, including *ndes*, *cnt*, and *fir*, the LPA is much better than both the HSA and the FSA, because the LPA can focus on allocating the basic blocks on the WC-path into the SPM to effectively reduce the WCET.

As expected, the EHSA is the best among all these four algorithms, because it is not only WCET-oriented, but also cache-aware. The EHSA is especially effective for the multiple-path benchmarks. For example, the EHSA can reduce the WCET of *ndes* by 8.1% as compared to the FSA. For the single-path benchmarks, the EHSA can still achieve better WCET by exploiting the cache analysis information. On average, the EHSA reduces the WCET of the FSA by 5.4% for all
the benchmarks.

Figure 5.4. Comparing the WCET of different SPM allocation algorithms with the default configuration, which is normalized to the WCET of the FSA algorithm.

5.5.3 Average-Case Performance Results

Figure 5.5 gives the ACET of the four different SPM allocation algorithms, which is normalized to the ACET of the FSA algorithm. We find that for the single-path benchmarks, the FSA and the HSA have exactly the same ACET, because the longest path is the same as the average-case path in this case. For single-path benchmarks, we also observe that both HSA and EHSA can actually achieve better ACET. The reason is that both the HSA and the EHSA can reduce the worst-case cache misses by exploiting the SPM, which also reduce the average-case cache misses for the single-path programs.

For the multiple-path benchmarks, however, we find that the FSA can actually achieve slightly better ACET than the other three algorithms. This is because the FSA algorithm allocates SPM space based on the access frequency of each basic block, which is obtained through profiling. Consequently, the basic blocks selected by the FSA are likely to be on the average-case path in our
simulation, thus benefiting ACET. By comparison, both the LPA and the EHSA exploit WC-path information, and the HSA uses the worst-case cache timing analysis information, all of which are not guaranteed to improve ACET. Nevertheless, as can be seen from Figure 5.5, the average-case performance degradation by these three algorithms is not significant. On average, for the multiple-path benchmarks, the HSA, LPA, and EHSA degrade the ACET of the FSA by only 2.3%, 3.0%, and 1.8% respectively.

![Figure 5.5. Comparing the ACET of different SPM allocation algorithms with the default configuration, which is normalized to the ACET of the FSA algorithm.](image)

### 5.5.4 Sensitivity Study

Since the average-case performance of the cache and the SPM can be dependent on their sizes, we also conduct experiments to study the sensitivity of different SPM allocation algorithms with respect to various SPM and cache sizes, while trying to keep their total size fixed. Figure 5.6 compares the WCET of different SPM allocation algorithms for the hybrid SPM-Cache with a 96B SPM and a 32B cache, which is normalized to the WCET of the FSA algorithm with the same SPM-Cache configuration. As we can see, with a smaller cache and a larger
SPM, the HSA can reduce the WCET more significantly as compared to a 64B SPM and a 64B cache. This is because there are likely more capacity and conflict misses in a smaller cache. Therefore, keeping the Always Hit instructions into the cache can help improve the worst-case cache performance. For multiple-path benchmarks, the LPA can achieve even better (i.e. smaller) WCET than the HSA because the LPA is WCET-oriented. The EHSA is superior to both the HSA and the LPA for all benchmarks. For ndes, the EHSA can reduce the WCET by 10.1% as compared to the FSA. On average, the EHSA can reduce the WCET of the FSA by 9.4% for multiple-path benchmarks and by 7.3% for all benchmarks, indicating the effectiveness of this approach in reducing WCET.

Figure 5.6. Comparing the WCET of different SPM allocation algorithms for the hybrid SPM-Cache with a 96B SPM and a 32B cache, which is normalized to the WCET of the FSA algorithm.

Figure 5.7 compares the ACET of different SPM allocation algorithms for the hybrid SPM-Cache with a 96B SPM and a 32B cache. Compared to the 64B SPM and the 64B cache configuration, we find that on average, both the HSA and EHSA can improve ACET by 4.3% and 4.6% respectively better than that of the FSA. A possible reason is that with a smaller cache and a larger SPM, more AM
and NC instructions will be allocated into the SPM, which are more likely to be on
the average-case path for the multiple-path programs and are definitely on the
average-case path for the single-path benchmarks. Actually for the latter, we
observe up to 9.1% ACET reduction. One exception is the benchmark fir, for
which the ACET of the HSA and the EHSA is 4.1% and 3.4% respectively worse
than that of the FSA. For this benchmark, we find many instructions allocated
into the SPM are not on the simulated average-case path, leading to worse ACET.
Also, we find that compared to the FSA, the LPA leads to the same ACET for
single-path benchmarks, while its effect on ACET for multiple-path benchmarks
varies, depending on how many selected blocks are on both the WC-path and the
average-case simulated path.

Figure 5.7. Comparing the ACET of different SPM allocation
algorithms for the hybrid SPM-Cache with a 96B SPM and a
32B cache, which is normalized to the ACET of the FSA algo-

We also run experiments with a smaller SPM and a larger cache. More
specifically, we use a 32B SPM and a 128B cache. Figure 5.8 shows the WCET of

4We have attempted to use a 32B SPM and a 96B cache to keep the total size of the SPM-Cache
fixed. However, our cache simulator requires that the cache size must be a power of 2. Thus we
increase the cache size to 128B.
different SPM allocation algorithms for the hybrid SPM-Cache with a 32B SPM and a 128B cache. For such a relatively small SPM and a large cache, we find that while the HSA, LPA, and EHSA algorithms can still achieve better WCET than the FSA, the amount of WCET reduction is smaller. This is because, with a smaller SPM, fewer basic blocks can be stored into the SPM, thus there is less room for an SPM allocation algorithm to improve WCET. Also, with a larger cache, there are less cache misses, thus limiting the benefit of cache-aware SPM allocation.

Figure 5.8. Comparing the WCET of different SPM allocation algorithms for the hybrid SPM-Cache with a 32B SPM and a 128B cache, which is normalized to the WCET of the FSA algorithm.

Figure 5.9 presents the ACET of different SPM allocation algorithms for the hybrid SPM-Cache with a 32B SPM and a 128B cache, which is normalized to the ACET of the FSA algorithm with the same SPM-Cache configuration. Again, we observe that for single-path programs, the LPA and the FSA have the same ACET, and both the HSA and the EHSA can attain better ACET than the FSA. However, the amount of ACET reduction by both the HSA and the EHSA is smaller as compared to the ACET reduction for the 96B SPM and 32B cache.
This is similar to the trend of WCET reduction we have observed. The reason is that as the SPM size decreases and the cache size increases, there are less cache misses and less SPM space to reduce possible cache misses.

Figure 5.9. Comparing the ACET of different SPM allocation algorithms for the hybrid SPM-Cache with a 32B SPM and a 128B cache, which is normalized to the ACET of the FSA algorithm.

5.6 RELATED WORK

Several researchers have explored hybrid models consisting of both cache memory and SPM. Panda et al. [45] investigated partitioning scalar and array variables into SPM and data cache to minimize the execution time for embedded applications. Verma et al. [63] studied an instruction cache behavior based SPM allocation technique to reduce the energy consumption. Cong et al. [5] proposed an adaptive hybrid cache by reconfiguring a part of the cache as software-managed SPM to improve both performance and energy efficiency. Kang et al. [4] introduced a synergetic memory allocation method to exploit SPM to reduce data cache pollution. All these prior studies have focused on improving performance and/or energy efficiency. In contrast, this chapter investigates how to reduce the WCET of hybrid SPM-Caches.
There have also been many studies to exploit SPMs for better performance, energy efficiency, or time predictability. For example, a number of SPM allocation algorithms have been proposed to improve the average-case performance [46, 47, 48, 49], energy efficiency [50], or WCET [51, 52, 53, 54, 55]. However, to the best of our knowledge, all the prior studies on WCET-oriented SPM allocation have focused on pure SPMs, which may lead to suboptimal results because they cannot deterministically and cooperatively leverage the cache in the hybrid SPM-Cache architecture to minimize WCET. To the best of our knowledge, the EHSA approach proposed in this chapter is the first work to study WCET-oriented and cache-aware SPM allocation.

The WCET-oriented and cache-aware SPM allocation is based on prior work on WCET analysis [68], especially static timing analysis for caches based on Abstract Interpretation [10]. There are several other methods for cache timing analysis, for example the static cache simulation techniques [70, 71]. Also, a number of researchers have examined WCET-aware compiler optimizations. While this dissertation does not intend to have a complete survey of related work in this area, the studies close to this work include WCET-driven code positioning [72, 73], procedure positioning [74], and memory content selection to benefit from the instruction cache [75]. However, all these prior studies are focused on the instruction cache, which are not aware of the instruction SPM that is available in the hybrid SPM-Cache architecture.
5.7 CONCLUSIONS

In this chapter, we have explored four SPM allocation algorithms that differ by whether or not they are aware of the WCET and/or the cache. The FSA algorithm allocates SPM space based on the access frequency of each basic block from profiling, whereas the LPA attempts to allocate basic blocks with high access frequencies on the WC-path. Both the HSA and the EHSA algorithms can exploit the worst-case cache analysis information; however, the EHSA ensures that only basic blocks on the WC-path are allocated to the SPM. We have also extended the ILP-based timing analysis method [68] to predict the WCET for the hybrid SPM-Cache architecture, and our experiments indicate that the developed WCET analyzer is safe and reasonably accurate.

We have implemented all the four SPM allocation algorithms on our evaluation framework based on Trimaran compiler/simulator infrastructure [40]. Our evaluation indicates that the EHSA algorithm, which is both WCET-oriented and cache-aware, can achieve the best WCET for all benchmarks under all SPM-Cache configurations we have evaluated. The EHSA is especially more effective to reduce WCET with a smaller cache and a larger SPM. While the EHSA may lead to degradation of the average-case performance for some multiple-path benchmarks, its impact is insignificant. Actually, on average, the EHSA leads to better ACET than the FSA in our sensitivity study. Therefore, exploiting the cache and the SPM cooperatively is important for the hybrid SPM-Cache to enhance WCET, and its impact on ACET can be either positive or insignificant.
Also, our experiments show that the LPA algorithm outperforms both the FSA and the HSA in reducing WCET for most multiple-path programs. Compared to the EHSA, the LPA algorithm has significantly less time complexity and can run much faster in practice. On the other hand, the HSA algorithm can reduce the WCET for both single-path and multiple-path programs as compared to the LPA, although the time complexity of the HSA is much higher due to the invocation of cache time analysis. Among these four algorithms, the FSA has the least time complexity and can reduce the ACET more effectively for some multiple-path benchmarks under certain SPM-Cache configurations, but it is the least effective algorithm to reduce WCET.

In our future work, we would like to explore the hybrid SPM-Caches for storing data as well. Moreover, to use the SPM space more efficiently, we plan to study an optimal SPM allocation algorithm by using integer linear programming and explore dynamic SPM allocation algorithms for the hybrid SPM-Cache architecture for minimizing WCET without significantly impacting performance and energy consumption.
CHAPTER 6
CACHE-AWARE SPM ALLOCATION FOR MAXIMIZING PERFORMANCE ON HYBRID SPM-CACHE ARCHITECTURE

6.1 CHAPTER OVERVIEW

To address the growing gap between CPU and main memory performance, cache memories have been widely used in modern processors. Cache memories are based on the principle to make the common cases fast. However, there is no guarantee that the cache can also benefit the worst-case execution time (WCET), which is crucial for hard real-time systems. Actually, the cache performance is heavily dependent on the history of memory accesses, as well as the cache placement and replacement algorithms, making it hard to accurately predict the worst-case execution time. The WCET analysis for data caches is even harder, because the addresses of data accesses to the heap may not be predicted statically.

Scratch-Pad Memory is an alternative on-chip memory to the cache. SPM is also a small on-chip memory based on fast SRAM, but is directly and explicitly managed at the software level, either by the compiler or by the developer. Due to its area and energy efficiency, SPM has been increasingly used in embedded processors such as ARMv6, Motorola MCORE, Nvidia’s PhysX PPU (Physical Processing Unit) and the Cell multiprocessor jointly developed by IBM, Sony, and Toshiba. The SPM is particularly useful for real-time systems, because the SPM allocation is controlled by software and the latency to access the SPM is fixed,
both of which can be statically predicted. However, SPMs generally are not adaptive to runtime instruction and data access patterns, and thus may lead to inferior average-case performance.

Recently there are an increasing number of studies on hybrid on-chip memory architectures by placing caches and SPMs together to cooperatively improve performance, energy efficiency, or time predictability. Cong et al. [5] proposed an adaptive hybrid cache by reconfiguring a part of the cache as software-managed SPM to improve both performance and energy efficiency. Kang et al. [4] introduced a synergetic memory allocation method to exploit the SPM to reduce data cache pollution. Zhang et al. [6] studied hybrid on-chip memory architecture that can leverage the SPM to achieve time predictability while exploiting the cache to improve the average-case performance.

The HSC models have also been used in some prototypes or commercial processors such as TRIPS [1], ARM1136JF-S [2], and Nvidia Fermi [3]. For example, in the Nvidia Fermi architecture, the L1 on-chip SRAM memory is configurable to support both shared memory (i.e. SPM) and caching of local and global memory operations.

The hybrid SPM-Cache architecture brings new challenges and opportunities to further enhance the performance and energy efficiency of the on-chip memory. Traditionally, the SPM allocation, including both static and dynamic allocation, mainly focuses on the SPM alone. These cache-unaware SPM allocation algorithms are unlikely to harness the full potential of the hybrid SPM and cache.
To use the aggregate SPM and cache space more efficiently, we believe the SPM allocation for the hybrid SPM-Cache architecture must be aware of the cache performance to maximally optimize the execution time or energy consumption.

To this end, we design and comparatively evaluate 4 different SPM allocation algorithms. The first one is the Frequency-based SPM Allocation (FSA), which is not aware of the cache and is used as the baseline. The other three algorithms are all cache-aware, but exploit cache information in different ways. The Hybrid SPM-Cache Allocation (HSA) exploits cache profiling information. It tries to allocate the memory objects with the largest cache misses into the SPM. The remaining two algorithms are both based on the Stack Distance Analysis (SDA) [11, 12]. The Greedy Stack Distance based Allocation (GSDA) is a greedy algorithm, whereas the Optimal Stack Distance based Allocation (OSDA) is an optimal algorithm by using model checking. More details of these algorithms can be seen in the rest of the chapter. As the first step to exploiting the tight interaction between SPM allocation and cache performance in the HSC architecture, our study focuses on studying an instruction SPM-Cache as depicted in Figure 5.1(b). In the instruction HSC, both the SPM and the cache are used for storing instructions only. We plan to explore hybrid SPM-Caches for data accesses in our future work.

This chapter makes three main contributions as the follows.

- First, we propose a novel unified HSC analysis framework based on stack distance, in which the SPM is treated as additional “virtual” ways for the
cache and thus can be included in the stack distance analysis.

- Second, we develop a heuristic based GSDA algorithm with polynomial time complexity and an optimal OSDA algorithm based on model checking, both of which are built upon the unified stack distance analysis framework.

- Third, we have implemented and compared all the four SPM allocation algorithms, and find that all the three cache-aware algorithms attain superior performance than the FSA algorithm. In particular, the HSA and the GSDA improve the performance by 9% and 11% respectively as compared to the FSA. The OSDA always achieves the best performance, but requires significantly more memory space and longer running time and may not be scalable for larger benchmarks. The GSDA can achieve performance either the same as or very close to that of the OSDA.

6.2 RELATED WORKS

To use the SPM efficiently, researchers have done extensive study on SPM allocation. All the existing approaches can be classified into two classes: static allocation [45, 46, 48, 49, 50, 52] and dynamic allocation [47, 51, 54, 55, 76, 77, 78, 79, 80]. In static allocation, once an instruction or data is loaded into the SPM, its space cannot be allocated to other instructions or data. By comparison, in dynamic allocation, the SPM space can be reused by other instructions or data under the compiler’s control. While the dynamic SPM allocation can use the SPM space more efficiently, transferring instructions or data
from the main memory to the SPM takes time and energy, which must be considered by the dynamic allocation algorithms. All these prior studies on SPM allocation, however, have focused on pure SPMs, which may lead to suboptimal results for HSCs because they cannot cooperatively leverage the cache in parallel.

Several researchers have also explored hybrid models consisting of both cache memory and SPM. Panda et al. [45] investigated partitioning scalar and array variables into the SPM and the data cache to minimize the execution time for embedded applications. Kang et al. [4] introduced a synergetic memory allocation method to exploit the SPM to reduce data cache pollution for real-time tasks. In contrast, our study focuses on allocating SPM space to store instructions for the hybrid SPM-Cache architecture.

Verma et al. [63] studied an instruction cache behavior based SPM allocation technique to reduce the energy consumption. Their approach was based on the cache conflict graph, which used profiling information (i.e. weights) to get the number of conflicting misses for different instructions. They then proposed an Integer Linear Programming (ILP) based solution to minimize the number of the conflicting edges in the conflict graph and the overall energy consumption of the system. However, approximation is used to linearize the problem so that it is solvable by the ILP solver.

Despite the breadth of existing studies, our work differs from all the investigations above in the following two aspects.

First, prior studies on cache-aware SPM allocation [4, 45, 63] examine cache
and SPM separately. The SPM allocation is done after cache performance profiling and analysis, with the goal to minimize different kinds of cache misses for different objective functions. While this separation of concern reduces the complexity, it does not consider the impact of SPM allocation on the original cache profiling or analysis. For example, if instructions $A$ and $B$ are conflicting with each other. While allocating $B$ into the SPM may reduce the conflicting misses for $A$, it may change another instruction, say $C$, from hit to miss due to spatial locality if both $C$ and $B$ are stored in the same cache block. Also, if $A$ is conflicting with multiple instructions, including $B$, while allocating $B$ into the SPM may not reduce $A$’s number of misses, it may have positive impact in the future if other conflicting misses are also stored into the SPM. Since each SPM allocation may affect the cache hit/miss estimation, we develop a novel stack distance based analysis framework for HSC by treating the SPM as additional “virtual” ways for the cache, enabling us to study the interactions between SPM allocation and cache performance in a finer granularity.

Second, we have studied both heuristic and optimal cache-aware algorithms in this work. The first heuristic algorithm HSA allocates SPM space after cache profiling. The second heuristic algorithm GSDA is based on the SDA framework, which can take into account the impact of each SPM allocation on the cache performance and thus can achieve better performance. The optimal algorithm OSDA leverages model checking. While the OSDA in general demands much more memory space and execution time, it provides the basis to assess the effectiveness...
of the heuristic based algorithms.

6.3 BASIC SPM ALLOCATION ALGORITHMS

We first develop two basic SPM allocation algorithms: the Frequency-based SPM Allocation algorithm and the Hybrid SPM-Cache Allocation algorithm. While the former is cache-unaware, the later is cache-aware. Both algorithms are heuristic based, and can be implemented efficiently.

6.3.1 Frequency-based SPM Allocation

The FSA is based on the access frequency of each memory object of a given program. The heuristic is that the memory objects are stored into the SPM based on the decreasing order of their access frequencies, until the SPM is full or there is no enough space left to hold a memory object. This algorithm is straightforward, which is described in Algorithm 7.

The FSA algorithm only needs to check each memory object once and its complexity is dominated by sorting all the memory objects based on their access frequencies. Therefore the time complexity of FSA is $O(N \log(N))$, where $N$ is the number of memory objects in the given program.

6.3.2 Hybrid SPM-Cache Allocation

The HSA is a cache-aware SPM allocation algorithm designed for the hybrid SPM-Cache architecture. The idea is to allocate the memory objects with larger cache misses into the SPM in order to reduce the total number of cache misses.
Algorithm 7 FSA Algorithm

1: begin
2: allocations ← empty;
3: run simulation and get the frequency of each memory objects;
4: sort memory objects by frequency from high to low in fList;
5: while isNotFull(SPM) && isEmpty(fList) do
6:   get the first memory object B from fList;
7:   if sizeof(B) ≤ sizeof(available SPM space) then
8:      allocate B into SPM;
9:   end if
10:  remove B from fList;
11: end while
12: return allocations
13: end

We first simulate the benchmark and get the number of cache misses of each memory object without the SPM. Then we allocate the memory objects in the descending order of their numbers of cache misses. Algorithm 8 describes the HSA allocation algorithm.

The HSA algorithm only needs to check each memory object once and its complexity is dominated by sorting all the memory objects according to their numbers of cache misses. Thus the time complexity of the HSA is also $O(N \log(N))$. 
Algorithm 8 HSA Algorithm

1: begin
2: allocations ← empty;
3: profiling to get the number of cache misses of all memory objects;
4: sort the memory objects in m_list;
5: while isNotFull(SPM) && isNotEmpty(m_list) do
6:   get the first memory object B from m_list;
7:   if sizeof(B) ≤ sizeof(available SPM space) then
8:     allocate B into SPM;
9:   end if
10:  remove B from m_list;
11: end while
12: return allocations
13: end

6.4 STACK DISTANCE BASED SPM ALLOCATION

ALGORITHMS

6.4.1 Stack Distance

Stack distance [11] [12] has been widely used in cache performance analysis. Stack distance of a memory access can be defined as the number of accesses to unique addresses made since the last reference to the requested data [11]. The stack distance has an interesting property: in a $d$-way associative LRU cache, a reference with stack distance $s < d$ will hit, and a reference with stack distance $s \geq d$ will miss.

6.4.2 Stack Distance Analysis for The HSC Architecture

While the stack distance is very useful to analyze the cache performance, it is not directly applicable to the hybrid SPM-Cache architecture, as the instructions
stored into the SPM will disrupt the cache behavior. Therefore, we propose to treat the SPM as additional “virtual” ways of the cache to enable unified analysis for the HSC. Given an SPM with \( N \) words and an \( d \)-way set associative cache, we model the hybrid SPM-Cache as a virtually \((N + d)\)-way set-associative cache initially. This is because up to \( N \) words can be stored into the SPM, in addition to the \( d \) ways in the cache, to reduce the conflict misses. Thus stack distance based analysis can treat the HSC as a virtually \((N + d)\)-way set-associative cache.

However, unlike a regular \((N + d)\)-way set-associative cache, after allocating each word from the SPM, the set associativity is reduced by 1, until all the SPM space is allocated and the cache becomes a regular \( d \)-way set-associative cache.

For each memory access \( a_i \), we define the binary variable \( x_i \) as the following:

\[
  x_i = \begin{cases} 
  0, & \text{if } a_i \text{ is in the SPM} \\
  1, & \text{if } a_i \text{ is not in the SPM}
  \end{cases} \tag{6.1}
\]

We use the binary variable \( m_i \) to indicate whether the memory access is a cache miss or not.

\[
  m_i = \begin{cases} 
  0, & \text{if } a_i \text{ is in the SPM or cache hit} \\
  1, & \text{if } a_i \text{ is not in the SPM and cache miss}
  \end{cases} \tag{6.2}
\]

Then the total number of cache misses \( M \) of the benchmark after the SPM allocation can be calculated by Equation 6.3:
The stack distance of each memory access can be calculated according to the trace of all memory accesses. Assuming that stack distance of \( a_i \) is \( s_i \) before the SPM allocation, if \( a_i \) is in the SPM after the SPM allocation, then \( x_i = 0 \). If \( a_i \) is not in the SPM after the SPM allocation, then \( x_i = 1 \) and the stack distance of \( a_i \) becomes \( s'_i \). If \( s'_i \geq d \), then memory access \( a_i \) is a miss and \( m_i = 1 \).

In the hybrid SPM-Cache, \( s'_i \) is dependent on \( s_i \). Specifically, if \( s_i < d \), we definitely have \( s'_i < d \), so \( a_i \) is a cache hit, and \( m_i = 0 \). If \( d + N \leq s_i < \infty \), because we can only allocate up to \( N \) instructions into the SPM, \( s'_i \geq d \). Thus \( a_i \) is a miss, and \( m_i = 1 \). Also, if \( s_i = \infty \), we must have \( s'_i = \infty \), so \( a_i \) is still a miss, and \( m_i = 1 \). However, if \( d \leq s_i < d + N \), then \( a_i \) may become a hit or a miss, depending on how many of the interfering instructions are allocated into the SPM.

For example, if up to \( N \) interfering instructions are stored into the SPM, then \( s'_i < d \), so \( a_i \) becomes a hit.

Based on the discussion above, Equation 6.3 can be transformed into Equation 6.4.

\[
M = \sum_{all \ a_i} x_i \cdot m_i \quad (6.3)
\]

\[
M = \sum_{s_i \geq d+N} x_i + \sum_{d \leq s_i < d+N} x_i \cdot m_i \quad (6.4)
\]

If \( s'_i < d \), \( m_i = 0 \), and \( s'_i \geq d \), \( m_i = 1 \), so

\[
m_i = U(s'_i - d) \quad (6.5)
\]
U is the step unit function:

\[
U(x) = \begin{cases} 
0, & \text{if } x < 0 \\
1, & \text{if } x \geq 0 
\end{cases}
\]  
(6.6)

\(s'_i\) can be written as:

\[
s'_i = \sum_{i - s_i \leq j < i} x_j
\]  
(6.7)

In the above equation, \(j\) is the different memory accesses between \(a_i\) and the latest memory access to the same address of \(a_i\).

Combining Equations 6.4, 6.5 and 6.7, the total number of cache misses \(M\) of the benchmark after the SPM allocation can be described by the following equation:

\[
M = \sum_{s_i \geq d+N} x_i + \sum_{d \leq s_i < d+N} x_i \ast U(\sum_{i - s_i \leq j < i} x_j - d)
\]  
(6.8)

### 6.4.3 Stack Distance Based SPM Allocation Algorithms

Based on the unified stack distance analysis framework, we propose two SPM allocation algorithms to minimize the total number of cache misses.

**A Greedy Stack Distance Based SPM Allocation**

To get the best performance for the HSC, our goal is to minimize the total number of cache misses \(M\) in Equation 6.8. The problem is the \textit{satisfiability}(SAT) problem for the propositional calculus, which has been proved to be a NP-complete problem [81]. We first design a heuristic algorithm called
Greedy Stack Distance based SPM Allocation algorithm, which is described in Algorithm 9.

**Algorithm 9 GSDA Algorithm**

1: begin
2: allocations ← empty;
3: run simulation to get the memory accesses trace;
4: calculate the stack distance for the memory accesses;
5: while isNotFull(SPM) do
6:  \( M \leftarrow \text{MAX}_\text{INT}; \)
7:  \( B \leftarrow \text{NULL}; \)
8:  for each unallocated memory object \( b \) do
9:      allocate \( b \) into SPM;
10:     calculate cache miss number \( m \) use Equation(6.8);
11:    if \( m < M \) then
12:        \( M \leftarrow m; \)
13:        \( B \leftarrow b; \)
14:    end if
15:    unallocate \( b \) from the SPM;
16:  end for
17:  allocate \( B \) into the SPM;
18:  update the stack distance for the memory accesses;
19: end while
20: return allocations
21: end

The algorithm is a greedy one. In each iteration, this algorithm always tries to find the memory object that can minimize the current cache misses and allocate it into the SPM, and this process is repeated until the SPM is full. The complexity of GSDA is \( O(N \ast N_i \ast N_s) \), where \( N \) is the number of memory objects, \( N_i \) is the number of instructions in the memory access trace of the given program, and \( N_s \) is the number of memory objects the SPM can hold.
An Optimal Stack Distance Based SPM Allocation Algorithm

Although the problem is computationally expensive to solve, an optimal solution may still be derived by using the model checking for small size benchmarks and small SPMs. The optimal results can be used as a basis to check how close the heuristic-based algorithms can achieve. In this chapter, we use the SPIN model checker [8] to exhaustively and automatically check the developed model to find the optimal solution. Like other cache-aware algorithms developed in this chapter, the SPM allocation in OSDA is also based on the cache line block granularity (see Subsection 6.4.4 for details). We describe our OSDA allocation model by using the PROMELA (i.e. the verification modeling language of SPIN system), which is shown in Listing 6.1.

Listing 6.1. The SPIN Model for OSDA

```
bit arraylb[n];
int iAvailableSPM;
int iCacheMiss;
proctype allocation(){
    /\lb i /=
    atomic{
        if
            ::1 -> arraylb[i]=0;
            iAvailableSPM = iAvailableSPM - sizeof(lbi)
            ::1 -> arraylb[i]=1
        fi;
        if
            ::iAvailableSPM==0 -> goto endofallocation
            ::iAvailableSPM<0 -> arraylb[i]=1;
        goto endofallocation
```

119
::iAvailableSPM>0 -> skip
fi;
}
}
endofallocation: skip;
d_step{
iCacheMiss = \sum_{s_i \geq d+i} arraylb[i] 
+ \sum_{d \leq s_i < d+i} arraylb[i] \times [\sum_{i-s_i \leq j<s_i} arraylb[j] - d] \geq 0];
assert(iCacheMiss>TEST_VAL);
}
}
init{
atomic{
int i=0;
for(i: 0 .. n){
arraylb[i]=1
}
iAvailableSPM=M;
iCacheMiss=0;
run allocation();
}
}

For each line block of the given program, we generate an atomic allocation sequence. In the atomic sequence, we can have an if statement, because either path of the if statement (i.e. whether allocating this line block into the SPM or not) is executable, the SPIN will arbitrarily choose one of them based on its non-determinism. The variable iAvailableSPM is used to ensure that the total size of the allocated blocks will not exceed the total SPM size. After the allocation, the d_step statement (Lines 21-23) uses Equation 6.8 to calculate the
number of cache misses (i.e., \( iCachemiss \)) for this allocation. An `assert` statement (Line 24) is used to verify if there exists an allocation that can make \( iCachemiss \) less than or equal to the test value. The model checker will evaluate the assertion as a part of its search of the state space. If an error is reported in the verification stage, it indicates there is an allocation with the number of cache misses less than or equal to the test value. In this case, we can check another value less than the test value, until no error is reported. Then the last test value reporting an error is the optimal allocation we are looking for. Therefore, a binary search can be used to find the allocation with the minimum value of \( iCachemiss \).

Algorithm 10 describes the OSDA allocation algorithm.

**Algorithm 10 OSDA Algorithm**

1: begin  
2: allocations ← empty;  
3: run simulation to get the memory accesses trace;  
4: calculate the stack distance for the memory accesses;  
5: the upper bound of \( TEST_{VAL} \) ← result of GSDA;  
6: the lower bound of \( TEST_{VAL} \) ← 0;  
7: while lower bound ≤ upper bound do  
8: middle = (lower bound + upper bound)/2;  
9: \( TEST_{VAL} \) = middle;  
10: verify the model by SPIN;  
11: if no error report then  
12: lower bound ← middle;  
13: else  
14: upper bound ← middle;  
15: end if  
16: end while  
17: return the allocation of upper bound;  
18: end
It is worthy to note that to get the minimum number of cache misses, SPIN must exhaustively search all the possible allocations (the whole state space). More specifically, a state of a program is a set of values of its variables and location counters. In our allocation model, the state can be described by the state vector \( (arraylb[n], iAvailableSPM, iCachemiss, i) \), where \( i \) is the location counter of the program. A computation of a program is a sequence of states beginning with the initial state and continuing with the states that occur as each statement is executed. The verifier systematically checks that the correctness of the specifications held in all possible computations, which involves executing the program and backtracking over each choice of the next statement to execute the program nondeterministically.

To optimize the verification, we need to build a minimum allocation model in SPIN. We only declare necessary variables, and the types of the variables are as narrow as possible. We take advantage of the \texttt{atomic} and \texttt{d_step} statements to avoid middle states and to efficiently execute the statements if possible. Moreover, we only use the \texttt{assert} statement to check the allocation result and to verify the correctness of the model. However, an array whose elements are of type \texttt{bit} or \texttt{bool} is stored as an array of type \texttt{byte} (i.e., \texttt{arraylb}) in SPIN’s implementation [82]. To reduce the memory consumption of the state vector, we use multiple bit type variables instead of the array of bytes in our implementation if the number of the variables is less than 256. However, for the benchmarks that need more than 256 variables in the model, we still need to use \texttt{arraylb} because of the SPIN compiler
constraint.

Despite our efforts to reduce the memory consumption, the limitation of the OSDA is the state explosion. The whole state space includes all the possible allocations. Suppose all the line block sizes are equal, the SPM can hold $m$ line blocks, and there are $n$ line blocks for a given program, the total number of the allocations is $C^m_n + C^{m-1}_n + ... + C^0_n$. When the problem size increases, the state space grows very fast, which can quickly reaches the upper limit of the physical memory that is available in our experimental computer. Although SPIN provides some techniques such as hash table, partial order reduction, collapse compression and minimal automaton to reduce the memory consumption, they come at the cost of much longer execution time. As a result, the optimal results for large problems may not be practically solvable with limited resources (i.e., time and memory).

6.4.4 Side Effects of Basic Block Based Allocation

Many prior studies on pure SPM allocation for instructions are based on basic blocks. However, we find in the context of the HSC architecture, the basic block based allocation may be harmful to understanding the impact of SPM allocation on cache performance, which may result in suboptimal results. As the instructions of a program are typically placed continuously in the main memory and they are fetched into the cache in the unit of a cache line (also called line block in this dissertation), it is possible that the instructions from two continuous basic blocks are fetched into the same cache line. Typically, this can happen with the instructions in the end of a preceding basic block and the instructions at the
beginning of the following basic block. In this case, allocating the preceding basic block into the SPM can affect the cache hits for the instructions in the following basic block.

For example, as shown in Figure 6.1, a program segment contains three basic blocks. For simplicity, assume the cache has two cache lines and each cache line can hold two instructions, and there is an SPM whose size equals to the size of the cache. Inst 3 from BB1 and Inst 4 from BB2 are fetched into the Line 2 of the cache, and Inst 3 is placed in the head of the Line 2. Before the SPM is used, the number of cache misses is 4, which is caused by the accesses to Inst 1, Inst 3, Inst 5 and Inst 7 respectively. If the instructions are allocated into the SPM in the unit of a basic block, and BB1 and BB3 are allocated into the SPM, then two cache misses from Inst 1 and Inst 3 are reduced. However, the access to Inst 4 now turns into a cache miss, due to the elimination of spatial locality after Inst 3 is stored into the SPM. So the cache misses after SPM allocation is 3. In contrast, if we allocate the SPM space based on the line block granularity, both Inst 1 to Inst 4 will be stored into the SPM. Thus, the total number of cache misses after SPM allocation becomes 2 (i.e. Inst 5 and Inst 7 are still misses but there is no new miss).

To avoid this kind of side effect of basic block based allocation, we adopt line blocks as the allocation unit in SPM allocation for all the three cache-aware algorithms: HSA, GSDA and OSDA. Also, we assume the virtual memory support. After SPM allocation, the virtual to physical memory address is updated to ensure
the correct execution.

Figure 6.1. The example of side effect of basic block based allocation.

6.4.5 An Example To Compare HSA and GSDA

Compared to the HSA, which only considers cache performance before SPM allocation, the unified SDA framework for the HSC enables both the GSDA and the OSDA to take into account the impact of allocating each SPM line block on the cache performance. The updated cache performance provides more accurate information to guide the next allocation of the next SPM line block, which can result in better SPM allocation and higher cache performance. To illustrate the differences between the HSA and the GSDA, we provide an example and its control flow graph is shown in Figure 6.2. There are 6 basic blocks (from \(BB0\) to \(BB5\)), and \(BB3\) has a back-edge to \(BB1\), which iterates 2 times. Therefore, the basic blocks in this loop may have higher access frequencies than other basic blocks.

The configuration parameters of on-chip memories in the example are listed in Table 6.1. The SPM and cache have the same size, i.e., 4 words, and the line block size is 2 words. The execution order of basic blocks is listed in Table 6.2, and the memory address trace before the SPM allocation is shown in Table 6.3.

The stack distance for each memory access before the SPM allocation is
calculated by using cache block address (BA) instead of instruction memory address (A) to keep the spatial locality. One stack is maintained for each cache set to calculate the stack distance $s$. In this example, there are 2 cache sets, so the set index (SI) is either 0 or 1. Based on SDA, if $s \geq d$ (cache associativity), the memory access results in a cache miss. So the total number of cache misses before the SPM allocation is 13. Table 6.4 provides the 6 line blocks of this program, and line block $1$ is not executed and will not be considered in the allocation. The numbers of cache misses of each line block before the SPM allocation are calculated and shown in Table 6.4 as well.

![Figure 6.2. The control flow graph of the example code segment.](image)

If the HSA algorithm is used, the line blocks 0 and 4 will be stored into the SPM because they are the first two from the list of line blocks sorted in the descending order of the number of cache misses. The cache accesses after the allocation are shown in Table 6.5 (note the accesses to the SPM are not shown here), and the total number of cache misses becomes 6.

If the GSDA algorithm is used instead, all the line blocks are checked to find the first candidate as shown in Table 6.6. We can calculate the stack distance
Table 6.1. The SPM and cache parameters used in the example.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction size</td>
<td>1</td>
</tr>
<tr>
<td>SPM size</td>
<td>4</td>
</tr>
<tr>
<td>Cache size</td>
<td>4</td>
</tr>
<tr>
<td>Cache line size</td>
<td>2</td>
</tr>
<tr>
<td>Number of cache lines</td>
<td>2</td>
</tr>
<tr>
<td>Associativity(d)</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.2. The execution sequence of the basic blocks.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.3. The memory access trace before allocation. M=13.

(A: instruction address, BA: block address, SI: set index, SD: stack distance, M: number of cache misses)

<table>
<thead>
<tr>
<th></th>
<th>LB</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4. The number of cache misses of the line blocks before the SPM allocation.

(LB: line block, INSTR: instruction, M: number of cache misses)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>11</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h/m</td>
<td>m</td>
<td>m</td>
<td>h</td>
<td>m</td>
<td>h</td>
<td>m</td>
<td>h</td>
<td>m</td>
<td>h</td>
<td>m</td>
<td>h</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5. The memory access trace and cache misses after the SPM allocation by the HSA. (A: instruction address, BA: block address, SI: set index, SD: stack distance)
directly by using Equation 6.8 and get the number of cache misses for each case. We choose line block $0$ as the first candidate to minimize the current number of cache misses. Then all the remaining line blocks are checked to find the second candidate, which is line block $3$ as shown in Table 6.7. So the total number of cache misses after the SPM allocation by the GSDA is 5, which is smaller than that of the HSA.

6.5 EVALUATION METHODOLOGY

We use Trimaran compiler [40] to compile the benchmarks into the binary code for the target processor. The address information obtained from the compiler is used in the stack distance analyzer. The SPM allocator conducts the SPM allocation based on different algorithms and the SPIN model checker is used for the OSDA algorithm. The Trimaran simulator is used to simulate and report the performance of each benchmark. The model checking experiments are executed on a machine with the Intel Core i7 2.8GHz CPU and 16GB memory.

In our experiments, the baseline processor has 2 integer ALUs, 2 float ALUs, 1 branch predictor, 1 load/store unit, and 1-level on-chip memory. In our default configuration, to focus on the hybrid instruction SPM-Cache, the data cache is 128 bytes without any data SPM. The instruction HSC includes a 64 byte instruction SPM and a 64 byte instruction cache. The parameters of the caches include: 32 byte block size, direct-mapped, and LRU replacement policy. A cache hit takes 1 cycle and a main memory access takes 20 cycles. We also use two other configurations to do the sensitivity experiments, all of which are shown in Table
Table 6.6. Checking the cache misses for each line block to identify the first candidate by the GSDA-based SPM allocation. (A: instruction address, BA: block address, SI: set index, SD: stack distance, M: number of cache misses).
(a) Allocate line block 2 into the SPM.  
\[ M=6. \]

\[
\begin{array}{cccccccccc}
A & 11 & 6 & 7 & 8 & 11 & 6 & 7 & 8 & 9 & 10 \\
BA & 5 & 3 & 3 & 4 & 5 & 3 & 3 & 4 & 4 & 5 \\
SI & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
SD & \infty & \infty & 0 & \infty & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]

(b) Allocate line block 3 into the SPM.  
\[ M=5. \]

\[
\begin{array}{cccccccccc}
A & 11 & 4 & 5 & 8 & 11 & 4 & 5 & 8 & 9 & 10 \\
BA & 5 & 2 & 2 & 4 & 5 & 2 & 2 & 4 & 4 & 5 \\
SI & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
SD & \infty & \infty & 0 & \infty & 0 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]

(c) Allocate line block 4 into the SPM.  
\[ M=6. \]

\[
\begin{array}{cccccccccc}
A & 11 & 4 & 5 & 6 & 7 & 11 & 4 & 5 & 6 & 7 & 10 \\
BA & 5 & 2 & 2 & 3 & 3 & 5 & 2 & 2 & 3 & 3 & 5 \\
SI & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
SD & \infty & \infty & 0 & \infty & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]

(d) Allocate line block 5 into the SPM.  
\[ M=5. \]

\[
\begin{array}{cccccccccc}
A & 11 & 4 & 5 & 8 & 11 & 4 & 5 & 8 & 9 & 10 \\
BA & 5 & 2 & 2 & 4 & 5 & 2 & 2 & 4 & 4 & 5 \\
SI & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
SD & \infty & \infty & 0 & \infty & 0 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]

Table 6.7. Checking the cache misses for each line block to identify the second candidate by the GSDA based SPM allocation.  
(A: instruction address, BA: block address, SI: set index, SD: stack distance, M: number of cache misses).

<table>
<thead>
<tr>
<th>Configuration</th>
<th>I-SPM</th>
<th>I-Cache</th>
<th>D-Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>64B</td>
<td>64B, 32B cache line</td>
<td>128B, 32B cache line</td>
</tr>
<tr>
<td>configuration I</td>
<td>128B</td>
<td>128B, 32B cache line</td>
<td>256B, 32B cache line</td>
</tr>
<tr>
<td>configuration II</td>
<td>64B</td>
<td>64B, 16B cache line</td>
<td>128B, 16B cache line</td>
</tr>
</tbody>
</table>

Table 6.8. Three memory configurations in our experiments.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Code size (bytes)</th>
<th>Total_exe_cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>cyclic redundancy check computation on 40 bytes of data</td>
<td>664</td>
<td>65314</td>
</tr>
<tr>
<td>edn</td>
<td>finite impulse response (FIR) filter calculations</td>
<td>13504</td>
<td>162944</td>
</tr>
<tr>
<td>lms</td>
<td>lms adaptive signal enhancement</td>
<td>2136</td>
<td>1015329</td>
</tr>
<tr>
<td>matmult</td>
<td>matrix multiplication of two $20 \times 20$ matrices</td>
<td>480</td>
<td>395755</td>
</tr>
<tr>
<td>ndes</td>
<td>complex embedded code</td>
<td>3580</td>
<td>336728</td>
</tr>
<tr>
<td>state mate</td>
<td>automatically generated code</td>
<td>10476</td>
<td>8829</td>
</tr>
</tbody>
</table>

Table 6.9. General information of all benchmarks

6.8.

We randomly select 6 real-time benchmarks from Mälardalen WCET benchmark suite [41] for the experiments. The salient characteristics of all benchmarks are shown in Table 6.9.

6.6 EXPERIMENTAL RESULTS

6.6.1 Performance of Different Algorithms

Table 6.10 shows the cache misses of all 4 allocation algorithms for all the benchmarks in our default configuration. As we can see, by using the cache-aware SPM allocation algorithms at line block granularity, the instruction cache misses of all the three cache-aware algorithms are decreased compared to the baseline FSA. Among the three cache-aware algorithms, both the GSDA and the OSDA can reduce the cache misses more than the HSA for all the benchmarks. We also observe that for all benchmarks except state mate, the GSDA can achieve the same results as the OSDA, which are optimal. Even for the benchmark state mate, the result of the GSDA is only about 1% worse than that of the OSDA.
Table 6.10. The cache misses of all 4 allocation algorithms in default configuration.

Figure 6.3 compares the overall performance of different algorithms in the default configuration, which is normalized to the total number of execution cycles of the FSA. We find that the HSA and the GSDA can improve the overall performance by 9% and 11% respectively on average. The OSDA achieves the best performance for all benchmarks. However, the GSDA can achieve the same performance as the OSDA for all the benchmarks except the benchmark \textit{statemate}, for which it is only 0.5% worse than the OSDA.

Figure 6.3. The performance of all 4 algorithms in default configuration, which is normalized to the total number of execution cycles of the FSA.

Although the GSDA and OSDA can reduce more cache misses and achieve
better performance than both the FSA and the HSA, it should be noted that both
the GSDA and the OSDA have higher time complexity and thus can take much
longer time to compute. Table 6.11 presents the running time of SPM allocation
for each algorithm. Since our SPM allocation algorithms are all static, the
allocation will only be performed once, and the results of allocation will be used
many times in the benchmark execution on the HSC platforms. Therefore, it is
worthy for a designer to spend more time offline to find a better allocation to
improve the performance of hybrid SPM-Cache system in run time. For most
benchmarks, both the FSA and the HSA can finish allocation within 1 millisecond,
while the GSDA and the OSDA take significantly longer time.

It should be noted that the running time of the OSDA can be varied by
different upper bound of the $TEST\_VAL$, as listed in Table 6.11. OSAD(1) is only
the time to perform one round of verification by checking all possible allocations.
The actual allocation time can be $O(\lg n)$ times of the verification time in the
worst case, where $n$ is the upper bound set for the $TEST\_VAL$ in Algorithm 10. In
case there is no reference result that can be used from other allocations, we can
simply use the number of cache misses without the SPM as the initial $TEST\_VAL$
(OSDA(M) in Table 6.11 ). Otherwise, we can set the initial $TEST\_VAL$ as the
results of the FSA(OSDA(F) in Table 6.11), the HSA(OSDA(H) in Table 6.11) or
the GSDA(OSDA(G) in Table 6.11). In practice, since we may already get the
cache miss result by using the GSDA first, the $TEST\_VAL$ can be set to be the
result of the GSDA minus 1. Because for most cases, the GSDA can find the
Table 6.11. The running time (in msec) of all 4 allocation algorithms in default configuration.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FSA</th>
<th>HSA</th>
<th>GSDA</th>
<th>OSDA(1)</th>
<th>OSDA(F)</th>
<th>OSDA(H)</th>
<th>OSDA(G)</th>
<th>OSDA(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>0.586</td>
<td>0.605</td>
<td>63</td>
<td>1</td>
<td>11.586</td>
<td>11.605</td>
<td>74</td>
<td>13</td>
</tr>
<tr>
<td>edn</td>
<td>1.922</td>
<td>0.812</td>
<td>3461</td>
<td>3230</td>
<td>38761.922</td>
<td>38760.812</td>
<td>42221</td>
<td>38760</td>
</tr>
<tr>
<td>lms</td>
<td>0.686</td>
<td>0.602</td>
<td>1815</td>
<td>50</td>
<td>750.686</td>
<td>750.602</td>
<td>2565</td>
<td>750</td>
</tr>
<tr>
<td>matmult</td>
<td>0.669</td>
<td>0.599</td>
<td>205</td>
<td>1</td>
<td>11.669</td>
<td>10.599</td>
<td>215</td>
<td>14</td>
</tr>
<tr>
<td>ndes</td>
<td>0.770</td>
<td>0.790</td>
<td>896</td>
<td>410</td>
<td>5330.77</td>
<td>5330.79</td>
<td>6636</td>
<td>5740</td>
</tr>
<tr>
<td>statemate</td>
<td>0.750</td>
<td>0.726</td>
<td>40</td>
<td>40</td>
<td>320.75</td>
<td>320.726</td>
<td>360</td>
<td>280</td>
</tr>
<tr>
<td>average</td>
<td>0.897</td>
<td>0.689</td>
<td>1080</td>
<td>622</td>
<td>7531.231</td>
<td>7530.856</td>
<td>8678.5</td>
<td>7592.833</td>
</tr>
</tbody>
</table>

optimal or near optimal results, the OSDA may only need one round of verification instead of using the binary search. As a result, the execution of the OSDA can be greatly reduced for small benchmarks. However, as the benchmark size increases, the running time of the OSDA increases much faster than that of the GSDA. For example, the running time of the OSDA for one round verification for \textit{edn} is about 3230 times longer than that of the \textit{crc}, while it is only about 55 times longer for the GSDA.

6.6.2 Sensitivity to the Cache Size

For sensitivity study, we run two more groups of experiments. First, the size of both the SPM and the cache is increased to 128 bytes while keeping the same cache line size as the default configuration, which is the Configuration I. Second, the cache line size is decreased to 16 bytes while using the same size of the SPM and the cache as the default configuration, which becomes the Configuration II. Both the Configurations I and II increase the number of line blocks that the SPM
can hold, thus increasing the complexity of the allocation problem.

Table 6.12 gives the number of cache misses for different algorithms in Configuration I. Due to the increase of the SPM and cache size, the number of cache misses decreases as compared to that of the default configuration. With a larger SPM and cache size, we observe that all the three cache-aware algorithms still lead to much less cache misses than the FSA algorithm, and both the GSDA and the OSDA are superior to the HSA. For all benchmarks, the GSDA can achieve the results either the same as or very close to those of the OSDA. For the two benchmarks that the GSDA does not achieve the optimal results, i.e., edn, ndes and statemate, the GSDA is only 0.1%, 3.7% and 1.1% worse than the OSDA respectively.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FSA</th>
<th>HSA</th>
<th>GSDA</th>
<th>OSDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>26</td>
<td>19</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>edn</td>
<td>1896</td>
<td>1695</td>
<td>1598</td>
<td>1596</td>
</tr>
<tr>
<td>lms</td>
<td>21320</td>
<td>19060</td>
<td>9122</td>
<td>9122</td>
</tr>
<tr>
<td>matmult</td>
<td>55</td>
<td>15</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>ndes</td>
<td>7919</td>
<td>7250</td>
<td>7031</td>
<td>6770</td>
</tr>
<tr>
<td>statemate</td>
<td>195</td>
<td>193</td>
<td>193</td>
<td>191</td>
</tr>
</tbody>
</table>

Table 6.12. The cache misses of all 4 allocation algorithms with the Configuration I.

Figure 6.4 demonstrates the overall performance of all the four algorithms in Configuration I, which is normalized to the total execution cycles of the FSA under Configuration I. On average, the HSA, the GSDA, and the OSDA can improve the overall performance of the FSA by 2.8%, 7.7%, and 8.4% respectively,
indicating the effectiveness of cache-aware SPM allocation.

![Figure 6.4](image)

**Figure 6.4.** The performance of all 4 algorithms with Configuration I, which is normalized to the total execution cycles of the FSA under Configuration I.

### 6.6.3 Sensitivity to the Block Size

The numbers of cache misses of different algorithms in Configuration II are presented in Table 6.13. As the block size decreases, the number of cache misses increases for most benchmarks due to the reduced spatial locality. In Configuration II, both the HSA and the GSDA can decrease the cache misses by 22.3% and 28.5% on average as compared to the FSA. For all the benchmarks whose optimal results can be obtained by the OSDA, the GSDA can achieve results either the same as or very close to those of the OSDA.

Figure 6.5 compares the performance of all the four algorithms in Configuration II, which is normalized to the total execution cycles of the FSA under Configuration I. The HSA and the GSDA can improve the overall performance of the FSA by 9.6% and 10.5% respectively on average. The GSDA attains the optimal performance for the benchmarks *crc, edn, lms* and *ndes*. For *matmult* and *statemate*, the GSDA is only 0.1% and 0.15% worse than the OSDA.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FSA</th>
<th>HSA</th>
<th>GSDA</th>
<th>OSDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>1386</td>
<td>1384</td>
<td>1381</td>
<td>1381</td>
</tr>
<tr>
<td>edn</td>
<td>8402</td>
<td>5201</td>
<td>4699</td>
<td>4699</td>
</tr>
<tr>
<td>lms</td>
<td>42965</td>
<td>31087</td>
<td>27435</td>
<td>27435</td>
</tr>
<tr>
<td>matmult</td>
<td>2642</td>
<td>1063</td>
<td>1063</td>
<td>1044</td>
</tr>
<tr>
<td>ndes</td>
<td>16236</td>
<td>16153</td>
<td>15855</td>
<td>15855</td>
</tr>
<tr>
<td>state mate</td>
<td>388</td>
<td>386</td>
<td>386</td>
<td>385</td>
</tr>
</tbody>
</table>

Table 6.13. The cache misses of all 4 allocation algorithms with the Configuration II.

Figure 6.5. The performance of all 4 algorithms with Configuration II, which is normalized to the total execution cycles of the FSA under Configuration II.
6.6.4 Running Time Under Configuration I and II

We have also compared the running time of all SPM allocation algorithms for both Configurations I and II, which is shown in Table 6.14 and Table 6.15 respectively. It can be observed that the allocation time of the FSA and the HSA does not vary too much with different configurations. The allocation time of the GDSA in Configuration I and II is increased by 1.8 and 4.1 times respectively on average as compared to that of the default configuration, due to the increased number of line blocks in the SPM. However, for the OSDA, the running time increases dramatically. Taking the benchmark *lms* for example, the verification time for the default configuration is only 50ms, while it becomes 292 times longer in Configuration I and 8520 times longer in Configuration II. This is because in the OSDA algorithm, as the problem size grows, the number of states explodes, thus requiring deeper compression to solve the problem. Table 6.16 shows the compression ratio to the memory usage for all configurations. The higher the compression ratio, the longer time it may take to finish the verification.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FSA</th>
<th>HSA</th>
<th>GSDA</th>
<th>OSDA(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>crc</strong></td>
<td>0.745</td>
<td>0.562</td>
<td>95</td>
<td>10</td>
</tr>
<tr>
<td><strong>edn</strong></td>
<td>0.859</td>
<td>0.852</td>
<td>6409</td>
<td>6.04E+08</td>
</tr>
<tr>
<td><strong>lms</strong></td>
<td>0.851</td>
<td>0.937</td>
<td>3388</td>
<td>14600</td>
</tr>
<tr>
<td><strong>matmult</strong></td>
<td>0.788</td>
<td>0.687</td>
<td>391</td>
<td>10</td>
</tr>
<tr>
<td><strong>ndes</strong></td>
<td>0.801</td>
<td>0.715</td>
<td>1611</td>
<td>3480000</td>
</tr>
<tr>
<td><strong>statemate</strong></td>
<td>0.82</td>
<td>0.773</td>
<td>60</td>
<td>87800</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td>0.811</td>
<td>0.754</td>
<td>1992.333</td>
<td>1.01E+08</td>
</tr>
</tbody>
</table>

Table 6.14. The allocation time (in msec) of all 4 allocation algorithms in Configuration I.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FSA</th>
<th>HSA</th>
<th>GSDA</th>
<th>OSDA(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>0.608</td>
<td>0.618</td>
<td>175</td>
<td>190</td>
</tr>
<tr>
<td>edn</td>
<td>0.757</td>
<td>0.792</td>
<td>14455</td>
<td>1.71E+09</td>
</tr>
<tr>
<td>lms</td>
<td>0.681</td>
<td>0.726</td>
<td>7361</td>
<td>426000</td>
</tr>
<tr>
<td>matmult</td>
<td>0.693</td>
<td>0.677</td>
<td>754</td>
<td>160</td>
</tr>
<tr>
<td>ndes</td>
<td>0.749</td>
<td>0.829</td>
<td>3527</td>
<td>39600000</td>
</tr>
<tr>
<td>statemate</td>
<td>0.732</td>
<td>0.799</td>
<td>160</td>
<td>23800000</td>
</tr>
<tr>
<td>average</td>
<td>0.703</td>
<td>0.740</td>
<td>4405.3</td>
<td>2.96E+08</td>
</tr>
</tbody>
</table>

Table 6.15. The allocation time (in msec) of all 4 allocation algorithms in configuration II.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>default</th>
<th>configuration I</th>
<th>configuration II</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>None</td>
<td>0.77%</td>
<td>15.56%</td>
</tr>
<tr>
<td>edn</td>
<td>2.32%</td>
<td>99.99%</td>
<td>99.99%</td>
</tr>
<tr>
<td>lms</td>
<td>None</td>
<td>13.99%</td>
<td>38.42%</td>
</tr>
<tr>
<td>matmult</td>
<td>None</td>
<td>0.62%</td>
<td>15.23%</td>
</tr>
<tr>
<td>ndes</td>
<td>10.57%</td>
<td>97.61%</td>
<td>99.71%</td>
</tr>
<tr>
<td>statemate</td>
<td>5.3%</td>
<td>11.14%</td>
<td>99.99%</td>
</tr>
</tbody>
</table>

Table 6.16. The compression ratio of OSDA model during verification.
6.7 CONCLUSIONS

In this chapter, we develop 4 SPM allocation algorithms for the HSC architecture: FSA, HSA, GSDA and OSDA. While the FSA is cache-unaware, all other three algorithms are aware of the cache, which can reduce more cache misses and achieve much better performance than the FSA.

Both the GSDA and the OSDA are based on the unified stack distance analysis framework, which can consider the interaction between the SPM allocation and the cache performance. The OSDA is an optimal algorithm based on model checking; however, it may take significantly more memory and longer time to run as compared to other algorithms. The GSDA is a greedy algorithm, which can run efficiently and achieve optimal or near-optimal results for most benchmarks. Overall, we believe the GSDA is a good SPM allocation algorithm to harness the full potential of the HSC architecture efficiently.

In our future work, we would like to explore both heuristic-based and optimal cache-aware SPM allocation algorithms for data accesses as well. Also, we plan to study SDA based dynamic SPM allocation for the HSC architecture.
CHAPTER 7
CACHE-AWARE SPM ALLOCATION FOR MAXIMIZING ENERGY EFFICIENCY ON HYBRID SPM-CACHE ARCHITECTURE

7.1 CHAPTER OVERVIEW

The traditional SPM allocation algorithms, including both static and dynamic allocation, mainly focus on the SPM alone and are cache-unaware. They are unlikely to harness the full potential of the hybrid SPM and cache. We also believe that the SPM allocation for the hybrid SPM-Cache architecture must be aware of the cache performance to maximally optimize the energy consumption.

In this chapter, we design two energy-oriented algorithms: the Greedy Stack Distance based Allocation for Energy (GSDA-E) and the Optimal Stack Distance based Allocation for Energy (OSDA-E), which are extend from the GSDA and OSDA in chapter 6. We also comparatively evaluate all the 6 different SPM allocation algorithms, including the 4 SPM allocation algorithms in chapter 6. The first one is the Frequency based SPM Allocation (FSA), which is not aware of the cache and is used as the baseline. The other five algorithms are all cache-aware, but exploit cache information in different ways. The Hybrid SPM-Cache Allocation (HSA) exploits cache profiling information. It tries to allocate the memory objects with the largest cache misses into the SPM. The remaining four algorithms are all based on the Stack Distance Analysis (SDA) [11], [12], including two performance-oriented algorithms, i.e., the Greedy Stack Distance based
Allocation (GSDA) and the Optimal Stack Distance based Allocation (OSDA),
and two energy-oriented algorithms, i.e., the Greedy Stack Distance based
Allocation for Energy (GSDA-E) and the Optimal Stack Distance based Allocation
for Energy (OSDA-E). The GSDA and GSDA-E are greedy algorithms, whereas
the OSDA and OSDA-E are optimal algorithms by using model checking. More
details of these algorithms can be seen in the rest of the chapter.

As the first step to exploiting the tight interaction between SPM allocation
and cache energy in the HSC architecture, our study focuses on studying an
instruction SPM-Cache only as we do in chapter 6. In the instruction HSC, both
the SPM and the cache are used for storing instructions only. We plan to explore
hybrid SPM-Caches for data accesses in our future work.

This chapter makes three main contributions as the follows.

• We develop the heuristic based GSDA-E algorithms with polynomial time
  complexity and the optimal OSDA-E algorithms based on model checking,
  all of which are built upon the unified stack distance analysis framework we
  proposed in chapter 6.

• We have implemented and compared all the six SPM allocation algorithms,
  and find that GSDA-E can reduce the energy either the same as or close to
  the optimal results attained by the OSDA-E, while achieving performance
  close to the OSDA and the GSDA.
<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>associativity</td>
</tr>
<tr>
<td>$N$</td>
<td>the maximal number of memory objects stored the SPM</td>
</tr>
<tr>
<td>$A$</td>
<td>the total number of memory accesses</td>
</tr>
<tr>
<td>$a_i$</td>
<td>a memory access</td>
</tr>
<tr>
<td>$x_i$</td>
<td>binary: a memory access is in SPM(0) or not(1)</td>
</tr>
<tr>
<td>$m_i$</td>
<td>binary: a memory access is a hit(0) or a miss(1)</td>
</tr>
<tr>
<td>$M$</td>
<td>total number of cache misses</td>
</tr>
<tr>
<td>$e_h$</td>
<td>cache energy consumption per access</td>
</tr>
<tr>
<td>$e_s$</td>
<td>SPM energy consumption per access</td>
</tr>
<tr>
<td>$e_M$</td>
<td>main memory energy consumption per access</td>
</tr>
<tr>
<td>$e_m$</td>
<td>$e_h + e_M$</td>
</tr>
<tr>
<td>$E_{mem}$</td>
<td>energy consumption of the memory subsystem</td>
</tr>
<tr>
<td>$E_{total}$</td>
<td>total energy consumption of the processor and memory</td>
</tr>
</tbody>
</table>

Table 7.1. The symbols used in the equations.

7.2 STACK DISTANCE BASED SPM ALLOCATION ALGORITHMS FOR ENERGY

7.2.1 Stack Distance Analysis for HSC on Energy Consumption

We listed all the symbols used in this chapter in Table 7.1.

When exploring the energy consumption of the memory accesses, i.e., $E_{mem}$, we have the equation 7.1,

$$E_{mem} = \sum_{a_i} x_i \cdot e_i$$ (7.1)
in which, $e_i$ is the energy consumption for each memory accesses, and

$$
e_i = \begin{cases} 
e_s, & \text{if } a_i \text{ is in SPM} \\ 
e_h, & \text{if } a_i \text{ hits in cache} \\ 
e_m, & \text{if } a_i \text{ misses in cache} \end{cases} \quad (7.2)$$

Therefore, the total energy consumption of the memory acceses can be calculated as the following:

$$E_{mem} = \sum_{\forall a_i} [(1 - x_i) * e_s + x_i * (1 - m_i) * e_h + x_i * m_i * e_m] \quad (7.3)$$

Combining with Equation 7.4 derived from chapter 6, Equation 7.3 can be written as Equation 7.5, where $A$ is the total number of the memory accesses, and $e_M$ is the main memory energy consumption per access.

$$M = \sum_{s_i \geq d+N} x_i + \sum_{d \leq s_i < d+N} x_i * U(\sum_{i-s_i \leq j < i} x_j - d) \quad (7.4)$$

$$E_{mem} = A e_s + (e_h - e_s) \sum_{\forall a_i} x_i + (e_m - e_h) * M$$

$$= A e_s + (e_h - e_s) \sum_{\forall a_i} x_i$$

$$+ e_M * (\sum_{s_i \geq d+N} x_i + \sum_{d \leq s_i < d+N} x_i * U(\sum_{i-s_i \leq j < i} x_j - d)) \quad (7.5)$$

### 7.2.2 Exploit Cache Stack Distance to Improve SPM Allocation

A Greedy Stack Distance Based SPM Allocation Algorithm for Energy (GSDA-E)

Similar to the GSDA, we design a heuristic algorithm called Greedy Stack Distance Based SPM Allocation Algorithm for Energy minimizing using Equation
7.5, which is described in Algorithm 11. Same as the GSDA, the complexity of GSDA-E is \(O(N \times N_i \times N_s)\), where \(N\) is the number of memory objects, \(N_i\) is the number of instructions in the memory access trace of the given program, and \(N_s\) is the number of memory objects the SPM can hold.

\begin{algorithm}
\caption{GSDA – E Allocation Algorithm}
\begin{algorithmic}[1]
\State \textbf{begin}
\State \textit{allocations} $\leftarrow$ empty;
\State \textit{run simulation to get the memory accesses trace;}
\State \textit{calculate the stack distance for the memory accesses;}
\While {\textit{isnotfull} (SPM)}
\State \textit{E}$_{\text{mem}}$ $\leftarrow$ MAX\_INT;
\State \textit{B} $\leftarrow$ NULL;
\For {each \textit{allocate memory unit} \textit{b}}
\State \textit{allocate} \textit{b} into SPM;
\State \textit{calculate} total memory energy \(\epsilon_{\text{mem}}\) use Equation\( (7.5)\);
\If {\(\epsilon_{\text{mem}} < E_{\text{mem}}\)}
\State \textit{E}$_{\text{mem}}$ $\leftarrow$ \(\epsilon_{\text{mem}}\);
\State \textit{B} $\leftarrow$ \textit{b};
\EndIf
\State \textit{unallocate} \textit{b} into SPM;
\EndFor
\State \textit{allocate} \textit{B} to the SPM;
\State \textit{update} the stack distance for the memory accesses;
\EndWhile
\State \textbf{return} \textit{allocations}
\State \textbf{end}
\end{algorithmic}
\end{algorithm}
An Optimal Stack Distance Based SPM Allocation Algorithm for Energy Optimization (OSDA-E)

An optimal solution OSDA-E can also be designed by using model checking to optimally reduce the energy consumption for HSC. A similar Algorithm 12 to OSDA can be used by the OSDA-E as well. The allocation model of OSDA-E using the PROMELA (the verification modeling language of SPIN system) is shown in Listing 7.1.

Listing 7.1. The SPIN Model for OSDA-E

```plaintext
1 bit arraylb[n];
2 int iAvailableSPM;
3 int iEnergy;
4 proctype allocation(){
  /*lbi*/
  5 atomic{
    6 if
    7 ::1 -> arraylb[i]=0;
    8 iAvailableSPM = iAvailableSPM - sizeof(lbi)
    9 ::1 -> arraylb[i]=1
    10 fi;
    11 if
    12 ::iAvailableSPM==0 -> goto endofallocation
    13 ::iAvailableSPM<0 -> arraylb[i]=1;
    14 goto endofallocation
    15 ::iAvailableSPM>0 -> skip
    16 fi;
    17 }
    18 endofallocation: skip;
    19 d_step{
    20 iEnergy = Ae + (e_h - e_s) * \sum_{i} a_i arraylb[i]
    21 }
```
\[ e_M = (\sum_{i \geq d+1}^N \text{arraylb}[i]) \]
\[ + \sum_{d \leq i \leq d+1}^N \text{arraylb}[i] * ((\sum_{i \leq j < i} \text{arraylb}[j] - d) \geq 0) \]
\[ \text{assert (iEnergy > TEST_VAL);} \]

\begin{verbatim}
init{
atomic{
    int i=0;
    for(i: 0 .. n){
        arraylb[i]=1
    }
    iAvailableSPM=M;
    iEnergy=0;
    run allocation();
}
}\end{verbatim}

7.3 EVALUATION METHODOLOGY

We use Trimaran compiler [40] to implement the proposed SPM allocation algorithms. In our experiments, the baseline processor has 2 integer ALUs, 2 floating point ALUs, 1 branch predictor, 1 load/store unit, and 1-level on-chip memory. The instruction on-chip memories include a 64B SPM and a 64B cache. The parameters of the caches include: 32 Byte block size, direct-mapped, and LRU replacement policy. A cache hit takes 1 cycle and a main memory access takes 20 cycles. We use Cacti [57] to estimate the SPM and cache energy consumption. We randomly select 6 realtime benchmarks from Mälardalen WCET benchmark suit [41] for the experiments.
Algorithm 12 OSDA – E Allocation Algorithm

1: begin
2: allocations ← empty;
3: run simulation to get the memory accesses trace;
4: calculate the stack distance for the memory accesses;
5: the upper bound of TEST\_VAL ← results of GSDA – E;
6: the lower bound of TEST\_VAL ← 0;
7: while lower bound ≤ upper bound do
8:     middle = (lower bound + upper bound)/2;
9:     TEST\_VAL = middle;
10:    verify the model by SPIN;
11:    if no error report then
12:        lower bound ← middle;
13:    else
14:        upper bound ← middle;
15:    end if
16: end while
17: return the allocation of upper bound;
18: end
7.4 EXPERIMENTAL RESULTS

7.4.1 Memory Energy Consumption

We compare the memory energy of all the 6 allocation algorithms in Figure 7.1, which is normalized to the memory energy consumption of the FSA. The OSDA-E has the lowest memory energy dissipation for all the 6 benchmarks since it is the optimal energy oriented allocation. We also find that the OSDA-E and the GSDA-E can both reduce the memory energy consumption more than other four algorithms. For all the benchmarks, the GSDA-E can get the same or very close results comparing to the OSDA-E. The average energy consumption of the GSDA-E is only 0.3% higher than that of the OSDA-E, which proves that the GSDA-E is an effective heuristic algorithm.

![Figure 7.1. The memory energy of all the allocation algorithms in default configuration (normalized to OSDA).](image)

For the benchmark lms, the energy consumption of the GSDAE is 2% higher than that of the OSDA-E. This is because in lms, there are a number of line blocks that have close access frequencies and similar numbers of cache misses, making the GSDA-E less effective to choose the global optimal line blocks for minimizing memory energy. However, for all the other 5 benchmarks, we find that the GSDA-E can get exactly the same allocation as that of the OSDA-E.
We also observe that the memory energy consumption of the OSDA and the GSDA are almost the same because most of the GSDA allocation are the same with the OSDA. Compared to both the OSDA and the GSDA, on average, the OSDA-E and GSDA-E reduce the memory energy consumption by 10% and 9.7% respectively, indicating that the SPM allocation optimized for performance does not necessarily produce the best energy dissipation for the memory subsystem. This is mainly because in the HSC, although accesses to the SPM and the cache take the same latency, they consume different energy, which can only be effectively optimized by the energy-oriented algorithms.

Figure 7.1 also indicates that the HSA has the largest memory energy consumption. This is because the HSA can neither minimize the number of cache misses compared to the performance-oriented algorithms, nor maximize the number of SPM accesses compared to the energy-oriented algorithms. On average, the memory energy consumption of the HSA is 3.7% higher than that of the OSDA, and 15% higher than that of the OSDA-E. The FSA, however, can reduce the energy consumption compared to the OSDA and the GSDA by having more SPM accesses, because it tries to put the most frequently accessed instructions into the SPM. However, the memory energy consumption of the FSA is still 2% worse than the OSDA-E on average because of the energy dissipation caused by more cache misses.
7.4.2 Performance Results

Figure 7.2 compares the performance of all the allocation algorithms, which is normalized to the execution time of the FSA. Compared to the FSA, all the other five algorithms lead to better performance because they are all cache aware. Among all the algorithms, the OSDA has the best performance as it is the optimal algorithm to minimize the execution time. The GSDA can achieve performance either the same or very close to the OSDA, indicating its effectiveness in reducing the total execution time. While both the OSDA-E and the GSDA-E aim at optimizing memory energy consumption, they also result in better performance than both the FSA and the HSA because the energy-oriented allocation also requires to reduce the number of cache misses and increase the number of SPM accesses, both of which can benefit performance as well.

Figure 7.2. The total energy of all the allocation algorithms in default configuration (normalized to FSA).

7.4.3 EDP Results

Figure 7.3 compares the Energy-Delay Product (EDP) for all the SPM allocation algorithms. It can be seen that the EDP of the OSDA-E is the smallest on average because it can achieve optimal energy consumption with near-optimal
performance results. The EDP of the GDDA-E, on average, is only 0.1% larger than that of the OSDA-E, indicating its effectiveness in improving both energy consumption and performance. While the performance of the OSDA and the GSDA is close to the OSDA-E and the GSDA-E, the EDPs of the OSDA and the GSDA are much larger than those of the OSDA-E and the GSDA-E on average because both the OSDA and the GSDA consumes much more memory energy. Also, the EDPs of the FSA and the HSA are worse than the OSDA-E and the GSDA-E on average because they have both longer execution time and higher memory energy consumption.

Figure 7.3. Compare the EDP of all the allocation algorithms in default configuration (normalized to FSA).

7.5 CONCLUSIONS

In this chapter, we extend the GSDA and the OSDA from chapter 6 to optimize the energy consumption of the hybrid SPM-Cache architecture. We propose two energy-oriented allocation algorithms based on cache stack distance analysis, and find in general the cache-aware SPM allocation can lead to better performance and/or energy consumption than the cache-obliviouis SPM allocation algorithm (i.e. FSA). Also, we discover that for the HSC architecture, the
energy-oriented algorithms can lead to better EDP than the performance-oriented algorithms. In particular, our experiments indicate that the GSDA-E can reduce the energy consumption either the same as or close to the optimal results attained by the OSDA-E, while achieving performance close to the optimal results obtained by the OSDA.
CHAPTER 8
CONCLUSION REMARKS

This dissertation explores the hybrid SPM-Cache Architectures to improve the performance and energy efficiency of the real-time systems:

- How can we precisely estimate the WCET of the real-time applications?
- How can hybrid SPM-Cache Architectures improve energy efficiency besides the performance and the time predictability of the real-time systems?
- How can we improve the WCET of the real-time applications on hybrid SPM-Cache Architectures by SPM allocation algorithms?
- How can we design SPM allocation algorithms to optimise the execution time of the real-time applications on hybrid SPM-Cache Architectures?
- How can we design SPM allocation algorithms to reduce the energy consumption of the real-time applications on hybrid SPM-Cache Architectures?

Chapter 3 proposes a model checking based approach to bounding the worst-case performance of a multicore processor with shared L2 instruction caches. To alleviate the state explosion problem, we propose several techniques for reducing the memory consumption without compromising the quality of WCET analysis. Our experimental results show that the model checking based approach is safe and improves the tightness of WCET estimation as compared to the static
analysis approach [9]. However, due to the inherent complexity of multicore WCET analysis, the state explosion problem, and the physical memory constraint, this approach currently can only solve small benchmarks, while larger benchmarks with more interfering instructions will cause out-of-memory fault. However, it is possible to combine the model checking based method with the static analysis to benefit larger real-time applications.

Chapter 4 Built upon the prior work in [6] to study the performance and time predictability of hybrid SPM-cache architectures, we investigates the energy consumption of seven different SPM-caches. We find that all these seven hybrid on-chip memory architectures consume less energy than the pure SPM based architecture. Three hybrid SPM-cache architectures, including the IH-DC, the IHDH, and the IH-DS, can reduce the total energy consumption than the IC-DC. By considering both energy consumption and performance, the IC-DH, IH-DC, and IH-DH can achieve energy-delay product less than both the pure cache-based and SPM-based architectures. Among all the hybrid on-chip memory architectures, our evaluation indicates that the IH-DH architecture is the best in terms of both total energy consumption or EDP. More specifically, on average, the IH-DH architecture can reduce the total energy consumption by 22% and 16%, and reduce the EDP by 38.1% and 16.4% as compared to that of the IS-DS and the IC-DC respectively. Therefore, in addition to reconciling performance and time predictability as revealed in [6], we demonstrates that the hybrid on-chip memory architectures, in particular the IH-DH, can also make better tradeoffs between
performance and energy consumption, making it a very attractive design option for real-time and embedded systems.

Chapter 5 have explored four SPM allocation algorithms that differ by whether or not they are aware of the WCET and/or the cache. The FSA algorithm allocates SPM space based on the access frequency of each basic block from profiling, whereas the LPA attempts to allocate basic blocks with high access frequencies on the WC-path. Both the HSA and the EHSA algorithms can exploit the worstcase cache analysis information; however, the EHSA ensures that only basic blocks on the WC-path are allocated to the SPM. We have also extended the ILP-based timing analysis method [9] to predict the WCET for the hybrid SPM-cache architecture, and our experiments indicate that the developed WCET analyzer is safe and reasonably accurate. Our evaluation indicates that the EHSA algorithm, which is both WCET-oriented and cache-aware, can achieve the best WCET for all benchmarks under all SPM-cache configurations we have evaluated. The EHSA is especially more effective to reduce WCET with a smaller cache and a larger SPM. While the EHSA may lead to degradation of the average-case performance for some multiple-path benchmarks, its impact is insignificant.

To improve the performance, Chapter 6 develops 4 SPM allocation algorithms for the HSC architecture: FSA, HSA, GSDA and OSDA. While the FSA is cache-unaware, all other three algorithms are aware of the cache, which can reduce more cache misses and achieve much better performance than the FSA. Both the GSDA and the OSDA are based on the unified stack distance analysis
framework, which can consider the interaction between the SPM allocation and the cache performance. The OSDA is an optimal algorithm based on model checking; however, it may take significantly more memory and longer time to run as compared to other algorithms. The GSDA is a greedy algorithm, which can run efficiently and achieve optimal or near-optimal results for most benchmarks. Overall, we believe the GSDA is a good SPM allocation algorithm to harness the full potential of the HSC architecture efficiently.

Last but not the least, Chapter 7, extend the GSDA and OSDA to reduce the energy consumption to GSDA-E and OSDA-E. We evaluate them together with the four different SPM allocation algorithms from Chapter 6, and find in general the cache-aware SPM allocation can lead to better performance and/or energy consumption than the cache-oblivious SPM allocation algorithm (i.e. FSA). Also, we discover that for the HSC architecture, the energy-oriented algorithms can lead to better EDP than the performance-oriented algorithms. In particular, our experiments indicate that the GSDA-E can reduce the energy consumption either the same as or close to the optimal results attained by the OSDA-E, while achieving performance close to the optimal results obtained by the OSDA.

8.1 FUTURE WORK

In our future work, we would like to seamlessly integrate static analysis with the model checking based method to attain safe and tight WCET results with much smaller memory consumption and less computation time. Moreover, we would like to explore the hybrid SPM-caches for storing data as well. We plan to
explore both heuristic-based and optimal cache-aware SPM allocation algorithms for data accesses as well. Also, we plan to study SDA based dynamic SPM allocation for the HSC architecture. In addition, we intend to study the hybrid SPM-cache architecture to multi-level on chip memory and multi core platform.
REFERENCES


[16] R. Wilhelm et al., “The worst case execution time problemOverview of


[32] R. Wilhelm, “Why ai+ilp is good for wcet, but mc is not, nor ilp alone,” In *Verification, Model Checking and Abstract Interpretation (YMCAI).* Lecture


[41] Mälardalen WCET research group, “Mälardalen wcet benchmark suite,”


Symposium (RTSS), 2004.


Proc. 14th international conference on Compilers, architectures and synthesis
for embedded systems. ACM, 2011.

[74] P. Lokuciejewski et al., “WCET-driven cache-based procedure positioning
Conference on. IEEE, 2008

[75] S. Plazar et al., “WCET-driven Cache-aware Memory Content Selection,”

Proc. Object/Component/Service-Oriented Real-Time Distributed Computing


[77] M. Kandemir et al., “Dynamic management of scratch-pad memory space,”


management,” Proc. Parallel Architectures and Compilation Techniques,

[79] B. Egger et al., “Dynamic scratchpad memory management for code in
portable systems with an mmu,” ACM Transactions on Embedded

low power embedded processors,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4, 8, August 2006.
