ABSTRACT

KANG, SANGYEOL. Memory Allocation for Real-Time Embedded Systems. (Under the direction of Alexander G. Dean.)

This study proposes memory allocation methods to acquire predictably fast and energy efficient memory access for real-time embedded systems. First, this study presents how scratchpad memory (SPM) reduces data cache contention for preemptive real-time embedded systems using both fast memory systems together. We classify data cache misses into intrinsic misses and interference misses. This new cache miss analysis reveals average interactions between data of real-time tasks in the data cache. The proposed allocation methods build on analysis to identify data sets which disrupt the data cache states frequently during the real-time scheduling. Our experiments report significantly reduced inter-task cache pollution as well as eliminated intrinsic cache misses of the data sets themselves after allocating those data sets into SPM. This process improves the cache performance since the SPM allocation increases effective capacity of the data cache without performance degradation. This interactive benefit is recognized as synergy between SPM and data cache. The proposed schemes guide us to enhance the sensitive and dynamic data cache through a synergetic process. Second, this study proposes that complementary advantages of both SPM and data cache can be accommodated together in a balanced manner. We submit that a small capacity of SPM improves the data cache performance and eventually the system performance since a large amount of memory access is generated by a small portion of data variables. Our performance simulation results from an integer linear programming (ILP)-based SPM allocation explain that this memory access concentration leads to diminishing returns as SPM capacity increases. In addition to those observations, various hybrid configurations of SPM and data cache are evaluated in terms of memory access latency, data cache miss counts, dynamic energy consumption, and the energy-access latency product. The evaluation results demonstrate that the synergy is escalated at certain balanced capacities of SPM and data cache. Third, memory systems using only SPM are also explored to see the potential of SPM as an alternative to data cache. This study extends a previously proposed static SPM allocation scheme to a full implementation under realistic circumstances. The major leap in the implementation described here is the proposed viable assembly rewriting technique to deploy procedure stack frames across multiple memory units including SPM. All these proposed SPM allocation methods, experiments, evaluation and analysis provide a foundation for research using SPM and cache memory together or solely SPM with better timing predictability for further developed performance of real-time embedded systems.
Memory Allocation for Real-Time Embedded Systems

by

Sangyeol Kang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina
2012

APPROVED BY:

_________________________  _______________________

Eric Rotenberg               Yan Solihin

_________________________  _______________________

Frank Mueller               Alexander G. Dean
Chair of Advisory Committee
DEDICATION

To My Parents.
Sangyeol Kang received his Bachelor of Science degree in Electrical Engineering from Seoul National University (Seoul, Republic of Korea) in 2003, and his Master of Science degree in Computer Engineering from North Carolina State University (Raleigh, North Carolina) in 2008. From 2004 to 2006, he developed hardware and software for mobile phones in Korea. He started his graduate study under direction of Dr. Alexander G. Dean at North Carolina State University in 2006. He developed a method providing static timing analysis support for an ARM7 processor platform during studying his masters degree. For his doctoral degree, he studied issues in timing predictably fast memory systems for real-time embedded systems. His research interests widely include design and performance issues in processor architecture, memory architecture, operating systems and softwares for embedded systems.
ACKNOWLEDGEMENTS

I would like to thank many persons who helped me with this study. My sincere gratitude goes to Dr. Alexander G. Dean. He encouraged me academically and inspired me to seek for many valuable aspects in my life while serving as my advisor. I appreciate Dr. Eric Rotenberg, Dr. Yan Solihin and Dr. Frank Mueller. They served as my advisory committee members. Their enthusiasm for research is the most valuable textbook for me. I learned many things from their lectures and discussion. Warm thanks go to Shaolin Peng, Gregory Parsons, Karthik Sundaramoorthy, and Avik Juneja, my colleagues at NCSU for their friendship and helpful comments. I also appreciate Byoungdon Kong, Won So and Seongbeom Kim, my SNU alumni and my friend Sangchang for their sincere consulting. Finally, my parents and my wife Eunha always believe in me and that I am heading in the right direction.
# TABLE OF CONTENTS

List of Tables ................................................................. vii

List of Figures ................................................................. viii

Chapter 1 Introduction ......................................................... 1
   1.1 Memory Systems of Real-Time Embedded Systems .................... 1
   1.2 Timing Variability ................................................... 4
   1.3 Outlines ............................................................. 6
   1.4 Contributions ....................................................... 7

Chapter 2 Review of Related Work ........................................ 12
   2.1 Program Component Allocation into SPM ............................. 12
       2.1.1 Program Code ............................................... 13
       2.1.2 Program Data ............................................... 14
   2.2 Timing Predictability of Cache Memory ............................. 21
       2.2.1 Bounding of Cache Misses .................................. 21
       2.2.2 Improving Predictability of Cache Memory ................. 23

Chapter 3 Stack Split ....................................................... 25
   3.1 Introduction ........................................................ 25
   3.2 Stack Frame Models ............................................... 26
       3.2.1 Original Stack Frame Model ................................. 26
       3.2.2 New Stack Frame Model by Stack Split .................... 28
   3.3 Stack Frame Splitting Schemes ................................... 29
       3.3.1 Default Split and Dense Split .............................. 29
       3.3.2 Concerns in Real World Application ....................... 30

Chapter 4 Data Allocation with Real-Time Scheduling .................. 33
   4.1 Introduction ........................................................ 33
   4.2 DARTS Concept .................................................... 33
       4.2.1 DARTS Cycle ................................................ 34
       4.2.2 Memory Access Time and Data Allocation .................. 36
   4.3 Experiments ........................................................ 41
       4.3.1 Taget System ............................................... 41
       4.3.2 Experiment Setup .......................................... 43
   4.4 Results ............................................................ 44
       4.4.1 Single-Task Perspective Discussion ......................... 45
       4.4.2 Multi-Task Perspective Discussion ......................... 49
   4.5 Summary ............................................................ 52
# LIST OF TABLES

| Table 4.1 | Target hardware system features and toolchain | 42 |
| Table 4.2 | Properties of tasks used for experiments | 42 |
| Table 4.3 | Speedup comparison between DARTS (using 2-cycle SPM) and 1-cycle data cache | 50 |
| Table 5.1 | Target hardware system features and toolchain | 63 |
| Table 5.2 | Test programs used for experiments | 64 |
| Table 5.3 | Cache contention degree table used in fine granularity level experiments | 67 |
| Table 5.4 | Number of scheduler ticks encountering by \( MgSort \) and \( Rotate \) | 69 |
| Table 5.5 | Best task combinations allocated into SPM and corresponding cache misses counts from coarse granularity level experiments (\( Controlled-Allocation \)) | 69 |
| Table 5.6 | Best task combinations allocated into SPM and corresponding cache misses counts from coarse granularity level experiments (\( Real-Allocation \)) | 71 |
| Table 5.7 | Portion of memory access counts and intrinsic cache miss counts by each task during hyperperiod | 72 |
| Table 5.8 | Best combinations under limited SPM capacity at coarse granularity (\( Controlled-Allocation \)) | 73 |
| Table 5.9 | Best combination under limited SPM capacity in fine granularity level experiments (\( Controlled-Allocation \)) | 74 |
| Table 6.1 | Memory access distributions used in simulation | 86 |
| Table 6.2 | Test programs used for evaluation and experiments | 90 |
| Table 6.3 | Memory parameters used for evaluation | 91 |
| Table 6.4 | Cache miss rate and SPM allocation saturation point of test programs | 94 |
| Table 6.5 | Hybrid configurations showing best memory access performance | 98 |
| Table 6.6 | Hybrid configurations showing best energy saving and energy-access latency product | 98 |
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Static RAM cell consisting of six transistors [81]</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Simplified structures of two fast memory systems</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Adjusted CPU utilization bounds for rate monotonic scheduling</td>
<td>5</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Example code and stack frames</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Pseudo code of stack split (dense split)</td>
<td>31</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>DARTS application flow and components provided by toolchain</td>
<td>34</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Example of preemption graph embedding call graphs inside</td>
<td>35</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Memory footprint of data objects at task, procedure and variable level</td>
<td>37</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Performance comparison of ADPCM</td>
<td>45</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Performance comparison of Huff</td>
<td>46</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Performance comparison of MgSort</td>
<td>47</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Performance comparison of Dijkstra</td>
<td>48</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Performance comparison of SHA</td>
<td>49</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>Improved real-time performance</td>
<td>51</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Sensitivity of data cache performance in single-threaded system</td>
<td>55</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Example of new cache miss analysis</td>
<td>60</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Cache contention degree graph used in coarse granularity level experiments (in Rotate-centric view)</td>
<td>65</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Cases of negative interference miss counts</td>
<td>68</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Reduction of cache misses by SPM allocation using heuristic I and other methods on exhaustive search from coarse granularity level experiments</td>
<td>70</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Reduction of cache misses by SPM (1KB) allocation into using heuristic I and exhaustive search at fine granularity level</td>
<td>73</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Correlation of actual cache performance vs. cache contention degree estimated by heuristic II and SPM usage (Controlled-Allocation)</td>
<td>75</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Call graph to obtain example call graph matrix $G_{eg}$ (SHA[30])</td>
<td>84</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Captured memory access distributions</td>
<td>85</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Diminishing returns of memory access performance by ILP-based SPM allocation</td>
<td>87</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Performance estimation of hybrid configuration of SPM and data cache</td>
<td>89</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Fraction of SPM access counts to first level memory access counts</td>
<td>93</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>Diminishing returns as increasing SPM capacity with 8KB data cache</td>
<td>94</td>
</tr>
<tr>
<td>Figure 6.7</td>
<td>Memory access performance of various hybrid configurations from estimation and simulation</td>
<td>96</td>
</tr>
<tr>
<td>Figure 6.8</td>
<td>Dynamic energy consumption of various hybrid configurations from estimation</td>
<td>97</td>
</tr>
<tr>
<td>Figure 6.9</td>
<td>Energy-access latency product of various hybrid configurations from estimation</td>
<td>99</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Memory Systems of Real-Time Embedded Systems

As technology evolves, more power is available to computer systems, including embedded systems. Development of faster, more powerful and more versatile embedded devices has not paused. As a reaction to this achievement in modern technology, designing computer systems becomes further complicated. Design engineers of computer systems are asked to balance performance and cost in spite of ever-increasing complexity. Developing embedded systems involves more complexity since the system resources are likely to be limited, but manufacturing cost is sensitive to even small changes in the design. Design engineers of real-time embedded systems especially must take one more challenging design factor into consideration on top of the others, which is real-time schedulability.

Processors and memory systems are the major components in modern computer systems. Processors manipulate information and the memory stores the information, such as program code and data, to keep the computation continual. System performance depends on the performance of these components. However, a performance gap exists between processors and memory systems [34] and many researchers have been encouraged to bridge the gap in tremendously novel ways.

Cache memory and scratchpad memory (SPM) are a marvelous idea in this sense. These two types of memory system provide small but quickly accessible memory space near to the processors so that the processors can reduce access time to the code and data. Both cache memory and SPM have static RAM cells consisting of 6 transistors to store their contents, as in Figure 1.1 [81]. However, they are different in architectural aspects, and the different hardware schemes are naturally related to different control mechanisms and characteristics. Figure 1.2 depicts two fast memory systems [117, 10].

A data cache is composed of tag array, data array, tag matching circuitry and output cir-
cuitry such as sense amplifiers, output drivers and so on. The cache memory is not assigned to a part of system’s address space particularly but covers all cacheable memory accesses. Contents of the cache memory are loaded and replaced automatically by the hard-wired algorithm. Recently used contents and their neighboring contents are cached since they are more likely to be requested by the processors. These temporal and spatial localities of the code and data are utilized to enhance the cache performance whilst overcoming its limited capacity. Hence, the cache memory takes advantage of multi-level memory systems covering all cacheable memory access automatically.

Unfortunately, however, no matter whether the real-time constraints are soft or hard, the cache memory used in the real-time systems raises one issue – timing predictability of its behavior [33]. This is because utilization of the localities fails often due to the limited capacity and the automatic operation mechanism. This situation causes content misses in the cache memory, and some performance penalty must be paid to recover from the misses. Moreover, the content misses are not easy to estimate or bound statically since the cache misses are dynamically determined by previous states of the cache lines, which relies on a program’s input workload, memory layout and memory access pattern, cache pollution by other programs and so on. Although the general purpose system may endure this situation, real-time embedded systems are susceptible because the misses add extra timing uncertainty in designing the systems in addition to the recovery cost. Thus, much research has suggested methods to bound cache misses safely and to include timing variation into real-time schedulability analysis.
Scratchpad memory (SPM) has been regarded as an alternatively faster memory in place of the cache memory in real-time embedded systems. SPM is a fast on-chip memory located near to the processors whose memory cell is similar to that of the cache memory. The difference is that it does not have hardware controllers such as the tag array or control logic supporting tag matching to load and evict the contents automatically (as the cache memory does). Consequently SPM consumes less energy per access. SPM has its own address space, which is part of the system address space. The SPM contents are controlled by software and SPM responds to memory access only within itself. Thus, SPM takes advantage of heterogeneous memory systems and has versatile functionality managed by software methods.

Since SPM contents are under the control of programmers without dynamic misses, it offers better timing predictability than the cache memory, in addition to the same fast access latency and energy saving. The SPM contents can be managed either statically or dynamically. Static managements load the SPM contents when programs are loaded into a target system. Dynamic methods load and evict the contents during the program’s running, similar to how the cache memory does but via software control. In both cases, it is known which contents are available in SPM at a particular execution time point of the program at design phase. More details about the timing predictability provided by SPM are discussed in Chapter 2. However, the SPM control
mechanism burdens the programmers with content selection, and content selection affects the system performance directly. Furthermore, the embedded systems are equipped simultaneously with heterogeneous memory units whose characteristics such as access latency, access bus width, and energy consumption are diverse. Therefore, researchers have developed various schemes to select and manage the SPM contents for the best performance.

Wise and efficient memory allocation for real-time embedded systems is not straightforward but also cannot be overlooked. To improve system performance, the cache memory may be used, but the system suffers from the timing variability. SPM may be used to replace cache memory, but the programmers are responsible for choosing the most beneficial data on behalf of the system, considering many factors. This study looks into data allocation problems in real-time embedded systems using SPM and data cache memory, and those with solely SPM. Ultimately this study aims to investigate memory allocation methods enhancing real-time embedded systems beyond many conflicting design factors, including memory access performance, energy consumption and timing predictability.

1.2 Timing Variability

Timing variability makes developing a real-time embedded system difficult since it directly impedes timing analysis, which is an essential design step to guarantee schedulability of the system. There are many sources of timing variability and timing noise in contemporary embedded systems, such as out-of-order execution, speculation techniques in processor pipelines, non-uniform memory access latency and so on [60, 33, 112]. Behaviors of these features are so dynamically changing that it is not easy to estimate the behaviors by tightly bounding. Even if the timing analysis is performed appropriately, the system is likely to be over-provisioned to give more timing margin for schedulability.

One example helps us understand the impact of timing variability to a system’s schedulability. Let a target system schedule \( n \) tasks in a rate monotonic way. In order for the system to work properly in the rate monotonic scheduling, a CPU utilization test must meet the theoretical upper bound, which is given in Equation 1.1 [56, 57]. A task \( i \)'s period is given as \( T_i \) and its execution time is \( C_i \).

\[
U \leq n \left(2^{\frac{1}{n}} - 1\right) \quad \text{where} \quad U = \sum_{i=1}^{n} \frac{C_i}{T_i}
\] (1.1)

Now consider a new CPU utilization which is composed of two parts: analyzable part \( U_a \) and timing variable part \( U_v \). The analyzable part can be calculated or analyzed statically so that we can know it during the designing of the system. The timing variable part accounts for some unexpected variation in actual CPU utilization which is not discovered during the design.
Figure 1.3: Adjusted CPU utilization bounds for rate monotonic scheduling

phase. The actual CPU utilization becomes the sum of the two parts, as shown below, when timing variation is introduced into each task of the system by $\Delta_i$.

$$U = U_a + U_v = \sum_{i=1}^{n} \frac{C_i}{T_i} + \sum_{i=1}^{n} \frac{\Delta_i}{T_i}$$

Provided simply that the timing variation is uniformly rated for all the tasks by $\delta$ of their period $T_i$ respectively, i.e., $\Delta_i = \delta T_i$. Then,

$$U_v = n\delta$$

When the system is designed, $U_v$ is not expected but actually imposed onto the system during its running. Hence, in order to guarantee the feasibility of the system, a new utilization test must be considered as in Equation 1.4, whose upper bound is less than that of the original in Equation 1.1.

$$U_a \leq n \left(2^{\frac{1}{n}} - 1\right) - n\delta$$

Several cases of the new CPU utilization bound are calculated for this example in Figure 1.3.
The topmost black solid line plots the original upper bound from Equation 1.1, and other lines show the adjusted bounds corresponding to Equation 1.4. As observed in the plots, realistic upper bounds are less than the original one. For example, when the timing variability of each task is 1% of its period ($\delta = 0.010$) in the system running 10 tasks ($n = 10$), it is observed that the gap between the original bound and the adjusted one is 10%. Furthermore, as the timing variability increases so that $\delta$ also increases, more timing margin is required. This situation gets worse as more tasks run on the system. Although this calculation is based on slightly over estimated parameters and simplified assumptions, the impact of the small timing variability to the system is demonstrated well in the plots.

The above example addresses rate monotonic scheduling, which is one static priority scheduling method. A system accepting dynamic priority scheduling also suffers from the same problem. For example, success of the earliest-deadline-first (EDF) scheduling [56] or the least slack time (LST) scheduling [64] also leans on tasks’ actual execution time or estimation of the execution time. Hence, unexpected timing variability makes the system design difficult because: (1) it is not estimated or analyzed easily and (2) it requires more conservative and over-provisioned design.

1.3 Outlines

As described in the previous sections, needs for better performance and safer real-time behavior in real-time embedded systems necessarily intersect at providing fast but timing predictable memory systems. Since huge efforts have been already made on SPM and cache memory, part of the research in diverse fields which have motivated, inspired and informed this study are briefly reviewed in Chapter 2. We look through research addressing SPM allocation methods of program code and data, and management methods of the SPM contents. Efforts to improve timing predictability of cache memory are also reviewed.

One challenging job after obtaining a new data allocation is implementing the data allocation to a target system so that we can actually use it. When automatic data variables are moved to different memory units apart from their original locations, original stack frames containing the variables may no longer be maintained within their original memory unit. Chapter 3 proposes stack split, an assembly rewriting technique to deploy the procedure stack frames across multiple memory units at a fine granularity level. The stack split technique is actually applied for the DARTS implementation discussed in Chapter 4.

Predictably fast SPM can be a good substitute for data cache. A fast memory system built only on SPM rather than data cache is explored in Chapter 4. The Data Allocation with Real-Time Scheduling (DARTS) method has been already proposed by Ghattas et al. [23, 25]. But development of the method stops at simulation targeting a simple 8-bit AVR processor.
platform. Chapter 4 extends the DARTS method toward a full implementation under further realistic circumstances. The allocation method is transplanted to the popular ARM processor platform. Real test benchmarks are examined to evaluate the scheme with the help of the stack split scheme on real hardware. Memory access performance obtained from DARTS is comprehensively compared to that of real data cache hardware.

This study leverages SPM to improve data cache performance beyond taking its alternative benefits (such as fast access latency and better timing predictability) to data cache. Allocating some parts of data into SPM reduces cache contention by the data and finally improves the cache performance. Of course, the data stored in SPM do not experience any performance degradation. This interactive benefit is recognized as synergy between SPM and data cache in Chapter 5. The chapter also proposes synergetic SPM allocation heuristics, which can alleviate inter-data cache contention of tasks and cache misses in multi-threaded preemptive real-time embedded systems. For this, a new concept of cache miss analysis is also proposed, where we classify cache misses into intrinsic misses and interference misses.

A question about how to balance limited hardware resources (e.g. die area or fast memory capacity) between the two fast memory systems remains unanswered after discussing the synergetic SPM allocation scheme. Chapter 6 empirically investigates ways to expand the synergy by balancing the hybrid configurations of SPM and data cache in aspects of memory access latency, data cache performance, dynamic energy consumption and energy-access latency products. Various hybrid configurations of SPM and data cache are simulated and evaluated with those metrics. The chapter also observes concentration of memory access to a small portion of data variables, which leads to diminishing returns as SPM capacity increases. Some guiding insights established from the simulation and evaluation results are supported by those observations.

To summarize, the hybrid configuration of SPM and data cache are studied in Chapter 5 and 6, and the memory configuration using only SPM is explored in Chapter 4. Chapter 3 explains a supporting technique to exploit SPM for automatic type stack variables. This chapter (Chapter 1) introduced this study with examples of the timing variability problem in the real-time embedded systems, compared SPM to data cache qualitatively, and discussed outlines and contributions. Chapter 2 introduces related research work briefly. Chapter 7 summarizes all chapters and lists future work beyond the findings of this study.

1.4 Contributions

Throughout all the chapters as outlined above, this study demonstrates several contributions distinguishing itself from the prior achievements by other researchers reviewed in Chapter 2.

This study actively considers cache memory along with SPM to data allocation schemes
targeting real-time embedded systems. Most of the SPM allocation studies compare benefits such as memory access performance, estimated worst case execution time (WCET), energy saving or die area when using SPM to those metrics when using cache memory [5, 37, 10, 95, 94, 21, 112, 61, 1, 77]. Those studies regard SPM as an alternative to cache memory. Only a few studies investigate the hybrid configuration consisting of SPM and cache memory [20, 108, 110, 76, 75]. Mamidipaka et al. also report some benefit in data cache access counts after employing the on-chip stack memory [62]. Other studies propose cache locking and cache partitioning [46, 47, 79, 78, 80, 102, 103] in order to improve timing predictability of cache memory directly. Little work has promoted SPM to enhance cache memory, and relatively less attention has been paid to the synergy between SPM and cache memory, even though hybrid configurations can provide better performance with little timing variability.

The work presented in Chapter 5 [42] differs from all these studies in several ways: (1) our proposed schemes take advantage of both SPM and cache memory, rather than one or the other, in order to mitigate inter-data cache pollution while still providing fast memory access for all data, (2) this study presents a new approach toward data cache miss analysis, which guides a way to identify data to allocate into SPM, (3) the allocation schemes are thus based on the new data cache miss analysis rather than data access count analysis or data access frequency analysis in multi-threaded real-time embedded systems, and (4) our work considers data allocation and data cache rather than instruction allocation and instruction cache; where cache behavior is ever-changing, and analyzing it involves much more complexity. In other words, this study reports the synergy between SPM and data cache that exists in memory access performance, which can be extracted via the new cache miss analysis concept and the allocation heuristics proposed in this work.

As a next step in studying this synergy, Chapter 6 establishes several other differences between this work and current research: (1) it discovers that a small portion of data variables generate a large amount of data memory access counts, (2) the concentrated memory access distributions are modeled with exponential distributions and simulated to show the diminishing returns as SPM capacity increases due to the concentrated memory access, and (3) various hybrid configurations of SPM and data cache are evaluated by four metrics: total memory access latency, total data cache miss counts, total dynamic energy consumption and energy-access latency products. To summarize, this work is the first work which reveals that performance benefit attainable from SPM is correlated to the memory access distribution, and the benefit diminishes as SPM capacity increases. The chapter also reports that the synergy can be escalated at certain balanced capacities between SPM and data cache and establishes some guiding insights on employing the hybrid fast memory configurations from all these observations.

The method to deploy procedure stack frames across multiple memory units is also compared to other studies. When procedure stack frames are distributed across SPM and other memory
units, automatic type data organized in the stack frames can be managed at two different granularity levels, one at an entire procedure stack frame level [17, 77, 22, 88] and another one at an individual data object level [8, 9, 23, 25, 101]. Both the approaches induce large runtime overhead since they employ a dynamic SPM management scheme (explained in Chapter 2) or require fastidious attention on implementation since the traditional stack frames are not designed for a situation using the multiple frames.

This study proposes the stack split technique, which is a viable stack frame splitting approach and toolchain support described in Chapter 3 [41]. The proposed method is novel in several ways: (1) the stack split method is an assembly rewriting technique rather than a technique adjusting stack pointer values or requiring hardware support, (2) this approach enables fine-grain use of the SPM for stack access without the help of compilers, while preserving the stack-based allocation of the automatic variables, (3) overhead is not large (under 20 clock cycles per additional stack pointer in our implementation) since only several additional instructions are inserted per procedure, and (4) it implements a static SPM management method but automatically supports dynamic management of the SPM contents thanks to the innate properties of the stack frames. To summarize, the stack split scheme maintains the dynamic stack-based allocation/deallocation nature of automatic variables through assembly rewriting. The method provides efficient memory use, and this is especially valuable for SPM since its size is limited.

The last aspect of this study to be highlighted is a rediscovery of the benefit attainable from the integer linear programming (ILP)-based SPM allocation and a memory system using only SPM. Various SPM allocation methods modeled the allocation problem into the 0-1 Knapsack Problem [43] solved by ILP [8, 9, 90, 10, 95, 94, 107, 109, 108, 110, 112, 72, 96, 97, 16, 19, 23, 24, 22]. Those methods transform the SPM allocation problem to a problem maximizing or minimizing objective functions which consist of linear sums of benefits and costs by different types of memory access. Furthermore, only a few studies explore improvement of the worst case performance [16, 96, 97, 23, 25], which is critical in designing hard real-time embedded systems. In addition, multi-threaded executions were not discussed widely, except for in a few studies [21, 106, 75, 74, 97], although those studies worked with non-real-time settings. For example, Panda et al. commingled memory trace randomly to emulate context switching [75], and Francesco et al. and Verma et al. considered simple round-robin scheduling [21, 106]. Suhendra et al. traced disjoint lifetime of program code in simple preemptive multi-tasking embedded systems [97]. Beyond this series of research, DARTS integrated the ILP-based SPM allocation scheme, multi-threaded real-time task scheduling and the disjoint lifetimes of data on the worst case execution path into ILP formulas at a different granularity level of data [23, 25].

Chapter 4 [41] extends DARTS to a full implementation, targeting the ARM processor platform. Due to the complexity of implementation, very few studies actually provided a
toolchain allocating data automatically to SPM, applied the new allocation to real programs, and measured real performance on a real hardware system. This work evaluates the performance benefit of DARTS with a real target system having an ARM processor core and a preemptive real-time operating system modified to support the preemption threshold scheduling. Automatic type data are deployed into SPM by the stack split technique. Test programs having all types of data access and larger data sets are experimented, which diminishes the benefit since dynamic type data are not handled optimally by the current version of the toolchain. This study shows that the suggested method may be stronger since performance is improved significantly even under those realistic circumstances with some constraints.

This study discusses the synergy when using the hybrid configuration of SPM and data cache (Chapter 5 and 6) as well as the sole configuration of SPM (Chapter 4) in real-time embedded systems. The synergy can be explained qualitatively; allocating code and data into SPM increases the effective capacity of cache memory without performance loss. Then, residual code and data can take advantage of the larger cache capacity with less contention. This improves the cache performance and eventually system performance with little timing variability by memory access. Particularly the discussion on the hybrid configuration aims at verifying two hypotheses: (1) allocating data variables into SPM employs some synergy in memory access performance by improving data cache performance, and (2) the synergy is escalated at a certain balanced capacity between SPM and data cache.

Regarding these hypotheses, several points are articulated here; the discussion in Chapter 5 and 6 does not address directly how the synergy improves the worst case performance or the worst case performance analysis which is considered critically when designing hard real-time embedded systems. The chapters lack of the worst case timing analysis but focus on average case performance, when SPM and data cache are provided simultaneously, although such worst case timing studies remain a proposed direction for future study. This study also does not claim that SPM always fits better for real-time embedded systems. As long as cache misses can be bounded safely by the static cache miss analysis methods introduced in Chapter 2, our target systems can be designed feasibly in the real world. This work does not claim that SPM always outperforms cache memory since we do not explore design space for choosing the most beneficial cache parameters.

However, this study addresses that employing SPM along with cache memory can reduce cache misses directly and significantly thanks to the synergy. The synergy between SPM and data cache reduces cache contention. Furthermore, according to the results reported in Chapter 6, a small portion of data variables generate a large amount of data memory accesses. Even small SPM can manage those memory accesses effectively. Hence it is anticipated that reduction of cache misses on the average case possibly and sufficiently improves the worst case performance and helps the bounding of the worst case performance of our target real-time systems. Moreover,
energy consumption by SPM is much less than that of data cache, and many modern systems are equipped with SPM and data cache at the same time. Therefore, recognizing and balancing the synergy between SPM and data cache are worth exploring.

The contributions of this study can be summarized as follows:

• This study reveals performance synergy between SPM and data cache. A novel cache miss analysis concept – intrinsic miss and interference miss – and SPM allocation heuristics are suggested to accommodate the synergy for multi-threaded preemptive real-time embedded systems.

• This study discovers that the memory access of data variables are concentrated to a small portion of the data variables, and this distribution leads to diminishing returns as SPM capacity increases. This phenomenon establishes guiding insights to balance the synergy between SPM and data cache memory in various hybrid configurations.

• This study proposes stack split, an assembly rewriting scheme, to deploy the traditional stack frame of procedures across multiple memory units with small runtime overhead.

• This study extends data allocation with real-time scheduling (DARTS) to a full and feasible implementation targeting the ARM processor platform. The DARTS scheme is reevaluated with experimental results obtained from a real ARM target system, real test benchmarks and real SPM and data cache.
Chapter 2

Review of Related Work

2.1 Program Component Allocation into SPM

Allocating program components efficiently into SPM has been actively studied so far. Several orthogonal bases can categorize the existing research achievements into several groups. Program code and data are allocated into SPM either separately or together. Some work targets multi-threaded systems, while other work focuses on single-threaded systems. Some studies consider cache memory along with SPM into their SPM allocation process, while other studies do only SPM. Management methods of the SPM contents also categorize the work. However, regardless which allocation method is accommodated across all these categories, timing predictability of memory access is improved since the program components serviced by SPM are recognized predictably. Especially the two different management methods of the SPM contents are explained here since later parts of this section divide the already existing research studies by the SPM management methods they address as well as by other factors.

One method called static management locates and loads the SPM contents statically before a target system starts running and does not update the SPM with new contents once it is loaded. This method is not concerned with how to replace the SPM contents during the execution of a program. It lacks extra code to manage the SPM contents dynamically. Hence, the static method imposes no extra code execution and very low runtime overheads on target systems. But this method utilizes the SPM space less by limiting exploitation of spatial and temporal reuse patterns of the program components, since the SPM contents reside in fixed locations until end of the program’s execution.

Another method called dynamic management replaces the SPM contents while the target system is running. At the appropriate time point in the middle of execution, the SPM contents are loaded by the inserted code for the contents transfer. With dynamic management, SPM can behave more like cache memory, i.e., it takes advantages from predictable spatial and temporal
locality of the program code and data for some given SPM capacity. The dynamic method is usually more flexible than the static method at the cost of much higher run-time overheads, and the overheads increase as more components are allocated into SPM. Furthermore, if the loading cost of some components is larger than performance benefit, the components have no reason to allocate into SPM even though SPM space is available.

2.1.1 Program Code

Access patterns to program code are more regular than access patterns to data. Multiple instructions unite to a basic block, one large execution block where the program’s control flows on straight. The control flow changes only on branches, jumps, function calls and returns. In most cases, these changes in control flow head toward instructions whose addresses are known explicitly. This regularity of the access pattern promotes allocation of the program code into SPM.

One stem of work in this field [4, 5, 108, 110] takes the static approach, and another one [94, 20, 37, 97] takes the dynamic approach. Egger et al. considers both of the management methods at the same time in one of their studies [19]. In addition to all these studies, one body of work [10, 95, 107, 109, 106, 72, 73] manipulates both the program code and data, which is discussed in Section 2.1.2.

Angiolini et al. modeled the SPM allocation problem of program code into SPM to some dynamic programming problems [4, 5]. Their former work [4] performed the SPM allocation by synthesizing partitioned SPMs for particular target applications, as several other studies did [13, 12, 11]. However, this method restricts its applicability to other applications since the SPMs are synthesized only for some particular applications, and source code analysis is required for synthesizing the best partitioned SPMs. Their later work [5] widened the applicability by employing a patching tool, which modifies selected executable code segments at a post-compiler stage, connecting execution of the instructions in SPM and in the main memory. Although this work focused on code allocation and did not consider cache memory actively, several hybrid configurations of SPM and instruction cache were examined. Their simulation results alluded to some performance synergy between SPM and the instruction cache, which is described in Section 5 and 6 of this study.

Verma et al. considered both SPM and instruction cache more actively into their instruction selection algorithm, which improves instruction cache performance and saves more energy [108]. Instruction cache behavior was modeled with a conflict graph whose arcs present conflict misses between memory objects of instructions. The graph provides parameters for the 0-1 knapsack problem [43] modeling, and actual instructions to reside in SPM are selected by solving the problem with integer linear programming (ILP) [43]. Their later work [110] extended this
approach.

Steinke et al. moved instructions (basic blocks) into SPM dynamically by inserting copier functions to original source code [94]. Their proposed algorithm selects target instructions copied into SPM and possible positions of the copier functions by comparing the copying cost. The final selection of the instructions is performed by solving the 0-1 Knapsack Problem based on ILP formulas. Their simulation results show that SPM performs better than cache memory in both performance and energy saving.

Egger et al. modeled code selection problems by the 0-1 Knapsack Problems for performance improvement and energy saving [19, 20]. One of their studies [19] classified code blocks into three categories: ones in SPM statically, other ones in DRAM statically and the others paged dynamically between SPM and DRAM. This work implemented an on-demand paging manager and modified executable binary to call the paging manager at a post-compiler stage. Their other work [20] also categorized the program code into three different groups: uncached, cached and paged. In this work, only the paged code blocks are loaded into SPM on demand, and the cached blocks are supported by a mini instruction cache. The on-demand page loading takes advantage of virtual memory environments [35], and this work is the first approach to access physically-indexed SPM in virtual memory environments. Although these studies are different from each other in terms of SPM management, both methods support machine binary modification techniques, which expands the applicability of the methods since no source code is required.

Janapsatya et al. copied program code dynamically into SPM [37] by using special custom instructions suggested in their previous work [36]. Two metrics called concomitance and self-concomitance capture probabilities of two code blocks, including a duplicated single basic block, and are executed in a temporally correlated manner. Selection of the basic blocks is based on these metrics.

Suhendra et al. presented a method supporting dynamic allocation of code into SPM for preemptive, multi-tasking embedded systems (potentially with multiple processors) to minimize the worst case response time [97]. A chart called the Message Sequence Chart (MSC) captures the sequence of communications between the processes. An iterative analysis on the MSC identifies disjoint lifetimes of the processes and the variables within the processes. These variables can share the same SPM locations.

2.1.2 Program Data

Vast amounts of studies manipulated both the static type data and the automatic type data, while only small amount of research [1, 61, 18, 63] managed the dynamic type data variables. Several studies [62, 77, 17, 22, 88] paid attention only to stack frames of procedures. In each
discussion of the data variable type, the static management methods [76, 75, 10, 95, 107, 112, 89, 90, 8, 9, 23, 25, 72, 96] are reviewed first and the dynamic management methods [39, 38, 40, 100, 101, 109, 106, 21, 51, 16, 115, 116] follow.

In one example of early work in this field, the partitioning of scalar and array variables to SPM and data cache was studied by Panda et al. [76, 75]. Their work is close to this study in the sense that SPM and data cache are considered actively at the same time. The former work [76] focused on a single program system, and the later work [75] extended to cover multiple levels of memory hierarchy and context-switching of multiple programs, although the context-switching is emulated as randomly switching memory traces of three tasks. These works aim at minimizing cross-interference between different scalar and array variables by using several metrics reflecting the variable’s lifetime, size, access frequency, and potential cache conflicts inside of loops. The potential cache conflicts were estimated statically and loosely by examining the program source code for array accesses. All of this information is required a priori for the allocation process. The synergy between SPM and data cache was partially reflected in their experimental results, such as execution time and external memory access counts.

To our best knowledge, one study by Banakar et al. [10] and another study by Steinke et al. [95] are the first trials which compared performance and energy consumption by SPM to that of cache memory. The former work [10] used circuit models provided by CACTI [117] to estimate memory cell area, energy consumption and performance of SPM and cache memory. The selection algorithm of SPM contents was demonstrated in the other work [95], which detailed the algorithm to reduce energy consumption by a program through extending a compiler. The energy consumption by program code and data was modeled into a 0-1 Knapsack Problem, and the most frequently accessed program components were selected for allocation into SPM. In some trace-based simulations, SPM was estimated to occupying less chip area, consuming less energy but achieving better performance.

One work by Verma et al. [107] extended the work by Steinke et al. [95] to enable maximal SPM usage. They partitioned array variables into smaller pieces so that the SPM usage could be improved at a finer granularity level. One study by Wehmeyer et al. [112] compared average case performance and WCET estimation of SPM equipped systems to those of unified cache equipped systems. They used the allocation algorithm suggested by Steinke et al. [95] and the energy consumption model demonstrated by Banakar et al. [10].

Some methods of Sjödin et al. [89] addressed the allocation of static variables (e.g. globals) with a greedy heuristic. Their later work [90] allocated both static type data and automatic type data (i.e. stack variables).

Avissar et al. proposed an allocation scheme covering global and stack data variables based on the 0-1 Knapsack Problem [8, 9]. Their studies found the most frequently accessed global and stack variables to minimize average case memory access latency of a program. These studies
are close to DARTS [23, 25], in the sense that the lifetimes of the stack variables captured from
the call graph is brought into the ILP formulas. However, the extended DARTS described in
Chapter 4 considers the lifetimes of data objects at task level with the preemption graph and at
the procedure level with the call graph together. Another similarity about maintaining multiple
stack frames for one procedure is discussed later. One more interesting thing is that diminishing
returns as SPM capacity increases is also observed in their result, although the authors do not
mention this. This study investigates this phenomenon comprehensively in Chapter 6.

The allocation algorithm of Avissar et al. was used in two studies by Nguyen et al. [72, 73].
The authors proposed software loader routines to implement new data allocation into SPM,
assuming the SPM size is unknown at compile time. The loader discovers the SPM size,
selects program code, static data and automatic data, and finally modifies machine binary.
The binary has been already compiled considering the modification by the loader routines.
Hence, the loader routines stretch portability of the target applications toward different size
SPMs, although source code of the application needs to be recompiled for this. One later study
[73] extended the previous work by renaming the loader to the installer, detailing the SPM
contents selection algorithms and addressing real world issues such as handling library and
separate compiling.

Suhendra et al. optimized the allocation of data variables of the worst case [96] rather than
those of the average case. Since the profiled information contains infeasible paths, a branch-
and-bound search was suggested as well as two heuristics in order to reduce the running time
of the search.

Ghattas et al. combined a real-time task scheduling scheme with a SPM allocation scheme,
which is named to Data Allocation with Real-Time Scheduling (DARTS) [23, 25]. Unnecessary
preemptions are disabled by Preemption Threshold Scheduling (PTS) [111, 85] to increase
disjoint lifetimes of automatic type data variables at task level. This, in turn, increases SPM
utilization further since those data variables can share the same SPM space in an overlaid
manner, as Suhendra et al. did [97]. It handles the data variables on the worst case execution
path and multi-threaded real-time task scheduling through the proposed preemption graphs.
All these are integrated into ILP formulas at a different granularity level of data. In their
implementation, static variables are not managed dynamically so that the runtime overhead
to manage them in the dynamic management scheme is avoided. Distributed procedure stack
frames across multiple memory units implemented the dynamic management scheme due to
the nature of automatic variables. However, they have evaluated the benefit of the scheme by
simulating randomly synthesized workloads and targeted the AVR 8-bit processor platform.

The dynamic management of the data in SPM has been implemented through compiler
supports since some data transfer code is required to update the SPM dynamically. Kandemir
et al. focused on data arrays accessed inside of well-structured loop kernels of some specific
filed applications such as image and video processing [39, 38, 40]. One part of the work [39, 40] decomposed the arrays into tiles so that the arrays could be loaded partially into the small size of SPM. They suggested a series of compiler techniques to partition the SPM space, generate loop access patterns using the tiles, and restructure program code for the data transfer. In another study [38], the authors suggested a different compiler technique, leaning on the reuse vector and matrix for multi-level SPM hierarchy. They proposed their upgraded compiler could be used when designing new SPM hierarchies from applications as well when allocating for already existing hierarchies.

Udayakumaran et al. also suggested dynamic SPM management methods supported by compilers [100, 101]. A profile-driven cost model supported by the compilers identifies possible program points such as functions, conditionals or loops entries and exits into which data transfer code can be inserted. The Data Program Relationship Graph (DPRG) associates timestamps to the program points of interest to determine lifetime of the SPM contents. From this analysis, the most promising data are selected by a greedy heuristic. Their later work [101] extended this approach to present actual SPM layout and code generation methods. However, implementation of the new SPM allocation is performed through indirect methods since stack variables were converted into global type variables. For the global variables, extra variable symbols are inserted, which copy their original values when they are about to be accessed in SPM. (This implementation method is discussed in more detail together with Avissar et al.’s scheme later.) Udayakumaran et al.’s work overcomes limitations of the work by Kandemir et al. [39, 40]. Kandemir et al. limited their study to well-structured loop nests. But Udayakumaran et al. worked with a fully general dynamic method covering global and stack variables. Furthermore, they supported a whole-program analysis across all control structures, while Kandemir et al. manipulated each loop nest independently.

Verma et al. applied a global register allocation scheme [15, 29] for SPM allocation that overlays multiple program components onto the same SPM space [109, 105]. Live range analysis on profiled program information identifies the overlaying SPM contents. For better utilization of SPM, the ILP formulas finally determine the SPM contents, and inserted SPM spilling code manages the SPM contents dynamically. However, the work lacks explanation of practical implementation methods for new allocation, such as how to assign the same memory address to multiple memory objects and how to handle branch instructions. Another work by Verma et al. [106] suggested methods for sharing SPM space with multiple processes for energy saving, although only round-robin static scheduling was assumed. They examined three different SPM management schemes: partitioned and exclusively assigned to a process, non-partitioned and fully shared by all the processes, and hybrid types of the two methods.

Francesco et al. suggested a new hardware and software scheme together to manage SPM contents dynamically [21]. The hardware scheme uses Direct Memory Access (DMA) which
can reduce data transfer cost from main memory to SPM. To make the DMA-based data transfer easier, software application programming interfaces (API) are also supported at high level language. They simulated performance and energy consumption, and their management scheme outperforms explicit data transfer schemes inserting copier function into code. However, program source code needs to be rewritten with the APIs in order to apply this scheme.

Li et al. proposed a compiler technique called memory coloring to allocate data arrays into SPM [51]. In their scheme, SPM is partitioned to pseudo registers having predefined sizes, and live ranges of the data arrays are analyzed and split to fit the SPM partition. Then, register allocation algorithms based on Graph Coloring [15] maps the partitioned SPM to the data arrays. Some data transfer code is also inserted.

Deverge et al. managed both static type and automatic type data variables in SPM dynamically to reduce a single task’s WCET [16], while Suhendra et al. used a static method for the same purpose [96]. Hence, their experimental evaluation involved modifying compilers to generate executable code, including data transfer code, to analyze timing of tasks, and to adjust a task’s resulting timing by compensating the data transfer cost. They also managed pointer-referenced memory access.

Although much research has allocated static type and automatic type data into SPM, moving automatic type data variables into SPM is not straightforward. The automatic type data variables of a procedure are located in a stack frame of the procedure, and the traditional stack frame is organized by compilers which are not aware that the stack frames can be deployed across multiple memory units. Several studies [8, 9, 62, 77, 23, 25, 22, 88] paid attention to this problem.

Avissar et al. proposed splitting stack frames across different memory units [8, 9]; their concept is most closely related to the stack split scheme devised by this study in Chapter 3. Their method also employs extra stack pointers to keep multiple stack frames in SPM and other memories. They suggested two alternative strategies to mitigate the expected overhead from managing different versions of the stack pointer at the prologue and epilogue of a procedure; (a) for short-running procedures, using one stack pointer and forcing all the variables in one procedure into the same memory and (b) for long-running procedures, using distributed stacks for the procedure’s variables. Method (a) avoids the overhead, while method (b) tolerates the overhead. In other words, the work presented methods to avoid splitting frames except for long-running functions. However, no actual overhead cost was presented. Udayakumaran et al. and Ghattas et al. also mentioned that multiple stack frame pointers can cause heavy overhead during runtime and did not present concrete pictures of implementation [101, 23, 25].

These anticipations by the authors has resulted in an unfounded belief that supporting distributed and split stack frames for a procedure is too costly to be viable [8, 9, 100, 101, 96, 97]. However, one study by Udayakumaran et al. [100] has recognized the memory inefficiency of
representing automatic variables as statics and has proposed performing data liveness analysis to determine which variables interfere, for example by examining the call graph. Hence, to the best of our knowledge, all previous implementations converted automatic variables into static variables in order to simplify experimentation.

The trouble is that this conversion eliminates the possibility of memory reuse because the lifetimes of the variables are expanded, increasing memory requirements. All such converted variables are now live for the entire program, rather than merely being live for the function. Hence these variables all interfere with each other and must be allocated exclusive storage, reducing the number of variables which will fit in SPM and thus reducing program speedup. Another drawback of this alternation is that accessing static variables is likely to need more instructions than automatic variables (e.g. in the ARM architecture) although such requirements vary according to processor architecture.

Rather than splitting stack frames, three studies [17, 22, 88] moved stack data at procedure granularity level. Dominguez et al. tried allocation of stack frames of recursive procedures [17]. Their proposed method updates the values of stack pointers selectively to maintain some stack frames of the recursive procedures inside of SPM within endurable recursion depths. Gauthier et al. selected the most frequently accessed stack frames by solving the 0-1 Knapsack Problem and moved the frames into SPM via a fully software manner [22]. Shrivastava et al. implemented a software SPM management (SPMM) system, which manges stack frames to be located in SPM [88]. The SPM space is managed by the SPMM in a circular way so each frame can be evicted in a round-robin manner. Hence, their technique requires neither profiling nor SPM size known as a priori. Runtime overhead due to the SPMM is reduced by consolidating the SPMM function calls to the best.

One interesting discussion in the work of Gauthier et al. [22] is their mention of two issues encountered when moving automatic type data variables into SPM at the variable level; (1) in some situations, a source code is not available, such as library functions and (2) it is costly to update address values of a stack pointer every time when data variables are accessed across in SPM and main memory. In the case of the first issue, the stack split scheme also suffers from the same problem. But we can recompile the library so that the library can also benefit from SPM. Or we can simply leave them so that other data variables can take advantages of SPM. In the case of the second issue, the stack split scheme employs more stack frame pointers rather than updating the address value of the single stack pointer.

Conventional stack frames hold automatic-type data variables of procedures and activation records of the procedures such as the return addresses from the procedure calls, evacuated values of general registers, and so on. Mamidipaka et al. show that a severe portion of memory access traffic is caused from the activation records, and they added an on-chip stack memory being separated from main memory to hold those data [62]. They proposed reduction of data bus
traffic, cache pollution and finally energy consumption and reported reduction of cache accesses and consequent energy saving by evaluating one of their proposed schemes. This benefit is one kind of the synergy between SPM and data cache, although the authors did not mention that.

The previous work outlined above keeps the assigned address to SPM constant throughout a program’s execution. Thus, frequent updating of address values maintained by the stack pointers is required to deploy the procedure stack frames across multiple memory units. If necessary, source code or machine binary also needs to be rewritten. Park et al. assign different addresses dynamically to SPM with help of a memory management unit (MMU) [77]. Thus, the stack frames of a program can be positioned in SPM while neither modifying the source code nor updating the address value in the stack pointer. When accessed addresses are overflowed or underflowed within a limited capacity of SPM due to a certain address mapping, the MMU’s fault handlers keep track of the code executed in SPM and manage replacement of the SPM contents. This method was also used in one study by Egger et al. [20]. In their experiment results, the synergy is observed that one hybrid configuration of SPM and data cache outperforms data cache only configurations and SPM only configurations, although the authors were not aware of it.

So far, studies to manipulate static type and automatic type data variables have been introduced. Comparing the static type and the automatic type data variables, dynamic type data variables stored in heap memory regions are not easy to manage since their locations move around all ways in the memory. Despite much leaning towards dynamic type data which can be accommodated by data cache rather than scratchpad memory, two studies [2, 1, 61] showed that such assumptions may result from a misunderstanding.

Absar et al. utilized SPM for dynamic applications whose data access patterns are irregular or non-manifest [2, 1]. Their former work [2] investigated a scheme of dynamically-managed SPM. The later work [1] compared the performance of SPM to that of a data cache analytically using hit rates of data objects and a corresponding statistical model. In their analytical model, the hit rate of a data object is represented as a function of the access probability, and the access probability is calculated in a simple analytical manner since they considered dynamic data types as having obvious memory access sequences such as linked lists and trees. Hence, the scheme requires a priori information about the dynamic type data objects by examining the program source code. In the case of the other types of data, the complexity of the access probability calculation scales badly, as the analysis scope grows to the whole program or the whole system scope due to complex program control flows and task scheduling behavior.

The work of Mamagkakis et al. [61] analyzed the spatial locality of dynamically allocated data through instrumentation and a custom dynamic memory management library. Based on this information, the dynamic type data are grouped into pools, and the degree of activity of each pool is computed by the amount of access in the pool divided by the pool size. The
allocation into SPM follows a decreasing degree of activity under some SPM capacity.

Dominguez et al. allocated the heap memory region into SPM [18]. Data transfer between the SPM bins and main memory is managed dynamically by inserted code. To overcome the uncertainty of dynamic type data size at compile time, they partitioned SPM into fixed size bins. The bins hold only part of dynamic type data variables (when the data size is larger than the bin size). McIlroy et al. solved this problem by supporting full size dynamic type data positioned in SPM by their SPM allocator [63]. They managed SPM space with coarse grained bins which have fixed size, and fine grained bins which have variable size at the same time.

2.2 Timing Predictability of Cache Memory

Much work aims to improve timing predictability of the cache memory while reducing dynamicity of cache content misses. This problem has been tackled by many researchers via two tracks. One track of the work [55, 45, 49, 48, 7, 32, 31, 65, 70, 67, 68, 69, 114, 113, 71, 3, 58, 60, 59, 26, 27, 104, 83, 84, 91, 93] estimated and bounded cache misses or response time of real-time tasks so that the dynamic behavior of the cache memory can be reflected safely into timing analysis of a target system. Another track of work [46, 47, 66, 79, 78, 80, 102, 103] enhanced the predictability of cache memory directly by locking the cache lines or partitioning the cache space.

2.2.1 Bounding of Cache Misses

A series of studies [55, 45] calculated memory references directly. Lim et al. computed WCET considering an instruction cache and a data cache for RISC processors [55]. They associated each program construct with Worst Case Timing Abstraction (WCTA), which contains timing information of possible worst case execution paths based on the timing schema [86]. Kim et al. extended the work to covering static and dynamic memory accesses but neither arrays nor pointers [45]. They suggested two schemes using global data flow analysis and data dependency analysis, which reduce miss-classification of memory accesses to the dynamic load and store and tighten the WCET estimation consequently. Those two studies tackled direct mapped data cache.

Arnold et al., Mueller et al. and Heally et al. analyzed timing behavior of instruction cache through static cache simulation based on categorization of instructions [7, 32, 31, 65, 70, 67, 68, 69, 114, 113]. Arnold et al. [7] and Mueller et al. [65, 70] categorized instructions into four groups – first miss, always hit, always miss and first hit – by analyzing the Control Flow Graphs and the Call Graphs. The simulation results are fed into WCET estimation.

Later studies by Mueller [67, 68, 69] employed the data flow analysis (DFA) framework in addition to the static simulation. In one study of the author [69] integrated all those achievements
as an extension. In that study, timing analysis considering instruction caches was performed via static cache simulation. Possible cache line contents were computed and estimated by concept of the abstract cache state (ACS) and the data flow analysis (DFA), which was followed by a timing tree based WCET estimation. White et al. extended this categorization concept to simulating direct mapped caches [114] and their later work [113] applied the DFA scheme as well.

Li et al. also modeled direct mapped instruction caches into the WCET analysis [52] and calculated possible cache contents along with a program’s timing using the ILP formula [53]. Their later work [54] extended the earlier work to set associative instruction caches.

Lundqvist et al. used symbolic execution to detect predictable and unpredictable memory accesses [58], and other work [59] moved non-analyzable memory accesses into a non-cacheable memory region to improve the timing predictability. They also have introduced timing anomalies due to cache misses so that some cache misses cause shorter execution times than cases of cache hits [60].

Lee et al. estimated the worst case response time by bounding cache related preemption delay (CPRD) due to inter-task cache pollution with the useful cache block (UCB) concept [49, 48]. Negi et al. extended this work to estimate the CPRD for both the preempted and the preempting task with a program path analysis [71]. All these studies covered only the instruction cache, even though Lee et al. mentioned the extendability of the concept to data cache [48]. Altmeyer et al. suggested a more detailed concept, named definitely cached UCB (DC-UCB), and tightened the CPRD estimation [3] more than the former work. In the UCB analysis, cache misses are counted doubly on timing analysis and context switching cost analysis. At every possible preemption point, the scheme statically estimates cache blocks that will be used by tasks after that time point.

Another group of work [26, 27, 104, 83, 84] predicted data cache misses from memory access patterns. Ghosh et al. solved cache miss conditions through vector equations in the iteration space of loops with some restrictions such as direct mapped cache, aligned cache lines and so on [26, 27]. This work affected other studies [104, 82, 84], which also analyzed and estimated cache misses statically. Since the initial concept of the cache miss equation (CME) by Ghosh et al. was built on several constraints, making application of the equation less practical, Vera et al. developed it further [104]. They computed the reuse vectors to calculate cache accesses within loops, provided that all array accesses are affine functions of loop induction variables and the loops are perfectly nested.

One study by Ramaprasad et al. [83] estimated cache miss statically based on the CME. They extended it to scalar variables and more complicated loop structures while relaxing the restriction of the CME and suggested the forced loop fusion scheme to make iteration spaces of the loops rectangular. The data cache miss patterns were obtained statically from the
extended CME and fed into their static WCET analysis framework. This study was extended to support multi-tasking preemptive environments later [84]. They analyzed preemption points, corresponding preemption delay, and finally worst-case response time of tasks.

Staschulat et al. estimated WCET with direct mapped data cache [93]. They classified memory accesses to a predictable group and a non-predictable group. For the predictable group, data cache misses were obtained from data cache simulation and data flow analysis. For the unpredictable group, their own analysis framework was used, which is based on the concept of useful cache block used in some previous studies [48, 71, 92].

### 2.2.2 Improving Predictability of Cache Memory

Some changes in hardware and software can improve the timing predictability of cache memory. Cache partitioning keeps impact of dynamic cache behaviors within partitions rather than the all cache lines so that data in other partitions can avoid unnecessary disruption of cache states. In other words, cache partitioning reduces interference on cache lines between tasks by isolating partitioned cache space to the tasks. Kirk et al. divided cache space into one large shared partition and small segmented partitions [46, 47]. Tasks requiring predictable cache hits are allocated into the small partitions, while others are assigned to the shared pool. Mueller focused on compiler support for this situation, i.e. non-linear addressing to instructions located in separated partitions [66].

Cache locking techniques freeze the replacement of cache lines with software and hardware support. Static locking schemes lock cache lines for the entire lifetime of a program after the cache contents are loaded into the cache lines. If the cache contents can be replaced by the software, the scheme is called dynamic locking. Three studies by Puaut et al. [79, 78, 80] investigated instruction cache locking. One of the studies [79] suggested two algorithms to select and statically manage the instruction cache contents for multitasking real-time systems. One of the approaches purposes minimizing processor utilization while the other one aims at reducing cache interferences between task preemptions. Both the schemes propose optimization of WCET. The dynamic locking cache has been suggested in their later work [78], which proposes two algorithms to select the target instructions and reloading points. Another study by the authors [80] compared the worst case performance of instruction cache to that of SPM based on more generalized selection algorithms proposed in their prior work [78]. Their experimental results showed that the locked instruction cache and SPM show almost the same performance, but sometimes cache block sizes and basic block sizes can make a difference.

Vera et al. combined a data cache locking scheme with a compile-time static cache analysis scheme to tighten estimation of the worst case execution performance further [102, 103]. They modified a program by inserting locking and unlocking instructions whose target data were
identified by the reuse vector analysis suggested in Wolf et al. [118, 119]. Data which the analysis identifies as unpredictable is locked. After this transformation a static data cache analysis scheme based on the CME estimated the worst case performance, including the data memory access performance.

In research of general purposed systems, cache memory has been more widely studied. Reduction of cache contention also has been researched continuously. Taylor et al. proposed a technique named page coloring, which reduces cache contention by mapping sequential virtual pages to physical pages so as not to conflict with each other [98]. They assigned different colors to the sequential physical memory pages with a maximum number which is determined by size and associativity of cache memory, and a free page list is also constructed to have different colors, too. However, this scheme can lead to inter-address-space contention in multi-process execution since commonly used virtual addresses from different address spaces may be mapped into the same physical addresses. Hence, Kessler et al. developed the scheme further by hashing process identifiers (PID) into the coloring [44]. The PID hashing reduces the inter-address space contention between multi-processes. These two studies affect many other studies [14, 87, 120] later. These are not discussed in depth in this study since all those studies target general purpose systems operating virtual memory systems with multi-level cache memory or multiple processors. However, all this work also stands in the same line with this study in the sense that it also tries to reduce unnecessary cache contention.
Chapter 3

Stack Split

3.1 Introduction

Data variables can be categorized depending on where they are stored. Static type variables are allocated to a *data* section (if initialized) or a *block started by symbol* section (*bss*) (if not initialized). Those variables have fixed addresses which are determined statically during compiling and linking. Automatic type variables are associated with procedures and are located in the procedure’s stack frame. Their addresses are decided automatically by the current stack frame. Addresses of dynamic type variables are defined in a heap memory region and managed dynamically by management libraries, also during the program’s execution. (This type data are not handled in this study.)

The data variables also can be categorized based on their visibility scope within a program as either global or local variables. Global variables are generally static types or dynamic types, and can be accessed through the whole lifetime of a program. Local variables are automatic type, static type or dynamic type, and can be referenced and updated locally inside of a procedure. Static type local variables are allocated into *data* or *bss* sections, yet they are only accessible within the procedure.

The static type data variable’s location in a memory system can be controlled by a linker script. The linker script specifies the memory location to be loaded with the static data variables including globals and static locals. The static variables occupy the memory location from the program’s initiation until the the program’s end. Thus new locations of the static variables can be specified in the linker script. Particular locations in SPM also can be assigned for the static variables by the linker script.

The automatic variables are allocated into a procedure stack frame by a compiler. Traditionally, a stack frame of a procedure or a task consists of contiguous memory blocks completely contained within a single memory unit, and a procedure maintains only one stack frame. The
automatic variables are accessed by offsets from a stack frame base of a procedure. But the stack frame in the single memory unit is not an optimal and efficient use of memory space because data variables in the stack frame are accessed with different frequencies. Hence, allocating a stack frame across different memory units is required for efficient memory utilization, but involves changes in compilers, linkers and programming conventions.

This work proposes stack split, a practical way to solve this problem; two schemes to statically split a single stack frame of a procedure into multiple memory units are proposed. The schemes can be accommodated without support by compilers and linkers since the assembly code of a program is rewritten. The schemes support the granularity level of each data memory access so that coarser granularity levels, such as procedure or task levels, are also supported automatically.

The remainder of this chapter is organized as follows. Section 3.2 introduces stack frame models for before applying schemes and for after applying schemes. Section 3.3 explains the schemes in detail.

3.2 Stack Frame Models

As a procedure calls other procedures, the stack frames of the called procedures are augmented on top of the caller’s stack frame and discarded when returning back to the caller procedure. The automatic variables of the called procedures are accessed by offsets from the new stack frame bases. This conventional stack frame model is converted into a new one to set up a baseline for the proposed scheme.

3.2.1 Original Stack Frame Model

There are many stack frame variants possible, due to their dependence on processor architectures, procedure calling conventions, and compilers. However some common features can be extracted, and this study regards these features as an existing stack frame model. Several common characteristics of widely used modern popular platforms, such as ARM [6], are listed below:

- Either a stack pointer (sp) or a frame pointer (fp) provides reference addresses when accessing the stack.
- \( sp \) indicates the top of current stack.
- \( fp \) indicates the base address of stack frame of a procedure in the stack.
- At prologue and epilogue of a procedure, \( sp \) and \( fp \) are adjusted for allocating and deallocating a stack frame of the procedure.
void wolf(int n) {
    int i, j;
    /* wolf’s body */
    foo(i, j);
    return;
}

int foo(int a, int b) {
    int h, k, m, r;
    /* foo’s body */
    return(r);
}

(a) Example code

(b) Example of original stack frame before applying stack split

(c) Example of new stack frame after applying stack split

Figure 3.1: Example code and stack frames
Before and after a procedure call, the current stack frame is modified to support arguments passing and code addresses saving for the procedure call.

Since this study focuses on a method for splitting a conventional stack frame, operating systems issues are not considered. For example, it is obvious that \( sp \) and \( fp \) are saved and restored, respectively, when a current running task is switched out.

Figure 3.1b depicts an example snapshot of stack memory between two functions – wolf() and foo(). This example is simplified for explanation, but it is still based on the GNU ARM cross compiler [28]. The left-most diagram in the figure shows the stack before jumping to the callee function foo(). The next shows the stack after arguments of the callee function foo() are pushed into the stack frame. The right-most is obtained after storing activation records of the callee function foo(), and allocating local variables of the callee function. The numbers in the parentheses of \( fp \) and \( sp \) are time stamps to distinguish different versions of the register values.

### 3.2.2 New Stack Frame Model by Stack Split

To split the traditional stack frame, extra stack frame pointers are employed. Those registers are called as *sub stack frame pointers* or *sub frame pointers* simply and denoted by \( fp' \). Each sub frame pointer indicates a different stack frame from an original stack frame of a procedure. The stack frames located in new memory units are called *sub stack frames* or *sub frames* simply. Therefore, the number of sub frame pointers is same as the number of sub stack frames. For example, if a procedure’s stack frame is split onto three different memory units, the procedure has two sub frame pointers, \( fp'_{1} \) and \( fp'_{2} \) as well as its original frame pointer \( fp \).

Notice that no extra stack pointers are required. This is because the proposed scheme is a static method so that the size of the sub stack frames is already known when the original frame is split. Then, the new stack frame model can be described as below:

- Either stack pointer (\( sp \)) or frame pointer (\( fp \)) are employed to provide reference addresses when accessing an original stack.
- Multiple sub stack frame pointers (\( fp' \)) are employed to give reference addresses when accessing sub stack frames.
- \( sp \) indicates the top of the current stack.
- \( fp \) indicates the base address of the stack frame of a procedure in the stack.
- Each \( fp' \) indicates the base address of each sub stack frame of a procedure.
- At the prologue and epilogue of a procedure, \( sp \), \( fp \), and \( fp' \) are adjusted for allocating and de-allocating all stack frames of the procedure.
Before and after a procedure call, the current original stack frame is modified to support argument passing and code addresses saving for the procedure call.

To summarize this model, the sub stack frames keep part of an original stack frame, and they are responsible only for keeping the automatic variables of a procedure allocated to them. Other functionalities of the stack frame still remain at the original stack frame.

### 3.3 Stack Frame Splitting Schemes

Moving some parts of a procedure stack frame is a challenging task since it is not supported by current compilers. Moreover, DARTS assigns multiple automatic variables of different exclusive tasks to the shared SPM space. Hence the overlaying of data in SPM must be supported. Two assembly rewriting schemes – **default split** and **dense split** – are suggested to improve efficiency of memory utilization by procedure stacks. The difference between the schemes is the memory layout of the original stack frame after applying stack split.

#### 3.3.1 Default Split and Dense Split

The stack split technique modifies assembly code. We modify load or store operations which reference an original stack frame pointer with a data variable’s original offsets to new assembly code which references sub stack frame pointers with new offsets instead. At prologue of the procedure, sub stack frame pointers are adjusted to point at starting points of the procedure’s sub stack frames in different memory units. At epilogue of the procedure, the adjusted sub stack frame pointers’ values are rolled back to their prior values. This is the same as how traditional stack frame pointers are managed. Runtime overhead to adjust the sub stack frame pointers at the prologue and the epilogue is not huge since only several assembly codes lines are newly inserted. Only two lines are enough per procedure in our implementation. If required, other types of instructions which refer to the original stack frame pointer are modified so that they also can access the sub stack frames correctly.

After splitting stack frames, an original stack frame is retained (including unused space) but sub stack frames contain no unused space. This lets the original stack frame remain unchanged and sets up a new frame in a packed manner in a different memory unit as well as minimizing code modification of original assembly code. The unused space is due to data variables moved into other memory units. The **default split** scheme stops the assembly rewriting here. Thus, this scheme can save programming effort to modify assembly code and reduce risk of programs corruption resulting in incorrect computing results. This approach uses the same space for the original stack frame compared to the dense split explained below.
With *dense split*, the unused space in the original stack frame is removed to save memory space. Each automatic data variable occupies its space in exactly one memory unit over all system memory units. There is no extra space unused by the variable in its original stack frame or in its sub stack frames. This implies that the original stack frame must be modified to remove the unused space, which was used by the data variable but is not used now. Therefore, offsets of all automatic variables of the same procedure must be replaced with new stack frame offsets. For this, all assembly code which references the original stack frame must be examined and modified if necessary. The process also requires changing the prologue and epilogue of a procedure to adjust the original stack size to the new one. This scheme can optimize memory usage and improve performance by using faster memory most efficiently (densely), but it is more risky since more instructions are examined and modified compared to the *default split*.

Figure 3.1c shows a new stack frame after applying those schemes for the same example given above. In this example, variable \( k \) and \( m \) are allocated into SPM and SDRAM, and they are referenced by \( fp'_p \) and \( fp'_d \), respectively. When applying the default split technique, the variables \( k \) and \( m \) have duplicated memory space over the system memory units. Other variables still stay in the original frame stack without any modification. If the dense split technique is applied, the unused spaces for the variable \( k \) and \( m \) in the original stack frame are deallocated for saving memory space. For this, the code in the prologue of the function foo() needs to be modified to reduce the amount of allocation for the local variables in the original stack frame. As the original stack frame changes, all other offsets inside of the original frame also need to be recalculated. The offset for the variable \( h \) does not need to be recalculated. But the offset for the variable \( r \) must be recalculated so that it can be newly assigned to the space for the variable \( k \).

Figure 3.2 presents a pseudo code for the dense split technique targeting the ARM architecture. The notation \( i(op, dest, src1, src2) \) means an instruction operating \( op \) with register \( src1 \) and \( src2 \) (optionally) into destination register \( dest \). This scheme supports allocation of the finest granularity level of data and explicitly referenced stack data variables since it handles a stack frame offset of a data object directly. Consequently, it also supports splitting coarser granularity levels such as procedure level and task level (e.g. stack frame, or entire stack). For the default split scheme, the lines to handle data variables remaining in the original stack frame (line 6 – 7) and the lines to manage allocation and deallocation of original stack frame (line 8 – 9, line 10 – 11) can be skipped.

### 3.3.2 Concerns in Real World Application

Some practical concerns may arise when actually applying the stack split schemes. In this subsection we discuss issues we encountered in our development and application of the stack
PROC StackSplit(proc) :
01: FOR \( \forall \ i(op, dest, src1, src2) \in \text{proc} \) DO
02: IF \( i.op=\text{LDR/STR} \) AND \( i.src1=fp \) THEN
03: IF \( \exists \) new offset for \( i \) THEN
04: \( i.src1:=fp' \)
05: \( i.src2:=\text{new offset} \)
06: ELSE IF \( i.src2 \) needs to be updated THEN
07: \( i.src2:=\text{offset to be updated} \)
08: ELSE IF \( i=\text{instruction to allocate stack space} \) THEN
09: replace the original stack size with new stack size
10: ELSE IF \( i=\text{instruction to deallocate stack space} \) THEN
11: replace the original stack size with new stack size
12: ELSE IF \( i=\text{last instruction of prologue} \) THEN
13: \( i_{\text{new}}=(\text{SUB}, fp', fp', \text{sub stack frame size}) \)
14: insert \( i_{\text{new}} \) after \( i \)
15: ELSE IF \( i=\text{first instruction of epilogue} \) THEN
16: \( i_{\text{new}}=(\text{ADD}, fp', fp', \text{sub stack frame size}) \)
17: insert \( i_{\text{new}} \) before \( i \)

Figure 3.2: Pseudo code of stack split (dense split)

split. These limitations prevented us from splitting out some stack variables, yet significant performance improvements overall are still obtained.

One possible concern is about debugging since debugging information for local variables will be lost or distorted after splitting stacks. When a debugger examines the value of a local variable within a stack frame, an offset from the stack frame base is referenced. However, the offset information is no longer valid after applying the scheme. But, it is certainly possible to make debug support which recognizes stack frame splitting. This is, in fact, yet another trade-off between performance (optimization) and ease of debugging: increasing levels of code optimization require corresponding debug support. We leave this for future work.

When general registers are used for the sub stack frame pointers, the sub stack frame pointers may be corrupted by external library procedures which are not aware of the stack splitting but using those registers. In our implementation, a few general registers are chosen since they are not used in compiled assembly code. But some external library procedures use the registers so that their values are corrupted. As a solution, sub stack frame pointers may be assigned to different registers rather than the one used so far, particularly for this case. If registers for sub stack frame pointers are not available, memory spills can be considered.

Pointer type variables may make splitting more difficult. This problem is encountered when targets of the pointer variables are allocated into other memory units by the stack split. Since
the targets of the pointer variables are not knowable at analysis time, splitting becomes difficult. However, the pointer variables themselves can be allocated into other memory units without any trouble.

Some data variables can be accessed by indirect referencing instead of by stack frame register. In this case, address calculation are performed before accessing the data variable, and the variable is accessed by the register which contains the final address from the calculation. Then if it is not possible to follow the address calculation, the variable is not split out of the original stack frame.
4.1 Introduction

In this chapter, an SPM-equipped memory system is leveraged aggressively by overlaying data objects into the same memory locations based on the Data Allocation with Real-Time Scheduling (DARTS) method \[23, 25\]. DARTS provides predictably fast memory for multi-threaded real-time systems by using integer linear programming (ILP)-based optimization of data objects’s allocation and preemption threshold scheduling (PTS) \[111\] of tasks.

This study contributes methods and a real, complete and improved framework to allocate automatic and static data variables into heterogeneous memory systems using SPM. The toolchain of this work operates as a post-compiler stage to simplify code development. It analyzes a program’s assembly code statically and performs allocation of data variables. The benefits of the DARTS-based approach are evaluated, and the system aspects of DARTS behavior are discussed using an RTOS-based ARM7 real-time system with several test benchmarks.

Remainder of this chapter is organized as follows. Section 4.2 introduces methods to allocate data objects statically via the DARTS concept. Section 4.3 describes the experiments performed. In section 4.4, the results from the experiments are discussed. Section 4.5 summarizes this chapter.

4.2 DARTS Concept

The DARTS concept was introduced and evaluated previously \[23, 25\]. Here we provide an overview of the DARTS concept and then present the details of how to obtain a new data allocation which utilizes SPM efficiently.
4.2.1 DARTS Cycle

The DARTS scheme allocates data objects into multiple memory banks through three steps; (1) statically analyzing timing and data access of tasks, (2) allocating data variables by solving a 0-1 Knapsack Problem with integer linear programming (ILP), and (3) rescheduling the tasks by preemption threshold scheduling (PTS). See the steps marked with blue color in Figure 4.1. The last two steps (ILP solving and preemption threshold scheduling) can iterate multiple times as they are interdependent with respect to task’s timing. This iteration differentiates the DARTS from other ILP-based allocation schemes [75, 8, 9, 109, 96] as well as combining data allocation scheme with real-time task scheduling algorithm does. While the previous ILP-based SPM allocation schemes obtained new data allocation from single iteration of the ILP solving, DARTS accumulates benefit of SPM via the virtuous cycle.

The DARTS cycle starts with statically determining the total memory access latency of static and automatic data variables on the worst case path. The total memory access latency of each task in a target real-time system is represented as a linear sum of total read and write latency of all data variables of the task, which makes up objective functions for the next step (allocating data objects by ILP solving).

The objective functions representing the WCET of each task are minimized by ILP solving. Only the memory access time portion of the WCET can vary according to a particular allocation of data objects since program code is not handled by DARTS. The constraints for ILP solving are given from the system parameters such as available memory capacity, access latency of each memory bank, task period and priority, and call graphs of procedures. More detailed problem modeling and ILP solving will be explained later.

The new memory allocation obtaining from the ILP solution step enables tasks to be resched-
uled because the task’s execution time may fall. PTS [111] is involved in this phase. The Preemption Threshold Scheduling splits a traditional scheduling priority of a task into two parts; a priority to determine whether a task can preempt other tasks and another priority (preemption threshold) to determine whether the task is preempted by the other tasks. If a task’s preemption threshold is higher than a priority of another task which is trying to preempt the task, the task is not preempted by the another task. Thus, an off-line preemption threshold assignment determines which preemptions of lower priority tasks can be disabled without resulting in missing any deadlines.

After PTS rescheduling, DARTS can repeat the virtuous cycle between ILP-based allocation step and PTS-based rescheduling step. Tasks which cannot preempt each other can share the same memory location in SPM (or other faster memory) for their automatic data variables thanks to their exclusivity of execution in time frame. Hence, this new memory allocation sharing same memory location in SPM results in faster execution for both tasks. The new preemption relationship is given as one of constraints for the ILP solving. Consequently the ILP solving phase repeats, then PTS can seek to disable more preemptions based on the reduced execution time. ILP-based allocating and PTS-based rescheduling can iterate until each task’s timing does not change. When the estimated WCET of each task reaches a fixed point, the final allocation has been obtained.
This concept can be explained with an example in Figure 4.2. The graph of Figure 4.2 is a preemption graph [23, 25] embedding a call graph inside of each task node. In the preemption graph, nodes represent tasks, and directed arcs depict preemption relationships between the tasks. If task $A$ can preempt task $B$ then an arc is drawn from task $B$ to task $A$. The direction of the arc is opposite to actual preemping relationship. However, this notation is helpful to capture the maximum memory usage. In this example, a hexagon represents a task, and a rectangle shows a procedure, and an ellipse depicts a data variable. From the definition of the graph, we know that task $T_{(3)}$ has the highest priority and task $T_{(1)}$ has the lowest priority because task $T_{(3)}$ can preempt task $T_{(2)}$, and the task $T_{(2)}$ can preempt task $T_{(1)}$. $T_{(0)}$ is an idle task.

At the initial state before applying DARTS, all the tasks $T_{(1)}$, $T_{(2)}$, and $T_{(3)}$ cannot share any memory space in the fastest memory, since they have to keep their context simultaneously in the worst case of preemption. This occurs when $T_{(1)}$ preempts $T_{(0)}$, $T_{(2)}$ does $T_{(1)}$ and finally $T_{(3)}$ preempts $T_{(2)}$, i.e. the longest path in the preemption graph.

At the initial state of the example graph, there are 4 preemption cases possible; $T_{(3)} \rightarrow T_{(0)}$, $T_{(3)} \rightarrow T_{(1)} \rightarrow T_{(0)}$, $T_{(3)} \rightarrow T_{(2)} \rightarrow T_{(0)}$, and $T_{(3)} \rightarrow T_{(2)} \rightarrow T_{(1)} \rightarrow T_{(0)}$. In the ideal case, all the tasks can be exclusive to each other, and the preemption threshold value of each task is the same as the highest priority of the system. Then, there exist only three possible preemption cases; $T_{(3)} \rightarrow T_{(0)}$, $T_{(2)} \rightarrow T_{(0)}$, $T_{(1)} \rightarrow T_{(0)}$. The task $T_{(1)}$, $T_{(2)}$ and $T_{(3)}$ will become leaf nodes separately in a new preemption graph. Thus PTS increases exclusivity of the task’s execution from the help of ILP based allocation using faster memory.

### 4.2.2 Memory Access Time and Data Allocation

Allocation of data objects into heterogeneous memories using SPM is modeled by the 0-1 Knapsack Problem solved by ILP, where the objective function is the memory access time or execution cost with candidate data objects allocated into each memory bank. The required memory space and other system parameters constrain possible solutions. The ILP solver finds values of binary variables which determine a particular variable’s allocation under the constraints.

Memory subsystems may have multiple heterogeneous units such as SPM, external SRAM and external SDRAM. Each memory unit has multiple memory banks inside. Here we use the term memory bank rather than memory unit to imply a different memory chunk available for allocation. However, the memory bank can be interpreted as the memory unit.

In order to present different level of complexity on applying the scheme, the prior DARTS works [23, 25] considered three different levels of data variable granularity: task, procedure and variable level. Figure 4.3 illustrates an example memory footprint. There are multiple tasks ($T_{(i)}$ and so on). Within a task ($T_{(i)}$), there are multiple procedures ($P_{(i,j)}$ and so on).
Figure 4.3: Memory footprint of data objects at task, procedure and variable level

Within one procedure \( P_{(i,j)} \), again, there are multiple automatic variables \( V_{(i,j,k)} \) and so on. Static and global variables occupy other regions separate from the stack in memory. The task stacks including procedure stack frames are typically allocated by the operating system and are separate from each other. The actual sequence of stack frames depends on the procedure call sequence within the task.

The size and access information of data objects are determined by simple arithmetic based on the granularity level of interest. Data variables \( V_{(i,j,k)} \) whose first subindex starts with \( i \) are grouped into one conceptual data object \( x^T_{(i)} \) for working with the task level granularity for \( T_{(i)} \). Similarly, data objects at procedure level granularity is also obtained from the variable level. For an example, in Figure 4.2, to obtain a task level data object for \( T_{(1)} \) (a hexagon), all data variables inside of the task is gathered into a single a data object, which includes the 5 ellipses.

The following terms are defined to describe the problem model over three different granularity levels of data objects.

Memory related:
- \( N_U \) = total number of memory banks
- \( U_n = n^{th} \) memory bank such that \( n \in [1, N_U] \)
- \( r_n^{r/w} \) = read/write access latency of memory bank \( U_n \)

Data object granularity related:
\( N_T = \text{total number of tasks} \)

\( T(i) = i^{th} \text{ task such that } i \in [1, N_T] \)

\( N_P(T(i)) = \text{total number of procedures of task } T(i) \)

\( P(i,j) = j^{th} \text{ procedure of task } T(i) \text{ such that } j \in [1, N_P(T(i))] \)

\( N_V(P(i,j)) = \text{total number of automatic variables of procedure } P(i,j) \)

\( V(i,j,k) = k^{th} \text{ data variable of procedure } P(i,j) \text{ in task } T(i) \text{ such that } k \in [1, N_V(P(i,j))] \)

\( N_G = \text{total number of static variables including globals} \)

Data objects related:

\( x^G_m = m^{th} \text{ static data variable such that } m \in [1, N_G] \)

\( x^T(i) = \text{all automatic data variables of task } T(i) \)

\( x^T(i,j) = \text{all automatic data variables of procedure } P(i,j) \text{ in task } T(i) \)

\( x^T(i,j,k) = k^{th} \text{ automatic data variable of procedure } P(i,j) \text{ in task } T(i) \text{ (equivalent to } V(i,j,k)) \)

\( Z(x) = \text{size of data object } x \)

\( \mu^{r/w}(x) = \text{number of read/write of data object } x \)

Notice that at variable level \( V(i,j,k) \) are identical to \( x^T(i,j,k) \) since that is the finest granularity handled. In addition to the terms, let a binary variable \( I_n(x) \) indicate the location of a data object \( x \) as defined below.

\[
I_n(x) = \begin{cases} 
1 & \text{if } x \text{ is allocated into } U_n \\ 
0 & \text{otherwise} 
\end{cases} \quad (4.1)
\]

**Task Level**

By definition, \( \mu^r_i(x^G_m) \) and \( \mu^w_i(x^G_m) \) mean the number of reads and writes for static data object \( x^G_m \) by task \( T_i \) respectively. The total memory access cycles of the system at a task level granularity \( (MAC_T) \) becomes

\[
MAC_T = \sum_{n=1}^{N_U} \sum_{i=1}^{N_T} \sum_{m=1}^{N_G} I_n(x^G_m) [\tau^r_n \mu^r_i(x^G_m) + \tau^w_n \mu^w_i(x^G_m)] Z(x^G_m) \\
+ \sum_{n=1}^{N_U} \sum_{i=1}^{N_T} I_n(x^T(i)) [\tau^r_n \mu^r_i(x^T(i)) + \tau^w_n \mu^w_i(x^T(i))] Z(x^T(i)) \quad (4.2)
\]

Two constraints come from the rule that each data variable must be allocated in exactly one memory bank.
\[
\sum_{n=1}^{U_N} I_n(x_G^m) = 1 \quad \text{for } \forall m \in [1, N_G] \quad (4.3)
\]
\[
\sum_{n=1}^{U_N} I_n(x_T^{(i)}) = 1 \quad \text{for } \forall i \in [1, N_T] \quad (4.4)
\]

The limited capacity of each memory bank introduces the last constraint (4.5). \( \Omega \) is a set of all paths to reach leaf nodes in the preemption graph, and \( \omega \) is a path belonging to \( \Omega \). For the example in Figure 4.2, \( \Omega = \{T(3) \rightarrow T(2) \rightarrow T(1), T(3) \rightarrow T(2), T(3) \rightarrow T(1), T(3)\} \), and \( \omega \) is one of these paths. Therefore the inequality below states that total memory usage of a particular memory unit must be less than its capacity in any case of preemptions among tasks. By minimizing equation (4.2) under the constraints (4.3), (4.4) and (4.5), a new allocation is obtained.

\[
\sum_{m=1}^{N_G} I_n(x_G^m)Z(x_G^m) + \sum_{T(i)\in\omega} I_n(x_T^{(i)})Z(x_T^{(i)}) \leq Z(U_n) \quad \text{for } \forall \omega \in \Omega \quad (4.5)
\]

**Virtuous Cycle of DARTS:** The virtuous cycle of DARTS can be explained with these formulas and an example preemption graph in Figure 4.2. Let’s assume that \( T(3) \)’s execution time can be reduced by allocating some part of its stack space into the fastest memory. At this phase, we have \( \Omega = \{T(3) \rightarrow T(2) \rightarrow T(1), T(3) \rightarrow T(2), T(3) \rightarrow T(1), T(3)\} \). After the allocation by the ILP solving, \( T(3) \) may not need to preempt \( T(2) \)’s execution any more because \( T(3) \) can meet its deadline thanks to its faster execution time without preempting \( T(2) \). Then \( T(2) \) also avoids preemption by \( T(3) \) by PTS which sets \( T(2) \)’s preemption threshold to the preempting priority of \( T(3) \). This updates the preemption graph so that \( \Omega = \{T(3) \rightarrow T(1), T(2) \rightarrow T(1), T(3), T(2)\} \).

In the next iteration cycle of DARTS, stack space of \( T(2) \) is allocated to share the same memory space of \( T(3) \) in the fastest memory by the ILP formula. Consequently \( T(2) \)’s execution time also can be reduced by the new allocation. By the same manner, in turn, \( T(1) \) also may avoid preemption by \( T(2) \) and \( T(3) \) if possible. Finally this cycle reduces the number of preemptions among the tasks, as well as execution time. The bottom line of this idea is that the same memory space in faster memory banks can be shared by different data variables from different tasks thanks to the exclusivity among the tasks’ execution.

**Procedure Level**

At procedure granularity level, all data variables in a procedure are packed together and regarded as a single variable. Let \( x_T^{(i,j)} \) represent a single data object of the procedure \( P_{(i,j)} \). Then the objective function \( MAC_P \) becomes:
\[ MAC_P = \sum_{n=1}^{N_U} \sum_{i=1}^{N_T} \sum_{m=1}^{N_G} I_n(x^G_m)[\tau_n^r \mu^r_i(x^G_m) + \tau_n^w \mu^w_j(x^G_m)]Z(x^G_m) \]
\[ + \sum_{n=1}^{N_U} \sum_{i=1}^{N_T} \sum_{j=1}^{N_P(T_{(i)})} I_n(x^T_{(i,j)})[\tau_n^r \mu^r_i(x^T_{(i,j)}) + \tau_n^w \mu^w_j(x^T_{(i,j)})]Z(x^T_{(i,j)}) \] (4.6)

The constraints from (4.3) to (4.5) are also reformulated as (4.7) to (4.9) respectively.

\[ \sum_{n=1}^{U_n} I_n(x^G_m) = 1 \quad \text{for } \forall m \in [1, N_G] \] (4.7)
\[ \sum_{n=1}^{U_n} I_n(x^T_{(i,j)}) = 1 \quad \text{for } \forall j \in [1, N_P(T_{(i)})], \forall i \in [1, N_T] \] (4.8)

\[ \sum_{m=1}^{N_G} I_n(x^G_m)Z(x^G_m) + \sum_{P_{(i,j)} \in \omega'} I_n(x^T_{(i,j)})Z(x^T_{(i,j)}) \leq Z(U_n) \quad \text{for } \forall \omega' \in \omega, \forall \omega \in \Omega \] (4.9)

Here \( \omega \) and \( \Omega \) are same as those of the task level granularity. \( \omega' \) is a possible procedure execution sequence, which is derived from call graphs of the tasks on the path \( \omega \). in Figure 4.2, the task \( T_{(1)} \) has 3 procedures and its call graph says that it has only one sequence of procedures possibly executed, which is \( P_{(1,1)} \rightarrow P_{(1,2)} \rightarrow P_{(1,3)} \). The task \( T_{(2)} \) has two sequences \( (P_{(2,1)} \rightarrow P_{(2,2)} \rightarrow P_{(2,4)}, P_{(2,1)} \rightarrow P_{(2,3)} \rightarrow P_{(2,4)}) \), and the task \( T_{(3)} \) has also two \( (P_{(3,1)} \rightarrow P_{(3,2)} \rightarrow P_{(3,4)}, P_{(3,1)} \rightarrow P_{(3,3)} \rightarrow P_{(3,4)}) \). \( \omega' \) is a combination of them, when a task’s sequence is selected by \( \omega \).

**Variable Level**

At variable level granularity level, all data variables remain as single data objects so that the objective function \( MAC_V \) is shown below.

\[ MAC_V = \sum_{n=1}^{N_U} \sum_{i=1}^{N_T} \sum_{m=1}^{N_G} I_n(x^G_m)[\tau_n^r \mu^r_i(x^G_m) + \tau_n^w \mu^w_j(x^G_m)]Z(x^G_m) \]
\[ + \sum_{n=1}^{N_U} \sum_{i=1}^{N_T} N_P(T_{(i)}) \sum_{j=1}^{N_V(P_{(i,j)})} I_n(x^T_{(i,j,k)})[\tau_n^r \mu^r_i(x^T_{(i,j,k)}) + \tau_n^w \mu^w_j(x^T_{(i,j,k)})]Z(x^T_{(i,j,k)}) \] (4.10)
The constraints are reformulated as presented below.

\[
\sum_{n=1}^{U_N} I_n(x^G_m) = 1 \quad \text{for } \forall m \in [1, N_G] \tag{4.11}
\]

\[
\sum_{n=1}^{U_N} I_n(x^T_{i,j,k}) = 1 \quad \text{for } \forall k \in [1, N_V(P_{i,j})], \forall j \in [1, N_P(T_i)], \forall i \in [1, N_T] \tag{4.12}
\]

\[
\sum_{m=1}^{N_G} I_n(x^G_m)Z(x^G_m) + \sum_{V_{i,j,k} \in \omega'} I_n(x^T_{i,j,k})Z(x^T_{i,j,k}) \leq Z(U_n) \quad \text{for } \forall \omega' \in \omega, \forall \omega \in \Omega \tag{4.13}
\]

Here \( \Omega \) and \( \omega \) are the same as those in the task level and the procedure level. It is different from the procedure level formula above in that each variable is handled as an individual data object. The control flow graph of a procedure does not affect the path enumeration of inequality (4.13) due to automatic variables having the same lifespan (the scope of the procedure).

### 4.3 Experiments

Various experiments are carried out to evaluate the characteristics and the virtues of DARTS with a real target system. This section introduces the experimental setup as well as the system examined.

#### 4.3.1 Taget System

The target hardware system has a LPC2888 MCU whose processor core is ARM7TDMI-S running at 36MHz, and various memories; 64KB internal SRAM (SPM), 2MB external SRAM, 2MB internal flash ROM, and 8KB unified cache. The flash ROM is used to store the code region. The unified data cache is set to work only as data cache in the experiments. The other features used in the experiments are listed in Table 4.1.

The SPM has faster read and write latency than the external SRAM does, and the data cache shows better performance than the SPM. However, many other SPMs show the same access latency as the data cache; 1 clock cycle for read and write. This prevents fair comparison of performance by the SPM to that by the data cache. This 1 clock cycle penalty to SPM is compensated by counting memory accesses appearing on the external memory bus of the target. When a data variable is placed in SPM, accesses to the data variable will not be captured on the external memory bus. So the penalty can be compensated as long as we know the original
Table 4.1: Target hardware system features and toolchain

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td>NXP LPC2888 (ARM7TDMI-S, 36MHz)</td>
</tr>
<tr>
<td>Int. SRAM (SPM)</td>
<td>64KB, 32-bit width, 2/2 clock cycles for read/write of 32 bits</td>
</tr>
<tr>
<td>Ext. SRAM</td>
<td>2MB, 16-bit width, 17/14 clock cycles for read/write of 32 bits</td>
</tr>
<tr>
<td>Flash ROM</td>
<td>2MB, 128-bit width</td>
</tr>
<tr>
<td>Unified Cache</td>
<td>8KB, 2-way set associative, only data cache is enabled, 1/1 clock cycle for read/write of 32 bits</td>
</tr>
<tr>
<td>RTOS</td>
<td>FreeRTOS, 1 ms scheduling period</td>
</tr>
<tr>
<td>Toolchain</td>
<td>GNU arm-elf-gcc 4.1.1 and arm-elf-as 2.17 with newlib 1.14.0</td>
</tr>
</tbody>
</table>

Table 4.2: Properties of tasks used for experiments

<table>
<thead>
<tr>
<th>Task</th>
<th>Scheduling Priority</th>
<th>Period (ms)</th>
<th>Memory Usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Statics</td>
</tr>
<tr>
<td>ADPCM</td>
<td>1</td>
<td>27,000</td>
<td>16,844</td>
</tr>
<tr>
<td>Huff</td>
<td>2</td>
<td>18,000</td>
<td>8,216</td>
</tr>
<tr>
<td>MgSort</td>
<td>3</td>
<td>12,000</td>
<td>8,216</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>4</td>
<td>9,000</td>
<td>25,144</td>
</tr>
<tr>
<td>SHA</td>
<td>5</td>
<td>3,000</td>
<td>112</td>
</tr>
</tbody>
</table>

The number of memory accesses appearing on the external memory bus and the improved number of memory accesses by allocating data to the SPM. The difference between the memory access counts accounts for the counts toward SPM. So if we assume 1 clock latency SPM, we can easily compensate the difference of the memory access counts from the original execution time. However, compensation for only the shortest task SHA will be discussed because of the limited memory capacity of our logic analyzer.

To implement a multi-threaded system, a preemptive real-time operating system (FreeRTOS [99]) schedules 5 tasks; ADPCM, Huff, MgSort, Dijkstra, and SHA. They are collected from open source domain and MiBench [30]. FreeRTOS is modified to support PTS and some measurements for the experiments. Table 4.2 presents real-time properties and memory usage of the tasks. The initial priority of each task is assigned rate-monotonically. Higher priority is
represented by a higher number. The system is designed so that each task loads the processor with about 10% utilization, so the overall target for processor utilization becomes about 50%. Overhead due to the scheduler and some extra workload for experiments raise actual utilization up to 58.9% (see Figure 4.9a).

As sample workloads for the tasks, five different 8KB samples are selected from five classic texts: The Adventures of Huckleberry Finn, The Catcher in the Rye, The Lord of the Rings, The Odyssey, and The Three Musketeers. Their character code values are fed into the tasks as input workloads. For example, in case of Dijkstra, the character code values are assigned as weight of edges on a network graph. These five inputs induce timing variability in execution time of the tasks at workload level. To make the hyperperiod of the system more feasible for experiments, ADPCM uses only 1KB samples, MgSort uses 4KB samples and the others use 8KB samples. This helps adjust the execution time of the tasks so that the hyperperiod becomes feasible. Here it is 108 seconds.

Among the test programs, Huff and Dijkstra use dynamically allocated memory which is not covered by the current version of the toolchain. Moreover, some procedures of Huff, Dijkstra, and MgSort are recursive. The recursion makes budgeting of SPM difficult because static bounding of the recursion depth is not simple. Dynamically allocated data objects and automatic variables of recursive procedures are also excluded from the experiments. This degrades the benefit of DARTS somewhat because those data variables which are not handled by our toolchain may lose the chance to be allocated into the faster memory (SPM) although they are accessed frequently. However, despite of this limit of the current toolchain, the experimental results show that DARTS still delivers significant benefits in many aspects of the system such as task’s execution times, system utilization and the number of preemptions.

Correct computing results must be guaranteed after implementing allocation. This concern is relieved by the compiler and the linker in case of static variables. However, the automatic variables allocated into stack frames are rearranged by the proposed stack split scheme. Hence some code to verify each task’s result is inserted. After executing a task’s job, the computed result of each task is compared to the known-good result.

4.3.2 Experiment Setup

Three design space dimensions are explored in the experiments: available SPM budget, granularity level of applying DARTS and availability of PTS. As mentioned in the previous sections, task, procedure and variable level granularity are examined along with 6 SPM budgets and the PTS option.

The SPM is budgeted to 256B, 512B, 1KB, 2KB, 4KB and 8KB. This considers the dimensions of the cache memory of the hardware system. 256 bytes is the same size as one cache line.
of the cache memory, and 8KB is the same capacity as the cache memory.

The reason why the different granularity level of data variables are evaluated is scalability problem with ILP solving. Our allocation problem solved by ILP is a NP-complete problem. Hence as the number of data objects increases, the time required for obtaining the optimized allocation is also increased exponentially. Although variable level granularity shows the best performance, under a situation where practical applicability of DARTS at the level is not good, approaching at other granularity level would be alternative solution.

The option PTS means PTS is enabled so that the allocation and rescheduling follow the full DARTS cycle explained above. The option noPTS skips the step to reschedule the tasks using PTS, and PTS is disabled so that allocation of data variables is not as aggressive. Thus, noPTS case represents other prior schemes suggested, where data allocation by solving ILP formula is conducted only once, not iteratively as DARTS does.

In addition to 6 SPM budgets, one baseline memory configuration called SRAM case is examined, where all data objects are allocated into the external SRAM. These data objects include activation records of procedures and other data variables no matter whether they are identified by the static timing analysis tool or not. In this case, the standard memory allocation from the GNU toolchain [28] is not modified, and the system is assumed to use only the external SRAM as data memory. The SRAM case shows the worst performance in many aspects.

Another memory configuration SRAM+D$ case is also examined. SRAM+D$ is the same cases as the SRAM except only the data cache is enabled. Thanks to the data cache, this case shows the best performance. But the data cache does not guarantee predictability of task’s timing at memory access level.

During an experiment for one point of the design space, four metrics are measured; each task’s worst case execution time and worst case response time, total processor utilization and number of preemptions among the tasks. The worst case execution times and the worst case response time are normalized to those from the default configuration SRAM case. The maximum SPM usage by each allocation is also measured by a water-mark method. This purposes to check the allocation by DARTS satisfies the memory capacity constraint. The system is examined for one hyperperiod of the tasks.

4.4 Results

The benefit by DARTS is multi-faceted and depends on various factors. Hence the result is discussed with two different views of the experiments. In the discussion from the single-task perspective, behaviors of DARTS with respect to one task are discussed, relying primarily on the results using noPTS option, which are less related to multi task systems. This will help us figure out benefit obtaining from data allocation into SPM. In the discussion from the multi-
task perspective, the discussion mainly uses the results with the PTS option. This discussion help us figure out the more accumulated benefit by DARTS.

The task’s execution times and response times are presented in Figure 4.4 through Figure 4.8 in order of the task’s scheduling priority. The processor utilization is presented in Figure 4.9a, and the number of preemptions among the tasks is plotted in Figure 4.9b. As mentioned above, the execution times and the response times of the tasks are normalized to those of the baseline memory configuration SRAM.

### 4.4.1 Single-Task Perspective Discussion

The task’s execution time is reduced as the allocatable SPM budget increases. As more SPM space is available, more data variables are allocated into SPM and the execution time of the associated tasks is improved. From Figure 4.4 to Figure 4.8, the task’s worst case execution time decreases as more SPM budget is given, although the benefit depends on several other factors such as the PTS option, the granularity level, and so on.

Only 256 bytes of SPM are needed to reap most of the performance improvement over the baseline configuration (SRAM). At variable level granularity using noPTS option, only 256 bytes SPM reduces the execution times to 66.0% for ADPCM, 73.3% for Huff, 65.4% for MgSort, 73.4% for Dijkstra, and 53.3% for SHA respectively of the execution times by the baseline SRAM configuration. This means even a small size of SPM can improve the system performance dramatically, because a small portion of data variables account for a large portion.
of the total memory accesses. Our static timing analysis also explains this. In the analysis only 25 data variables among all 369 data variables cause 90.3% of total memory accesses. Both SPM and data cache rely on this to improve system performance at the memory access level while decreasing overhead.

At the maximum SPM budget (8KB) and the variable granularity level with noPTS option, the task’s execution times are reduced to 62.8%, 71.1%, 57.2%, 68.0%, and 53.0% against the execution time of SRAM configuration respectively for ADPCM, Huff, MgSort, Dijkstra, and SHA. On the other side, SRAM+D$ shows 41.9%, 39.9%, 37.1%, 38.9%, and 41.5% of the baseline execution time respectively for ADPCM, Huff, MgSort, Dijkstra, and SHA. Huff and Dijkstra access dynamically allocated data variables which are not covered by the current toolchain. This appears as a somewhat smaller reduction (improvement) of execution times compared with other tasks.

On average our scheme delivers 64.4% of the speedups attained by a data cache. Our experiments are in fact biased toward the data cache due to hardware characteristics of our system. The SPM used in these experiments takes 2 clock cycles for a read or write of 32-bit data, while the data cache takes 1 clock cycle for the same operation. Table 4.3 shows speedups by data cache and DARTS at variable level granularity calculated over the baseline configuration (SRAM) for particular SPM budgets. The maximum speedups with DARTS using SPM compared with those from the data cache are up to 66.7% for ADPCM, 56.1% for Huff, 64.7% for MgSort, 61.0% for Dijkstra, and 73.5% for SHA. Consider the 8KB SPM budget with noPTS option in Table 4.3.
Most SPMs have the same latency as the data cache, as pointed out in the previous section. To predict the performance of DARTS on such a system, we eliminate the 1 clock cycle penalty by counting the number of memory accesses with a logic analyzer as explained above. But only SHA is examined due to the limited memory capacity of our logic analyzer. The execution time of SHA at variable level and 8KB SPM using noPTS reaches 49.0% of the baseline execution time, and corresponding speedup approaches to $2.04 \times$. This is closer to the data cache’s speedup ($SRAM+D$), which is $2.57 \times$ as shown in Table 4.3. The original speedup before the compensation is $1.89 \times$. Hence we expect that an SPM will deliver somewhat better performance than shown in Table 4.3 in a system having SPM with 1 clock cycle access latency.

When we allocate the task’s whole stack frame including activation records and all static data variables into SPM as well as the data variables handled with our toolchain, SPM shows better performance than data cache. This experiment is performed in a single-threaded system since the SPM is not enough large to contain the whole multi-threaded system. The SHA-only single-threaded system delivers $2.486 \times$ speedup, while data cache returns $2.391 \times$. The system using SPM shows better performance than the system using the data cache but does not suffer from cache misses. The speedup from the data cache in this single threaded system ($2.391 \times$) differs from that in the multi-threaded system ($2.57 \times$) because of timing noise due to the scheduler in the multi-threaded system and changes in data variable locations. Notice that different data memory layout (different locations of a data variable) lead to different patterns of data cache interference.

The next observation is that finer granularity yields better performance. Variable-level
granularity provides the best overall benefits of all granularity levels since smaller granularity leads to less fragmentation and better utilization of the available space in SPM. For larger data objects, there is in effect less allocatable faster memory space available. With finer granularity the limit from the size of data object is mitigated, producing a better allocation. For example, at 256 bytes SPM budget with noPTS option, ADPCM’s SPM usage varies largely along with the granularity levels; none at task level, 20 bytes at procedure level, and 59 bytes at variable level. Correspondingly, the execution times are shortened to 98.0%, 72.8%, and 66.0% respectively of each SRAM case.

This in turn makes the finer granularity level improve performance even faster. At the variable level, the task’s execution time is already saturated only with 256 bytes SPM. On the contrary, at the task level 256 bytes and 512 bytes SPM space are not fully utilized since ADPCM and Huff need much larger space than those SPM capacity. Although the two task’s maximum SPM usage is not too large (125 bytes and 133 bytes respectively), they cannot be allocated into SPM. This is because all variables of a task are grouped into one conceptual data object as stated in section 4.2 without considering call graphs of procedures inside of a task. That is, at task level actual SPM usage is smaller than what the allocation assumes and calculates. This assumption wastes SPM space, and consequently, diminishes the benefit at the task level comparing to other level of granularity.

At certain SPM budgets, the SPM usage of higher priority tasks increases slowly than for other tasks. This is because the data accesses by other tasks are more frequent and intensive than those by the higher priority tasks, so that the tasks lose the opportunity to utilize SPM.
For example, from 256 bytes to 1024 bytes SPM budget at variable level, SHA’s SPM usage increases slowly while ADPCM and Huff do relatively faster, since allocating SPM space to those two tasks are more beneficial at the overall system perspective. The toolchain aims for improvement of the whole system’s performance. Even when data variables of a particular task are not allocated into SPM, data variables of other tasks still can be allocated into SPM, if it can help improve the system’s overall performance. Beyond 1K or 2K, all automatic variables are allocated into SPM at all granularity levels. Only small numbers of static variables remain (large buffer arrays), diminishing the benefit from rising SPM budgets.

### 4.4.2 Multi-Task Perspective Discussion

Reducing the execution time of the tasks improves the overall system’s processor utilization. See Figure 4.9a. The utilization of the baseline memory configuration SRAM is 58.9%, and SRAM+D$ is 30.0%. The highest utilization is 52.1% by using 256 bytes SPM at the task level granularity with noPTS option, and the smallest is 40.7% by using 8KB at the variable level granularity. Processor utilization shows similar behavior with the execution time discussed in the single-task view discussion so the larger SPM budget and the finer granularity reduce utilization. Using PTS further cuts utilization in some cases.

In the case of noPTS, the maximum SPM usage is calculated by summing up the SPM usages by all the tasks and the static (global) variables. At the worst case, all the tasks are preempted by their higher tasks in series from ADPCM to SHA. This case results in the maximum SPM
Table 4.3: Speedup comparison between DARTS (using 2-cycle SPM) and 1-cycle data cache

<table>
<thead>
<tr>
<th>PTS option</th>
<th>Task</th>
<th>SRAM+D$</th>
<th>SPM-256 (Vars)</th>
<th>SPM-8192 (Vars)</th>
</tr>
</thead>
<tbody>
<tr>
<td>noPTS</td>
<td>ADPCM</td>
<td>2.389</td>
<td>1.515</td>
<td>1.593</td>
</tr>
<tr>
<td></td>
<td>Huff</td>
<td>2.507</td>
<td>1.363</td>
<td>1.406</td>
</tr>
<tr>
<td></td>
<td>MgSort</td>
<td>2.699</td>
<td>1.529</td>
<td>1.747</td>
</tr>
<tr>
<td></td>
<td>Dijkstra</td>
<td>2.412</td>
<td>1.363</td>
<td>1.470</td>
</tr>
<tr>
<td></td>
<td>SHA</td>
<td>2.570</td>
<td>1.878</td>
<td>1.888</td>
</tr>
<tr>
<td>PTS</td>
<td>ADPCM</td>
<td>2.390</td>
<td>1.600</td>
<td>1.593</td>
</tr>
<tr>
<td></td>
<td>Huff</td>
<td>2.506</td>
<td>1.402</td>
<td>1.405</td>
</tr>
<tr>
<td></td>
<td>MgSort</td>
<td>2.700</td>
<td>1.644</td>
<td>1.746</td>
</tr>
<tr>
<td></td>
<td>Dijkstra</td>
<td>2.413</td>
<td>1.347</td>
<td>1.470</td>
</tr>
<tr>
<td></td>
<td>SHA</td>
<td>2.570</td>
<td>1.887</td>
<td>1.887</td>
</tr>
</tbody>
</table>

usage. In the case of PTS, the maximum SPM usage is the sum of static variable’s usage and the maximum SPM usage among the 5 tasks.

Compared to noPTS cases, PTS cases show shorter execution time, better processor utilization, and better SPM utilization both at same SPM budget and at same granularity. The PTS case makes more SPM space available to the tasks. For example, at the procedure level, SPM usage of ADPCM increases from 20 bytes to 87 bytes. Dijkstra increases from 20 bytes to 44 bytes, MgSort does from 10 bytes to 40 bytes. SHA also uses more SPM space from 20 bytes to 44 bytes. This also repeats at the variable level granularity, although the detailed SPM usage information is not presented here due to space limit.

In case of PTS, no task can preempt any other task at all because the preemption threshold values are the highest priority for all tasks. This means the preemption threshold value of all the tasks are set to the highest priority. The preemption threshold value is assigned by our PTS module in the DARTS cycle as we explained, and the DARTS cycle iterates only twice in our this experiment. All these situation comes from that tasks are sufficiently exclusive. We use real hardware and real test programs but not a real a application system. Hence DARTS cycle iteration does not repeat intensively. But in Ghattas et al.’s work in [23, 25], the more accumulated improvement of the system performance is simulated. The better performance with PTS i.e., full DARTS cycle is also observed even though PTS does not seem to bring significant extra benefit relatively comparing to the benefit from noPTS.

The increased exclusivity of the tasks are also observed in Figure 4.9b. The plot shows reduced number of preemptions among the tasks. Because of their long execution time and lower priority, ADPCM and SHA are preempted seven and six times respectively at the baseline.
memory configurations (SRAM) within the hyperperiod. Using data cache removes most of the preemption, so that task ADPCM is preempted once at SRAM+D$ case. The smallest number of preemptions is two when using 8KB SPM at the variable granularity level. Using PTS removes all preemptions. Figure 4.9b shows the number of preemption only when PTS is not used. The figure implies that DARTS not only reduces execution time of the tasks but also enhances the system’s real-time performance.

Reduced execution time, increased SPM utilization, and reduced number of preemptions ultimately result in reduced task response times. From Figure 4.4b to Figure 4.8b present the response time of the tasks. The response time of the tasks are reduced 60.9% for ADPCM, 60.7% for Huff, 60.8% for MgSort, 64.5% for Dijkstra, and 53.0% for SHA with noPTS option comparing to the response time when using their baseline configuration SRAM. This improvement is achieved only by using the same size of SPM as the data cache. Using the data cache shows reduction of the response time down to 38.7% for ADPCM, 36.8% for Huff, 38.9% for MgSort, 40.8% for Dijkstra, and 38.9% for SHA.

Using PTS option shows similar results for 4 tasks except SHA. The response time when using PTS option shows the shorter response time under the same design space so that 58.8% for ADPCM, 64.5% for Huff, 60.7% for MgSort, and 60.8% for Dijkstra. In case of SHA, the response is increased up to 629.6%. This is because of the preemption-limiting characteristic of PTS. Since the PTS iterations in DARTS result in preemption threshold values that do not allow preemptions, the highest priority task SHA cannot preempt other 4 tasks any more. It must wait for other task’s job is finished rather than preempt it. PTS reduces response time
of the lower priority task at the cost of increased response time of the higher priority task. In 
this experiment, $SHA$ pays the cost. However, recall that PTS ensures that all tasks, including 
$SHA$, will always meet their deadlines.

4.5 Summary

System performance can be enhanced by using faster memory more effectively. In this work 
we present new methods which allocate data objects more aggressively by considering the 
total memory access cost from multi-threaded tasks and leveraging SPM to provide timing 
predictability at memory access level. The suggested methods are implemented in a realistic 
and feasible toolchain, and performance is evaluated for real code.

The DARTS approach improves system performance significantly, even with a small SPM 
size. The size of data objects affects the allocation, so that supporting finer granularity brings 
better performance. Although our experiments show somewhat less performance enhancement 
than a traditional data cache memory system, the worst-case performance is easily calculated, 
making real-time analysis and design accurate and tractable. Furthermore when the access 
latency for the SPM and data cache used in the experiments is equalized, the performance gap 
narrows further.

DARTS improves the real-time performance in the experiments. It reduces the response time 
of lower priority tasks at the cost the highest priority task’s increased response time (though 
still meeting all deadlines), so that overall schedulability of a system is improved. The number 
of task preemptions is also reduced, reducing context-switching overhead.

There are various areas for future work. In this study, static variables are not decomposed 
into finer granularity data objects since this will result in a large overhead during execution 
time for loading data and the risk of implicit memory references. There may be more aggressive 
methods which exploit cache memory and SPM simultaneously. These remain as future works.
Chapter 5

Extracting Synergy between Scratchpad Memory and Data Cache

5.1 Introduction

This chapter proposes data allocation methods that actively exploit both fast memory subsystems together for improving the data cache performance and ultimately the system performance with little timing variability. To achieve these goals, a new perspective to understand data cache misses is proposed; data cache misses of a system are classified in two types – intrinsic misses and interference misses. The data cache misses of a preemptive real-time system are analyzed with this perspective, and particular data are selectively allocated into SPM. Allocating data into SPM this way reduces data cache misses from a task’s own use of its data as well as interference cache misses from other tasks since the selected data are managed by SPM, and only residual data still remain cacheable so that they can take advantage of increasing effective capacity of the data cache. Overall data cache performance is improved since inter-data cache pollution is alleviated. This interactive benefit is recognized as performance synergy between SPM and data cache.

The suggested approach is evaluated with a real hardware system which runs several tasks on a preemptive real-time operating system. During the real-time scheduling of the tasks, context switching occurs and disrupts data cache states. This is because the hardware resources of the data cache are limited and hence shared simultaneously by multiple threads of the real-time tasks. Even though a data cache may be large enough to contain all the data required for a task, context switching itself can pollute data cache lines unnecessarily, which depend upon on the memory layout of the other task’s cacheable data. This study also addresses this inter-task cache pollution problem by applying the proposed synergetic SPM allocation methods.

This work builds heuristic methods based on the new cache miss analysis approach, which
actually select a combination of task data to improve data cache performance. The heuristics are examined at coarse granularity level (task-level) of data and at fine granularity level. These experiments show that many data cache misses come from inter-task or inter-data cache pollution, and they can be reduced effectively by using SPM. One supplementary experiment is also performed, which shows that the data cache is sensitive not only to its dynamic contents but also to data memory layout changes. Our schemes guide us quite well to leverage data cache and SPM synergetically overcoming the sensitivity.

In spite of the interesting results mentioned above, this chapter does not demonstrate directly that timing predictability of the data cache is improved thanks to the proposed methods since the worst case timing analysis is not performed. However it is qualitatively anticipated that reduction of cache contention by employing SPM can contribute to improving the worst case performance and the worst case timing analysis by lowering timing jitters caused from data cache access.

The remainder of this chapter is organized as follows. Section 5.2 discusses the motivating case study on the sensitivity of the data cache to input workloads, data memory layout changes and the cache’s dynamic contents. Section 5.3 explains the new concept of data cache miss decomposition proposed by this work and presents heuristics to allocate task data into SPM. Section 5.4 describes the experiments setup, followed by the results in Section 5.5. Section 5.6 concludes this chapter by summarizing findings and proposing direction for further research.

5.2 Motivating Example

Data cache behavior is not easy to estimate due to its dynamicity. Many factors determine data cache behavior during execution of a program. We begin by performing a case study to examine the impact of three kinds of variation on the number of data cache misses of the tasks. This experiment helps explain the difficulties of static cache analysis, providing motivation for our work in subsequent sections. We find that sometimes the performance of a data cache is surprisingly sensitive to small changes in data memory layout and its dynamic contents. The three experimental factors are: input workload to test programs, memory layout of data, and data cache contents before starting a new job of a task.

The hardware and the test programs used in this experiment are presented in Table 5.1 and Table 5.2, respectively. (For more details, see Section 5.4.1.) The test programs are fed with five different input workloads. Different input workloads steer the program control flow in different ways, which changes the data cache states, and, consequently, the data cache performance also varies.

Each program runs two jobs back-to-back for one identical input workload. Before the first job, the data cache is flushed thoroughly to remove jitters (cold cache case). To emulate the
best case between the two neighboring jobs, the warm cache case does not flush the cache before starting the second job, which means the data cache is not polluted by other programs. The worst case is cold cache case, where the cache is flushed out thoroughly just before performing any job.

To alter data memory layouts, three memory layout variants are implemented: Reversed Statics, LRLA Dynamics, and 1.5 Cache Line Offset Stack. The control case is named Default. For the Reversed Statics configuration, the order of definition or declaration of static variables gets reversed within each source code file. This lets the linker assign different memory address to the static data variables. The configuration LRLA Dynamics adopts a last-returned-last-allocated policy to allocate heap memory chunks to dynamic type data, where the most recent freed memory block is not allocated until previously freed blocks have been allocated. Default configuration uses LRFA policy meaning last-returned-first-allocated, where the most recently freed memory block is allocated first for the next allocation. These different management policy for heap memory blocks also allocate different memory addresses to the dynamic data variables. For the 1.5 Cache Line Offset Stack configuration, a stack pointer at entry points of the programs is adjusted intentionally by one and half cache lines (48 bytes). Since task stacks grow up and down from the initial address the stack pointer stores, this also changes addresses of the automatic data variables. All other factors which can affect cache misses are controlled as described in Section 5.4.1.

Figure 5.1 shows the sensitivity of data cache performance in a single-threaded system. Each number of cache misses is normalized to the average number of cache misses of the first job.
for the five inputs running with Default configuration (the first column of each task in Figure 5.1a). Each column represents the average number of cache misses for the five inputs with an blue error bar showing the worst and the best number of cache misses for one data memory layout.

The input workload variation affects the cache miss rate for some programs (Dijkstra and TinyJPEG). In the Default configuration, Dijkstra varies 11.6%, and TinyJPEG varies 0.2%. Other benchmarks show a constant or almost constant number of cache misses, so different inputs do not generate different control flows. This behavior depends on the program’s logic.

Memory layout changes bring an unpredictable impact. In the cold cache case, Dijkstra generates 77.8%, 105.2%, and 231.8% of cache misses of its Default configuration on average for Reversed Statics, LRLA Dynamics, and 1.5 Cache Line Offset Stack respectively. Other benchmarks show from 99.1% to 102.4% of cache misses of their Default cases on average for the various memory layouts.

In case of the warm cache, data cache behavior changes more dramatically. The second jobs of ADPCM, MgSort, FFT2D, and Rotate never experienced any cache misses for all inputs and all memory layouts. This implies their working data from the first job are still kept in the data cache perfectly. In Default configuration, Dijkstra and TinyJPEG show 87.2% and 87.5% of their original cache misses respectively in average. This tells the second jobs of these programs still cause significant data cache misses across different memory layout variants.

Dijkstra needs to be discussed more due to its quirky behavior. First, there is dramatic variation with different data memory layouts. Second, even though the data cache (8KB) is large enough to hold all of the data (5,564 bytes), it does not eliminate data cache misses for the second job. The warm cache case still generates many cache misses. All these behaviors are explained by cache contention. Dijkstra’s static, automatic, and dynamic variables thrash the cache. The three different types of data are located in separated memory regions, but they are mapped into the same cache line in case of Dijkstra. It cannot avoid contentious mapping into the same cache lines due to their location in data memory. Consequently, memory layout changes lead to different patterns of cache misses, and cache pollution by the task itself delivers many cache misses even in the warm cache case.

This implies that static cache miss analysis techniques by prior work may be fragile and be invalidated by minor memory layout changes. Moreover, multi-threaded real-time systems can suffer unnecessarily from cache pollution by context switching. From this observations, our work suggests SPM data allocation schemes; data cache misses are analyzed through simple measurements rather than deliberate efforts for precise analysis, and inter-data cache pollution is minimized by allocation heuristics based on the cache miss analysis.

This study examines how to use the advantages of both fast memory types in a complementary way. To leverage the particular benefits of SPM and data cache, each fast memory
holds different categories of data. SPM is used to manage data causing many cache misses and severe inter-data cache pollution, while the data cache covers the other remaining data in main memory. This improves performance at the memory access level. The data stored in SPM is accessed still as fast that as in data cache. This reduces cache pollution, and the total number of cache misses falls. This can be interpreted as the effective capacity of the cache being increased since the data allocated to SPM is not cached any more. Taken together, allocating data into SPM brings performance improvement of data cache with little timing variability improvement by reducing cache contention.

5.3 Allocation of Data into SPM

Our new data cache miss analysis classifies cache misses of a system in two groups: intrinsic misses and interference misses. We build on this concept to propose heuristics to allocate data into SPM.

5.3.1 Intrinsic Misses and Interference Misses

Before looking into the details of the two types of data cache misses, we define a term: a data set is a set of data variables which has unique and fixed memory address range. This also implies each data set is exclusive to others, and each data variable must belong to only one data set. For instance, all static variables, and all automatic variables of a program can be two different data sets as they do not share the same address. Similarly all data variables of a task in a real-time system may compose a data set at task level granularity. However, particular subgroups of automatic variables of a program may not compose one data set since their addresses may not be uniquely fixed. The stack of the program grows up and down as procedure calls and returns occur, so some data variables of different procedures may share same memory address. For the same reason, a particular group of dynamic variables of a multi-threaded system may not make up a data set since same addresses in heap memory may be allocated to multiple dynamic type variables of different tasks during task scheduling.

The intrinsic misses of a data are the data cache misses of the data set when only that data set (and no other) is cacheable in a program or a system. Thus the intrinsic misses of a data set are a characteristic of the data set because they are determined by the program logic, an input workload to the program, a particular data memory layout after building the program, and hardware running the program but not by other data sets.

**Definition 5.3.1.** The intrinsic miss count $m(V)$ of a data set $V$ is the total number of cache misses caused by $V$ when only $V$ is cacheable in a system.
The interference misses of a group of data sets are cache misses due to cache contention and cache pollution for the same cache lines by the multiple data sets of the data set group. This type of miss is determined by contending and interfering relationships among the data sets.

**Definition 5.3.2.** The interference miss count of data sets is the difference between the total number of cache misses of the data sets and sum of the intrinsic misses of all the data sets.

The intrinsic misses are the cache misses of a data set observed when only that data set uses the data cache, while the interference misses are unnecessary cache misses of the program or the system because all the data sets contend for the limited cache resource. In our analysis, we need not detail how the misses are generated in the system.

From these definitions, two characteristics are induced. The notation $m(x)$ represents number of intrinsic misses by a data set $x$. $m(x\#y)$ denotes the number of interference misses between two data sets $x$ and $y$. When $X$ is a super set of sub elementary data sets, $m(\#X)$ returns number of interference misses inside of the super set $X$ when all the sub elementary sets of $X$ are cacheable at the same time.

**Theorem 5.3.1.** In a system $V$, when only $n$ data sets ($V = V_1 \cup \ldots \cup V_n$) are cacheable at the same time, the total number of cache misses of the system is the sum of the intrinsic misses of all the $n$ data sets and the system’s interference misses, i.e.,

$$m(V) = \sum_{i=1}^{n} m(V_i) + m(\#V)$$

**Proof.** If some data sets share the same cache lines so that extra misses exist in addition to the intrinsic misses of the $n$ data sets, the total number of cache misses of the system is increased by a positive integer $\alpha$, i.e.,

$$m(V) = \sum_{i=1}^{n} m(V_i) + \alpha$$

$$\Leftrightarrow \alpha = m(V) - \sum_{i=1}^{n} m(V_i)$$

From Definition 5.3.2, $\alpha = m(\#V)$.

If no data sets share any cache lines or no extra misses are observed, the total number of cache misses of the system is not increased due to cache contention between the data sets, which means $m(\#V) = 0$. Then,

$$m(V) = \sum_{i=1}^{n} m(V_i)$$
From both the cases above,

\[ m(V) = \sum_{i=1}^{n} m(V_i) + m(\#V) \]

\[ m(V_i) = m(V) - m(V_{\Omega}) - m(V_{\Theta}) \]

**Theorem 5.3.2.** In a system \( V \), when only \( n \) data sets \( (V = V_1 \cup ... \cup V_n) \) are cacheable at the same time, and they are grouped into two super sets \( V_\Omega \) and \( V_\Theta \) exclusively and exhaustively, i.e., \( V = V_\Omega \cup V_\Theta \) while \( V_\Omega \cap V_\Theta = \emptyset \), then one identity is established as below.

\[ m(V_\Omega \#V_\Theta) = m(\#V) - m(\#V_\Omega) - m(\#V_\Theta) \quad (5.2) \]

**Proof.** Since all data sets from \( V_1 \) to \( V_n \) are exclusive to each other, the assumptions about \( V_\Omega \) and \( V_\Theta \) are also satisfied. Then, Theorem 5.3.1 is applicable.

\[ m(V) = m(V_\Omega \cup V_\Theta) \]

\[ = m(V_\Omega) + m(V_\Theta) + m(V_\Omega \#V_\Theta) \]

By applying Theorem 5.3.1 again to \( m(V_\Omega) \) and \( m(V_\Theta) \),

\[ m(V_\Omega) + m(V_\Theta) = \sum_{i=1}^{n} m(V_i) + m(\#V_\Omega) + m(\#V_\Theta) \]

Therefore,

\[ m(V) = \sum_{i=1}^{n} m(V_i) + m(\#V_\Omega) + m(\#V_\Theta) + m(V_\Omega \#V_\Theta) \]

Also,

\[ m(V) = \sum_{i=1}^{n} m(V_i) + m(\#V) \]

Finally,

\[ m(V_\Omega \#V_\Theta) = m(\#V) - m(\#V_\Omega) - m(\#V_\Theta) \]

Notice that Equation(5.2) is an identity originating from Equation(5.1) – a different perspective of the data sets.

One example explaining this new concept of data cache miss analysis is depicted in Figure 5.2. Two data sets (S1 and S2) are cached by a data cache. Data set S1 generates a memory
trace – a, b, a, b, while data set S2 causes another memory trace – x, y. All the memory accesses are mapped to the same cache lines, and the data cache is a 2-way set associative cache whose replacement policy is LRU. These memory traces are reordered by time (right to left) at the side of the data cache in Figure 5.2.

When only the memory access of S1 is cached, the data cache experiences 2 compulsory misses as illustrated. When only the memory access of S2 is cached, the data cache meets 2 compulsory misses too. These data cache miss counts are intrinsic cache misses of the two data sets respectively. When the two data sets are cacheable at the same time and their memory accesses are commingled – a, b, x, y, a, b – as like in Figure 5.2, 6 data cache misses are observed in total. This defines the interference miss counts between S1 and S2 as 2.

The concept of the interference misses may be used for explaining context switching misses when it is provided that data set S1 is all data of one task and data set S2 is all data set of another task. Then, the commingled memory trace above models memory accesses when the task of S2 preempts the task of S1. Beyond the context switching miss, the concept of interference miss can explain a more generalized case. For example, within in a task scope, if S1 is a stack of the task while S2 is a data set composed of all static type data variables of the same task, the commingled memory trace captures memory accesses by the task. In any cases, extra data cache misses appear due to cache contention between the data sets and the interference miss count is a conceptual metric to capture the degree of the data cache contention.

5.3.2 Allocation Scheme – Exhaustive Search

Provided that we allocate some data sets to SPM (here let $V_\Omega$ groups the target data sets), definitely it is beneficial to move data sets causing more cache contention to other data sets (here they are bundled up with $V_\Theta$). Theorem 5.3.2 suggests an approach to identify the data
sets. The term $m(V_\Omega \# V_\Theta)$ of Equation(5.2) is number of interference misses between two super sets $V_\Omega$ and $V_\Theta$, and the right hand side tells how to calculate it. Then we may choose one $V_\Omega$ among all possible combinations which maximizes the term $m(V_\Omega \# V_\Theta)$. For example, when 2 tasks are chosen from 6 tasks, $6C_2$ different combinations may be examined, and one of the combinations is selected if it maximizes the benefit.

The term $m(\#V)$ of the right hand side in Equation(5.2) is the interference miss count by all $n$ data sets. This can be computed by applying Theorem 5.3.1 after measuring the intrinsic misses of all the $n$ data sets and the total number of cache misses. The memory layouts are, of course, fixed during the $n + 1$ times of measurement.

Per each $V_\Omega$, the other terms of the right hand side in Equation(5.2) are recalculated. The second term $m(\#V_\Omega)$ is the interference miss between data sets composing $V_\Omega$. This can be computed by Theorem 5.3.1 and measurement of cache misses when only $V_\Omega$ is cacheable in the system. Notice that the intrinsic miss count of each set of $V_\Omega$ is already known from the first calculation above when computing $m(\#V)$. Since memory layout does not change during this analysis, the intrinsic misses from the first measurement can be reused. Similarly the third term $m(\#V_\Theta)$ also can be calculated in the same manner.

However the calculation above is trivial because we have already known which combination is the most beneficial one. We already measure the total cache miss counts by each possible $V_\Omega$ and corresponding $V_\Theta$. Those are actual cache misses which the system will experience after the allocation. Therefore the selection process becomes an exhaustive exploration although Theorem 5.3.2 is applied, and it requires significantly more effort as the number of sets increases. The complexity of this search when examining all combinations of $n$ data sets is $O(2^n)$.

5.3.3 Allocation Scheme – Heuristics

First of all, an undirected graph called cache contention degree graph is defined. The graph represents intrinsic misses of each data set and interference misses between each pair of data sets. A node of this graph represents one data set, and a node has arcs to other nodes annotated with the interference miss count between the nodes. The interference miss count between two nodes can be obtained from Theorem 5.3.1. Each node is also annotated with its intrinsic miss count. A graph used in this work is presented in Figure 5.3. Similarly cache contention degree table is also defined. Each data set is enumerated both in the top most row and the left most column in same sequence. The intrinsic miss counts of the data sets fill out diagonal cells of the table and the intrinsic miss counts fill out other cells crossed by a row of and a column of two data sets respectively. Table 5.3 is an example.

Two heuristic schemes are devised in order to avoid the computational complexity of the exhaustive scheme presented above. For the first heuristic scheme called heuristic I, one metric
is calculated from the graph for each node: the sum of the intrinsic miss count of a node $N$ and the fluxes from $N$ to all other nodes. This metric prioritizes all nodes, and allocation to SPM follows the order of the priority until the SPM is filled. When the SPM capacity is limited, large data sets may lose an opportunity for being allocated into SPM in spite of their high allocation priority.

The second heuristic scheme named \textit{heuristic II} uses another metric, which sums up all intrinsic misses and interference misses of all nodes remaining in their original memory rather than moving into SPM. This metric captures cache contention degree by data still remaining cacheable. Finally one combination of data sets minimizing the metric is selected for efficient use of SPM after examining all possible combinations of data sets under a certain SPM size.

Both metrics estimate cache contention of the data sets and give guidance to select more beneficial data sets in terms of cache performance. These schemes also imply that the interference miss count of a set with all other data sets is positively correlated with (even though it is not proportional to) the sum of the interference miss counts of each pair of data sets, including the set of interest. The rationale behind this is that the interference miss counts annotated in the graph account for the relationship between only two data sets. The interference misses between more data sets are greater than or equal to the annotated interference misses, although it varies based on particular cache mapping. The estimation appearing in the \textit{cache contention degree graph and table} is regarded as an estimation of first order of cache contention degree among data sets. The complexity of the proposed heuristic is $O(n^2)$. The complexity is only polynomial time, unlike the exponential complexity of the exhaustive approach. Obtaining a more precise estimate by considering additional task interferences will raise the computational complexity of the analysis.

5.4 Experiments

We carried out two sets of experiments (at the coarse granularity level and at the fine granularity level) to evaluate the proposed schemes and the effect of SPM on data cache misses. We use a real hardware platform running a real-time operating system with several tasks.

5.4.1 Target System

The hardware system used for the experiments has an ARM7TDMI-S processor core running at 36MHz, and a heterogeneous memory system; 64KB internal SRAM (SPM), 2MB external SRAM, 32MB SDRAM, 2MB internal flash ROM, and 8KB unified cache. The flash ROM stores program code. Only the data cache is enabled, so the capacity of the data cache is 8KB. The other details of the hardware are listed in Table 5.1.
### Table 5.1: Target hardware system features and toolchain

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td>NXP LPC2888 (ARM7TDMI-S, 36MHz)</td>
</tr>
<tr>
<td>Int. SRAM (SPM)</td>
<td>64KB, 32-bit width, not cacheable, 2/2 clock cycles for read/write of 32 bits</td>
</tr>
<tr>
<td>Ext. SRAM</td>
<td>2MB, 16-bit width, cacheable, 17/14 clock cycles for read/write of 32 bits</td>
</tr>
<tr>
<td>Unified Cache</td>
<td>8KB, 2-way set associative, 32 bytes cache line, LRU replacement, write-back, no allocation on write miss, only data cache enabled, 1/1 clock cycle for read/write of 32 bits</td>
</tr>
<tr>
<td>SDRAM</td>
<td>32MB, 16-bit width, not cacheable</td>
</tr>
<tr>
<td>Flash ROM</td>
<td>2MB, 128-bit width</td>
</tr>
<tr>
<td>RTOS</td>
<td>FreeRTOS, 1 ms scheduling period</td>
</tr>
<tr>
<td>Toolchain</td>
<td>GNU arm-elf-gcc 4.1.1, arm-elf-as 2.17 newlib 1.14.0</td>
</tr>
</tbody>
</table>

The SPM has faster read and write latency than the external SRAM. Unfortunately, in our system the SPM access time is longer than the data cache hit access time, which weakens the performance impact of our approach from its potential. Other common systems feature scratchpad memories which are as fast as the data cache. The gap in access latency between data cache and SPM affects the execution time of tasks, and consequently preemption, which increases number of cache misses much as discussed in Section 5.2. However, the maximum difference in the number of preemptions measured is so small that the gap is negligible.

Test programs are collected from open source domain and MiBench [30]: Dijkstra, ADPCM, TinyJPEG, MgSort, FFT2D, and Rotate. Dijkstra and TinyJPEG use dynamic type data, and Dijkstra and MgSort contain recursive procedure calls. These programs are chosen to demonstrate different types of code and data behavior.

As input workloads to all tasks except TinyJPEG, five different 8KB data sets are drawn from five classic texts: The Adventures of Huckleberry Finn, The Catcher in the Rye, The Lord of the Rings, The Odyssey, and The Three Musketeers. For TinyJPEG, five 64×48 pixel images are prepared. These five inputs add timing variability to the execution time of the programs. Once the system runs, these five inputs are provided in a round-robin fashion.

Each program is run as one task in our real-time system. Table 5.2 presents real-time properties and memory usage of the programs. A preemptive real-time operating system called FreeRTOS [99] schedules those tasks. The priority of each task is assigned rate-monotonically.
<table>
<thead>
<tr>
<th>Task</th>
<th>Priority</th>
<th>WCET (ms)</th>
<th>BCET (ms)</th>
<th>Period (ms)</th>
<th># Jobs</th>
<th>Memory Usage (Bytes)</th>
<th>Statics</th>
<th>Dynamics</th>
<th>Autos.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>1</td>
<td>681.3</td>
<td>605.5</td>
<td>6,500</td>
<td>30</td>
<td>4,388</td>
<td>1,020</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>ADPCM</td>
<td>2</td>
<td>623.1</td>
<td>598.7</td>
<td>6,500</td>
<td>30</td>
<td>948</td>
<td>0</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td>TinyJPEG</td>
<td>3</td>
<td>555.0</td>
<td>356.1</td>
<td>5,000</td>
<td>39</td>
<td>260</td>
<td>51,972</td>
<td>1,580</td>
<td></td>
</tr>
<tr>
<td>MgSort</td>
<td>4</td>
<td>474.3</td>
<td>473.7</td>
<td>5,000</td>
<td>39</td>
<td>4,100</td>
<td>0</td>
<td>552</td>
<td></td>
</tr>
<tr>
<td>FFT2D</td>
<td>5</td>
<td>262.3</td>
<td>262.1</td>
<td>2,500</td>
<td>78</td>
<td>4,100</td>
<td>0</td>
<td>568</td>
<td></td>
</tr>
<tr>
<td>Rotate</td>
<td>6</td>
<td>164.1</td>
<td>163.9</td>
<td>1,500</td>
<td>130</td>
<td>4</td>
<td>0</td>
<td>4,540</td>
<td></td>
</tr>
</tbody>
</table>

Higher priority is represented by a higher number in Table 5.2. The system is designed to have about 62.1% processor utilization, with about 10% per task. Overhead due to the scheduler and experiment-management code raises actual utilization to 69.7%. The actual sizes of the input workloads used by the tasks are adjusted to make the hyperperiod of the system feasible. Here the hyperperiod is 195 seconds.

Only memory accesses to the external SRAM are cacheable, and all factors which can affect cache misses are controlled except data of the test programs. For example, internal data such as task management lists, stacks of the RTOS itself, dynamic memory management library and cache miss measurement library are allocated into a non-cacheable memory region (SDRAM). Only the data used by the programs themselves are cacheable.

### 5.4.2 Experiment Setup

Our data cache performance during the hyperperiod is measured by reading a hardware cache miss counter inside of the processor per each data set combination allocated for SPM. The hardware counter makes our new cache miss analysis easy. For systems lacking that feature, many cache miss analysis techniques may be accommodated as Ramaprasad et al. [84] did. This does not necessarily degrade applicability of our schemes. Obtaining accurate estimation of cache miss is a subsidiary benefit by our scheme’s reducing of cache misses overall since total number of cache misses can be reduced by using a small amount of SPM, which will be presented in the experiment results. This work seeks a method with multiple goals; reducing cache misses by using SPM and providing a cache miss analysis scheme to support it.

### Coarse Granularity Level

At experiment of the coarse granularity level, all task combinations from the test tasks are searched exhaustively. We allocate data from 0 to 6 of tasks into the SPM, exploring 64
If a task is determined to be allocated into SPM, all of its static and automatic data are moved into SPM. The data of the other tasks remain in the external SRAM and are still cacheable. Dynamic type data are excluded from this experiment because the exclusivity condition of data sets is not guaranteed, as mentioned previously.

The prior section shows that slightly different data memory layouts may unpredictably affect the number of cache misses. Hence, we use two different types of allocation per each combination of the tasks to evaluate our heuristic excluding (Controlled-Allocation) and including (Real-Allocation) that noise. Every measurement is performed for both types of allocation (128 times in total).

Controlled-Allocation pins the data memory layout throughout all the exploration. To keep the data memory layout of the static data variables unchanged, each static variable has one dummy static variable with the same size. The dummy static data variables occupy the same location as the original static data in the external SRAM so that other static variables also can keep their original location, after its original variable is moved to SPM. Of course, the dummy variables are never accessed. In the case of automatic data variables, the task stack allocator
of FreeRTOS is modified so that each task is always assigned to the same memory location in the external SRAM.

No artificial adjustment to memory layout is applied to \textit{Real-Allocation}. \textit{Controlled-Allocation} is an idealized but wasteful memory allocation. On the other hand, \textit{Real-Allocation} is a default memory allocation actually performed by the linker and the RTOS stack allocator, although it does not fix a unique memory layout during the measurement.

The new data cache miss analysis is also performed to apply the heuristic methods. The cache miss analysis uses the \textit{Controlled-Allocation}. First, the intrinsic misses of the tasks are measured as well as the total number of cache misses. Then each pair of the tasks is moved to SPM. Finally we can compute interference miss counts between any pair of tasks. All these measurements are already included in the exhaustive measurement explained above. When the suggested heuristics are applied to real world situations, only these measurements are performed. Then the data cache misses can be analyzed as explained in the prior section, and we obtain new memory allocation consequently.

Finally a cache interference degree graph is drawn as in Figure 5.3. Each node in Figure 5.3 contains the number of intrinsic cache miss in parentheses and interference misses with other tasks. The heuristic I gives us a sequence of the data set for allocation. The resulting priority of the tasks to be allocated into SPM follows the order: \textit{Rotate} → \textit{Dijkstra} → \textit{FFT2D} → \textit{MgSort} → \textit{ADPCM} → \textit{TinyJPEG} from higher to lower. This order means that as the number of tasks being allocated into SPM increases, it is more beneficial to select the tasks starting with the \textit{Rotate} task. The graph is drawn to emphasize the \textit{Rotate} task having the highest priority.

One experiment traces a trajectory of the sequence of the tasks allocated into SPM under unlimited SPM capacity. Only heuristic I participates in this experiment. In order to check feasibility of the two heuristics in real world, three kinds of SPM capacity are also investigated: 4KB, 8KB and 16KB. The two heuristics select one combination of data sets from the six task data sets until the SPM budget is filled out.

\textbf{Fine Granularity Level}

In the fine granularity level experiments, static type and automatic type data variables of a task are separated to two different data sets rather than one large data set as are done in the coarse granularity. Since an automatic type data can have different addresses per its procedure instance due to varying stack depth, it is not easy to split into the finer level. Each static type data variable may consist one data set, which will increase effort for applying the proposed schemes quadratically since the complexity of the data cache miss analysis is \(O(n^2)\). Dynamic type data variables are also excluded for the same reason stated above. Other factors for the experiments are set up same to those of the coarse granularity level one.

The performance trajectory of the heuristic I is traced as like at the coarse granularity level.
with unlimited SPM capacity. However, 1KB SPM is used to avoid a tremendous amount of measurements, i.e. sweeping $2^{12}$ combinations. We have 12 data sets in total at this level (2 data sets of 6 tasks). In addition, three kinds of small SPM capacity are also evaluated: 512B, 1KB, and 4,544B for both the heuristics. The larger size SPM is not considered due to an excessive amount of calculations.

The cache miss analysis is also performed, and a corresponding cache contention degree table is exhibited in Table 5.3. In the table, a subscript $S$ means static type data variables and a subscript $A$ stands for automatic type data variables. For example, $D_A$ indicates a data set composed of all automatic type data variables of Dijkstra. The sequence of the data sets obtained from the table by the heuristic $I$ is $D_S \rightarrow R_A \rightarrow D_A \rightarrow F_S \rightarrow M_S \rightarrow T_A \rightarrow F_A \rightarrow A_A \rightarrow R_S \rightarrow A_S \rightarrow M_A \rightarrow T_S$.

Definition 5.3.2 and Theorem 5.3.1 assume that the interference miss counts are positive, but Table 5.3 contains some negative interference miss counts. Figure 5.4 explains two reasons why the interference miss counts are calculated to negative numbers. Those negative numbers are not dropped from the analysis. Our goal is estimating data cache contention degree to guide better SPM allocation, not acquiring precise data cache miss counts. These exceptional cases reflect difficult problems posed by real world systems.

Although exclusive memory address between data sets is provided, in the real world one cache line can be co-allocated to different multiple data sets. Thus, once one of the data sets in the co-allocation cache line is accessed, this access derives a prefetching effect to the other data sets of the cache line. This case is demonstrated in Figure 5.4a. One data X of Task2 is

### Table 5.3: Cache contention degree table used in fine granularity level experiments

<table>
<thead>
<tr>
<th></th>
<th>$D_S$</th>
<th>$D_A$</th>
<th>$A_S$</th>
<th>$A_A$</th>
<th>$T_S$</th>
<th>$T_A$</th>
<th>$M_S$</th>
<th>$M_A$</th>
<th>$F_S$</th>
<th>$F_A$</th>
<th>$R_S$</th>
<th>$R_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_S$</td>
<td>1,228</td>
<td>94,042</td>
<td>-1</td>
<td>1,712</td>
<td>0</td>
<td>4</td>
<td>4,162</td>
<td>8</td>
<td>7,768</td>
<td>5,769</td>
<td>0</td>
<td>38,896</td>
</tr>
<tr>
<td>$D_A$</td>
<td>94,042</td>
<td>34</td>
<td>14</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>9</td>
<td>4</td>
<td>14</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>$A_S$</td>
<td>-1</td>
<td>14</td>
<td>261</td>
<td>0</td>
<td>-2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$A_A$</td>
<td>1,712</td>
<td>4</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>-10</td>
<td>10</td>
<td>12</td>
<td>0</td>
<td>-4</td>
</tr>
<tr>
<td>$T_S$</td>
<td>0</td>
<td>11</td>
<td>-2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$T_A$</td>
<td>4</td>
<td>11</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>426</td>
<td>191</td>
<td>-2</td>
<td>307</td>
<td>10</td>
<td>5</td>
<td>15,354</td>
</tr>
<tr>
<td>$M_S$</td>
<td>4,162</td>
<td>11</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>191</td>
<td>385</td>
<td>8</td>
<td>657</td>
<td>2</td>
<td>0</td>
<td>23,580</td>
</tr>
<tr>
<td>$M_A$</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>-10</td>
<td>4</td>
<td>-2</td>
<td>8</td>
<td>99</td>
<td>-3</td>
<td>9</td>
<td>-10</td>
<td>-10</td>
</tr>
<tr>
<td>$F_S$</td>
<td>7,768</td>
<td>4</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>307</td>
<td>657</td>
<td>-3</td>
<td>1,153</td>
<td>9</td>
<td>-2</td>
<td>33,854</td>
</tr>
<tr>
<td>$F_A$</td>
<td>5,769</td>
<td>14</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>123</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>$R_S$</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>-10</td>
<td>-2</td>
<td>0</td>
<td>2</td>
<td>1,430</td>
</tr>
<tr>
<td>$R_A$</td>
<td>38,896</td>
<td>11</td>
<td>0</td>
<td>-4</td>
<td>0</td>
<td>15,354</td>
<td>23,580</td>
<td>-10</td>
<td>33,854</td>
<td>-2</td>
<td>1,430</td>
<td>4,291</td>
</tr>
<tr>
<td>Sum</td>
<td>153,588</td>
<td>94,176</td>
<td>276</td>
<td>1,796</td>
<td>15</td>
<td>16,315</td>
<td>28,999</td>
<td>102</td>
<td>43,757</td>
<td>5,946</td>
<td>1,436</td>
<td>117,400</td>
</tr>
</tbody>
</table>

67
co-allocated into a cache line to which some data of Task1 including data M and data N are allocated. Once either M or N is accessed at the first time, X will be loaded automatically to the cache line. Thus, a first memory access to data X will not experience any compulsory misses. When we calculate interference misses, memory access to data X will not experience compulsory misses for this reason since Task1 and Task2 are cacheable at the same time. However, compulsory misses to this cache line are duplicated when measuring intrinsic miss counts of the two tasks respectively. This case happens between \( D_S \) and \( A_S \), \( A_S \) and \( T_S \), and \( F_S \) and \( R_S \) in Table 5.3. Our linker allocates consecutive addresses to some static variables of those tasks.

Another case is because of some timing jitter of our real-time scheduler tick. Although the real-time scheduler tick occurs periodically, the timing which a task experiences may vary on several factors. Different memory allocations change the execution time of tasks, which affects timing of the scheduler tick for other tasks. The problem arises because this jitter can cause extra memory operations. When a scheduler tick occurs, the scheduler of our real-time system saves architectural registers and other house-keeping information on the stack of the currently-executing task. These memory operations are performed even when the current task will resume its execution (i.e. without being preempted by another task). Hence if the scheduler tick exposes some timing jitter and the memory operations for saving registers grow the stack beyond the current maximum depth, data cache miss counts can increase somehow.

Figure 5.4b illustrates this complicated situation, where stack depth is plotted over time. The original scheduler tick occurs at time A. At that time, our scheduler will cause some extra memory accesses (shaded region, \( d_2 - d_1 \)) on top of the current stack (\( d_1 \)). If there is no scheduler jitter, our next scheduler tick will not experience extra cache misses although the stack depth grows up (to level \( d_3 \)). However, if the scheduler tick lags a bit so that our scheduling tick comes up at time B, our scheduler generates extra memory accesses (\( d_4 - d_3 \))
Table 5.4: Number of scheduler ticks encountering by \textit{MgSort} and \textit{Rotate}

<table>
<thead>
<tr>
<th>Task</th>
<th>Data in SPM</th>
<th>Total number of Scheduler Ticks</th>
<th>Number of Non-switching</th>
<th>Number of Jobs</th>
<th>Number of Preemptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MgSort</td>
<td>$M_A$</td>
<td>20,292</td>
<td>20,240</td>
<td>39</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>$M_A \cup R_A$</td>
<td>20,275</td>
<td>20,223</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate</td>
<td>$R_A$</td>
<td>23,710</td>
<td>23,580</td>
<td>130</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$R_A \cup M_A$</td>
<td>23,579</td>
<td>23,449</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: Best task combinations allocated into SPM and corresponding cache misses counts from coarse granularity level experiments (\textit{Controlled-Allocation})

<table>
<thead>
<tr>
<th>Number of Tasks in SPM</th>
<th>Tasks in SPM</th>
<th>Intrinsic Miss</th>
<th>Interference Miss</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$R$</td>
<td>103,544</td>
<td>514,630</td>
<td>618,174</td>
</tr>
<tr>
<td>1</td>
<td>$R \cup D$</td>
<td>97,821</td>
<td>132,509</td>
<td>230,330</td>
</tr>
<tr>
<td>2</td>
<td>$R \cup D \cup F$</td>
<td>2,527</td>
<td>75,506</td>
<td>78,033</td>
</tr>
<tr>
<td>3</td>
<td>$R \cup D \cup F \cup M$</td>
<td>1,242</td>
<td>8,411</td>
<td>9,653</td>
</tr>
<tr>
<td>4</td>
<td>$R \cup D \cup F \cup M \cup T$</td>
<td>750</td>
<td>2</td>
<td>752</td>
</tr>
<tr>
<td>5</td>
<td>$R \cup D \cup F \cup M \cup T \cup A$</td>
<td>324</td>
<td>0</td>
<td>324</td>
</tr>
</tbody>
</table>

which will cause some extra compulsory misses.

Table 5.4 presents the measured number of scheduler ticks which two tasks experience with during the hyperperiod. Different memory configurations cause a different number of scheduler ticks, which is observed at the third column. The fourth column (non-switching) tells how many scheduler ticks occur but do not switch current task with other task. Their differences exactly match up with the sum of the number of jobs and the number of preemptions during the hyperperiod, of course. In fact, $M_A$ and $R_A$ data sets show more negative interference misses (−10) than other data sets.

5.5 Results

5.5.1 Allocation at Coarse Granularity Level

The system’s total number of cache misses during the hyperperiod is 618,174, shown in Table 5.5 and 5.6 as the case of zero tasks allocated into SPM. From Figure 5.3 the interference misses are estimated to be 83.3% of the total cache misses, and the intrinsic misses are only 16.7% in
Figure 5.5: Reduction of cache misses by SPM allocation using heuristic I and other methods on exhaustive search from coarse granularity level experiments

Controlled-Allocation. It is from cache contention between data sets and corresponding cache pollution that most cache misses arise.

The cache miss counts for all 64 task combinations are plotted in Figure 5.5, sorted hori-
Table 5.6: Best task combinations allocated into SPM and corresponding cache misses counts from coarse granularity level experiments (Real-Allocation)

<table>
<thead>
<tr>
<th>Number of Tasks in SPM</th>
<th>Tasks in SPM</th>
<th>Intrinsic Miss</th>
<th>Interference Miss</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>102,119</td>
<td>516,055</td>
<td>618,174</td>
</tr>
<tr>
<td>1</td>
<td>R</td>
<td>97,826</td>
<td>132,128</td>
<td>229,954</td>
</tr>
<tr>
<td>2</td>
<td>R ∪ D</td>
<td>2,522</td>
<td>74,546</td>
<td>77,068</td>
</tr>
<tr>
<td>3</td>
<td>R ∪ D ∪ F</td>
<td>1,242</td>
<td>6,069</td>
<td>7,311</td>
</tr>
<tr>
<td>4</td>
<td>R ∪ D ∪ F ∪ M</td>
<td>754</td>
<td>0</td>
<td>754</td>
</tr>
<tr>
<td>5</td>
<td>R ∪ D ∪ F ∪ M ∪ T</td>
<td>324</td>
<td>0</td>
<td>324</td>
</tr>
<tr>
<td>6</td>
<td>R ∪ D ∪ F ∪ M ∪ T ∪ A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

horizontally by the number of tasks with data allocated into SPM. The blue solid line shows the performance trajectory of a system using the heuristic I, i.e., the tasks are added into SPM in sequence of Rotate → Dijkstra → FFT2D → MgSort → ADPCM → TinyJPEG under unlimited SPM capacity.

In both the case of Controlled-Allocation and Real-Allocation, the heuristic is best at reducing the total cache misses of the system, so it gives very good guidance in choosing appropriate data for SPM. The combinations showing the best result for each number of tasks with data in SPM are listed in Table 5.5 and 5.6.

Prior research work selects data to allocate to SPM based on memory access counts or frequency [1, 61, 8, 96, 25]. We next examine how well these approaches work with our experimental configuration in comparison with our method. Table 5.7 presents the portion of total memory access counts and the portion of intrinsic cache miss counts by each task during the hyperperiod. Using this data we examine the performance trajectory of three allocation sequences, shown by the three black lines (dotted, dot-dashed, and dashed lines) in Figure 5.5.

The first is ordered by the task’s scheduling priority and results in the sequence Rotate → FFT2D → MgSort → TinyJPEG → ADPCM → Dijkstra. The second is ordered by the portion of total memory access counts and results in the sequence ADPCM → MgSort → FFT2D → Dijkstra → Rotate → TinyJPEG. The third sequence is ordered by the portion of intrinsic cache miss counts and result in the sequence Dijkstra → Rotate → FFT2D → MgSort → TinyJPEG → ADPCM. None of these allocation scheme works as well as our heuristic, although the intrinsic cache miss count-based scheme (the dashed lines) works closely. Those trajectories imply that the inter-task cache pollution is not determined solely by the scheduling priority (or the number of jobs during the hyperperiod since the target system adopts the rate-monotonic priority), the memory access amounts or the number of cache misses. It results
Table 5.7: Portion of memory access counts and intrinsic cache miss counts by each task during
hyperperiod

<table>
<thead>
<tr>
<th>Task</th>
<th>Memory Access Count</th>
<th>Intrinsic Cache Miss Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>12.8%</td>
<td>92.0%</td>
</tr>
<tr>
<td>ADPCM</td>
<td>34.3%</td>
<td>0.3%</td>
</tr>
<tr>
<td>TinyJPEG</td>
<td>1.2%</td>
<td>0.4%</td>
</tr>
<tr>
<td>MgSort</td>
<td>27.6%</td>
<td>0.5%</td>
</tr>
<tr>
<td>FFT2D</td>
<td>16.0%</td>
<td>1.3%</td>
</tr>
<tr>
<td>Rotate</td>
<td>8.1%</td>
<td>5.5%</td>
</tr>
</tbody>
</table>

from superposition of those factors. The proposed heuristic successfully addresses the complex
interactions among these factors.

It is interesting to note that by moving only one task (Rotate), the total number of cache
misses is reduced to 37.3% (Controlled-Allocation) and 37.2% (Real-Allocation) of the original
(i.e. 618,174). This would be counterintuitive without using our approach, since that the
intrinsic misses of Rotate account for only 0.9% of the total cache misses. However, allocating
Rotate into SPM reduces the total cache misses of the system dramatically.

Table 5.5 and 5.6 also list how much intrinsic and interference misses fall with the proposed
scheme. This shows the synergy between data cache and SPM which we have proposed. The
task data are serviced by fast SPM, so they will not experience cache misses. In addition,
data cache contention and pollution are reduced, helping the other tasks with data which still
remains in the cacheable memory region. This behavior continues as more tasks are allocated
into SPM.

The two allocation policies – Controlled-Allocation and Real-Allocation show similar behav-
ior but slightly different numbers of cache misses through all the exploration. The maximum
difference of number of cache misses between the two policies is 3.2% when the difference is
normalized to the original number of cache misses. This implies that different memory layouts
still perturb the result of static cache miss analysis slightly. However, our allocation reduces
the total number of cache misses and consistently overcomes the obstacles.

When SPM capacity is limited, the two heuristics indicate the same combination of data sets
for best cache performance as shown in Table 5.8. Due to data set sizes, the 4KB SPM capacity
supports only three combinations of data sets, the 8KB SPM supports 19 combinations, and
the 16KB SPM supports 55 combinations.

4KB, 8KB and 16KB SPM reduce the total number of cache misses to 91.2%, 31.3% and 0.2%
of the original respectively. Due to the large granularity of the task level data set, 4KB SPM
is exploited less efficiently. However, 4,544B SPM reduces the cache miss counts dramatically
Table 5.8: Best combinations under limited SPM capacity at coarse granularity (*Controlled-Allocation*)

<table>
<thead>
<tr>
<th>SPM Capacity</th>
<th>Best Combination</th>
<th>Heuristic I</th>
<th>Heuristic II</th>
<th>Cache Miss Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>A ∪ T</td>
<td>A ∪ T</td>
<td>A ∪ T</td>
<td>563,871</td>
</tr>
<tr>
<td>8KB</td>
<td>R ∪ A ∪ T</td>
<td>R ∪ A ∪ T</td>
<td>R ∪ A ∪ T</td>
<td>193,428</td>
</tr>
<tr>
<td>16KB</td>
<td>R ∪ D ∪ F ∪ A</td>
<td>R ∪ D ∪ F ∪ A</td>
<td>R ∪ D ∪ F ∪ A</td>
<td>1,100</td>
</tr>
</tbody>
</table>

Figure 5.6: Reduction of cache misses by SPM (1KB) allocation into using heuristic I and exhaustive search at fine granularity level

(a) *Controlled-Allocation*  
(b) *Real-Allocation*

to 37.3% of the original one (in case of *Rotate* allocated in SPM). Coarse granularity allocation may use SPM capacity less efficiently. However, too fine granularity allocation also may suffer from much abstraction of the cache contention degree, which will be discussed soon.

5.5.2 Allocation at Fine Granularity Level

Cache performance at the fine granularity level is also evaluated as the number of data sets located in SPM increases. However, since tracing all possible combinations of the data sets at this level under unlimited SPM capacity requires tremendous effort as stated above, only the 1KB SPM budget is examined. 37 combinations are exhaustively searched in all. The corresponding performance trajectory is plotted in Figure 5.6 which is also obtained by applying the heuristic I. As observed, the heuristic identifies the best data sets well.

The data points on the plots are separated into two groups. This is because whether selected
Table 5.9: Best combination under limited SPM capacity in fine granularity level experiments (Controlled-Allocation)

<table>
<thead>
<tr>
<th>SPM Capacity</th>
<th>Best Combination</th>
<th>Heuristic I</th>
<th>Heuristic II</th>
</tr>
</thead>
<tbody>
<tr>
<td>512B</td>
<td>D_A ∪ A_A ∪ R_S</td>
<td>D_A ∪ A_A ∪ R_S</td>
<td>D_A ∪ A_A ∪ R_S</td>
</tr>
<tr>
<td>1KB</td>
<td>D_A ∪ T_S ∪ F_A ∪ R_S</td>
<td>D_A ∪ T_S ∪ F_A ∪ R_S</td>
<td>D_A ∪ T_S ∪ F_A ∪ R_S</td>
</tr>
<tr>
<td>4,544B</td>
<td>R_S ∪ R_A</td>
<td>D_S ∪ D_A</td>
<td>D_S ∪ R_S</td>
</tr>
</tbody>
</table>

Data sets include the highest priority data sets for SPM allocation (here automatics of Dijkstra, D_A) or not makes significant difference in the cache miss reduction. The upper group of grey points do not allocate D_A into SPM while the lower group do that so those allocations take advantage of SPM further as the heuristic method guides.

In total 11 combinations are examined for 512B SPM capacity, and 37 combinations for 1KB SPM capacity. In case of 4,544B SPM, 22 combinations are measured among 280 combinations in total. The 22 combinations are selected from the topmost 15 in terms of the estimated cache contention degree by the heuristic II and corresponding SPM usage. It is assumed that those data set combinations are highly likely to include the best combination observed and induced from this study. Hence only 22 combinations are examined to save the experiment effort.

Comparing to the results of the coarse granularity level experiment, better cache performance is achieved with fine-grain allocation. An SPM of only 512B reduces the total cache miss counts to 82.9%, and 1KB SPM results in 79.9% of the original cache miss counts. At the coarse granularity level, the total number of cache misses is reduced only to 91.2% of the original level, even though a 4KB SPM is utilized.

When SPM capacity is limited to 512B and 1KB, the two heuristics identify the same data set combinations as the best combinations respectively and the combinations are exactly the same as the actual measured results from the exhaustive search. The combinations also utilize the specified SPM capacity to maximum. These data set combinations are listed in Table 5.9.

However, when the SPM capacity is set to 4,544B, our scheme does not identify the best combination. 4,544B is the size of Rotate task data set (both statics and automatics) chosen from the prior experiments. The heuristic I chooses D_S ∪ D_A, and the heuristic II selects D_S ∪ R_S while the actual best combination is R_S ∪ R_A. Consequently the total number of cache misses reduces to 58.0% by the heuristic I, and to 59.1% by the heuristic II, which are performed less comparing to the actual best combination (37.3%). This is because the cache miss analysis performed for allocation process estimates only the first order of cache contention degree. Only pairs of data sets are examined to determine interference misses between data sets. This estimation abstracts the complicated interactions and interference between many data sets too simply. The situation may be problematic with relatively large sizes of SPM.
Figure 5.7: Correlation of actual cache performance vs. cache contention degree estimated by heuristic II and SPM usage (Controlled-Allocation)

However, if the SPM capacity is large, there is less reason to use fine granularity level analysis since it requires more effort for analyzing data sets due to increased number of data sets but ignores higher order cache contention.

To address this problem further, we also investigated how well our cache miss analysis estimates cache contention degree between data sets. Correlation between actual data cache performance and the heuristic is studied by assigning ranks to each combination of data sets. For
this the heuristic II is used since the heuristic I identifies just an ordered sequence of data sets but the heuristic II examines all possible combinations under a certain SPM budget. Measured cache miss counts of all combinations are sorted in increasing order, and each combination is ranked starting from 1. Two other ranks are assigned by increasing order of the estimated cache contention degree of a data set combination calculated by the heuristic II, and by decreasing order of SPM usage by the combination. The rank of cache contention degree is plotted in horizontal axis and other two ranks are located in vertical axis. If the heuristic or our estimation of cache contention degree is strongly correlated to actual cache performance, the correlation plot will be a straight line with slope of 1.

The correlation plots for 16KB SPM budget at the coarse granularity case, 512B, 1KB and 4,544B at the fine granularity case are presented in Figure 5.7. Blue squares indicate estimated cache contention degree by the heuristic II and grey dots indicate SPM usage. SPM usage by the memory allocation is less correlated to the cache performance. In the case of 16KB SPM at the coarse granularity level, a strong correlation is observed between our estimation of cache contention degree and actual data cache performance. This accounts for why the heuristics work well for 4KB, 8KB and 16KB SPM capacity at the coarse granularity level. On the other hand, the correlation of 4,544B SPM case at the fine granularity level is less clear. This observation is also reflected by the portion of cache misses appearing on the cache contention degree graph or table to the total number of cache misses. The sum of all cache miss counts on Figure 5.3 takes 71.84% of the total number of cache misses, but the sum on Table 5.3 is only 38.17% of the total. The fine granularity level analysis captures less cache contention than the coarse granularity level.

5.6 Summary

Much research concentrates on tightly bounding data cache misses, or else exploiting SPM in place of a data cache for real-time embedded systems. However, data cache miss analysis requires careful attention since it is sensitive to many factors, as illustrated above. Using only SPM misses the performance synergy attainable when using cache and SPM simultaneously. This study suggests heuristic schemes to improve data cache performance by using SPM to hold troublesome data for the cache. We present a new concept of data cache miss analysis; classifying data cache misses as intrinsic or interference misses. The intrinsic misses depend only on a data set itself, while the interference misses reflect data cache contention between data sets. The proposed schemes identify which data sets are best to allocate into SPM for the sake of improvement of cache performance. However, since the suggested schemes rely on a first order estimation of cache contention degree between the data sets, an appropriate granularity level of data sets is necessary. For large capacity of SPM, coarse granularity level approach
saves effort for the analysis while capturing complicated interference between the data sets effectively. The fine granularity level approach is more beneficial to utilize small size of SPM efficiently, and our first order estimation works well.

We have revealed significant synergy between data cache and SPM by looking into inter-task cache pollution. The tasks with allocated data into SPM benefit from fewer intrinsic misses, while other tasks also benefit due to fewer interference misses from less data cache contention and pollution. Our experiments show the proposed heuristics provide good guidance for choosing these data sets. However, this chapter does not show how the suggested methods directly improve worst-case performance and worst-case analysis. Quantitative and qualitative investigation on these questions is left as future work, although it is anticipated that reduced cache contention can affect the worst-case performance positively. Reducing analysis complexity or static estimation of the cache contention addressed by this study are also left as future work.
Chapter 6

Balancing Scratchpad Memory and Data Cache for Synergy

6.1 Introduction

The exploitation of hybrid memory configuration of SPM and cache memory have been evaluated already through several studies [108, 76, 75]. Chapter 5 also notes that some performance synergy exists when the two fast memory systems work together. However, a comprehensive study about how to configure both types of fast memory has not been undergone yet. Furthermore, many systems (such as IBM Cell processors and ARM processors) provide SPM and data cache together. This study looks into how to balance the capacity of both fast memory systems to escalate the synergy for better memory access performance and less dynamic energy consumption.

For this purpose, this study establishes several threads of work: (1) integer linear programming (ILP) formulas performing a static SPM allocation are developed from previous ILP-based SPM allocation research [8, 9, 23, 25], (2) distributions of data memory access counts over data variables are collected from several test benchmarks and modeled with exponential distributions, (3) simulations are performed to show that the performance benefit obtained by the SPM allocation displays diminishing returns as SPM capacity increases due to the exponentially distributed memory access concentration and (4) various hybrid configurations of SPM and data cache are evaluated with test benchmarks examining four metrics: total memory access latency, total data cache miss counts, total dynamic energy consumption by memory accesses and the energy-access latency products. The first three threads provide us some baseline insights to understand the evaluation results of the last thread.

This study focuses on how to balance the hybrid configurations for the greatest benefit. Hence, design space exploration in order to obtain the best data cache configuration is not
conducted. Rather our simulation and evaluation results demonstrate that the reason why a small SPM can improve the memory access performance significantly, why the diminishing returns phenomenon is exhibited as SPM capacity increases, and finally ways to escalate the synergy by balancing capacity of SPM and data cache. This study also does not directly prove that using the hybrid configuration of SPM and data cache improve timing predictability of the hard real-time embedded systems. However, all those observations qualitatively and implicitly imply that using SPM may improve timing predictability of a data cache by reducing cache accesses and consequently reduce dynamic energy consumption by memory accesses.

The remainder of this chapter is organized as follows. Section 6.2 discusses two types of ILP formulas in matrix form. Section 6.3 models and examines the memory access distribution over data variables of a program and evaluates the performance of the static SPM allocation scheme. The evaluation methods for looking for the balance points of the synergy between SPM and data cache from the various hybrid configurations are explained in 6.4. The detailed evaluation of the hybrid configuration is described in 6.5. Section 6.6 summarizes this chapter.

6.2 ILP-based SPM Allocation

Much research relies on the ILP formula to obtain optimized memory allocation under limited capacity of SPM and other types of memory units [8, 9, 95, 94, 10, 107, 109, 108, 96, 19, 16, 23, 25, 22]. The 0-1 Knapsack Problem [43] models this optimization process over heterogeneous memory units including SPM. Total memory access latency or total energy consumption by a certain memory allocation are modeled as a linear sum of binary variables (indicating allocation to a particular memory unit) and several system parameters, such as access counts and size of program components, heterogeneous memory access latency of the memory units and so on. Finally, the solution (i.e. optimized memory allocation) is obtained by solving the formula with ILP.

6.2.1 Basic ILP Formula

Let a target system have $n$ program components like data variables, and $u$ memory units. Two $1 \times n$ matrices $R$ and $W$ contain read access counts and write access counts of the $n$ program components, respectively. One $n \times n$ matrix $Z$ has each program component’s size as its diagonal elements. Hence an $i$-th component has three parameters: $R_i$ for the read access counts, $W_i$ for the write access counts, and $Z_i$ for the size in memory word unit, as shown below.

$$R = \begin{bmatrix} R_1 & \cdots & R_n \end{bmatrix}$$ (6.1)
Each memory unit in the memory subsystems is defined by specifying its available capacity and access cost. One $u \times 1$ matrix $B$ specifies capacity of each memory unit in memory word unit. Two $u \times 1$ matrices $C_R$ and $C_W$ represent read cost and write cost respectively when accessing one memory word. Thus, a $j$-th memory unit is modeled by three parameters: $B^j$ for the available capacity in memory word unit, $C^j_R$ for the read access cost, and $C^j_W$ for the write access cost.

To build the ILP formula for the 0-1 Knapsack Problem, a binary variable is employed per each program component, which indicates whether the program component is allocated into a particular memory unit or not. Thus, a binary variable $I^j_i$ in a $n \times u$ matrix $I$ has value of 1 when the $i$-th component is allocated into the $j$-th memory unit. Otherwise it has value of 0.

To help the ILP formula building in matrix representation, two constant matrices are also defined. One $u \times 1$ matrix $U$ and $n \times 1$ matrix $V$ have their elements all ones.
Now an objective function is built up. Total memory access cost of the $i$-th program component in the $j$-th memory unit during execution of a program ($MAC^j_i$) can be computed as shown in Equation 6.10. If the program component is not allocated into the memory unit so that the binary variable $I^j_i$ is 0, the right hand side of Equation 6.10 becomes 0 automatically.

$$MAC^j_i = R_i \cdot Z_i \cdot I^j_i \cdot C_R^j + W_i \cdot Z_i \cdot I^j_i \cdot C_W^j$$  \hspace{1cm} (6.10)$$

Total memory access cost by an $i$-th component ($MAC_i$) is obtained by accumulating all $u$ kinds of $MAC^j_i$. Only one component is valid and others are 0 among the $u$ kinds of components, since no program component can be positioned in multiple memory units simultaneously.

$$MAC_i = \sum_{j=1}^{u} MAC^j_i = \sum_{j=1}^{u} \left( R_i \cdot Z_i \cdot I^j_i \cdot C_R^j + W_i \cdot Z_i \cdot I^j_i \cdot C_W^j \right)$$  \hspace{1cm} (6.11)$$

Then, total memory access cost by the program is represented by collecting all these memory accesses cost of the $n$ program components as in Equation 6.12. The last line of Equation 6.12 is a matrix representation.

$$MAC = \sum_{i=1}^{n} \sum_{j=1}^{u} MAC^j_i = \sum_{i=1}^{n} \sum_{j=1}^{u} R_i \cdot Z_i \cdot I^j_i \cdot C_R^j + W_i \cdot Z_i \cdot I^j_i \cdot C_W^j = R \cdot Z \cdot I \cdot C_R + W \cdot Z \cdot I \cdot C_W$$  \hspace{1cm} (6.12)$$

Our goal is to find a particular memory allocation which minimizes the total cost in Equation 6.12. Two constraints come from the conditions: (1) all the $n$ program components must be allocated into only one memory unit exactly once ($C1$) and (2) maximum memory usage of
any memory unit by a particular memory allocation must not exceed the given capacity of the memory unit \((C2)\). These conditions are described in Equation 6.13 and Equation 6.14. In Equation 6.14, the inequality symbol \(\leq\) compares matrices of both sides element by element. Hence the equation means that each element of the left hand side is less than or equal to the corresponding element of the matrix \(B\). A superscript \(T\) means a transposed matrix.

\[
\begin{align*}
C_1 : & \quad I \cdot U = V \\
C_2 : & \quad I^T \cdot (Z \cdot V) \leq B
\end{align*}
\]  

Finally the binary variable matrix \(I\) is obtained, which contains a particular memory allocation by optimizing (i.e., minimizing) the objective function in Equation 6.12 under the two constraints in Equation 6.13 and 6.14. This chapter regards the memory access costs as read and write access latency in terms of clock cycles since total memory access latency of a program matters. Program data variables are also considered as participating program components. However, notice that the cost may account for energy consumption or any other types metrics including even benefit rather than cost per each memory access, which is determined by allocation purpose. Similarly, program code can be the matter of interest.

In the mathematical representation above, it is assumed that the SPM space is allocated to a program component (e.g. data variable) permanently until end of the program’s execution. This assumption does not utilize the limited SPM capacity as efficiently as possible. An improved method accounting for limited lifetimes of program components is presented in Section 6.2.2.

### 6.2.2 ILP Formula Embedding Call Graphs

The previous section presents an ILP objective function and two constraints without considering the lifetimes of the program components. The ILP formula can be extended so that the SPM space can be utilized more efficiently, although this type of SPM allocation is a static allocation. SPM space used by a program component can be reused by other program components after that component’s lifetime ends. For example, when automatic data variables of a certain procedure are invalidated by a procedure call return, the memory space occupied by the data may be yielded for other data variables which are about to be instantiated.

Consideration of limited lifetime have been introduced already in prior research work [76, 75, 8, 9, 109, 23, 25, 97]. Several works considered call graphs to distinguish lifetimes of automatic type data variables and procedure code [8, 9, 72]. A live range analysis scheme widely used in compilers were also accepted in other researchers work [109, 51]. Ghattas et al. [23, 25] considered exclusive lifetime of task stack frames by using the task preemption graph and call graphs, while Suhendra et al. [97] considered exclusive residing time of code since they focused
on code allocation into SPM. The formula here considers only a call graph of a single program so that it simplifies the complex lifetime analysis for a single threaded system.

The objective function in Equation 6.12 does not change since it states total memory access latency by a particular memory allocation and it is nothing with lifetime of the SPM contents. The first constraint $C_1$ also remains unchanged. However the second constraint $C_2$ in Equation 6.14 must change since the maximum SPM usage depends on the changes in the lifetime of the SPM contents.

To rebuild the constraint $C_2$, one more matrix which contains call graph information of procedures is employed. One $g \times n$ matrix $G$ represents possible $g$ kinds of procedure call paths of $n$ program components in a binary format. Each row of the matrix represents one of the call paths. Each binary element of the row indicates whether a corresponding program component is instantiated in memory systems or not. Hence each program component is assigned to a column index and each row represents which components are instantiated until reaching leaf nodes in a call graph.

\[
G = \begin{bmatrix}
0/1 & \cdots & 0/1 \\
\vdots & \ddots & \vdots \\
0/1 & \cdots & 0/1
\end{bmatrix} \quad (6.15)
\]

Figure 6.1 shows an example call graph (SHA from MiBench [30]). Seven procedures are indexed with a sequential number inside of an ellipse. This example call graph has five call paths reaching the leaf nodes (init, transform and reverse). Hence while this program is running, the possible number of memory foot prints is also five. Then the matrix $G_{eg}$ shown below represents the possible call paths in the binary format. This representation was implemented in Ghattas et al.’s work [23, 25] but not mentioned. The preemption graphs also can be represented in this method, which is a coarse granularity representation of the lifetime of data variables.

\[
G_{eg} = \begin{bmatrix}
1 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 1
\end{bmatrix} \quad (6.16)
\]

Then, maximum memory usage of each call path must be less than the given memory capacity too. This can be represented as like Equation 6.17. Memory capacity matrix $B$ also must be extended to a $g \times u$ matrix $B'$ in Equation 6.18, where $B^T$ means transposed matrix of $B$. 

83
In case of components having infinite lifetime, their representation in the call graph matrix will be always 1. For example, main function has the same lifespan of the program and their binary values in all possible paths are always 1. Similarly static global type data variables also can be represented in this way since they also occupy assigned memory space until end of the program.

### 6.3 Memory Access Distribution and SPM Performance

Cache memory shows diminishing benefits in terms of miss rates or miss counts as more capacity is made available [34]. A similar phenomenon is also observed in the previous chapters (in
Data Variables

0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5

Fraction of Memory Access Count
(Sorted in decreasing order of memory access count)

ADPCM
BlowFish
HelixMP3
Rotate
TinyJPEG

Figure 6.2: Captured memory access distributions

Chapter 4 and Chapter 5), where a small SPM improved memory access performance dramatically but further increasing SPM size only helped marginally. This section models and repeats this observation by a simulation.

6.3.1 Memory Access Concentration

Although a program has many data variables, access counts of those variables are not uniform, i.e., some variables such as loop index variables or arrays used in nested loops are more frequently accessed. Figure 6.2 plots the fraction of total memory access count by each data variable such as static type and automatic type variables of five test programs listed in Table 6.2. The memory traces of the test programs are captured by an ARM simulator [50] and the total memory access count by each data variable is sorted in decreasing order. Consequently $R_1 + W_1$ is the largest amount of memory traffic by a certain data variable, and $R_n + W_n$ is the smallest amount of memory traffic by another data variable among $n$ data variables. All automatic type data variables of a procedure are collected into one representative variable. The envelopes of the distribution in the figure show that a very large portion of memory access traffic is generated by only a small number of data variables even though the envelopes due to characteristics such as program logic and input workloads.

One metric is developed to measure the memory access concentration to small portion of data variables. The metric accounts for how many data variables ($\tau$) among $n$ data variables of
Table 6.1: Memory access distributions used in simulation

<table>
<thead>
<tr>
<th>Value of $\mu$</th>
<th>0.9-$\tau$</th>
<th>0.8-$\tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>35</td>
<td>25</td>
</tr>
</tbody>
</table>

a program occupy a particular portion ($\pi$) of the total memory access counts, which is named to $\pi$-$\tau$ metric. The concentration portion of the total memory traffic is specified by $\pi$ where $0 \leq \pi \leq 1$. The number of the data variables generating $\pi$ of the total memory traffic is specified by $\tau$ where $1 \leq \tau \leq n$. This can be represented in an equation format like Equation 6.19. Similarly Equation 6.20 also represents the same measure.

$$\tau = \left\{ \min (k) \mid k \sum_{i=1}^{k} (R_i + W_i) \geq \pi \cdot \sum_{i=1}^{n} (R_i + W_i) \right\}$$  \hspace{1cm} (6.19)$$

$$\tau = \left\{ \max (k) \mid k \sum_{i=1}^{k} (R_i + W_i) \leq \pi \cdot \sum_{i=1}^{n} (R_i + W_i) \right\}$$  \hspace{1cm} (6.20)$$

The $\pi$-$\tau$ metrics of the five test programs are calculated in Table 6.2. The last two columns of Table 6.2 list how many data variables cause 90% of and 80% of the total memory accesses. Their portion varies from 13.8% to 23.8% for the 90% memory access concentration (0.9-$\tau$), and from 9.8% to 19.0% for the 80% memory access concentration (0.8-$\tau$). Obviously most of data accesses are generated by less than a quarter of the data variables.

### 6.3.2 Performance of ILP-based SPM Allocation

In order to simulate performance of a data allocation based on the ILP formula, several memory access distributions of a program are modeled by the exponential distribution $\left(f (x) = \frac{1}{\mu} e^{-\frac{1}{\mu} x}\right)$. The observation in Figure 6.2 decides the exponential distribution to be selected. However, fitting the real memory access distribution for the exact curves is not discussed here since necessary information for the simulation is an analytical distribution model which mimics the actual distribution curves similarly. Thus, as $\mu$ increases, memory accesses are dispersed across
Figure 6.3: Diminishing returns of memory access performance by ILP-based SPM allocation

more data variables, and vice versa.

Figure 6.3a depicts the distribution of a constant amount of memory access counts (1 million) over 100 data variables (n = 100) with 8 different values of µ from 5 to 15. The µ values are selected in order that 0.9-τ and 0.8-τ metrics of each the memory access distribution range from

Figure 6.3b shows...
12% to 35% and from 9% to 25% respectively. These values are close to the real values of the test programs ranging from 13.8% to 23.8% for 0.9-\(\tau\) and from 9.8% to 19.0% for the 0.8-\(\tau\) in the previous section. The memory access concentration metrics of the 8 different distributions are listed in Table 6.1. To drop data variables whose access counts are 0, the minimum access count of a data variable is set to 1.

Data variable sizes are randomly distributed from 1 byte to 1024 bytes. In total, 30 kinds of different sizes are synthesized per each data variable. Hence, all the 30 kinds of the data variable sizes are simulated per each value of \(\mu\). The mean value of the set of the data variable sizes is 18,477 bytes. Assigned SPM capacity increases from 1KB to 16KB considering to the average total data size (18,477 bytes).

SPM access latency is modeled as 1 clock cycle while main memory is as 10 clock cycles. The allocation process is performed through the ILP formula introduced in section 6.2.1 since lifetime of the data variables are not synthesized in this simulation. Figure 6.3b plots the average of the objective function values of the 30 kinds of data variable sizes per each \(\mu\) value, which corresponds to average of total memory access latency by the ILP-based SPM allocation scheme. All values of the average total memory access latency are normalized to a case where no SPM space is available, i.e., the one million of memory access to the 100 data variables are managed by the main memory whose access latency is 10 clock cycles.

To summarize, the constant 1 million memory access counts in total are distributed over the 100 data variables with the 8 different \(\mu\) values, and 30 kinds of data variable sizes are assigned to each data variable. The data variable size varies from 1 byte to 1024 bytes. The SPM capacity increases from 1KB to 16KB as the power of the two.

Two behaviors are observed from the simulation results. First, as memory access is dispersed across more data variables, less performance is achieved by the SPM allocation. For example, at 2KB SPM capacity, the normalized total memory access latency of the distribution with \(\mu = 5\) is 0.40 while that value of the \(\mu = 15\) distribution is 0.81. Second, more highly concentrated memory access distribution shows faster diminishing returns as SPM capacity increases. The total access latency of the \(\mu = 5\) distribution falls to 0.74 at 1KB SPM capacity and to 0.40 at 2KB SPM capacity. That value of the \(\mu = 15\) distribution drops to 0.96 at 1KB SPM capacity and to 0.81 at 2KB SPM capacity, but to 0.49 at 4KB and to 0.21 at 8KB.

The shapes of the memory access concentration explain these behaviors. With more highly concentrated memory access distributions, i.e., with smaller values of \(\mu\), SPM can cover more data accesses. On the contrary, with relatively sparse concentrations, i.e., with larger values of \(\mu\), many data variables lose a chance to utilize SPM space even though they still have larger access counts than other variables. These simulation results also explain why only a small size of SPM can improve the memory access performance dramatically. Thanks to the concentration of the memory accesses, even a small SPM can manage the data variables generating the majority
From the discussion above, one lemma is obtained that assigning small size of SPM can improve the memory access performance further including the data cache performance when only limited die area is available for data cache and SPM. If the synergy between SPM and data cache is exploited actively, the small SPM covers the major data variables statically while the data cache manages other data variables dynamically. The synergy is already discussed in the previous chapter (Chapter 5). The data being allocated into SPM will not be cached but be serviced just as fast. Of course there is no performance degradation for those data. In turn, this allocation increases effective capacity of the data cache, so that other data remaining cacheable may take advantage of data cache further.

This viewpoint can be addressed by a qualitative analysis as plotted in Figure 6.4. The horizontal axis represents the capacity of the fast memory systems and the vertical axis shows the total memory access latency or miss rate (in case of data cache) which is strongly correlated to total memory access latency through data cache. In the figure, the sum of SPM capacity and data cache capacity is set at $T$. As observed above, performance benefit (benefit in the total memory access latency) diminishes as more SPM capacity is available. Data cache performance also follows this trend [34]. The data cache curve is flipped over horizontally in the plot. When a hybrid of SPM and data cache is added to our system, the actual memory access performance will follow the sum of the two curves as the top line in the figure. Of course, actual performance curves are likely to be asymmetric but at least one thing can be inferred; there is a balanced performance point. Hence remaining question is how to achieve the balanced performance between SPM and data cache. The next section reveals some insights on this question via analysis and simulation.
Table 6.2: Test programs used for evaluation and experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Memory Usage (Bytes)</th>
<th>Number of</th>
<th>0.9-τ</th>
<th>0.8-τ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statics</td>
<td>Dynamics</td>
<td>Automatics</td>
<td>Variables</td>
</tr>
<tr>
<td>ADPCM</td>
<td>25,016</td>
<td>0</td>
<td>332</td>
<td>41</td>
</tr>
<tr>
<td>BlowFish</td>
<td>939,644</td>
<td>0</td>
<td>4,484</td>
<td>13</td>
</tr>
<tr>
<td>HelixMP3</td>
<td>513,528</td>
<td>23,604</td>
<td>916</td>
<td>60</td>
</tr>
<tr>
<td>Rotate</td>
<td>20,484</td>
<td>0</td>
<td>444</td>
<td>21</td>
</tr>
<tr>
<td>TinyJPEG</td>
<td>17,000</td>
<td>51,980</td>
<td>1,580</td>
<td>58</td>
</tr>
</tbody>
</table>

6.4 Experiments

The experiments of this study conduct memory trace analysis and data cache simulations because there does not exist a practical way to monitor various hybrid configuration of SPM and data cache simultaneously. Memory traces are captured from several test programs and evaluation metrics are extracted from the memory trace analysis and the data cache simulations. The evaluation metrics are total memory access latency, total data cache miss counts, total dynamic energy consumption and the energy-access latency product. All these experiments evaluate the balancing point of the synergy between SPM and data cache.

6.4.1 Target System

Several test programs are selected from MiBench [30] and some open source domains. The test benchmarks are listed in Table 6.2. One input workload is fed into each of the programs respectively. The static memory usage in the table is much larger than other types of data since large input buffers to feed the input workloads and output buffers to capture processed outputs are also accounted into the static memory usage.

The capacity of our SPM increases from 128 bytes up to 8KB by powers of two. Access latencies of SPM and data cache are assumed to be one clock cycle. The main memory capacity is set to 8MB which is large enough to contain all data variables and its access latency is 10 clock cycles. The dynamic energy consumption for a memory access from each of the memory units (SPM, data cache and main memory) is extracted from CACTI [117] for different sizes. Due to implementation limitations of the CACTI energy modeling, the smallest capacity of data cache is 1KB while the maximum capacity is set to that of SPM (8KB). The memory accesses latency and the dynamic energy consumptions per access of all the memory units are presented in Table 6.3.

The memory traces are dumped by an ARM processor simulator named FaCSim [50]. A
Table 6.3: Memory parameters used for evaluation

<table>
<thead>
<tr>
<th>Type</th>
<th>Capacity (Bytes)</th>
<th>Access Latency (Clock cycles)</th>
<th>Energy Consumption (pJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPM</td>
<td>128</td>
<td></td>
<td>0.942</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td></td>
<td>0.942</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td></td>
<td>1.177</td>
</tr>
<tr>
<td></td>
<td>1K</td>
<td>1</td>
<td>2.433</td>
</tr>
<tr>
<td></td>
<td>2K</td>
<td></td>
<td>2.231</td>
</tr>
<tr>
<td></td>
<td>4K</td>
<td></td>
<td>3.575</td>
</tr>
<tr>
<td></td>
<td>8K</td>
<td></td>
<td>5.416</td>
</tr>
<tr>
<td>Data Cache</td>
<td>1K</td>
<td>1</td>
<td>29.244</td>
</tr>
<tr>
<td></td>
<td>2K</td>
<td></td>
<td>29.244</td>
</tr>
<tr>
<td></td>
<td>4K</td>
<td></td>
<td>34.907</td>
</tr>
<tr>
<td></td>
<td>8K</td>
<td></td>
<td>45.803</td>
</tr>
<tr>
<td>Main Memory</td>
<td>8M</td>
<td>10</td>
<td>14.449</td>
</tr>
</tbody>
</table>

1 The energy consumption is estimated by CACTI 5.3 with settings: 45nm technology node, ITRS-LSTP transistor type, 350K temperature, 1 read/write port, 1 memory bank and 32-bit data bus.

data cache simulator analyzes the data cache performance from the dumped memory traces. Our data cache has 32 bytes cache line with LRU replacement policy, write-back policy and no-allocation policy on write miss. One first level data cache is considered only. Analysis of the dumped memory traces is performed by a custom memory trace analyzer which can accumulate the memory access latency as well as the dynamic energy consumption by each line of the memory trace.

6.4.2 Experiment Setup

A notation in format of $S+D$ indicates a hybrid memory configuration whose SPM capacity is $S$ and the first level data cache capacity is $D$ respectively. A memory configuration (0+8K) where only 8KB data cache is available is regarded as our baseline memory configuration called Default. SPM allocation is performed for each SPM capacity shown in Table 6.3 via the ILP formula described in Section 6.2.2. For this, call graphs of the programs are captured as a priori. All automatic data variables of a procedure are collectively regarded as one large variable since we conduct the experiments at a granularity level of procedure.

Several hybrid memory configurations are examined. One group of the configurations
(0+8K, 128+8K, 256+8K, 512+8K, 1K+8K and 2K+8K) fixes the data cache capacity to the maximum (8KB) while the SPM capacity increases from 0 byte up to 2KB. Another group (8K+0, 8K+1K and 8K+2K) alters the capacity to 0 byte, 1KB and 2KB since the minimum data cache capacity which CACTI can model is 1KB. One configuration having evenly partitioned capacity (4K+4K) is also examined.

To implement the obtained SPM allocation, linker scripts are used for static type data variables. For automatic type data variables, the actual access addresses to stack frames appeared on the memory traces are replaced with some particular addresses of the SPM. Dynamic type data variables are not considered in this study.

Four evaluation metrics are measured: the total memory access latency, the total cache miss counts, the total dynamic energy consumption, and the energy-access latency product. As the first step of the experiment, a memory trace of a test program is dumped by the ARM simulator. The metrics for Default case are gathered originating from the data cache simulation. The total access latency, the total dynamic energy consumption and the energy-access latency products are calculated from the cache simulation history while the total cache miss counts are obtained directly from the simulation. In cases of other memory configurations, the memory address translation is performed after obtaining the original memory traces. Then, the translated traces are fed into the cache simulator, and the analysis on the cache simulation history measure the metrics. All the numbers collected from these steps are normalized to the numbers of Default configuration.

### 6.5 Results

This section evaluates all the hybrid memory configurations with the analysis results in sense of the four metrics; the total memory access latency, the total data cache miss counts, the total energy consumption, and the energy-access latency product.

#### 6.5.1 Memory Access Performance

Before discussing the memory access performance such as the memory access latency and the data cache miss counts, some curves are introduced first in Figure 6.5, which capture the portions of the SPM access counts to the first level memory access counts. The first level memory access counts are one of three numbers: (1) sum of the SPM access counts and the main memory access counts when only the SPM is available, (2) sum of the SPM access counts and the data cache access counts when both the SPM and the data cache are available or (3) the data cache access counts when only the data cache is available. Thus this metric lets us know what portion of memory accesses are managed by the SPM. Notice that y-axis (the fraction of
SPM access) is represented upside down for convenient comparison with other curves explained soon.

The memory access distribution affects shapes of the SPM access fraction in the figure. Although more SPM capacity becomes available, the portion of the SPM access increases marginally since our SPM allocation scheme selects the most frequently accessed data for the SPM allocation. Granularity of the data variables and quantized SPM capacity also determine the curve shapes. Data variables of BlowFish and Rotate are coarser than other programs as seen in Table 6.2. One large buffer (4368 bytes) of BlowFish is allocated into the SPM when 8KB SPM capacity is available, which occupies 20.6% of the total memory access counts. Similarly one large buffer (4096 bytes) of Rotate takes advantage of SPM at 8KB SPM capacity, which occupies 7.7% of the total memory access counts. HelixMP3 and TinyJPEG contain dynamic type data variables whose portions are 11.4% for HelixMP3 and 16.8% for TinyJPEG of their total memory access counts respectively. The dynamic type data are not managed by our SPM allocation scheme. Thus this limits benefit from the SPM in many respects. All these factors saturate the SPM allocation around a certain configuration. Some particular memory configurations saturating the allocation are specified in Table 6.4.

The memory access distribution mentioned in the previous sections explains why more marginal benefit is returned as the SPM capacity increases. The increased SPM capacity can cover less frequently accessed data which diminishes contribution by the incremental SPM capacity to the memory access performance. The simulation results in Figure 6.3 already
unveiled this phenomenon under a SPM-only memory configuration with no data cache. The same behavior is observed too with the real programs running on the hybrid configurations of SPM and data cache. Figure 6.6 plots the normalized total memory access latencies and the normalized total data cache miss counts with a fixed capacity of data cache (8KB) as the SPM capacity increases from 0 byte to 2KB.

In the figure, only 128 bytes SPM reduces the total cache miss counts to 72.8% for ADPCM, 77.7% for BlowFish and 63.7% for Rotate of their original counts (Default configuration, 0+8KB) respectively. It is interesting that consequently Rotate can reduce its total memory access latency by 7.9% by using only 128 bytes of SPM. But, the total cache miss counts of HelixMP3 and TinyJPEG fall to 94.0% and to 93.6% respectively at the same SPM capacity. This is explained by Figure 6.5 where fraction of the SPM access of HelixMP3 is only 29.9% at the 128 bytes SPM. This portion of HelixMP3 increases gradually as the SPM capacity in-
creases but saturates since the portion of the dynamic memory access is 11.4% as stated above. In case of TinyJPEG, its SPM allocation is saturated early and its portion of the dynamic memory access is 16.8%.

The diminishing return behavior is also clearly demonstrated on the memory access latency curves. But the behavior is likely to be blurred since the actual memory access latency is affected by data cache miss rates which improves further with more SPM capacity. For example, data cache miss rate of ADPCM at the 8KB data cache is extremely low as revealed in Table 6.4. Hence, the actual improvement on the memory access performance by SPM is very small as shown in Figure 6.6a while the data cache performance curve in Figure 6.6b displays manifest diminishing returns. Although the degree of the diminishing profits and corresponding performance benefits vary from characteristics of the test programs, the diminishing return phenomenon itself is observed clearly.

Figure 6.7a extends the two memory access performance curves to all the memory configurations evaluated. Note that the left half of the plot increases SPM size with a fixed 8K cache size, while the right half decreases cache size. In these curves, the memory access performance worsens as the data cache capacity decreases beyond the 2K+8K points. This behavior is prominently observed with HelixMP3, Rotate and TinyJPEG. Beyond the 2K+8K points, the total cache miss counts and the total memory access latency of the three benchmarks rise up steeply although a large SPM is provided. In case of Rotate, this tendency stops at the 8K+2K configuration since the one large buffer can be allocated to SPM as mentioned above. Two other benchmarks (ADPCM and BlowFish) show almost constant performance in the configurations having the small size data cache since their SPM allocations are already saturated as demonstrated in Figure 6.5.

This abruptly worsening memory access performance originates from worse performance of smaller data cache capacity. The increasing cache miss penalty due to the smaller data cache capacity degrades the memory access performance as discussed in the Section 6.3.2 with Figure 6.4. HelixMP3 and TinyJPEG are more sensitive to the decrease of data cache capacity since they have data variables (dynamic type variables) not covered by the SPM allocation scheme. From this observation, one insight can be established; yielding die area of data cache for SPM does not seem as an ever-preferable solution in sense of memory access performance since data cache miss will increases correspondingly.

6.5.2 Dynamic Energy Consumption

The total dynamic energy consumption is estimated in Figure 6.8 as determined through CACTI modeling. The envelopes of the energy consumption curves provide additional insight into the system behavior.
The first observation is that remarkable amounts of dynamic energy are saved when using SPM. The energy consumption reduces to 2.6% for ADPCM, 12.1% for BlowFish, 17.7% for HelixMP3, 12.5% for Rotate and 15.2% for TinyJPEG of the original dynamic energy consump-
Figure 6.8: Dynamic energy consumption of various hybrid configurations from estimation respectively at each of the best configuration. The best configurations are shown in Table 6.6.

The second observation is that the energy consumption is directly affected by the amount of SPM accesses. The curve envelopes in Figure 6.8 follow the fraction of SPM access curves in Figure 6.5. As more memory accesses are managed by the SPM, less dynamic energy is consumed overall because the SPM consumes less energy than the data cache does as shown in Table 6.3.

However, when the data cache size is fixed at 8KB, the energy consumption increases slightly as the SPM capacity increases. In this process, the data cache performance improves as well as the memory access performance so that it can be expected that less energy is consumed. But the larger SPM capacity enlarges SPM circuit size, which directly increases the dynamic energy consumption. Finally overall more energy consumption is observed with a larger SPM. In the figure, curves of BlowFish, HelixMP3, Rotate and TinyJPEG show this behavior in windows mainly from 128+8K to 2K+8K.

Therefore, the energy consumption curves in the figure are not determined by a single factor. This is because the energy consumption curves demonstrate measure of the total dynamic energy consumption ultimately including the energy-performance tradeoff as well as the SPM-data cache tradeoff we are discussing now. Overall using SPM consumes less energy than using data cache. As more die area is assigned for SPM, more energy is consumed by an access to the SPM due to increasing capacity inside of the SPM. In the meantime, less energy is consumed
Table 6.5: Hybrid configurations showing best memory access performance

<table>
<thead>
<tr>
<th>Program</th>
<th>Memory Access Latency</th>
<th>Data Cache Miss Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADPCM</td>
<td>1K+8K, ...1</td>
<td>1K+8K, ...1</td>
</tr>
<tr>
<td>BlowFish</td>
<td>8K+2K, 8K+1K</td>
<td>8K+2K, 8K+1K</td>
</tr>
<tr>
<td>HelixMP3</td>
<td>1K+8K, 2K+8K</td>
<td>1K+8K, 2K+8K</td>
</tr>
<tr>
<td>Rotate</td>
<td>8K+2K, 8K+1K</td>
<td>8K+2K, 8K+1K</td>
</tr>
<tr>
<td>TinyJPEG</td>
<td>2K+8K</td>
<td>2K+8K</td>
</tr>
</tbody>
</table>

1 All 1K+8K, 2K+8K, 4K+4K, 8K+2K and 8K+1K configuration of ADPCM show the same result due to SPM allocation saturation at 512 bytes SPM.

Table 6.6: Hybrid configurations showing best energy saving and energy-access latency product

<table>
<thead>
<tr>
<th>Program</th>
<th>Dynamic Energy Consumption</th>
<th>Energy-Access Latency Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADPCM</td>
<td>512+8K</td>
<td>512+8K</td>
</tr>
<tr>
<td>BlowFish</td>
<td>8K+0</td>
<td>8K+2K, 8K+1K</td>
</tr>
<tr>
<td>HelixMP3</td>
<td>8K+0</td>
<td>512+8K</td>
</tr>
<tr>
<td>Rotate</td>
<td>8K+0</td>
<td>8K+2K, 8K+1K</td>
</tr>
<tr>
<td>TinyJPEG</td>
<td>8K+0</td>
<td>4K+4K</td>
</tr>
</tbody>
</table>

by data cache access thanks to improved data cache performance. The plots in Figure 6.8 show the impact of this complicated and interactive tradeoff process. However, all these correlated processes are directly affected by how efficiently SPM is exploited.

6.5.3 Comprehensive Perspective Discussion

Figure 6.9 depicts the products of the total dynamic energy consumption and the total memory access latency of all the hybrid configurations experimented. As expected, the curves in the figure follow the superposition of the dynamic energy consumption and the memory access latency demonstrated above. This implies that an optimal memory configuration of the test programs can be attained from a comprehensive perspective considering the related factors all together, not solely one factor (e.g. memory access performance or dynamic energy consumption).

The hybrid configurations which show the best performance are listed in Table 6.6 and Table 6.5. In the tables, the optimal configurations for the memory access performance differ from those for energy saving. With all the benchmarks except ADPCM, the SPM-only configuration (8K+0) consumes the least energy thanks to the energy saving by the SPM, but memory access
Figure 6.9: Energy-access latency product of various hybrid configurations from estimation

performance varies depending on characteristics of the programs as discussed previously. In case of ADPCM, its small memory footprint makes two similar configurations (512+8K and 1K+8K) the best configurations for the different metrics because the SPM allocation of ADPCM is saturated around 512 bytes SPM while covering most of memory accesses as explained above.

In spite of the complexity of determining detailed behavior of the benchmarks in the figure, one parameter can help obtain the balanced synergy from SPM and data cache. The degree of SPM utilization represented in Figure 6.5 determines how to achieve balanced performance. In other words, the test benchmarks can be categorized into two groups by the fraction of SPM access. At the 8KB SPM capacity of Figure 6.5, most data accesses of ADPCM, BlowFish and Rotate (more than 95% of the total first level memory access) are managed by the SPM while 66.3% and 81.3% are managed by the SPM for HelixMP3 and TinyJPEG respectively.

When the SPM allocation scheme manages most memory accesses as in the cases of ADPCM, BlowFish and Rotate, a large SPM handles the memory accesses best. This also saves the dynamic energy consumption by using SPM. In addition, a small data cache is enough to improve the memory access performance further by caching other remaining data accesses although they do not make up a large portion of the total memory accesses. Hence, this case takes advantage of fast but energy-beneficial and timing-predictable SPM actively at the same time that the data cache takes care of the residual data accesses automatically.

On the contrary, when many memory accesses are not handled by SPM as in HelixMP3 and TinyJPEG, a small SPM with a large data cache can work better. In this case, concentrated
memory access to small numbers of data variables are managed by the SPM. Other uncovered or unknown data access are managed by the data cache instead. The concentration of memory accesses to small portion of the data variables enables extracting synergy by using a small SPM, holding data variables which are likely to disrupt data cache states and consequently cause many cache misses.

6.6 Summary

SPM and data cache can improve system performance remarkably by using fast access memory cells. Much research work focused on exploiting either SPM or data cache. Although several works investigated the system performance improvement with the hybrid model, achieving the balanced synergy from both the memory systems is not straightforward since many design factors such as memory access performance and energy consumption are involved.

To approach our goal of using both SPM and data cache effectively, ILP formulas for implementing a static SPM allocation scheme are formulated in matrix form. The memory access distribution of data variables is also investigated and modeled with the exponential distribution. The actual memory access distributions of our test benchmarks also resemble the exponential distribution. By combining these two steps, it is demonstrated that the performance benefit attainable through the SPM allocation exhibits diminishing returns as SPM capacity increases. This explains why a small SPM improves memory access performance efficiently.

As the final step, several hybrid configurations of SPM and data cache are evaluated in terms of the total memory access latency, the total cache miss counts, the total dynamic energy consumption and the energy-access latency product. Our analysis results show that the fraction of SPM access can guide understanding of the metrics. A large SPM and a small data cache perform well for the case where most of memory accesses are managed by the SPM. When significant amounts of unknown or uncovered memory accesses exist, a small SPM and a large data cache are more beneficial. We find that these insights are supported by the observation of the memory access concentration of a small number of data variables in a program. We leave further investigation of these observations in environments using multiple processing units simultaneously, such as multi-cores, graphic processor units (GPU), heterogeneous cores and co-processors, where the memory contention can be more problematic.
Chapter 7

Summary and Future Work

Memory allocation methods for real-time embedded systems need to coordinate fact access latency, less energy consumption and timing predictability at the same time. Cache memory improves memory access performance by providing fast access to particular instructions and data which are likely to be requested by processors. The cache contents are selected by spatial and temporal locality, and this automatic selection induces some level of timing variability due to cache misses. On the other hand, SPM has been studied as a substitute for the cache memory to hurdle the timing predictability problem. Its fast but predictable access latency has inspired much research so that many researchers have developed wise and efficient methods to select the SPM contents. This study explores the data allocation problem in two directions: (1) the hybrid model of SPM and data cache is scrutinized and (2) the memory configuration using only SPM is examined.

In the hybrid fast memory system consisting of SPM and data cache, SPM enhances the data cache without any performance degradation while keeping little timing variability. Data serviced by SPM are not cached but accessed as quickly as by data cache. This SPM allocation increases effective capacity of the data cache and reduce cache contention. Finally, the data cache performance is also improved since residual data can exploit data cache resources further. This study recognizes this interactive benefit as performance synergy between SPM and data cache memory.

As an application of the synergy, we try to alleviate inter-data cache contention between real-time tasks. To identify the most beneficial data sets to allocate into SPM, heuristic methods are proposed with a new cache miss analysis concept. The new cache miss analysis decomposes cache misses into intrinsic misses and interference misses. The proposed SPM allocation heuristics approximate cache contention degree between the data sets to the first order and select the appropriate data sets. Our experiments demonstrate that many cache misses are generated unnecessarily by the cache contention between the data sets, and the proposed methods reduce
the cache contention even with a small size of SPM.

As a next step towards studying the synergy described in this study, various hybrid configurations of SPM and data cache are evaluated by metrics such as total memory access latency, total cache miss counts, total dynamic energy consumption and the energy-access latency product. Prior to this evaluation, the memory access distribution of data variables in a program is analyzed, and it is discovered that the large amount of memory access is generated by small portion of data variables. This memory access concentration results in *diminishing returns as SPM capacity increases*.

Our observations and evaluation results establish some guiding insights to balance the benefits from the hybrid configurations of SPM and cache memory. When a SPM allocation scheme covers most of target memory accesses, the large size of SPM and small data cache benefit more than other configurations. On the other hand, the allocation scheme does not cover some portion of memory access, the small size of SPM and large data cache are more beneficial. While data variables causing large portions of memory access are allocated to SPM first, residual data which are still cacheable cause little timing variability since the portion of their memory access is not significant. In other words, using SPM and data cache together provides better performance with little timing variability. The synergy is escalated at a certain balanced capacity between SPM and data cache, depending on workload characteristics.

Limited capacity of SPM only supports part of data variables. To overcome this constraint, the dynamic management of SPM contents has been widely studied in spite of its large runtime overheads. On the other side, the static management schemes take advantage of small runtime overheads and simple implementation. Thanks to the memory access concentration, the static schemes still demonstrate good performance. This study extends a previously proposed SPM allocation scheme named DARTS to a full implementation under realistic circumstances. In particular, *stack split* is proposed, a viable method to deploy stack frames of procedures across multiple memory units. Our experiments using a real hardware system powered by SPM and data cache demonstrate that the static SPM allocation scheme enhances the system even with a small SPM. This observation is also supported by the guiding insights described above. A memory system using only SPM also benefits the system no matter whether the SPM contents are managed dynamically or statically.

To summarize, SPM can substitute for data cache in terms of memory access performance, energy consumption and timing predictability. Its limited capacity and flexible control mechanism has altered content allocation and management methods. Wise selection of the SPM contents directly determines performance benefits and energy savings. However, even small SPM can contribute to performance improvement significantly since memory access is likely to concentrate to parts of data variables. The small size of SPM makes it able to store those data variables without performance loss. Cache performance is also improved dramatically since the
synergy reduces cache contention between the data variables. The concentrated memory access distribution and the synergy between SPM and data cache explain why the real-time embedded systems can achieve multiple goals, i.e. better memory access performance and timing predictability and less energy consumption from exploiting SPM.

This study leaves a question unanswered: how the SPM allocation in the hybrid of SPM and data cache helps the design of hard real-time embedded systems via worst-case performance analysis. Although it is anticipated that the worst case performance is also improved and that the worst case timing analysis is also accommodated more conveniently by using the hybrid configuration, seeking answers to this question remains for future work. Today’s embedded systems are powered by multiple processing units such as multi-cores, graphic processor units (GPU), multi-processors or co-processors, where memory contention becomes more problematic. In addition to the unanswered question above, an architectural study to devise new perspectives exploiting the virtues of SPM to alleviate memory contention also remains for future work.
REFERENCES


[22] Lovic Gauthier and Tohru Ishihara. Optimal stack frame placement and transfer for energy reduction targeting embedded processors with scratch-pad memories. In Embed-


