



Copyright Undertaking

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

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

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

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

DESIGN AND ANALYSIS FOR EMBEDDED
SYSTEMS TO MEET RESOURCE
CONSTRAINTS

WEI ZHANG

PhD

The Hong Kong Polytechnic University

2021

The Hong Kong Polytechnic University
Department of Computing

Design and Analysis for Embedded Systems to Meet
Resource Constraints

Wei Zhang

A thesis submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

April 2021

CERTIFICATE OF ORIGINALITY

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

_____(Signature)

Wei Zhang _____(Name of Student)

ABSTRACT

Embedded systems are generally subject to resource constraints due to the limited amount of available resources. To operate correctly, efficiently and sustainedly in the presence of resource constraints, the system should be precisely analyzed and elaborately designed. This thesis considers two typical resource constraints, timing constraints and energy constraints, and studies on designing or analyzing embedded software under these constraints.

Tasks in real-time systems have a limited amount of time to use. In order to satisfy the timing constraints, the worst-case execution time (WCET) of real-time tasks should be tightly and safely predicted. In the past decades, the WCET analysis of real-time tasks is well studied under the single-task environment. However, in reality, a task may be interfered with by other tasks, resulting in more cache misses and additional execution time. For instance, higher priority tasks may preempt the lower priority task's execution and evict memory blocks accessed by the lower priority task. Once the lower priority task is resumed, the memory reference that is originally cache hit may miss the cache, and additional time will be costed to reload these additional cache miss memory blocks. The additional execution time and the additional cache miss memory block are called cache-related preemption delay (CRPD) and useful cache block (UCB), respectively. The state-of-the-art method computes the CRPD by counting the number of UCB. However, it can not capture the dynamic behavior of data memory references, e.g., index-dependent array access, as a result, may either overestimate or underestimate the CRPD. To address this problem, this thesis adopts the temporal scope analysis to capture the dynamic behavior of data memory references and integrate it into the existing UCB computation method to tightly and safely bound the number of UCB. Experimentally, our approach conducts a tighter and safe result comparing with the state-of-the-art.

A real-time task may be preempted more than once during its one operating period, and thus the total CRPD caused by all the preemptions should be counted. The state-of-the-art method counts the number of UCB of different preemption points in isolation and sums the k -largest counts as the total number of extra cache misses of k preemptions. However, different preemption points may have the same UCB, and simply summing them may double count some UCB, causing overestimation. To address this problem, we efficiently explore the correlation on UCB computation among different preemption points and introduce a new analysis unit to produce a tighter total UCB. Experiments with benchmark programs show that the proposed technique leads to substantially tighter total CRPD estimation with multiple preemptions comparing with the state-of-the-art.

Similar to preemptions, shared cache contentions between parallel executing tasks may also cause additional cache misses and execution time. The state-of-the-art method considers that all the possible shared cache contentions between parallel executing tasks can happen together. In specific, a memory reference is considered as a shared cache miss as long as its parallel executing tasks access the same shared cache line with it. But, because each task's memory references are accessed in a certain order, a large portion of the shared cache contentions can not happen together in reality. Therefore the state-of-the-art method is pessimistic. To address this problem, this thesis models shared cache contentions between memory references and access orders among memory references together, and proposes an efficient algorithm for calculating the worst-case additional execution time. We conduct the experiments with the MRTC benchmarks. Experiment results show that our method can significantly improve the analysis precision over the state-of-the-art with most benchmark programs without sacrificing the analysis efficiency.

Energy harvesting systems, which harvest energy from the environment to operate, promise to power billions of IoT devices without being restricted by battery life. Since the ambient energy is generally weak and unstable, the system is subject to energy constraints; that is, it may suffer unpredictable and frequent power failures. To finish a task across power

outages, the state-of-the-art method frequently backup the system state, which costs lots of energy and time and adversely reduces the time and energy for the program's execution. This thesis proposes LATICS, a low-overhead adaptive task-based intermittent computing system, which adaptively skips some unnecessary state savings when the energy supply is sufficient. To reduce the backup overhead, we precisely analyze the minimum set of data that should be saved when its following state saving points are dynamically enabled/disabled at run-time rather than saving all the system state. Moreover, we disclose that the state saving point can not be blindly skipped even if the energy is sufficient. It is because skipping such state saving points can lead to more data to backup. In this thesis, we introduce the concept of breaking points to avoid skipping such points. Experimentally, LATICS significantly reduces state saving overhead and improves execution efficiency compared to existing solutions.

State backup which is performed frequently in energy harvesting systems, is considered as the performance bottleneck. Although its overhead can be reduced through some optimizing techniques, it always blocks the system's execution as they are performed in sequential. Performing state saving and system execution in parallel is feasible in modern MCU, e.g., by leverage DMA, but is not adopted by existing systems as it may cause data race and further system error. However, in energy harvesting systems, since the backup is performed frequently, the former incorrect backup can be overwritten soon by the following backups. Moreover, incorrect backup rarely happens as only a part of data may result in incorrect backup errors. Therefore, in this work, instead of avoiding the incorrect backup, we allow the incorrect backup to occur and propose an efficient method to precisely detect the incorrect backup. In addition, we design a fault-tolerant backup management mechanism to guarantee that the system can execute correctly in case of the state is backed up incorrectly. The proposed method is implemented as a run-time system, and experimental results show that the proposed method has a better performance than existing methods.

Keywords: static timing analysis, cache analysis, WCET, CRPD, shared cache contention, energy-harvesting system, parallel state backup, intermittent computing

PUBLICATIONS ARISING FROM THE THESIS

1. **Wei Zhang**, Songran Liu, Mingsong Lv, QiuLin Chen and Nan Guan. "Intermittent Computing with Efficient State Backup by Asynchronous DMA". in *Design Automation and Test in Europe (DATE)*, 2021.
2. Songran Liu, **Wei Zhang**, Mingsong Lv, Qiulin Chen, Nan Guan, "LATICS: A Low-overhead Adaptive Task-based Intermittent Computing System", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2020.
3. **Wei Zhang**, Nan Guan, Lei Ju, Yue Tang, Weichen Liu and Zhiping Jia. "Scope-Aware Useful Cache Block Calculation for Cache-Related Preemption Delay Analysis with Set-Associative Data Caches". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.
4. **Wei Zhang**, Nan Guan, Lei Ju, Weichen Liu. "Analyzing Data Cache Related Preemption Delay with Multiple Preemptions". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Nan Guan, for the continuous support of my Ph.D. study and research. I am very honored to be a student of Dr. Guan, and I want to thank him for his guidance which helps me finally know how to find research problems, fix these issues, write academic papers, and present the research work. The completion of the thesis would not have been possible without his support and help.

I must also thank Prof. Lei Ju at Shandong University, the supervisor of my master thesis. I finished my first academic paper under his professional supervision. The experience made me realize that doing research is such a great lifelong cause. Since that, a seed of being an excellent researcher is planted in the deepest of my heart. I also appreciate a lot of constructive advice and encouragement he gives to me during my Ph.D. study in both research and life.

I want to thank Dr. Mingsong Lyu from the Hong Kong Polytechnic University, for his patient guidance, stimulating discussions, insightful comments, and encouragement. I also express my gratitude to other members of Dr. Guan's research group - Dr. Songran Liu, Dr. Xu Jiang, Dr. Yue Tang, Dr. Tao Yang, Dr. He Du, Qingqiang He, Xuemei Peng, etc.—for their unconditional assistance during my Ph.D. study.

I would also express my gratitude to Dr. Yuanqing Zheng at the Hong Kong Polytechnic University, for kindly serving as the Chairman of the Board of Examiners (BoE). I also thank Prof. Guoliang Xing from the Chinese University of Hong Kong, and Dr. Cong Liu from The University of Texas at Dallas, for kindly taking the time from their busy schedules to serve as the external examiners.

I recognize that this thesis would not have been possible without the financial assistance from the Hong Kong Polytechnic University. I thank Dr. Guan and the Department of Computing for offering me grants to attend international conferences.

Finally, I want to thank my family, especially my parents and my wife. I can not finish my Ph.D. study without their unconditional support and endless love. I would also like to thank my daughter, Suisui. The birth of my daughter makes me stronger and braver, which helps me overcome all the difficulties and motivates me to finish my Ph.D. thesis.

TABLE OF CONTENTS

CERTIFICATE OF ORIGINALITY	iii
ABSTRACT	iv
PUBLICATIONS	vii
ACKNOWLEDGEMENTS	viii
LIST OF FIGURES	xiv
LIST OF TABLES	xvi
CHAPTER 1. INTRODUCTION	1
1.1 Timing Analysis for Real-time System Software	1
1.1.1 Cache-Related Preemption Delay Analysis	2
1.1.2 Share Cache Contention Analysis for WCET Estimation	4
1.2 Designing Efficient Energy Harvesting System	5
1.3 Thesis Organization	6
CHAPTER 2. CACHE-RELATED PREEMPTION DELAY ANALYSIS WITH SET-ASSOCIATIVE CACHES	8
2.1 Introduction	8
2.2 Related Work	10
2.3 Preliminary	12
2.4 Motivation	14
2.4.1 Underestimation at The Basic Block Boundary	14
2.4.2 Overestimation of UCBs for Set-associative Caches	16
2.4.3 Summary of The Observations	19
2.5 New Analysis Boundary	20
2.6 Temporal Scope Analysis	21
2.7 Scope-Aware UCB Analysis	26
2.7.1 Temporal Scope Normalization	27
2.7.2 RCS and LCS Calculation	31

2.7.3	UCB Calculation	36
2.8	Evaluation	38
2.9	Conclusions	43
CHAPTER 3. CACHE-RELATED PREEMPTION DELAY WITH MULTIPLE PRE-EMPTIONS		45
3.1	Introduction	45
3.2	Background	46
3.2.1	Temporal Scope Analysis	46
3.2.2	UCB Analysis	48
3.3	Motivation	49
3.4	Useful Memory Block Analysis	51
3.4.1	A Running Example	52
3.4.2	Analysis Unit	54
3.4.3	Useful Memory Block Classification	55
3.5	Useful Cache Block Analysis for Subsequent Preemptions	59
3.5.1	O-UMB Analysis for Subsequent Preemptions	59
3.5.2	I-UMB Analysis for Subsequent Preemptions	61
3.5.3	UCB Calculation for Subsequent Preemptions	62
3.5.4	Total CRPD Analysis	64
3.5.5	Discussion	67
3.6	Evaluation	68
3.7	Conclusion	72
CHAPTER 4. SHARED CACHE CONTENTION ANALYSIS FOR WCET ESTIMATION ON MULTI-CORES		73
4.1	Introduction	73
4.2	Related Work	74
4.3	Preliminary	76
4.3.1	System Model	76
4.3.2	The State-of-the-Art Method	78
4.4	Motivation	80
4.4.1	Overestimation I	82
4.4.2	Overestimation II	82
4.4.3	Summary and Discussion	83
4.5	Methodology	84

4.5.1	Access Order Analysis	84
4.5.2	Precise Shared Cache Contention Analysis Between URs	87
4.5.3	Precise Shared Cache Contention Analysis Between Tasks	88
4.6	Multi-Level Cache Analysis	94
4.7	Evaluation	95
4.8	Conclusion.....	99
CHAPTER 5.	LATICS: A LOW-OVERHEAD ADAPTIVE TASK-BASED INTER- MITTENT COMPUTING SYSTEM	101
5.1	Introduction	101
5.2	Overview	103
5.3	Analysis	106
5.3.1	Rationale.....	106
5.3.2	Analysis Target and the Definition of Breaking Point	109
5.3.3	Computing PWS and Inserting Breaking Points.....	111
5.4	The Run-Time System	120
5.4.1	The Main Work Flow	120
5.4.2	Core Components	122
5.5	Experiments and Evaluation	124
5.5.1	Experimental Setup	124
5.5.2	Evaluation Methods	125
5.5.3	Empirical results and evaluation	126
5.5.4	Further Discussion.....	130
5.6	Conclusion.....	134
CHAPTER 6.	INTERMITTENT COMPUTING WITH EFFICIENT STATE BACKUP BY ASYNCHRONOUS DMA	135
6.1	Introduction	135
6.2	Related Work	136
6.3	The Parallel State Backup Problem	137
6.4	Overview of Our Approach	138
6.5	Design	141
6.5.1	Software Model	141
6.5.2	State Backup Error Detection	142
6.5.3	Buffer Design for Fault-tolerant Backup Management	146
6.5.4	Reorganizing Memory Layout to Reduce Backup Errors	146

6.6 Experiments and Evaluation	147
6.6.1 Experimental Setup	147
6.6.2 Results and Evaluation	148
6.7 Conclusion.....	150
CHAPTER 7. CONCLUSION AND FUTURE WORK.....	151
7.1 Conclusion.....	151
7.2 Future Work	152
REFERENCES	153

LIST OF FIGURES

2.1	The control flow graph of a program fragment	15
2.2	A running example	17
2.3	Examples of continuous and noncontinuous iteration ranges	23
2.4	Memory blocks accessed by array A at different loop iterations	24
2.5	Temporal scope analysis	25
2.6	Program unrolling according to the normalization	32
2.7	Unrolled control flow graph for the RCS and LCS calculation of $D_2^{L_1[0],L_2[10]}$..	34
2.8	Analysis time of different benchmarks	39
2.9	Maximal UCB number on different benchmarks	42
2.10	Data memory accesses of $Adpcm$	43
2.11	UCB number of different loop points in $Bsort100$	44
3.1	Transformed CFG of a preempted task	50
3.2	A running example	54
3.3	Total number of UCBs of different benchmarks with different numbers of preemptions	70
4.1	A multi-core processor with shared cache	76
4.2	Control flow graph (CFG) of task T	77
4.3	CFGs of two parallel tasks T and T'	80
4.4	Cache states of C_1	82
4.5	$CM_{(P_T^a, P_{T'}^b)}$	89
4.6	A multi-level cache architecture	94
5.1	The WAR problem and breaking point insertion	105
5.2	Deciding breaking points for each leading task	108
5.3	An example to explain PWS computation and breaking points insertion	119
5.4	A running example of the run-time system	121
5.5	Execution time results under random power traces (normalized to InK's total execution time)	127
5.6	The program graph of CEM	128
6.1	An example of the parallel state backup problem	138

6.2	Overview of the proposed state backup approach.....	139
6.3	Fault-tolerant backup management	140
6.4	State backup error detection exemplified (principle and optimization)	143
6.5	Execution times of different benchmarks under different NVM speeds (Normalized to ASY-OPT)	148

LIST OF TABLES

2.1	UCB in different temporal scopes at P_1	19
2.2	Benchmark descriptions and array sizes	41
3.1	Benchmark descriptions and array sizes	69
3.2	Total CRPD of tasks preempted by different preempting tasks	71
4.1	Mapping among memory blocks, cache sets and memory references of T and T'	81
4.2	Reduced estimated WCET in percentage of our method with different benchmark programs	96
4.3	Average analysis time and the number of CEOPs of each benchmark	99
5.1	State saving size for all tasks (in bytes)	127
5.2	Execution time results under periodic power traces with different power cycles	129
5.3	The number of tasks finished in each power cycle	130
5.4	The minimal execution times (in ms) obtained by ES and LATICS under periodic power trace with $1ms$ power cycle	131
5.5	Results with different α values under periodic power trace with $1ms$ power cycle	132
5.6	Time cost (in μs) for different state copying methods under different data sizes (in bytes)	133
6.1	A survey of state backup methods	137
6.2	Average numbers of uncovered incorrect backups (UIB), incorrect backups (IB) and total backups (B)	149

CHAPTER 1

INTRODUCTION

Nowadays, embedded systems exist in almost everything in our lives, e.g., cars, robots, telephones, pacemakers, climate control systems, fire alarm systems, manufacturing systems, etc. Unlike the general-purpose system, embedded systems have limited resources to achieve their objective and therefore are subject to resource constraints. To satisfy the resource constraints, the system should be specifically designed and analyzed.

This thesis considers two typical resource constraints, timing constraints and energy constraints. In real-time systems, tasks should be finished before their deadline. Violating the timing constraints may lead to catastrophic consequences such as loss of human life. A safe and tight bound on the execution time of real-time tasks should be predicted in advance to guarantee the timing constraints can be satisfied. Energy harvesting systems harvest the energy from the ambient environment to operate. Since the ambient energy is weak and unstable, the system may suffer unpredictable and frequent power failures. Conventional software that always reboots from the system entry can not progress under such energy constraints. Hence, a new design of embedded software should be proposed.

1.1 Timing Analysis for Real-time System Software

In order to know whether real-time tasks can be finished before their deadline, a safe (no less than any possible execution time) and tight (as close as possible to the actual worst-case execution time) upper bound on the execution time, also known as the worst-case execution time (WCET), should be computed in advance. Analyzing whether a memory reference is a cache miss or hit is the foremost part of timing analysis, as the timing delay caused by a cache miss could be several orders of magnitude greater than a cache hit.

Static timing analysis has been studied for several decades and works well under the single-task environment [28]. Different tasks may compete to use the same cache line in practice, causing additional cache misses and a long delay in the execution time. To derive a safe and tight WCET bound, we need to tightly and safely bound the worst-case additional execution time caused by inter-task interferences. Existing methods are inaccurate in computing such additional shared cache misses, and thus may either overestimate or underestimate the WCET bound. The overestimated WCET bound may lead to a pessimistic schedulability test result and waste computational resource, while the underestimated WCET bound is unsafe and may cause system failures. This thesis considers two typical inter-task interferences, preemptions and share cache contentions, and proposes methods to tighten the WCET bounds while guaranteeing the soundness of the analysis result.

1.1.1 Cache-Related Preemption Delay Analysis

Preemptive scheduling is widely used in real-time systems to ensure more important or urgent tasks to be executed first. A higher priority task may preempt lower priority tasks, evicts memory blocks accessed by lower priority tasks from the cache, and makes the following accesses to these evicted memory blocks miss the cache. Once the lower priority task is resumed, additional time is spent to reload these additional cache miss memory blocks, and such additional execution time is called cache-related preemption delay (CRPD). It has been shown that CRPD contributes significantly to task execution time [40]. Therefore, a precise CRPD analysis is crucial for real-time system tests and design.

The state-of-the-art method computes CRPD by counting the number of useful cache blocks (UCBs) [4, 9], where a UCB of a program point is defined as a memory block that

1. is reside in the cache when the program reaching the program point,
2. will be re-referenced before it is evicted from the cache when leaving the program point.

If preemption occurs at a program point, the additional number of cache misses will no larger than the number of UCBs. In addition, the state-of-the-art method pointed out that the CRPD of different preemption points in a basic block is the same, and therefore the UCB computation can only be performed for each basic block rather than each instruction.

However, the state-of-the-art UCB computation method can not capture the dynamic behavior of data memory reference and therefore may overestimate the number of UCBs. For instance, an index-dependent array reference may access different memory blocks at different loop iterations, and a memory block is only a UCB of the program point at the loop iteration it is accessed. However, the state-of-the-art method can not distinguish which memory block is accessed at each iteration, and thus consider that all the memory blocks are UCBs at all the iterations, which is over-pessimistic. Besides, performing UCB computation at each basic block boundary may underestimate the number of UCBs. Underestimated WCET bound may lead to catastrophic consequences, which should be avoided. To address the above-mentioned problems, we adopt the scope-aware analysis [31] to capture the dynamic data memory access behavior and integrate it into the UCB computation to compute a safe and tighter bound of CRPD.

A task may be preempted several times (i.e., k) during its one period. The state-of-the-art method [7, 37] calculates the total number of additional cache misses by simply summing the k -largest UCBs. However, since UCBs of different preemption points may have correlations, simply summing the k -largest may double count some UCBs. This thesis first points out the pessimism of existing total CRPD analysis methods. Furthermore, we show that computing the worst-case additional cache misses for multiple preemptions is computationally intractable. According to the locality of memory references, propose an approximate method to efficiently and precisely compute the total CRPD. Experimental results show that, for most benchmarks, the total UCBs produced by our method is 50% less than that of the state-of-the-art method.

1.1.2 Share Cache Contention Analysis for WCET Estimation

Embedded real-time systems are shifting to multi-core platforms to meet both high-performance requirements and strict thermal and power constraints [26, 46]. Multi-core processors, while bringing great performance benefits, significantly complicate the cache behavior analysis. Most multi-core processors contain a shared cache which can be accessed by different cores simultaneously. Different tasks executed in parallel on different cores may contend to use the same shared cache line. We a memory reference is a shared cache hit if only one task is executed on the processor. If another task is executed in parallel and access the same shared cache set with it, its cache behavior may be changed to miss due to shared cache contentions. Therefore, in multi-core systems, the cache behavior of memory references depends not only on the local program's execution flow but also on the shared cache contentions with its co-runner tasks.

The state-of-the-art method [39] considers that all the shared cache contentions between parallel executing tasks can happen together, which is pessimistic. Specifically, a memory reference is regarded as a shared cache miss as long as other parallel-executing tasks access the same shared cache set with it. However, memory references are accessed in a certain order defined by the program control flow graph, and thus a lot of shared cache contentions can not happen together at runtime. We denote the set of shared cache contentions that can happen together as the feasible set. To derive a tight and safe WCET bound, the feasible set that causes the maximum shared cache misses (i.e., the most extra execution time) should be computed. Shared cache contentions between parallel executing tasks can form a massive number of feasible sets, and how to exactly find the objective one is challenged. In this thesis, we introduce a novel model to analyze the shared cache contentions. The new model can capture access orders among different memory references. Based on the new analysis model, we propose an efficient method to compute the maximum number of additional cache misses caused by shared cache contentions, and produce a tighter WCET bound. Experimentally, the proposed method can produce a much tighter WCET bound for most benchmark programs without sacrificing efficiency.

1.2 Designing Efficient Energy Harvesting System

Batteries are inconvenient to use and not environmentally friendly as they generally need to be charged or replaced periodically. Since the number of Internet-of-Things(IoT) devices will be expected to exceed 20 billion by 2025 [3], powering such a huge number of IoT devices with batteries is no longer a practical solution. A new technology, energy harvesting, is considered a promising technique to power a huge number of IoT devices in the future. Energy harvesting technology harvests energy from the ambient environment, e.g., solar, piezoelectric, RFID, wind and ,thermal, thus eliminating restrictions caused by batteries.

Energy harvesting systems operate when the energy is available and turn off when the harvested energy is depleted. However, due to the size constraints of IoT devices and the changeable environment, the harvested energy is weak and unstable, leading to unpredictable and frequent power failures (i.e., even more than a hundred times per second). Traditional embedded software will reboot from the very beginning every time the system is power-on, and therefore cannot progress under such frequent and unpredictable power failures. To provide sustainable service, a new computing paradigm, intermittent computing, is proposed. In intermittent computing, a program is decomposed into a set of program segments. In this thesis, we focus on task-based intermittent computing, in which a program segment is realized as a task. Once the system resumes power failures, the system continues its execution from the beginning of the latest executed task. Incrementally, the program can finish its execution task by task in the presence of such an unstable power supply.

However, to ensure memory consistency across power outages, the system should backup the system state before each task's execution. These state backups consume a lot of energy and time, which adversely impact the system's performance. It has been shown that the state backup can consume up to 90% total execution time [41], which is the performance bottleneck of the energy harvesting system. In this thesis, we propose two methods to reduce the backup overhead.

First, we propose an adaptive task-based intermittent computing system, which can coalesce some tasks when the energy is enough to skip some unnecessary state backup.

Counter-intuitively, we found that some state backups can not be skipped even if the energy is sufficient, as skipping it may lead to much more state to backup. In this thesis, we introduce the breaking point for each task to guide the task coalescing. A breaking point is a task that a compulsive state backup should be performed before its execution. Then, to minimize the overhead of each backup overhead, we propose an algorithm to precisely find the data that need to be backed up for each task rather than backup all the system state. We implement a run-time algorithm to implement the proposed method. Experimental results show that the proposed method significantly reduces state saving overhead and improves execution efficiency compared to existing solutions.

To further reduce the state backup overhead, we propose to perform the system state backup in parallel with the task's execution to hide the latency of the state backup. Most embedded MCUs contain specific hardware that can transfer data without interrupting the CPU, e.g., DMA. But all the existing methods perform the state backup in sequential. This is because performing the state backup in parallel with the task's execution may cause data race, and therefore backup an incorrect variable. All existing methods [23, 70] adopt the sequential backup method to avoid the happening of incorrect backup. In reality, we found that the probability of the happening of an incorrect backup is very small. Therefore, it is a waste to use an inefficient method to prevent a small probability thing. Moreover, an incorrect backup can be overwritten by the next backup soon. Therefore, we allow the backup error to occur by performing state backup in parallel. To avoid a system failure in the presence of incorrect backup, we propose an efficient method to detect backup error and a fault-tolerant backup buffer management method. Experimental results show that the proposed method can efficiently detect state backup errors, and by parallelizing state backup with program execution, system performance is considerably improved.

1.3 Thesis Organization

The rest of this thesis is organized as follows.

- Chapter 2 discloses the underestimation and the overestimation of existing CRPD analysis methods and presents a method to produce a tight and safe CRPD bound for a single preemption.
- Chapter 3 shows the pessimism of existing methods on computing total CRPD for multiple preemptions and presents a method to tight the CRPD bound of multiple preemptions.
- Chapter 4 presents a novel method to precisely compute the additional execution time caused by shared cache contentions between parallel executing tasks.
- In chapter 5, we present an adaptive task-based intermittent computing system to reduce the backup overhead of energy harvesting systems.
- In chapter 6, to further reduce the overhead caused by state backup, we, for the first time, propose to parallel the state backup with the task's execution and design a run-time system to ensure system correctness.

CHAPTER 2

CACHE-RELATED PREEMPTION DELAY ANALYSIS WITH SET-ASSOCIATIVE CACHES

2.1 Introduction

Static timing analysis is crucial for the design of real-time systems. Conventionally, the worst-case execution time (WCET) analysis of a task is performed assuming the task is executing in an uninterrupted environment. However, most real-time tasks may suffer inter-task interference during its execution. For example, in preemptive systems, a lower priority task is allowed to be preempted by higher priority tasks. Preemptive scheduling introduces additional cache misses, because the preempting task may evict some reusable cache blocks of the preempted task. The cost of reloading such additional misses is called cache-related preemption delay (CRPD) [37]. It has been shown that CRPD contributes significantly to task execution time [40]. Therefore, precise CRPD analysis is important to build correct and resource-efficient real-time systems.

Most existing work on CRPD analysis only considers instruction caches. The access behaviors of instruction caches and data caches are very different. The execution of an instruction always access the same memory block, and a memory block can only be accessed by neighboring instructions (if we inline all the functions). By contrast, an instruction may access different data memory blocks at its different executions (e.g., an index-dependent array access), and a data memory block may be visited by instructions distributed in different locations of a program. Existing CRPD analysis techniques fail to capture the dynamic data cache access behaviors, and therefore may either overestimate or underestimate the CRPD. To the best of our knowledge, the only existing work which takes the dynamic behavior of the

data memory references into account is [53], which only considers direct-mapped caches and suffers from strict restrictions on data access patterns (so only applicable to a very limited subset of realistic programs).

In this chapter, we study the CRPD analysis for data caches (called dCRPD analysis for short). CRPD is caused by cache block conflicts between the preempted task and preempting task, so CRPD analysis should be performed with both of them. The problem to be solved regarding the preempted task is how to bound the number of cache blocks that will be reused after a preemption, namely, the *useful cache blocks* (UCBs). On the other hand, the problem concerning a preempting task is how to estimate the number of cache blocks that will be occupied by the preempting task(s), namely, the evicted cache blocks (ECBs) [37]. Similar to many existing work (e.g., [4, 37]), we focus on the calculation of UCBs for the preempted task, which can be easily integrated with state-of-the-art joint UCB-ECB CRPD analysis (e.g., [8]) and CRPD-aware schedulability analysis (e.g., [12, 44]).

Previous work has studied UCB calculation for instruction caches, and claimed that the analysis can also be applied to data caches [6, 34]. However, this will lead to problems in soundness and precision:

- **Soundness.** It has been pointed out that the set of data UCBs may change at the instruction level, instead of at the basic block level for the instruction CRPD analysis [6]. In this chapter, we demonstrate, by means of a concrete counter-example, that the existing work [34, 52], which performs analysis at the block boundaries, underestimates the CRPD when applied to data cache. Further, to solve this problem, we introduce a new unit for dCRPD analysis, *data analysis blocks*, and perform analysis at their boundaries for safe UCB calculation.
- **Precision.** The traditional UCB calculation for instruction caches lacks the capability to predict the dynamic behavior of data accesses. In this chapter, we disclose that, for a preemption point that resides in a loop, its UCBs may be different in different loop iterations. The existing work [6, 8] that counts all the UCBs at different loop iterations may significantly overestimate the dCRPD. To solve this problem, we consider the

temporal scope of each memory block in UCB calculation to precisely predict the dynamic behavior of the data memory references, which significantly improves the analysis precision.

We evaluate our method with benchmark programs containing frequent data memory accesses from the MRTC WCET benchmark [2]. In particular, we compare our method with the state-of-the-art CRPD analysis technique for set-associative caches in [34]. Note that the technique in [34] is designed for instruction caches, and as discussed above, is not sound for data caches. Nevertheless, we use [34] as the bottom line in the comparison to show the precision improvement of our method consisting in counting the temporal scopes of memory blocks and eliminating the memory blocks doomed to be evicted before the next visit from UCB. Experiment results show that for most benchmark programs our method can reduce more than half of the UCBs and thus significantly improve the CRPD analysis precision.

2.2 Related Work

CRPD analysis on the preempted task bounds the number of useful cache blocks (UCBs) for each possible preemption point. Lee et al. modeled the cache state with a set of possible memory blocks that may reside in a cache block at each preemption point for the UCB calculation [37] (the set-based approach). Negi et al. enhanced the accuracy of UCB calculation by incorporating path analysis such that cache states resulting from different program paths are captured (the state-based approach [52]). [63] proposed a CRPD analysis framework that balances the scalability of the set-based approach with the accuracy of the state-based approach. Altmeyer et al. integrated UCB calculation with cache behavior modeling in WCET analysis to deliver a safe and tight bound of preemption costs due to CRPD [4]. Kleinsorge et al. extended the state-based approach to set-associative instruction caches [34]. [9] noted that not every UCB will contribute to CRPD, because the UCBs will not be evicted from the cache by preempting tasks if the preempting tasks do not access enough memory blocks. Hence, with a consideration of the associativity of the cache, Altmeyer et al. tightened the CRPD using the concept of *Resilience*. Among all the above-mentioned techniques, [37, 52, 63] focus

on the direct-mapped cache analysis, and [9, 34] focus on the set-associative cache analysis.

Evicted cache block (ECB) analysis determines the cache blocks that may potentially be occupied by higher-priority preempting tasks. In [65], the *UCB-Union* approach considered that a single preemption may evict the cache blocks of multiple preempted tasks to provide a tighter CRPD analysis. [8] provided a comprehensive analysis framework *ECB-Union* to capture the interleaving between multiple preempting and preempted tasks. In [57], a new cache replacement policy was proposed to reduce the CRPD cost due to cache conflicts. UCB and ECB calculation can be easily integrated to provide more precise CRPD analysis results than applying any of them individually [7].

In addition to the CRPD cost of a single preemption, the number of preemptions must be computed to integrate CRPD analysis with system-level schedulability analysis. Nested and indirect preemptions were considered to bound the total number of preemptions in fixed-priority [7] and EDF scheduling [44]. [54] utilized the tasks' best-case and worst-case execution times to further tighten the number of preemptions. The integration of CRPD analysis and fixed-priority preemption threshold scheduling was studied in [12]. In case of multiple preemptions occurred in one period of preempted tasks, the UCBs of each preemption point may have correlations. Work in [72] detected the correlations and performed an accurate total CRPD analysis for multiple preemptions.

The above-mentioned CRPD analysis techniques mainly focus on instruction caches. Some of them [8, 37] even claimed that they can be extended to data cache analysis. Due to the lack of ability to predict the dynamic behavior of data references, they may suffer from significant overestimation. Various methods have been proposed to analyze data cache behavior, e.g., the cache miss equation (CME) framework [24] and the Presburger Arithmetic formulations [13]. The CME framework has been utilized for data cache behavior modeling [67] and dCRPD analysis [55], which provides precise analysis results for data references with regular patterns. However, the CME framework and the Presburger Arithmetic formulations have inherent limitations in handling programs with input-dependent branches and data references. Furthermore, it is not easy to integrate the CME-based approach with ex-

isting AI-based frameworks for instruction CRPD analysis and CRPD-WCET co-analysis (e.g., [4]).

[62] proposed a framework that categorizes data memory accesses as predictable or unpredictable, where the impact of unpredictable memory accesses on the data cache is bounded for WCET analysis. AI-based data cache analysis frameworks that handle both regular and input-/data- dependent accesses for WCET estimation were proposed in [60] and [31]. In particular, [31] introduced the concept of "temporal scope" to capture the dynamic behavior of data caches for WCET analysis. In this work, we enhance the concept of "temporal scope" in [31] for CRPD analysis.

2.3 Preliminary

As mentioned before, CRPD is caused by the interference on the cache blocks between the higher priority tasks (ECB) and the lower priority tasks (UCB). ECB of a program, corresponding to all the cache blocks that the program may access during its execution, is easy to compute. Compared with the ECB calculation, the UCB calculation is far more complicated. A UCB for a program point is defined as:

Definition 2.3.1 (UCB). *A UCB is a memory block which is in the cache when the program reaches the program point and may be re-referenced in the following execution before being replaced by another memory block.*

For a program point P , UCB_P is a collection of all the UCBs at P . In most of existing work [34, 52], the computation of UCBs of a program preemption point is done through an iterative calculation for reaching cache states (RCS) and living cache states (LCS) [34]:

- **RCS:** The Reaching Cache States at a program point P of a program, denoted as RCS_P , is the set of possible cache states when P is reached via any incoming path.
- **LCS:** Given a program, the living cache states at a point P , denoted as LCS_P , are defined as the possible first references to cache blocks via any outgoing path from P .

Note that, the path mentioned here is the path on the CFG (control flow graph) of the pre-empted task. The iterative method used to calculate the RCS_P and LCS_P for a program point P is shown below.

$$\begin{aligned}
RCS_P^{IN} &= \bigcup_{p=\text{pre}(P)} RCS_p^{OUT} \\
RCS_P^{OUT} &= \{\text{lru}(r, \text{gen}_P) \mid r \in RCS_P^{IN}\} \\
LCS_P^{OUT} &= \bigcup_{s=\text{succ}(P)} LCS_s^{IN} \\
LCS_P^{IN} &= \{\text{lru}(l, \text{gen}_P) \mid l \in LCS_P^{OUT}\}
\end{aligned}$$

When the fixed point is reached in this iterative analysis, we set $RCS_P = RCS_P^{OUT}$, $LCS_P = LCS_P^{OUT}$. In the above, the $\text{pre}(P)$ ($\text{succ}(P)$) is the set of immediately preceding (succeeding) program points of P . In order to simplify the presentation, we only give the high level description of the lru function and the gen function, and a more detailed description presented in [34]. The function $\text{lru}(s_1, \text{gen}_P)$ defined in [34] returns the cache state s after access of program point P under a LRU replacement policy with an original cache state s_1 . And the gen function returns all the memory blocks accessed by a program point. The RCS_P^{IN} , RCS_P^{OUT} , LCS_P^{IN} and LCS_P^{OUT} are initially assigned with:

$$RCS_P^{IN} = \emptyset, \quad RCS_P^{OUT} = \{\text{gen}_P\}$$

$$LCS_P^{OUT} = \emptyset, \quad LCS_P^{IN} = \{\text{gen}_P\}$$

Finally, if a memory block resides both in RCS_P and LCS_P , the method in [34] considers the memory block to be a UCB. The computation of the UCBs is performed at an abstract domain. In the abstract domain, an abstract cache state is defined as a vector with n -elements $\{\text{ca}_1, \dots, \text{ca}_n\}$ to denote the RCS_P , LCS_P and UCB_P , where n is the number of cache sets. Each ca_i denotes the i -th cache set and it contains the memory blocks which will map to the cache set. In this chapter, as with most embedded processors (e.g., ARM 11 [21]), we assume the size of each cache block is 32 Bytes.

The iterative calculation for the RCS_P and LCS_P is based on the CFG (control flow graph) of the objective program. In the CFG, we use BB_i to denote a basic block and define the following concepts.

- **Program point:** Each program point corresponds to an instruction (on the executable level). More specifically, a program point locates right after the corresponding instruction.
- **Preemption point:** When a program is preempted at a program point, this program point is called a preemption point.
- **Execution point:** A program point may execute for more than one time when it is located in a loop. An execution point refers to a program point in a particular loop iteration.

In this work, to simplify the presentation, we assume a fully-associative cache with least recently used (LRU) replacement policy, and our techniques can be easily extended to set-associative caches since the analysis for one cache set does not affect other cache sets.

2.4 Motivation

The state-of-the-art UCB calculation method [34] (for set-associative instruction caches) is performed on the basic block boundaries. However, this may *underestimate* the UCB and lead to unsound CRPD analysis results when dealing with data caches, as shown in Section 2.4.1. On the other hand, the method in [34] is designed for instruction caches and may significantly overestimate the UCB with data caches, as it will be shown in Section 2.4.2.

2.4.1 Underestimation at The Basic Block Boundary

The work in [37], which focused on instruction analysis, proved that the number of useful cache blocks at any execution point within a basic block is the same. The following work on

CRPD analysis [52], [63], [34], performed their computations at the basic block boundary to reduce the computational complexity. The work in [6] noted that the set of UCBs may change at the instruction level and not only at the basic block level for data cache analysis. Since their work focused on the instruction analysis, they didn't give a clear conclusion about either unsoundness or pessimism and still performed their analysis on the basic block boundary. In this section, we present this phenomenon and claim that performing dCRPD at the basic block boundary may underestimate the number of UCBs. How to avoid this unsoundness is presented in Section 2.5.

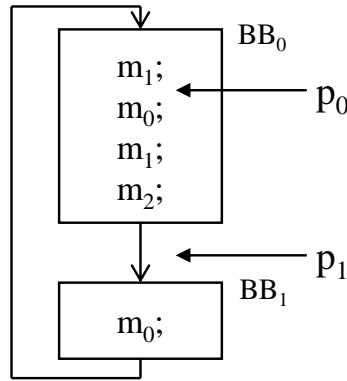


Figure 2.1. The control flow graph of a program fragment

We have a control flow graph of a program fragment in Fig. 2.1. In this example, the program contains 2 basic blocks BB_0 and BB_1 . In BB_0 , the program always accesses memory block m_1 , m_0 , m_1 and m_2 successively. In BB_1 , the program always accesses memory block m_0 . Consider two possible preemption points, P_0 and P_1 , where P_0 is located before the access to m_0 (a program point inside BB_0) and P_1 is located at the end of BB_0 (the boundary of BB_0). In this subsection we assume the associativity is 2. Based on the RCS/LCS analysis technique presented in Section 2.3, we have the following RCS and LCS of the preemption point P_1 :

$$RCS_{P_1} = \{(m_2, m_1)\}$$

$$LCS_{P_1} = \{(m_0, m_1)\}$$

Since m_1 is the only memory block that resides both in RCS_{P_1} and LCS_{P_1} , the number of useful cache blocks of P_1 is 1.

Similarly, we have the RCS and LCS of preemption point P_0 as follows:

$$RCS_{P_0} = \{(m_1, m_0)\}$$

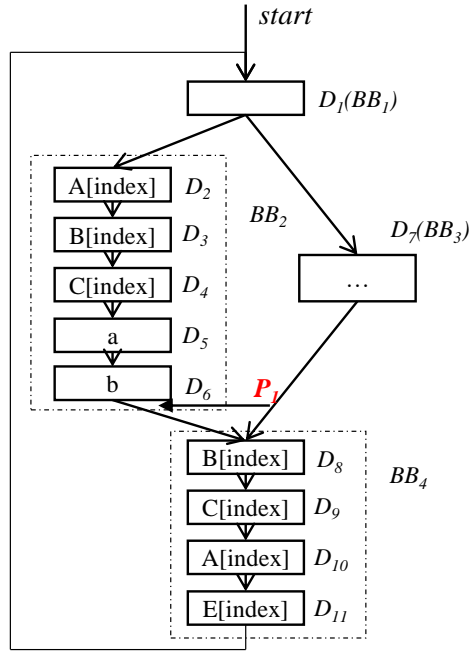
$$LCS_{P_0} = \{(m_0, m_1)\}$$

So the number of UCBs of P_0 is 2 (m_1 and m_0) which is larger than the number of UCBs of P_1 . Since the number of UCBs at the basic block boundary (P_1) may be less than other points (P_0) inside the basic block, the preemption occurred inside the basic block may lead to more additional cache misses. Therefore, using the number of UCBs at the basic block boundary to represent other points inside the basic block is unsafe. Hence, we make the following general observation for dCRPD analysis.

Observation 1. *In a program where data references at different program locations access the same data memory block, it will lead to an underestimation if the UCB analysis is performed at the basic block boundaries.*

2.4.2 Overestimation of UCBs for Set-associative Caches

In the following, we will use the example in Fig. 2.2 to illustrate the pessimism problem. Fig. 2.2(a) shows the control flow graph (CFG) of a program fragment, which contains four basic blocks, BB_1 to BB_4 , and they all reside in a loop L_1 with *index* of range $[0, 31]$. We assume BB_1 and BB_3 do not access data memory and only the data references in BB_2 and BB_4 are listed. Our example contains both array accesses and scalar variable accesses: Arrays **A**, **B**, **C** and **E** which contain 32 integers; **a** and **b** which are scalar variables. The accessed memory blocks and their corresponding temporal scopes are shown in Fig. 2.2(b). Since we assume a fully-associative cache, all the memory blocks map to the same cache set.



(a) The transformed CFG

index	0-7	8-15	16-23	24-31
A[index]	m_0	m_1	m_2	m_3
B[index]	m_4	m_5	m_6	m_7
C[index]	m_8	m_9	m_{10}	m_{11}
E[index]	m_{12}	m_{13}	m_{14}	m_{15}
a	m_{16}			
b	m_{17}			

(b) Memory blocks and corresponding temporal scopes

Figure 2.2. A running example

Most of the literature on CRPD analysis technique primarily focuses on instruction caches. When these techniques are extended for data caches (e.g., [37], [4], [34]), pessimistic results will be obtained due to the lack of ability to predict the dynamic behavior of data references. When such techniques address the dynamic behavior of data cache references, the original CRPD analysis may assume that each data reference instruction may access any of the *possible* memory blocks at any loop iteration (e.g., $A[index]$ may access either m_0 , m_1 , m_2 or m_3 during any iteration of loop L_1 in the example of Fig. 2.2(a)). However, each memory block can only be accessed at a particular loop iteration, as shown in Fig. 2.2(b), and it can be determined by a static analysis, which will be discussed in the context of our temporal scope analysis in Section 2.6.

Given the temporal scope information shown in Fig. 2.2(b), different memory blocks will become referenced at various loop iterations. Based on the analysis framework presented in Section 2.3, we have the following fixed-point analysis results for program point P_1 at

various loop iterations.

Iteration[0,7]

$$RCS_{P_1} = \{(m_{17}, m_{16}, m_8, m_4)\}$$

$$LCS_{P_1} = \{(m_4, m_8, m_0, m_{12})\}$$

Iteration[8,15]

$$RCS_{P_1} = \{(m_{17}, m_{16}, m_9, m_5)\}$$

$$LCS_{P_1} = \{(m_5, m_9, m_1, m_{13})\}$$

Iteration[16,23]

$$RCS_{P_1} = \{(m_{17}, m_{16}, m_{10}, m_6)\}$$

$$LCS_{P_1} = \{(m_6, m_{10}, m_2, m_{14})\}$$

Iteration[24,31]

$$RCS_{P_1} = \{(m_{17}, m_{16}, m_{11}, m_7)\}$$

$$LCS_{P_1} = \{(m_7, m_{11}, m_3, m_{15})\}$$

According to the accurate UCB calculation presented above, the number of UCBs and the corresponding useful memory block are shown in Table 2.1. Since a data reference may access different memory blocks in its different execution instances, the RCS and LCS of preemption points also changed in different loop iterations. Consequently, the UCBs of a preemption point at different loop iterations changed. Due to the lack of the temporal scope information of every memory block, the conventional framework considers that all memory blocks may be accessed in any loop iteration and this may lead to overestimation. Without the consideration of the temporal scope of every memory block, the UCB calculation method considers that the following memory blocks

$$m_4, m_5, m_6, m_7, m_8, m_9, m_{10}, m_{11}$$

Table 2.1. UCB in different temporal scopes at P_1

Loop iterations	number of UCB (UCBs)
[0,7]	$2(m_4, m_8)$
[8,15]	$2(m_5, m_9)$
[16,23]	$2(m_6, m_{10})$
[24,31]	$2(m_7, m_{11})$

are the useful cache blocks. So the conventional method considers that the maximum number of UCB is 4 (which can not be greater than the associativity). However depending on the above RCS and LCS of P_1 in different loop iterations and the UCBs in different loop iterations shown in Table 2.1, we can find for every particular loop iteration, there are at most only 2 useful memory blocks, so the maximum number of UCB at P_1 is 2. To summarize, the following observation is made in this chapter.

Observation 2. *A memory reference may access different memory blocks in its different references, especially for array access in the loop. As a result, the UCBs of a program point that is in a loop may be different at different loop iterations. To obtain a tight UCB count for dCRPD analysis, it is crucial to utilize the loop-affine reference information to identify the temporal scopes of the memory blocks.*

2.4.3 Summary of The Observations

In the above subsections, we point out that the data CRPD analysis can't be performed at the basic block boundary, otherwise it will lead to an underestimation; the data CRPD analysis should consider the temporal scope information of every memory block to capture its behavior, otherwise it will lead to overestimation. In the hard real time systems, underestimation

is unacceptable since the result is unsound. Moreover the overestimation will produce a pessimistic result, and then have a pessimistic effect on the scheduling analysis.

2.5 New Analysis Boundary

As pointed out in Section 2.4.1, conducting UCB analysis at basic block boundaries may produce unsound results. To solve this problem, we define a new type of block as follows and perform the analysis at its boundary:

Definition 2.5.1 (Data Analysis Block). *A data analysis block (DAB) is a maximal sequence of instructions within a basic block and contains at most one data memory referencing instruction at the beginning of the sequence.*

The original CFG is converted into a transformed CFG on the basis of DABs:

- A basic block in the original CFG without accessing any data memory block remains as a DAB in the transformed CFG.
- A basic block containing accesses to data memory blocks is divided into a set of DABs, where in each DAB, the data memory accessing instruction is the first instruction of the DAB.

During the execution of a DAB without data memory blocks, the program does not access any data memory, so the data cache state is unchanged. For a DAB containing accesses to data memory blocks, the program only accesses the data memory at the first instruction, then the data cache states are steady during the execution of the remaining instructions of the DAB (in the same loop iteration). Therefore, we have the following theorem:

Theorem 2.5.1. *The UCBs are equal at every point of a DAB (at each particular loop iteration if the DAB is a loop).*

The proof of the theorem is straightforward and thus omitted. Therefore, performing UCB analysis at the boundaries of DABs is safe.

Fig. 2.2(a) shows the transformed CFG of the example. In the transformed CFG, basic block BB_2 is divided into five DABs D_2, D_3, D_4, D_5 and D_6 , each containing a data memory reference at the beginning (i.e., B in D_3 and a in D_6). Basic block BB_4 is divided into D_8, D_9, D_{10} and D_{11} . Finally, basic blocks BB_1 and BB_3 , which do not have any data memory block references, are directly converted into DABs D_1 and D_7 in the transformed CFG respectively.

2.6 Temporal Scope Analysis

According to Observation 2 in Section 2.4.2, traditional UCB analysis may cause overestimation when applying to data caches because it fails to predict the dynamic behavior of data accesses. To solve this problem, we integrate the concept of *temporal scope* into UCB analysis to better predict the behavior of data accesses. The temporal scope was first introduced in [31] in the context of WCET analysis, which helps to more precisely handle the dynamic behaviors of data accesses and input/data-dependency.

In a previous work [31], the temporal scope was defined on a may-basis. In CRPD analysis, for a preemption point, the reaching cache state and the living cache state should be computed as the input of the subsequent CRPD analysis. If at a particular loop iteration, a memory block must be accessed by a data memory reference, the access is a must access, otherwise it is a may access. We assume that the age of a memory block is 0 when it is accessed, meanwhile the age of other memory blocks that has the smaller age in the same cache set increases by 1. Considering that data access may include must-access and may-access, must-access and may-access should be considered separately when computing the cache state. For example, in a set-associative cache analysis, when a program may access a memory block, the age of each cache block will not increase because it is possible that the program will not access the memory block. To perform a more accurate CRPD analysis, we use two different types of temporal scopes, must-scope and may-scope, to denote the must-

access and may-access, respectively, and the definitions of the must temporal scope and the may temporal scope are as follows:

Definition 2.6.1. (May Temporal Scope) A may temporal scope regarding data memory block m and DAB D , denoted by \widetilde{TS}_m^D , is defined as:

$$\widetilde{TS}_m^D = \left(\widetilde{m}, D, \bigcup_{D \in L_i} \{L_i[l, u]\} \right)$$

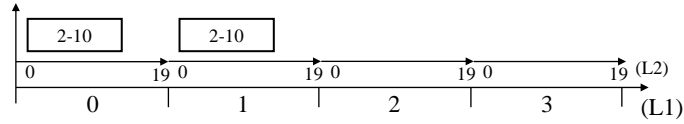
where $\bigcup_{D \in L_i} \{L_i[l, u]\}$ characterizes a continuous range of iterations for loops containing D , and data memory block m **may** be accessed in DAB D in any loop iteration in $\bigcup_{D \in L_i} \{L_i[l, u]\}$, where l and u represent the minimal iteration and the maximal iteration of loop L_i that m may be accessed respectively.

Definition 2.6.2. (Must Temporal Scope) A must temporal scope regarding data memory block m and DAB D , denoted by \overline{TS}_m^D , is defined as:

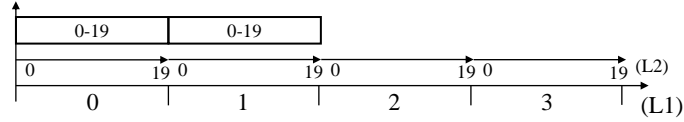
$$\overline{TS}_m^D = \left(\overline{m}, D, \bigcup_{D \in L_i} \{L_i[l, u]\} \right)$$

where $\bigcup_{D \in L_i} \{L_i[l, u]\}$ characterizes a continuous range of iterations for loops containing D , and data memory block m **must** be accessed in DAB D in any loop iteration in $\bigcup_{D \in L_i} \{L_i[l, u]\}$, where l and u represent the minimal iteration and the maximal iteration of loop L_i that m is accessed respectively.

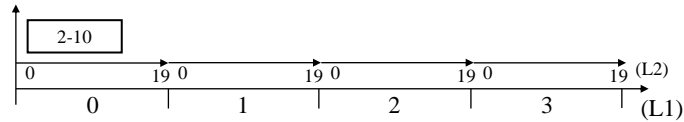
Note that, in the definition of *must* temporal scope and *may* temporal scope, we only capture a continuous range of iterations. This is because in Section 2.7.2 when unrolling the program, we need to get the continuous range of iterations that each must data reference only accesses one memory block in this range. The tilde and bar used on the memory block is only used to indict whether the memory block is from a *must* temporal scope or *may* temporal scope. The union set in the above definitions is used to capture all the range of iterations for different loops that the memory block is accessed. The *must* temporal scopes



(a) Noncontinuous iteration range



(b) Continuous iteration range



(c) Continuous iteration range

Figure 2.3. Examples of continuous and noncontinuous iteration ranges

and *may* temporal scopes have essentially the same form, but differ in their semantics (the data memory block *must* be accessed or *may* be accessed). In other words, a *may* temporal scope is an *over-approximation* of the actual possible loop ranges of a memory block, while a *must* temporal scope is an *under-approximation*.

In our work, the *may* temporal scope analysis is performed using the same approach presented in [31], which performs address analysis at the disassembly code with the program binary. Note that the iteration range of resulting *may* temporal scope by this approach must be continuous. Fig. 2.3 illustrates the difference between continuous and noncontinuous iteration ranges. In Fig. 2.3-(a), the iteration ranges of the outer loop L_1 and inner loop L_2 are $[0, 1]$ and $[2, 10]$, which lead to noncontinuous iteration range. Its safe continuous over-approximation is shown in Fig. 2.3-(b), where the loop range of the inner loop L_2 is extended to $[0, 19]$, where 19 is the loop upper bound of L_2 . In Fig. 2.3-(c), the inner loop does not reach lower or upper bound of the loop. However, since its outer loop only covers one iteration, it is also a continuous iteration range. In general, an iteration range is continuous if the following condition holds: scan all the loops from the outermost to innermost, once a loop has an interval iteration range, all the inner loops must have iteration

range covering the whole valid ranges. For example, we show an array access in Fig. 2.4. In the example, array A must access memory block m_1 in different iteration ranges $[0, 7]$ and $[16, 24]$. Since A accesses m_1 in noncontinuous iteration range, we get the safe continuous over-approximation of m_1 $[0, 24]$. Therefore, in range $[0, 24]$ A does not must access m_1 , and the temporal scope of m_1 is a may temporal scope. Note that, accessing a memory block in noncontinuous iteration range is common in reality programs, i.e., traversing a matrix with column major order.

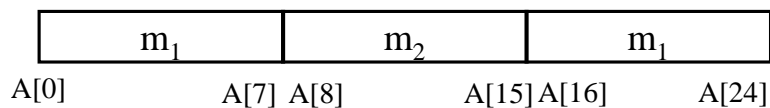


Figure 2.4. Memory blocks accessed by array A at different loop iterations

In principle, *must* temporal scopes can be obtained using similar techniques with [31]. In this work, we use a simple approach to do this: we obtain the *must* temporal scopes by selecting among the *may* temporal scopes obtained above that actually characterize *precise* memory access patterns. The simple approach generally contains two steps: 1. we obtain all the *may* temporal scopes; 2. for a *may* temporal scope, if the obtained iteration ranges of all other temporal scopes do not overlap with its, it is a *must* temporal scope. By doing this, we only get the temporal scopes that are both *may* and *must* temporal scopes (and thus is actually the exact temporal scope for the memory access). If a *must* temporal scope is obtained, we can simply discard the corresponding *may* temporal scope (i.e., the *may* temporal scope is masked by the *must* temporal scope).

We use the example in Fig. 2.5 to illustrate the *must* and *may* temporal scopes. Assume the program is running on a 32-bit machine with a memory block size of 32 bytes, and all the array accesses never exceed the array boundary. Also the memory block size can be changed depending on the real platform architecture. In this chapter, we assume that the memory block size is 32 bytes, as with lots of embedded processors [21]. The program contains a scalar access (variable x), non-scalar access (array B) and input-dependent access

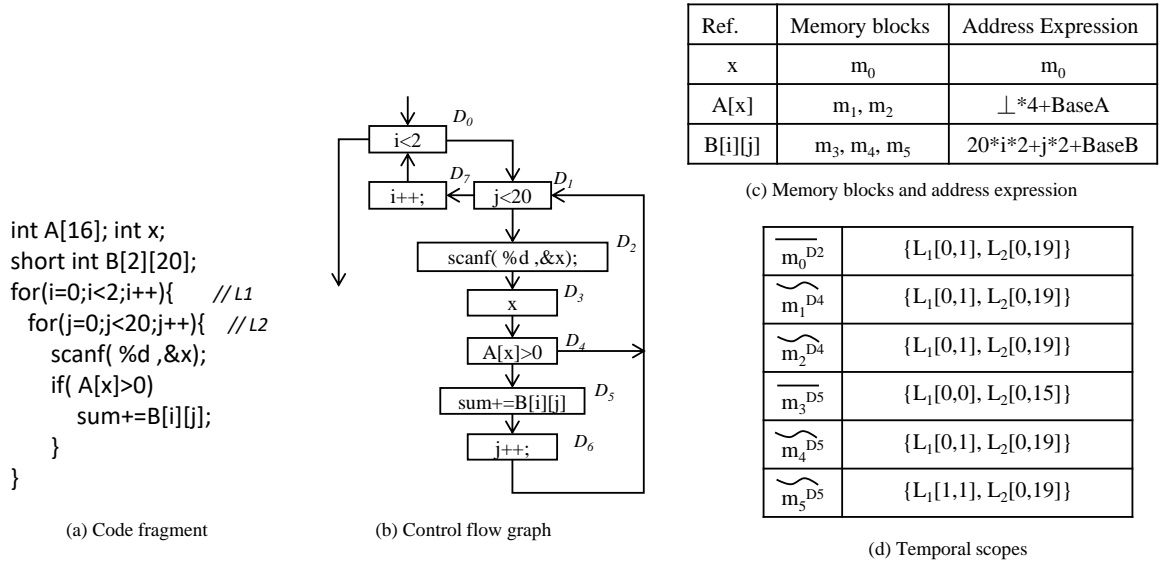


Figure 2.5. Temporal scope analysis

(array A). Three DABs D_2 , D_3 and D_4 all contain accesses to m_0 , the data memory block of x . The temporal scope analysis is automatically performed with the assembly code of each benchmark. With the assembly code, we can get the address expression of each data memory reference. In the following we discuss how to get the temporal scope of each memory block according to the address expression with the example.

For the access to x , all references to x (in DABs D_2 and D_3) always access memory block m_0 in any loop iteration of L_1 and L_2 . Therefore, the *must* temporal scopes of m_0 accessed by x regarding D_2 and D_3 are

$$\overline{\text{TS}}_{m_0}^{D_2} = (\overline{m_0}, D_2, \{L_1[0, 1], L_2[0, 19]\})$$

$$\overline{\text{TS}}_{m_0}^{D_3} = (\overline{m_0}, D_3, \{L_1[0, 1], L_2[0, 19]\})$$

The memory blocks accessed by array A depend on the value of input variable x , which cannot be determined at compile time. Since the program is running on a 32-bit machine and each integer occupies 4 bytes in memory, the access of $A[x]$ may cover two memory blocks, denoted by m_1, m_2 in our example, and the address expression of the array is $\perp \times 4 + \text{BaseA}$, where BaseA denotes the start address of array A (also the address of

the first byte of memory block m_1), \perp denotes the unknown input-dependent access index. Since \perp is unknown, we have to assume that array A can access any memory block in the valid range in different iterations. Therefore, we cannot claim any *must* temporal scope for m_1 and m_2 , and have their *may* temporal scopes as follows:

$$\widetilde{TS}_{m_1}^{D_4} = (\widetilde{m}_1, D_4, \{L_1[0, 1], L_2[0, 19]\})$$

$$\widetilde{TS}_{m_2}^{D_4} = (\widetilde{m}_2, D_4, \{L_1[0, 1], L_2[0, 19]\})$$

For the loop affine access array B , the data memory block accessed in a loop iteration depends on the value of loop indices i and j . The address expression of array B is

$$(20 \times i + j) \times 2 + \text{BaseB}$$

where BaseB is the start address of B (also the start address of its first memory block m_3). Array B contains 40 short integers, where each short integer occupies 2 bytes in memory. Memory block m_3 has the address range $[\text{BaseB}, \text{BaseB} + 31]$, so it is accessed with $i = 0 \wedge 0 \leq j \leq 15$. Therefore, m_3 has a *must* temporal scope:

$$\overline{TS}_{m_3}^{D_5} = (\overline{m}_3, D_5, \{L_1[0, 0], L_2[0, 15]\})$$

The address range of data memory block m_4 is $[\text{BaseB} + 32, \text{BaseB} + 63]$, m_4 can be accessed from $(i = 0, j = 16)$ and $(i = 1, j = 11)$. Therefore its actual access range is continuous. But the iteration ranges of j are not equal when i is equal to 0 or 1. So we use a minimal continuous over-approximation of its actual iteration range $L_1[0, 1], L_2[0, 19]$, and have *may* temporal scope as shown below.

$$\widetilde{TS}_{m_4}^{D_5} = (\widetilde{m}_4, D_5, \{L_1[0, 1], L_2[0, 19]\})$$

2.7 Scope-Aware UCB Analysis

This section presents our scope-aware UCB analysis based on the new analysis boundaries and *must/may* temporal scopes introduced in the above sections. Similar to [34], our UCB

analysis is also performed by computing the RCS and LCS. However, we will compute the UCB of a DAB at each loop iteration, instead of computing a single UCB for all the loop iterations as in [34].

In [34], RCS (LCS) is computed by iteratively collecting memory blocks into RCS (LCS) until a fixed point is reached. This method is not suitable to our problem since (1) the RCS and LCS of the same DAB at different loop iterations are different and (2) the data memory blocks accessed in different loop iterations may be different. A naive method to compute the RCS (LCS) in different loop iterations is to simply unroll all the loops and search backwards (forwards) along the unrolled CFG to check which memory blocks may still reside in the cache when reaching the considered DAB in each particular loop iteration, which is extremely inefficient due to the exploded size of the unrolled CFG. In the following, we will introduce a scalable approach to compute the RCS (LCS) of a DAB in different loop iterations.

2.7.1 Temporal Scope Normalization

The first step is to normalize the temporal scopes such that no two temporal scopes have overlapping loop iteration ranges. Intuitively, this will identify the boundary loop iterations of the temporal scopes. The iterations between two consecutive boundaries are similar and thus do not need to be unrolled.

The method of how to normalize given temporal scopes is given in Algorithm 1. In the algorithm, for each outmost loop, we first check all the temporal scopes residing in the same outmost loop. For each outmost loop L and a list of all the temporal scopes accessed in L , we use $normalize(list, L)$ to do the normalization. The basic idea of our algorithm is to split the temporal scopes from its outmost loop to the inner loop. After the split of outer loop, all the temporal scopes do not overlap at this loop. Then for the temporal scopes which have the same loop interval at this loop, we split on its inner loop, iteratively, until we split to the inner most loop, then all the split temporal scopes in the same outmost loop do not overlap with each other.

Algorithm 1 *normalize(list, L)*

```
1: for i=0; i < L.index; i++ do
2:   if check(list, L, i) then
3:     split(list, L, i);
4:   end if
5: end for
6: {SLa, SLb, ...} = group(list)
7: if L has inner loop then
8:   denoting the inner loop of L as Lx
9:   for all the S of group(list) do
10:    normalize(SLx, Lx);
11:   end for
12: end if
```

In the algorithm 1, $L.index$ denotes the iterations of loop L . For a given outmost loop L and a temporal scope list, we traverse all the loop iterations of this loop from the first to the last (line 1), where the temporal scope list contains all the temporal scopes in L . We use $check(list, L, i)$ to check whether there is a temporal scope TS in $list$ satisfying

$$TS.L.l == i \vee TS.L.u == i$$

If there exists such a temporal scope, it means the temporal scope's boundary at L is i . Then we use the function $split(list, L, i)$ to split other temporal scopes to avoid their overlap at this loop, where the $TS.L.l$ denotes the l at L of TS , and the $TS.L.u$ denotes the u at L of TS . Function $split(list, L, i)$ checks all the temporal scopes in the list which satisfies:

$$TS.L.l \leq i \wedge TS.L.u > i$$

For each of such temporal scopes TS , we create a new temporal scope $TS' = TS$ assuming

$$TS.L.u = i; TS'.L.l = i + 1$$

and insert TS' into the *list*. When all the iterations of L have been checked, all the temporal scopes do not overlap in this level. And then we use the function $group(list)$ to separate the *list* to different temporal scope lists S_{L_x} . In each list, temporal scopes have the same loop interval at L , and they have the same inner loop of L denoted as L_x . Iteratively, we reuse the function $normalize$ to normalize the temporal scope of each S_{L_x} at loop L_x until all the temporal scopes do not overlap at every loop level. The computational complexity of Algorithm 1 is I^N , where I denotes the maximum iterations of loops and N denotes the maximum nesting depth of loops.

In the following, we illustrate how Algorithm 1 works. Rather than using temporal scopes of memory blocks in Fig. 2.5, in order to better illustrate our normalization method, we use five new temporal scopes. Among these five sample temporal scopes, there are four must temporal scopes ($\overline{TS}_1, \overline{TS}_2, \overline{TS}_3, \overline{TS}_4$) and a may temporal scope (\widetilde{TS}_5)

$$\overline{TS}_1 = (\overline{m}_1, D_1, \{L_1[0, 4], L_2[0, 19]\})$$

$$\overline{TS}_2 = (\overline{m}_2, D_1, \{L_1[5, 8], L_2[0, 19]\})$$

$$\overline{TS}_3 = (\overline{m}_3, D_2, \{L_1[0, 0], L_2[0, 9]\})$$

$$\overline{TS}_4 = (\overline{m}_4, D_2, \{L_1[1, 1], L_2[0, 9]\})$$

$$\widetilde{TS}_5 = (\widetilde{m}_5, D_2, \{L_1[0, 8], L_2[0, 19]\})$$

In these temporal scopes, for their outmost loop L_1 , they have five boundaries: 0 ($\overline{TS}_1, \overline{TS}_2, \widetilde{TS}_5$), 1 (\overline{TS}_4), 4 (\overline{TS}_1), 5 (\overline{TS}_2), 8 (\widetilde{TS}_5). The boundary of temporal scopes at a particular loop is the upper bound and the lower bound of their loop iteration range, and they can be found using the $check(list, L, i)$ function in Algorithm 1. In order to avoid the overlap of all the temporal scopes at loop L_1 , we split every temporal scope whose loop interval covers any

other's boundaries. After the split we get the following temporal scopes.

$$\overline{\text{TS}}_{1.1} = (\overline{\text{m}}_1, \text{D}_1, \{\text{L}_1[0, 0], \text{L}_2[0, 19]\})$$

$$\overline{\text{TS}}_{1.2} = (\overline{\text{m}}_1, \text{D}_1, \{\text{L}_1[1, 1], \text{L}_2[0, 19]\})$$

$$\overline{\text{TS}}_{1.3} = (\overline{\text{m}}_1, \text{D}_1, \{\text{L}_1[2, 4], \text{L}_2[0, 19]\})$$

$$\overline{\text{TS}}_2 = (\overline{\text{m}}_2, \text{D}_1, \{\text{L}_1[5, 8], \text{L}_2[0, 19]\})$$

$$\overline{\text{TS}}_3 = (\overline{\text{m}}_3, \text{D}_2, \{\text{L}_1[0, 0], \text{L}_2[0, 9]\})$$

$$\overline{\text{TS}}_4 = (\overline{\text{m}}_4, \text{D}_2, \{\text{L}_1[1, 1], \text{L}_2[0, 9]\})$$

$$\widetilde{\text{TS}}_{5.1} = (\widetilde{\text{m}}_5, \text{D}_2, \{\text{L}_1[0, 0], \text{L}_2[0, 19]\})$$

$$\widetilde{\text{TS}}_{5.2} = (\widetilde{\text{m}}_5, \text{D}_2, \{\text{L}_1[1, 1], \text{L}_2[0, 19]\})$$

$$\widetilde{\text{TS}}_{5.3} = (\widetilde{\text{m}}_5, \text{D}_2, \{\text{L}_1[2, 4], \text{L}_2[0, 19]\})$$

$$\widetilde{\text{TS}}_{5.4} = (\widetilde{\text{m}}_5, \text{D}_2, \{\text{L}_1[5, 8], \text{L}_2[0, 19]\})$$

Once no temporal scopes overlap at loop L_1 , we split all the temporal scopes with the same loop interval at L_1 on their inner loops. After the split, the given temporal scopes are normalized to:

$$\{L_1[0, 0]\}$$

$$\overline{TS}_{1.1.1} = (\overline{m}_1, D_1, \{L_1[0, 0], L_2[0, 9]\})$$

$$\overline{TS}_{1.1.2} = (\overline{m}_1, D_1, \{L_1[0, 0], L_2[10, 19]\})$$

$$\overline{TS}_3 = (\overline{m}_3, D_2, \{L_1[0, 0], L_2[0, 9]\})$$

$$\widetilde{TS}_{5.1.1} = (\widetilde{m}_5, D_2, \{L_1[0, 1], L_2[0, 9]\})$$

$$\widetilde{TS}_{5.1.2} = (\widetilde{m}_5, D_2, \{L_1[0, 1], L_2[0, 19]\})$$

$$\{L_1[1, 1]\}$$

$$\overline{TS}_{1.2.1} = (\overline{m}_1, D_1, \{L_1[1, 1], L_2[0, 9]\})$$

$$\overline{TS}_{1.2.2} = (\overline{m}_1, D_1, \{L_1[1, 1], L_2[10, 19]\})$$

$$\overline{TS}_4 = (\overline{m}_4, D_2, \{L_1[1, 1], L_2[0, 9]\})$$

$$\widetilde{TS}_{5.2.1} = (\widetilde{m}_5, D_2, \{L_1[1, 1], L_2[0, 9]\})$$

$$\widetilde{TS}_{5.2.2} = (\widetilde{m}_5, D_2, \{L_1[1, 1], L_2[10, 19]\})$$

$$\{L_1[2, 4]\}$$

$$\overline{TS}_{1.3} = (\overline{m}_1, D_1, \{L_1[2, 4], L_2[0, 19]\})$$

$$\widetilde{TS}_{5.3} = (\widetilde{m}_5, D_2, \{L_1[2, 4], L_2[0, 19]\})$$

$$\{L_1[5, 8]\}$$

$$\overline{TS}_2 = (\overline{m}_2, D_1, \{L_1[5, 8], L_2[0, 19]\})$$

$$\widetilde{TS}_{5.4} = (\widetilde{m}_5, D_2, \{L_1[5, 8], L_2[0, 19]\})$$

2.7.2 RCS and LCS Calculation

In this section, we present how to compute the RCS and LCS of a DAB. In the following, we use the example in Fig. 2.6 to present how our technique works. In this example, the program contains two loops L_1 and L_2 and 7 DABs as shown in the CFG (Fig. 2.6 (a)). The unrolled CFG of the example in Fig. 2.6 (a) based on the normalized temporal scopes in Fig.

2.6 (b) is shown in Fig. 2.6 (c). For loops in our example, loop L_1 contains 2 iterations and is the outer loop of L_2 which contains 16 iterations.

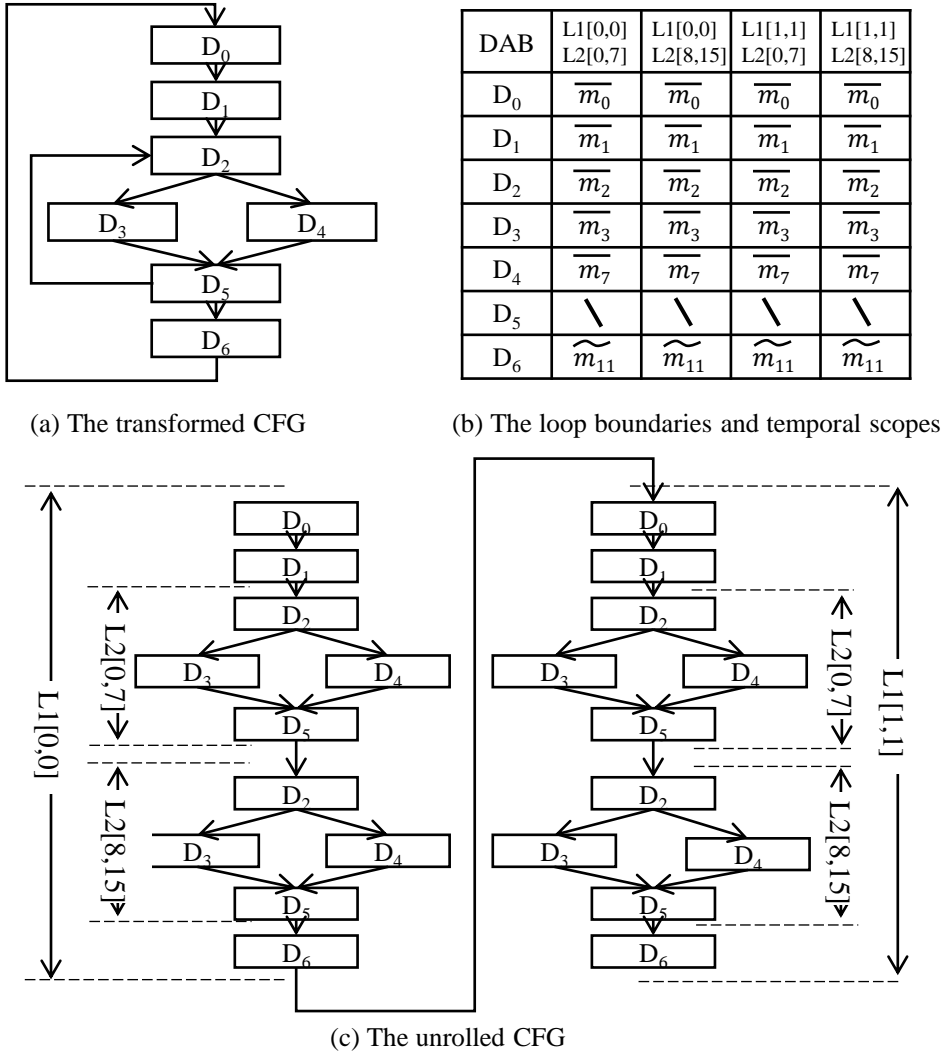


Figure 2.6. Program unrolling according to the normalization

As presented in Section 2.4.2, the RCS and LCS of a DAB are different in its different instance of different loop iterations if the DAB is in a loop. So, in our work, the calculation of RCS and LCS is performed on each loop iteration of a DAB. We first introduce some notations. For a DAB D , we define D^ℓ as the instance of D in a particular loop iteration $\ell = \cup_{D \in L_i} \{L_i\langle a \rangle\}$, where $L_i\langle a \rangle$ denotes the particular loop iteration of loop L . For the example in Fig. 2.6, $D_2^{L_1[0], L_2[10]}$ represents the D_2 at the first loop iteration of L_1 and the

11-th iteration of L_2 .

After the normalization in Section 2.7.1, the iteration ranges of any two temporal scope residing in the same loop are not overlapping, i.e., either they are equal, or one is earlier than the other one. Therefore, instead of unrolling the CFG with each single loop iteration, we can unroll the CFG in a more compact way, based on the boundaries of the iteration ranges of the temporal scopes.

The unrolled CFG is composed by different loop iteration ranges. Since the cache state of a DAB at the border of the range may be different with that of iteration inside the range, we compute RCS_{D^ℓ} and LCS_{D^ℓ} for each D^ℓ . We first locate the D^ℓ on the unrolled CFG. For the example shown in Fig 2.6, assume we compute the RCS and LCS of $D_2^{L_1[0],L_2[10]}$. First we locate the $D_2^{L_1[0],L_2[10]}$ on the unrolled CFG, which is D_2 in the iteration range of $L_1[0, 0], L_2[8, 15]$. In case of that, we extract the objective iteration from the range in order to capture the access behavior of the program in this loop range denoted as $G_{D_2^{L_1[0],L_2[10]}}$ as we show in the Fig. 2.7.

With the unrolled control flow graph $G_{D_i^\ell}$, we define the *distance* of two vertices on a path as follows:

Definition 2.7.1 (Distance(v_1, v_2)). *The distance between two vertices in the unrolled CFG is the minimum number of the must temporal scopes of distinct memory blocks on the path from v_1 to v_2 .*

For the distance calculation, when counting the number of the must temporal scopes, the temporal scope in v_2 should be counted and the temporal scope in v_1 should not. In case of $v_1 = v_2$, the distance is 0.

The $RCS_{D_i^\ell}$ and $LCS_{D_i^\ell}$ are represented by a vector of k elements $c[0, \dots, k-1]$. The RCS_{D^ℓ} , is defined as:

$$RCS_{D^\ell}^x[i] = (\bar{m}, \tilde{m} | \forall TS(m) \in G \wedge Dis(TS(m), D^\ell) = i)$$

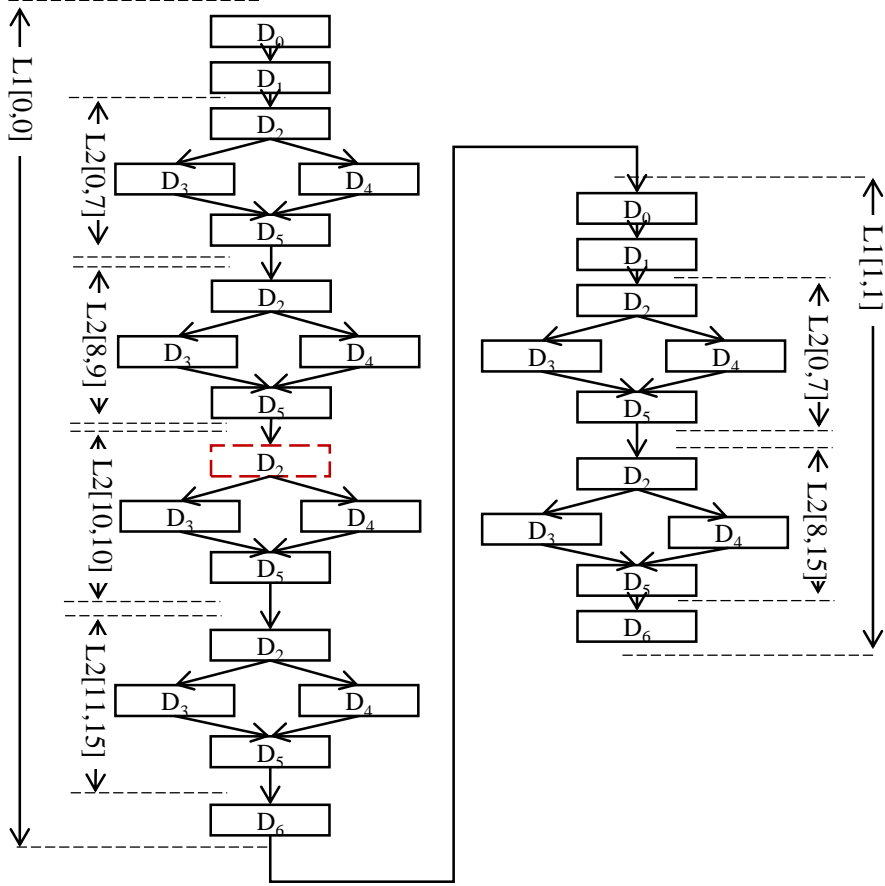


Figure 2.7. Unrolled control flow graph for the RCS and LCS calculation of $D_2^{L_1[0],L_2[10]}$.

The LCS_{D^ℓ} is defined as:

$$LCS_{D^\ell}^x[i] = (\bar{m}, \tilde{m} | \forall TS(m) \in G \wedge Dis(D^\ell, TS(m)) = i + 1)$$

where the \bar{m} denotes the memory block from the must temporal scope, \tilde{m} is the memory block from the may scope, $TS(m)$ denotes vertex on G_{D^ℓ} of the temporal scopes of m and the D_i^ℓ denotes the corresponding DAB on G_{D^ℓ} .

After the extraction of $D_2^{L_1[0],L_2[10]}$, for the $RCS_{D_2^{L_1[0],L_2[10]}}$ calculation, we traverse the $G_{D_2^{L_1[0],L_2[10]}}$ from the located DAB ($D_2^{L_1[0],L_2[10]}$) in a reverse direction, until we find k distinct memory blocks in every possible reverse path. For different paths, we have different

$RCS_{D_2^{L_1[0],L_2[10]}}$:

$$RCS_{D_2^{L_1[0],L_2[10]}}^1 = \{(\overline{m_2}), (\overline{m_4}), (\overline{m_3}), (\overline{m_1})\}$$

$$RCS_{D_2^{L_1[0],L_2[10]}}^2 = \{(\overline{m_2}), (\overline{m_4}), (\overline{m_7}), (\overline{m_1})\}$$

$$RCS_{D_2^{L_1[0],L_2[10]}}^3 = \{(\overline{m_2}), (\overline{m_8}), (\overline{m_7}), (\overline{m_1})\}$$

$$RCS_{D_2^{L_1[0],L_2[10]}}^4 = \{(\overline{m_2}), (\overline{m_8}), (\overline{m_3}), (\overline{m_1})\}$$

Therefore, among all the paths, we get the minimum distance of each memory block at different paths and obtain the $RCS_{D_2^{L_1[0],L_2[10]}}$ as follows:

$$RCS_{D_2^{L_1[0],L_2[10]}} = \{(\overline{m_2}), (\overline{m_4}, \overline{m_8}), (\overline{m_3}, \overline{m_7}), (\overline{m_1})\}$$

Similarly for the LCS calculation, we traverse the program in a forward direction, then we have $LCS_{D_2^{L_1[0],L_2[10]}}$:

$$LCS_{D_2^{L_1[0],L_2[10]}}^1 = \{(\overline{m_8}), (\overline{m_2}), (\widetilde{m_{11}}, \overline{m_0}), (\overline{m_1})\}$$

$$LCS_{D_2^{L_1[0],L_2[10]}}^2 = \{(\overline{m_4}), (\overline{m_2}), (\widetilde{m_{11}}, \overline{m_0}), (\overline{m_1})\}$$

By selecting the minimum index of all the memory blocks among all the $LCS_{D_2^{L_1[0],L_2[10]}}^x$, we have:

$$LCS_{D_2^{L_1[0],L_2[10]}} = \{(\overline{m_8}, \overline{m_4}), (\overline{m_2}), (\widetilde{m_{11}}, \overline{m_0}), (\overline{m_1})\}$$

For memory block m in $RCS_{D_i^\ell}$ or $LCS_{D_i^\ell}$, we have the following lemma:

Lemma 2.7.1. *All the memory blocks that may be in the cache when the program reach to D_i^ℓ are in $RCS_{D_i^\ell}$. If the first access of a memory block after D_i^ℓ is a cache hit, it must in $LCS_{D_i^\ell}$.*

Proof. If m is in the cache when the program reaches to D_i^ℓ , there must exist a path from the latest access of m before D_i^ℓ to D_i^ℓ that the program access less than k memory blocks.

Therefore, in the unrolled CFG, there must exist a path from $TS(m)$ to D^ℓ that $Dis(TS(m), D^\ell)$ must be less than k , and thus the memory block must be in a $RCS_{D_i^\ell}^x$.

Similarly, if the first access of memory block is a cache hit, there must exist a path from D_i^ℓ to the first access of the memory block after D_i^ℓ that the program accesses less than k memory blocks. Therefore, $Dis(D^\ell, TS(m))$ must be less than k and the memory block must be in a $LCS_{D_i^\ell}^y$. \square

\square

2.7.3 UCB Calculation

In this section, we present the method of how to compute the UCB of a D at ℓ , which is denoted as UCB_{D^ℓ} . Before the UCB calculation, we first compute the useful memory block (umb) depending on the RCS and LCS which are computed in Section 2.7.2. The umb is defined as:

Definition 2.7.2 (Useful memory block (umb)). *A useful memory block of D^ℓ is a memory block which is in the cache when the program runs to D^ℓ and the next access to the memory block after D^ℓ is a hit.*

For a pair of $RCS_{D_i^\ell}^x$ and $LCS_{D_i^\ell}^y$, before computing umbs, we first define $\{MID_{D^\ell}(m_u)\}$ for the memory block(m_u) that both reside in the RCS_{D^ℓ} and LCS_{D^ℓ} . $\{MID_{D^\ell}(m_u)\}$ is defined as:

$$MID_{D^\ell}(m_u) = \{\bar{m} | \text{index}(\bar{m}) < \text{index}(m_u)\}$$

Where the $\text{index}(m)$ returns the index of m in the $RCS_{D_i^\ell}^x$ or $LCS_{D_i^\ell}^y$. Then we define $MID_{\min_{D^\ell}}(m_u)$ as the one among all $\{MID_{D^\ell}(m_u)\}$ with the minimal cardinality. Therefore, we have the following lemma:

Lemma 2.7.2. *Suppose m is in $RCS_{D_i^\ell}^x[r]$ and $LCS_{D_i^\ell}^x[l]$, if m is a umb, $|MIDmin_{D^\ell}(m)|$ must be less than k .*

Proof. 1. Suppose m is a umb of D_i^ℓ , between the latest access of m before D_i^ℓ and the first access of m after D_i^ℓ , the program at most access $k - 1$ memory blocks no matter which path it executes.

2. Since each $RCS_{D_i^\ell}^x[r]$ corresponds a path from the latest access of m to D_i^ℓ and each $LCS_{D_i^\ell}^x[l]$ corresponds a path from D_i^ℓ to the first access of m after D_i^ℓ , each $MID_{D^\ell}(m_u)$ contains all the memory blocks that must be accessed on a path from the latest access of m before D_i^ℓ and the first access of m after D_i^ℓ .

3. By step 1, the program at most accesses $k - 1$ memory blocks on every path. Therefore, the cardinality of all the $MID_{D^\ell}(m_u)$ are no larger than k . Thus $MIDmin_{D^\ell}(m)$ must be less than k , and the lemma is proved. \square

\square

By lemma 2.7.1, if a memory block m is a umb of D_i^ℓ , it must be in $RCS_{D_i^\ell}$ and $LCS_{D_i^\ell}$. By lemma 2.7.2, if m is a umb of D_i^ℓ , $|MIDmin_{D^\ell}(m)|$ must be less than k . Therefore, we can get all the umbs for D_i^ℓ by selecting the memory blocks (m_u) from $RCS_{D_i^\ell}$ and $LCS_{D_i^\ell}$ as follows:

$$m_u \in RCS_{D_i^\ell} \wedge m_u \in LCS_{D_i^\ell} \wedge |MIDmin_{D^\ell}(m_u)| < k$$

In our example, we compute the UCB of $D_2^{L_1[0],L_2[10]}$. According to the $RCS_{D_2^{L_1[0],L_2[10]}}$ and $LCS_{D_2^{L_1[0],L_2[10]}}$ computed in Section 2.7.2, the memory blocks m_2 , m_4 , m_8 and m_1 both reside in $RCS_{D_2^{L_1[0],L_2[10]}}$ and $LCS_{D_2^{L_1[0],L_2[10]}}$.

For memory block m_2 we have two $MID_{D_2^{L_1[0],L_2[10]}}(m_2)$:

$$\{m_8\}, \{m_4\}$$

Apparently, their cardinalities are all less than 4, so m_2 is a *umb*. Similarly, m_4 and m_8 are *umb* too. For memory block m_1 , we have 8 $MID_{D_2^{L_1[0],L_2[10]}}(m_1)$:

$$\begin{aligned} & \{\overline{m_2}, \overline{m_4}, \overline{m_3}, \overline{m_8}, \overline{m_2}, \overline{m_0}\} \{\overline{m_2}, \overline{m_4}, \overline{m_3}, \overline{m_4}, \overline{m_2}, \overline{m_0}\} \\ & \{\overline{m_2}, \overline{m_4}, \overline{m_7}, \overline{m_8}, \overline{m_2}, \overline{m_0}\} \{\overline{m_2}, \overline{m_4}, \overline{m_7}, \overline{m_4}, \overline{m_2}, \overline{m_0}\} \\ & \{\overline{m_2}, \overline{m_8}, \overline{m_7}, \overline{m_8}, \overline{m_2}, \overline{m_0}\} \{\overline{m_2}, \overline{m_8}, \overline{m_7}, \overline{m_4}, \overline{m_2}, \overline{m_0}\} \\ & \{\overline{m_2}, \overline{m_8}, \overline{m_3}, \overline{m_8}, \overline{m_2}, \overline{m_0}\} \{\overline{m_2}, \overline{m_8}, \overline{m_3}, \overline{m_4}, \overline{m_2}, \overline{m_0}\} \end{aligned}$$

Since the cardinalities of all the above $MID_{D_i^\ell}(m_1)$ s are more than 4, m_1 is not a *umb*.

When all the useful memory blocks of D_i^ℓ are obtained, we have the useful memory block set which contains all the useful memory blocks of D_i^ℓ denoted as $UMB_{D_i^\ell}$. So the number of *ucb* of D_i at loop iteration ℓ is;

$$UCB_{D_i^\ell} = \min\{|UMB_{D_i^\ell}|, k\}$$

So if the preemption point occurred at D_i^ℓ , the maximum CRPD is:

$$CRPD_{D_i^\ell} = UCB_{D_i^\ell} * \text{ReloadTime}$$

where the *ReloadTime* denotes the cache reload time.

In our example, depending on the computed useful memory block of $D_2^{L_1[0],L_2[10]}$, we have:

$$UMB_{D_2^{L_1[0],L_2[10]}} = \{m_2, m_4, m_8\}$$

So we have:

$$UCB_{D_2^{L_1[0],L_2[10]}} = 3$$

So if the program is preempted at $D_2^{L_1[0],L_2[10]}$ by the preempting task, the maximum CRPD is: $3 * \text{ReloadTime}$.

2.8 Evaluation

We evaluate the proposed dCRPD analysis approach with the WCET benchmark [2] in this section. Since our approach focuses on the data cache analysis, we only conduct our

dCRPD analysis on programs with frequent data memory references. In addition, we select representative programs with identical data memory access behaviors (e.g., among various bubble-sort- or selection-sort-based programs) because they have the same dCRPD analysis results. The other selected 11 benchmarks used in our evaluation explore different program structures and memory access patterns, including nested loops, non-rectangular loop nests, row-/column-based matrix accesses, and data-/input-dependent branches. Table 2.2 shows the benchmark name, a brief description, and the input array size. Moreover, we compare the analysis time of our technique and the conventional method in Fig. 2.8 to evaluate the overhead.

In the experiment, we modify the Chronos WCET analyzer ([38]) to obtain the transformed control flow graph with DABs and the temporal scope for all the memory blocks accessed by each data memory reference. We assume that each benchmark is executed on a processor architecture with a 5-stage pipeline, in-order execution, perfect branch prediction, and separate *L1* instruction and data caches. Both instruction and data caches are 4-way set-associative caches and have a total cache size of 8 kB with an LRU replacement policy. We assume that each cache block has a block size of 32 B; therefore, the instruction cache and data cache both contain 64 cache sets.

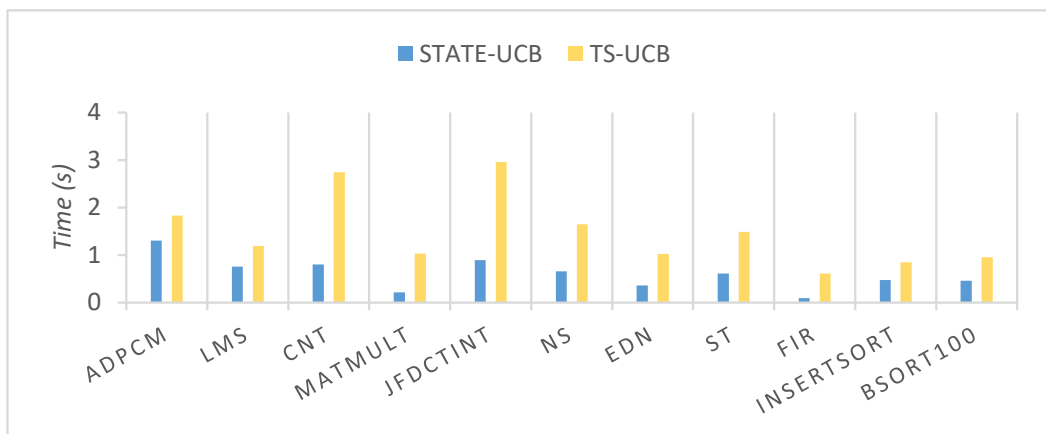


Figure 2.8. Analysis time of different benchmarks

Current dCRPD approach [53] is limited to addressing direct-mapped caches, and

since we focus on *set-associative* cache analysis, we will not compare our method with this approach. We compare our approach with the straightforward extension of the original approach [34], which focuses on LRU set-associative instruction cache analysis. To apply the original approach for safe dCRPD, we modify the *gen* function (as discussed in Section 2.3) of each data referencing instruction to generate an abstract cache state that contains all data memory blocks that are possibly referenced by the instruction at any time. Furthermore, in contrast to instruction cache analysis where the corresponding instruction memory block is guaranteed to be visited during an instruction fetch, the CRPD analysis for a data cache must operate on a may-basis for non-scalar data accesses and perform at the DAB boundary to obtain a safe upper bound.

The maximal UCB is the most important parameter for bounding the preemption cost. As discussed above the CRPD of the preempted task is the cache reload time multiple the intersection of the UCB and ECB. In our evaluation, we report the UCBs of all the programs listed in Table 2.2 between the original approach (ORI-UCB) and our proposed temporal-scope-aware UCB calculation (TS-UCB). Since the ORI-UCB fails to precisely identify which data memory blocks are visited at each program point, it has to conservatively assume that any of the memory blocks in the address space of the visited data array may become referenced. Therefore, for the analysis results of all the benchmarks, when the program with an array performs an access in a loop, the ORI-UCB thinks that all the cache blocks mapped by all the memory block are accessed by the array as the UCB. As a result, for all benchmarks, ORI-UCB treats all 256 data cache blocks or the working set of the program as UCBs. For example, *Matmult* has a working set of 216 memory blocks over the entire execution, which is smaller than the data cache size, and then, they are considered as the UCBs in ORI-UCB. Compared with ORI-UCB, our approach achieves a significant improvement in the maximal UCB calculation because our TS-UCB approach integrates the temporal scope information of every memory block.

The *Cnt* benchmark is a simple program that counts the non-negative numbers in a matrix. In this benchmark, the program uses a loop with a depth of two to access the matrix sequentially. During the sequential accessing of the matrix, a memory block of the matrix is

Table 2.2. Benchmark descriptions and array sizes

Benchmark	Benchmark description	Array Size	Number of DABs
Adpcm	Adaptive pulse code modulation	2048	57
Lms	Adaptive signal enhancement	1024	24
Matmult	Matrix multiplication	24×24	19
Cnt	Count non-negative in matrix	128×128	15
Jfdctint	DCT of pixel blocks	256×64	49
Bsort100	Bubblesort program	2048	13
Edn	(FIR) filter calculations	1024	38
Ns	Search multi-dimensional array	$16 \times 16 \times 16 \times 16$	16
St	Statistics program	2048	27
Fir	Finite impulse response filter	720	11
Insertsort	Insertion sort	2048	14

only accessed in eight successive loop iterations; thus, for the eight loop iterations, the UCB is 1. Therefore, fully considering the loop, the UCB number is 1 in every loop iteration, as shown in Fig. 2.9. The sequential access of an array or a matrix is very common in programs, like benchmarks *St* and *Ns*. According to the analysis result shown in Fig. 2.9, our TS-UCB achieves a significant improvement when addressing loop affine array references.

In contrast to data analysis approaches based on linear equations, temporal scope analysis is more applicable because it can address data references with input/data dependencies. The *Adpcm* benchmark includes an input-dependent and data-dependent situation. The CFG of *Adpcm* is shown in Fig.2.10, where the 3 different array accesses are labeled as (A),

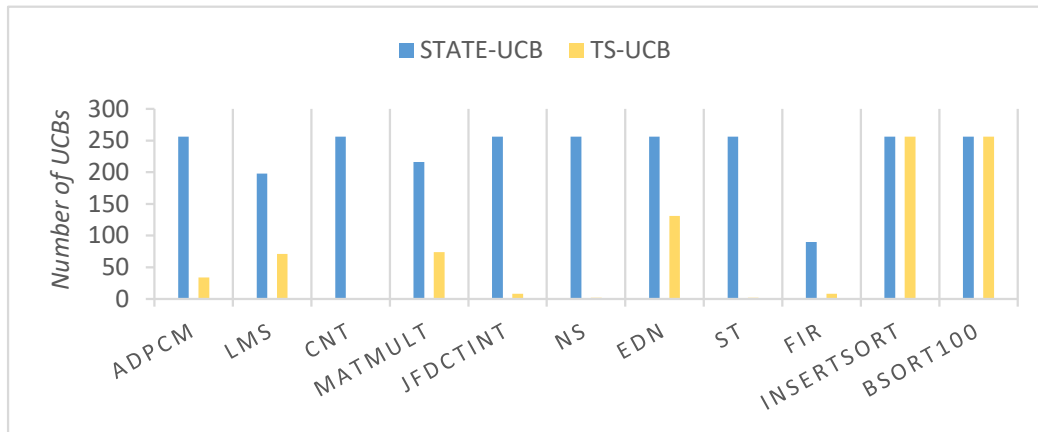


Figure 2.9. Maximal UCB number on different benchmarks

(B), and (C). Due to the array access (A) in the data-dependent branches (i.e., *bufferstep*), as well as the input-dependent array access (B) (i.e., *delta* is computed according to the input stream), our temporal scope analysis produces a pessimistic analysis result for (A) and (B); however, for array (C), the temporal scope analysis can also provide an accurate behavior prediction for every memory block of this reference. As a result, (A) is treated as an irregular array access in our analysis, where each access *may* visit any of the 33 memory blocks in array *inp_de*. Array access (B) is input dependent, therefore, we consider that it *may* visit any of the 3 data memory blocks in array *indexTable*. The temporal scope of the memory block referenced is a *may* scope. Finally, access to *outp_de* is loop affine. Therefore, although the *outp_de* array contains 2048 short integers (i.e., spanning over 129 memory blocks in the worst case), the maximal UCB resulting from accesses to *outp_de* is only 1. Finally, TS-UCB reports that the overall maximal number of UCBs is 37.

The *Bsort100*, as well as the *insertsort* program, where the boundary of the inner loop decreases when the outer loop's index increases. Due to the decrease in the work space of the inner loop, the data reference accesses fewer memory blocks. At the first iteration of the outer loop, the inner loop sequentially accesses all the memory blocks of the array and bubbles the objective value to the first place in the array. According to Table 2.2, the array size of *Bsort100* is 2048; therefore, the array may access 256 memory blocks. Then, when

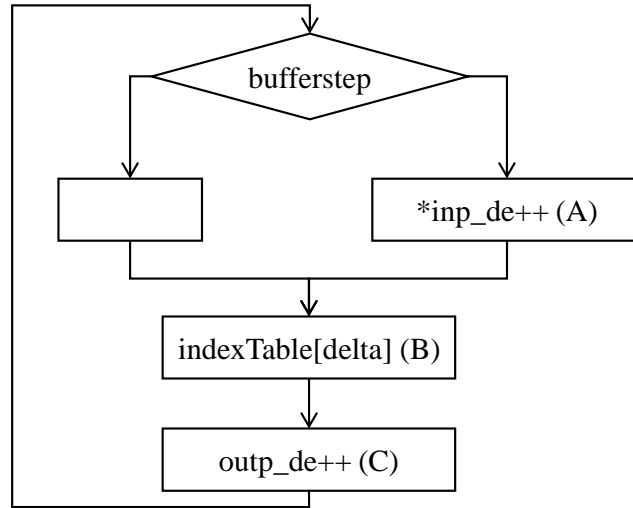


Figure 2.10. Data memory accesses of *Adpcm*

the outer loop reaches the second iteration, the inner loop accesses the 256 memory blocks again; therefore, the UCB number is 256, which is the same as the result of ORI-UCB. When the outer loop executes the 8th iteration, the inner loop accesses 255 memory blocks because the inner loop only executes for 2040 times; therefore, the UCB number of *Bsort100* is 255. As shown in Fig. 2.11, the UCB number decreases during the execution of the program. However, ORI-UCB finds that the UCB number of *Bsort100* is always 256. As a result, although both analyses give the same maximal UCB number, the proposed analysis accurately captures the changes in the UCB over the program execution, potentially leading to a substantially improved joint UCB-ECB CRPD and schedulability analysis in which possible preemption points are considered.

2.9 Conclusions

The utilization of cache in modern computer systems makes static timing analysis more complicated, especially for data caches with a more unpredictable access behavior. This chapter presents a general and accurate data CRPD analysis framework, which can produce a safe and accurate data CRPD analysis result compared with the state-of-the-art CRPD

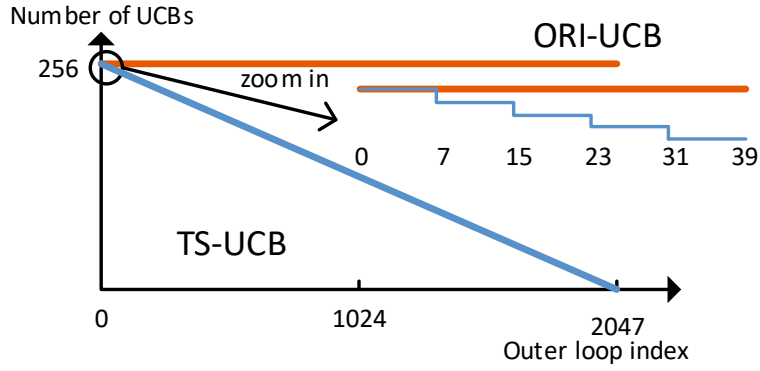


Figure 2.11. UCB number of different loop points in *Bsort100*

method [34]. Moreover, our work is the first approach that targets on the set-associative data cache-related preemption delay analysis.

According to the transformed CFG and the temporal scope of every memory block, we do the normalization of all the temporal scopes and unroll the program depending on the normalization analysis result. Compared with the method which fully unrolls the program, our framework is more efficient. With a distance calculation of each execution point, we can compute the RCS and LCS of program points at different loop iterations thus getting the UCBs. Finally, we evaluate our approach with the MRTC benchmark programs. The experimental result shows our approach can obtain a more accurate dCRPD analysis result than the existing works.

CHAPTER 3

CACHE-RELATED PREEMPTION DELAY WITH MULTIPLE PREEMPTIONS

3.1 Introduction

Work presented in chapter 2 show the CRPD analysis for a single preemption. In general, a lower-priority task may be preempted multiple times during the execution of each period. Existing CRPD analysis techniques calculate the UCB of different program points independently and sum the k -largest number of UCBs as the total number of UCBs for k preemptions.

The above-mentioned state-of-the-art CRPD analysis techniques, although work well with instruction caches, may lead to significant overestimation when dealing with data caches. An instruction may access different data memory blocks in different loop iterations, and the same memory block may be accessed by different instructions at different program points. Consequently, a data memory block may be contained in the UCB of different program points, and simply using the sum of the k -largest number of UCBs as the total number of UCBs is pessimistic.

In this chapter, we present a new CRPD analysis technique to address the pessimism described above for direct-mapped data caches. Our technique computes the UCBs of a program point based on the analysis of UCBs of program points where previous preemptions are assumed to happen, to remove the double-counting of UCBs of different preemptions. We define a new analysis unit which makes it more efficient to compute the redundant memory blocks in the subsequent preemptions.

We conduct experiments with benchmarks in [2]. The number of UCBs is widely

used to bound the CRPD if the preempting tasks are nondeterministic. Therefore, we compute the total number of UCBs of each benchmark for different number of preemptions. Moreover, we also compute the total CRPD with concrete preempted tasks and preempting tasks. According to the experimental results, we can conclude that, our method can substantially tighten the total number of UCBs for multiple preemptions and significantly reduce total CRPD with concrete preempting tasks.

3.2 Background

Related work in CRPD analysis is discussed in section 2.2. This chapter focuses on CRPD analysis for the direct-mapped cache, which is different from the set-associative cache analysis presented in chapter 2. Our previous work [71] has the most precision on CRPD analysis for the direct-mapped cache of a single preemption, and is considered as the base of the technique proposed in this chapter. As the background knowledge of this chapter, In this section, we briefly introduce again the temporal scope analysis and UCB computation in section 3.2.1 and 3.2.2, respectively. Section 2.5 and [71] show that using basic blocks as the analysis unit for data CRPD analysis is unsafe, and introduce a new analysis unit called DAB for safe UCB calculation. The UCBs of any program points in a DAB are equal, so when we talk about the UCB calculation, the program point and the DAB are equivalent.

3.2.1 Temporal Scope Analysis

The concept of temporal scope was introduced in [31] for WCET analysis, defined on *may* semantics in the sense that it captures memory blocks that may be accessed by a data reference instruction in particular loop iterations. Temporal scope analysis is performed automatically via an address analysis at the disassembly code from the program binary. Therefore, loop boundary and memory address of a program should be known at compile time in our method. Moreover, we do not deal with the program with virtual address and dynamically allocated data.

[71] adopted temporal scope for CRPD analysis, and extend it to include both *may* semantics and *must* semantics (memory blocks must be accessed by a data reference instruction in particular loop iterations). Since the latter work can capture the data access behavior more precisely, we use the latter temporal scope analysis.

Definition 3.2.1. (May Temporal Scope) A may temporal scope regarding data memory block m and DAB D , denoted by $\widetilde{\text{TS}}_m^D$, is defined as

$$\widetilde{\text{TS}}_m^D = \left(\widetilde{m}, D, \bigcup_{D \in L_i} \{L_i[l, u]\} \right)$$

where $\bigcup_{D \in L_i} \{L_i[l, u]\}$ characterizes a continuous range of iterations for loops containing D , and m is a data memory block that may be accessed in DAB D in any loop iteration in $\bigcup_{D \in L_i} \{L_i[l, u]\}$.

Definition 3.2.2. (Must Temporal Scope) A must temporal scope regarding data memory block m and DAB D , denoted by $\overline{\text{TS}}_m^D$, is defined as:

$$\overline{\text{TS}}_m^D = \left(\overline{m}, D, \bigcup_{D \in L_i} \{L_i[l, u]\} \right)$$

where $\bigcup_{D \in L_i} \{L_i[l, u]\}$ characterizes a continuous range of iterations for loops containing D , and m is a data memory block that must be accessed in DAB D in any loop iteration in $\bigcup_{D \in L_i} \{L_i[l, u]\}$.

For simplicity of presentation, we also use TS_m^D to denote a temporal scope in general when it is not needed to distinguish whether it is a *must* or a *may* temporal scope. For a temporal scope TS_m^D , we call the combination of its DAB and loop iteration range as its scope. For example, we have the following temporal scope:

$$(\overline{m}_1, D_1, \{L_1[0, 7], L_2[0, 7]\})$$

It means m_1 is accessed in DAB D_1 , and the data reference instruction in D_1 only accesses m_1 in the iteration #0 to iteration #7 of loop L_1 and the iteration #0 to iteration #7 of loop L_2 .

Its scope is:

$$(D_1, \{L_1[0, 7], L_2[0, 7]\})$$

In this work, we refer to a data reference instruction as a data reference. If the temporal scopes of all the memory blocks accessed by a data reference are must temporal scopes, we define the data reference as a **Must data reference**. Otherwise, the data reference is a **May data reference**. Since a must data reference only contains must scopes, the memory block accessed by the reference at each loop iteration is deterministic. Therefore, when a must data reference accesses a new memory block, the memory blocks accessed before will never be accessed. If there is a preemption occurs in a particular loop iteration, among the memory blocks that accessed before by a *Must* data reference, at most one memory block can be reloaded into the cache by the *Must* data reference when the program resumes execution.

3.2.2 UCB Analysis

[71] used the temporal scope of every memory block to capture the dynamic behaviors of data accesses, and use the “temporal scope abstract cache state” (TS-ACS) to denote the content of all cache blocks. A TS-ACS is presented by a vector of n elements $c[0, \dots, n - 1]$ (n denotes the number of cache sets), where $c[i] = TS_m$ if cache block c_i holds memory block m , or $c[i] = \perp$ if c_i does not hold any memory block.

In order to compute the UCBs of each DAB, [71] use a fixed-point analysis to compute its reaching cache states (RCS) and living cache states (LCS), where the RCS and LCS are defined as follows.

RCS. The reaching cache states at a program point p , denoted by RCS_p , is the set of possible cache states when p is reached via any incoming program path.

LCS. Given a program, the living cache states at a point p , denoted by LCS_p , is defined as the possible first references to cache blocks via any outgoing program path from p .

The fixed-point RCS computation of a DAB iteratively collects its preceding DABs

via different possible incoming edges and computes the possible cache states when the execution reaches to the DAB. When the fixed point is reached, the RCS of the DAB is obtained. Similarly, the fixed-point LCS computation of a DAB uses a backward iteration by collecting states of successors in the control flow graph.

A memory block contained by some TS-ACSs in RCS and some TS-ACSs in LCS is called a **useful memory block (UMB)**, and the corresponding cache block is a **useful cache block (UCB)**. According to the temporal scopes of a UMB, [71] finds whether the memory block can be loaded into the cache in a particular loop iteration, thus computes the UCBs for a DAB at each loop iteration.

3.3 Motivation

ALL Existing CRPD techniques for instruction cache analysis [7, 37] compute the UCBs of each program point independently, and simply sum the k -largest number of UCBs among all program points as the total UCBs for k preemptions. [7] pointed out that the basic block containing the maximum number of UCBs often resides in a loop and the maximum number should take into account as often as the loop iterates when selecting the k -largest UCBs. [71] can significantly tighten the CRPD for each program point. However, its CRPD analysis for each program point is also performed independently. In this section we will show that the CRPD of a program point may decrease if there was a preemption occurred beforehand. Consequently, simply summing the k -largest CRPD will lead to a significant overestimation for total CRPD calculation.

We use the following example to illustrate this issue. The transformed control flow graph (CFG) of a preempted task is shown in Fig. 3.1. It includes 6 DABs and a loop L_1 . Loop L_1 iterates for 20 times and contains two DABs, D_2 and D_3 . We assume the example is running on a platform with a direct-mapped cache of 4 cache blocks, and a memory block m_i maps to cache block $C_{(i \bmod 4)}$.

Using the UCB computation method in [71], the RCS and LCS of D_2 and D_3 are

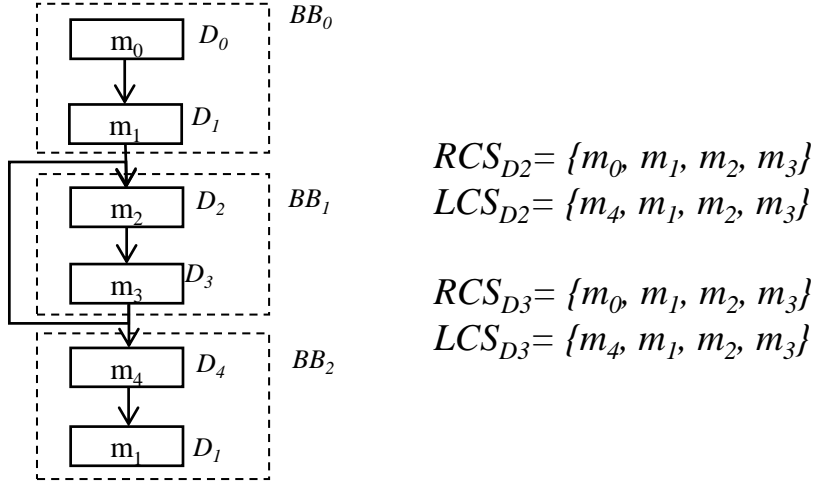


Figure 3.1. Transformed CFG of a preempted task

shown in Fig 3.1. As a result, the numbers of UCBs of D_2 and D_3 are 3, which are larger than other DABs'. Therefore, the maximum preemption cost of the preempted task is $t_{\text{reload}} \times 3$, where the t_{reload} denotes the cache block reloading time. Suppose the preempted task will be preempted for 8 times in one period. Since D_2 and D_3 both reside in L_1 and L_1 loops for 20 times (which is larger than the number of preemptions 8), the state-of-the-art methods [7, 37] consider that the 8 preemptions occur at D_2 or D_3 . Therefore, the total CRPD of 8 preemptions computed by the state-of-the-art method is:

$$t_{\text{reload}} \times 8 \times 3$$

However, this approach is very pessimistic, because the evicted UMBs by the former preemptions may never be loaded back into the cache. Thus, the number of UCBs of D_2 and D_3 actually may decrease in subsequent preemptions. Suppose the first preemption occurs at D_2 or D_3 , the CRPD of which is $t_{\text{reload}} \times 3$. The UMBs (m_1, m_2, m_3) of D_2 or D_3 are all considered to be evicted from the cache by the first preemption. Under this circumstance, when the task resumes execution, the RCS of D_2 and D_3 are changed to:

$$\text{RCS}_{D_2}^{2\text{nd}} = \{\perp, \perp, m_2, m_3\}$$

$$\text{RCS}_{D_3}^{2\text{nd}} = \{\perp, \perp, m_2, m_3\}$$

This is because, memory block m_1 will never be loaded back into the cache when the program resumes execution from the first preemption point (D_2 or D_3), while the LCS of D_2 and D_3 is the same as previous. Therefore, the number of UCBs of D_2 and D_3 is 2 if the program has been preempted at D_2 or D_3 before. As a result, the total CRPD of 8 preemptions occurred at D_2 or D_3 can be bounded by:

$$t_{\text{reload}} \times (3 + 7 \times 2)$$

Memory block m_1 which is considered as UMB of D_2 and D_3 in the first preemption should not be considered as UMB of D_2 and D_3 in the subsequent preemptions. The reason why m_1 is considered as UMB in the first preemption is that, m_1 is accessed by different data references, which reside in the preceding DAB (D_1) and the succeeding DAB (D_5) of D_2 respectively. However this situation can not happen in instruction cache analysis since an instruction memory block cannot be accessed at different locations in a program.

In summary, all existing CRPD techniques compute the UCBs of a program point independently. Therefore, in the traditional k -largest preemption cost calculation, a memory block m can be double counted as UMBs at different program points or the same program point in its different loop iterations. However, in many cases, once a memory block m is evicted from the cache due to preemption, it will not be revisited and loaded back into the cache within a certain program region. Therefore, m should be counted at most once in the total CRPD analysis for multiple preemptions in this region. In order to perform an accurate total CRPD analysis for multiple preemptions, we should re-compute the UCBs of each program point for the subsequent preemptions based on the UCBs already counted in the previous preemptions.

3.4 Useful Memory Block Analysis

In the following, we present a new method to compute the total CRPDs of a program for multiple preemptions to address the pessimism discussed in Section 3.3. Our method consists of

two parts: (1) the *first preemption analysis*, which assumes the preemption is the first preemption and analyzes the UCBs of each DAB, and (2) the *subsequent preemption analysis*, which analyzes the UCBs of each DAB assuming at least one preemption has happened. We use [71] for the first preemption analysis, and develop new analysis techniques for the second

3.4.1 A Running Example

This section first introduces a running example that will be used to illustrate our new method and some useful concepts. Then the new method to analyze UCBs for subsequent preemptions will be introduced in Section 3.5.

In this subsection, we introduce a running example as shown in Fig. 3.2 and present its first preemption analysis. Fig 3.2 (a) shows the transformed-CFG of a code fragment. The memory blocks accessed in each DAB and their temporal scopes are shown in the Fig 3.2 (b).

We assume the example is running on a direct-mapped cache with 8 cache sets ($c_0 \dots c_7$), and a memory block m_i maps to cache set c_i . Both loop L_1 and L_2 contain 32 loop iterations. The running example includes array accesses ($A[i]$) and accesses to a scalar variable at different locations (a, b). These two types of access patterns represent the main feature of data access.

By [71], the RCS and LCS of D_2 is:

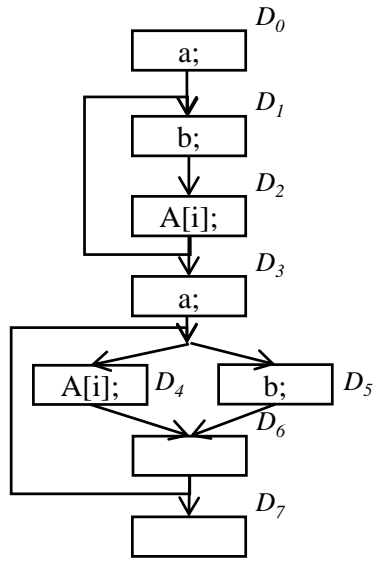
$$\begin{aligned}
\mathbf{RCS}_{D_2} &= \{(\overline{m_0}, D_2, L_1(0, 7)), (\overline{m_1}, D_2, L_1(8, 15)), \\
&\quad (\overline{m_2}, D_2, L_1(16, 23)), (\overline{m_3}, D_2, L_1(24, 31)), \\
&\quad (\overline{m_4}, D_0, \emptyset), (\overline{m_5}, D_1, (L_1(0, 31))), \perp, \perp\} \\
\mathbf{LCS}_{D_2} &= \{(\overline{m_0}, D_2, L_1(0, 7)), (\overline{m_1}, D_2, L_1(8, 15)), \\
&\quad (\overline{m_2}, D_2, L_1(16, 23)), (\overline{m_3}, D_2, L_1(24, 31)), \\
&\quad (\overline{m_4}, D_3, \emptyset), (\overline{m_5}, D_1, (L_1(0, 31))), \perp, \perp\} (\mathbf{LCS}_{D_2}^1) \\
&\quad \{(\overline{m_0}, D_4, L_2(0, 7)), (\overline{m_1}, D_4, L_2(8, 15)) \\
&\quad , (\overline{m_2}, D_4, L_2(16, 23)), (\overline{m_3}, D_4, L_2(24, 31)) \\
&\quad , (\overline{m_4}, D_3, \emptyset), (\overline{m_5}, D_5, (L_2(0, 31))), \perp, \perp\} (\mathbf{LCS}_{D_2}^2)
\end{aligned}$$

The LCS of D_2 represents two different cache states when the program leaves D_2 . $\mathbf{LCS}_{D_2}^1$ corresponds to " $D_2 \rightarrow D_1 \rightarrow D_2 \dots$ ". By \mathbf{RCS}_{D_2} and $\mathbf{LCS}_{D_2}^1$, we can get that the number of UCBs of D_2 is 3. Because according to the temporal scopes of m_0, m_1, m_2 and m_3 , there is at most one memory block among them that can be counted as UMB of D_2 at its each loop iteration. Moreover, memory blocks m_4 and m_5 are both UMBs of D_2 at its every loop iteration.

$\mathbf{LCS}_{D_2}^2$ corresponds to the program path " $D_2 \rightarrow D_1 \rightarrow D_2 \dots$ ". By \mathbf{RCS}_{D_2} and $\mathbf{LCS}_{D_2}^2$ the number of UCBs of D_2 is 6 and the UMBs are:

$$m_0, m_1, m_2, m_3, m_4, m_5$$

[71] finds the number of UCBs (6) only exist from the 25^{th} ($i=24$) iteration to the 32^{th} ($i=31$) iteration of loop L_1 . Because before the 25^{th} iteration of L_1 , m_3 have not been loaded into the cache and cannot be considered as a UMB. All the existing methods select the k -largest number of UCBs among all the program points and consider the time used to reload these UCBs as the maximum total CRPD of k -preemptions. In this example, when they select the k -largest number of UCBs, the number of UCBs (6) of D_2 will be selected 8 times.



(a) Control flow graph

LR ₀	a(D ₀)	m_4			
LR ₁	Index(L ₁)	0-7	8-15	16-23	24-31
	A[i] (D ₂)	m_0	m_1	m_2	m_3
	b(D ₁)	m_5			
LR ₂	a(D ₃)	m_4			
LR ₃	Index(L ₂)	0-7	8-15	16-23	24-31
	A[i] (D ₄)	m_0	m_1	m_2	m_3
	b(D ₅)	m_5			

(b) Memory blocks and temporal scopes in different loop regions

Figure 3.2. A running example

3.4.2 Analysis Unit

In this subsection, we introduce a new analysis unit for our multiple preemption CRPD analysis. According to the observation in Section 3.3, in order to accurately compute the total CRPD for multiple preemptions, we need to re-compute the UCBs for subsequent preemptions of each program point. However, the computation of the subsequent UCBs of each program point is very complex. The subsequent preemption cost of a DAB is different when previous preemptions occur at different program points. It would require to enumerate all the possible preemption sequences to precisely decide which one among them leads to the largest total CRPD, which is computationally intractable.

A UMB may only be double counted by different program points in a certain program region. In order to reduce the computational complexity, instead of computing the subsequent UCBs of a DAB affected by all the preceding preemption points, we shrink the analysis region. More specifically, we only compute the UCBs for the subsequent preemptions of a DAB affected by the DABs in a certain program region. Since different DABs in

the same loop have similar preceding and succeeding DABs. Therefore, they have similar RCS and LCS. As a result, all the DABs in the same loop may have much more common UMBs, and thus may have the same UMBs which may be double counted. In order to capture the DABs in the same loop, we define the loop region (LR) as follows and use it as our analysis unit.

Definition 3.4.1 (Loop region, LR). *A loop region is a set of DABs. The set either contains only one DAB that does not reside in any loop, or all the DABs that reside in an outmost loop.*

With different previous preemption points, different redundant UMBs of a DAB should be excluded in the subsequent preemptions. Therefore, an accurate UMBs computation of a DAB in subsequent preemptions should be performed every time when the previous preemption points are different. However, the redundant UMBs of a DAB in subsequent preemptions may be not much different if previous preemption points are in the same loop region as the DAB. Thus, an accurate UMBs of a DAB in subsequent preemptions can be obtained by one computation (only consider these same redundant UMBs).

For the example in Fig. 3.2-(a), the data reference in different LR is shown in Fig. 3.2-(b). D_0 is a DAB not residing in any loop, so it forms a loop region LR_0 itself. Similarly, D_3 forms a loop region LR_2 . The outmost loop L_1 contains D_1 and D_2 , so $LR_1 = \{D_1, D_2\}$, and similarly $LR_3 = \{D_4, D_5\}$. D_7 does not reside in any loop and form a loop region itself, since it does not contain any data reference, we omit it in Fig. 3.2-(b) for simplicity.

3.4.3 Useful Memory Block Classification

As shown in Section 3.3, the UCBs of a program point may be less if a preemption occurred before, since some UMBs may be evicted from the cache by the previous preemptions. In this subsection, we analyze which UMBs can be counted in the subsequent preemptions, and classify the UMBs of a DAB into three categories.

All the existing works consider a memory block that resides in both RCS_D and LCS_D as a UMB of D . However, a UMB of a DAB may be evicted from the cache by previous preemptions, and it can not be counted as a UMB for the DAB in subsequent preemptions, unless it will be loaded back into cache when the program reaches the DAB after resuming execution from previous preemptions. In order to find whether a UMB will be loaded back into the cache when the program resumes execution, we use a new notation to denote the UMB of a DAB D_i :

$$m\{S_{rcs}, S_{lcs}\}$$

where S_{rcs} denotes the scope of TS_m^D from RCS_{D_i} , which records the temporal scope of the latest access to m when the program reaches D_i . S_{lcs} denotes the scope of TS_m^D from LCS_{D_i} , which records the temporal scope of the first access to m when the program leaves D_i . For the example in Fig 3.2, m_0 is the UMB of D_2 , since it resides in RCS_{D_2} and $\text{LCS}_{D_2}^2$, which is denoted by:

$$m_0\{(D_2, L_1(0, 7)), (D_4, L_2(0, 7))\}$$

For the first preemption analysis, we define a vector $U^{1st}_{D_i}$ for a $\text{RCS}_{D_i}^a$ and a $\text{LCS}_{D_i}^b$, where a $\text{RCS}_{D_i}^a$ is a TS-ACS of RCS_{D_i} and a $\text{LCS}_{D_i}^b$ is a TS-ACS of LCS_{D_i} . The vector contains n elements $U_{D_i}^{1st}[0, \dots, n-1]$ (n is the number of cache sets). Each element $U_{D_i}^{1st}[x]$ in $U_{D_i}^{1st}$ is defined as:

$$U_{D_i}^{1st}[x] = \left\{ m\{S_{rcs}, S_{lcs}\} \mid m \in \text{RCS}_{D_i}^a[x] \wedge m \in \text{LCS}_{D_i}^b[x] \right\}$$

If there is no UMB in $U_{D_i}^{1st}[x]$, $U_{D_i}^{1st}[x] = \perp$.

A DAB may have several U^{1st} since a RCS or an LCS may contain several TS-ACS.

According to the RCS_{D_2} and the $LCS_{D_2}^1$ of the example in Fig 3.2, we have:

$$\begin{aligned} \mathbf{U}_{D_2}^{1st^1} = & \{(\overline{m_0}\{(D_2, L_1(0, 7)), (D_2, L_1(0, 7))\}), \\ & (\overline{m_1}\{(D_2, L_1(8, 15)), D_2, L_1(8, 15)\}), \\ & (\overline{m_2}\{(D_2, L_1(16, 23)), D_2, L_1(16, 23)\}), \\ & (\overline{m_3}\{(D_2, L_1(24, 31)), D_2, L_1(24, 31)\}), \\ & (\overline{m_4}\{(D_0, \emptyset), (D_3, \emptyset)\}), \\ & (\overline{m_5}\{(D_1, L_1(0, 31)), D_1, L_1(0, 31)\}), \perp, \perp\} \end{aligned}$$

According to the RCS_{D_2} and the $LCS_{D_2}^2$, we have:

$$\begin{aligned} \mathbf{U}_{D_2}^{1st^2} = & \{(\overline{m_0}\{(D_2, L_1(0, 7)), (D_4, L_2(0, 7))\}), \\ & (\overline{m_1}\{(D_2, L_1(8, 15)), D_4, L_2(8, 15)\}), \\ & (\overline{m_2}\{(D_2, L_1(16, 23)), D_4, L_2(16, 23)\}), \\ & (\overline{m_3}\{(D_2, L_1(24, 31)), D_4, L_2(24, 31)\}), \\ & (\overline{m_4}\{(D_0, \emptyset), (D_3, \emptyset)\}), \\ & (\overline{m_5}\{(D_1, L_1(0, 31)), D_5, L_2(0, 31)\}), \perp, \perp\} \end{aligned}$$

We classify UMBs of a DAB D_i into the following three categories:

1. Outside UMB (O-UMB): The S_{rcs} of the memory block is in a different loop region from D_i .
2. Inside UMB (I-UMB): The S_{rcs} is a must scope and in the same loop region as D_i .
3. Other UMB: The rest of UMBs excluding the O-UMBs and I-UMBs.

Since a memory block is considered as an O-UMB of a DAB D_i only depends on its S_{rcs} , where two O-UMBs $m_1\{S_{rcs1}, S_{lcs1}\}$ and $m_2\{S_{rcs2}, S_{lcs2}\}$ are equal iff

$$m_1 = m_2 \wedge S_{rcs1} = S_{rcs2}$$

Similarly, two I-UMBs $m_3\{S_{rcs3}, S_{lcs3}\}$ and $m_4\{S_{rcs4}, S_{lcs4}\}$ are equal iff

$$m_3 = m_4 \wedge S_{rcs3} = S_{rcs4}$$

An O-UMB counted as a UMB of D_i in the previous preemptions should not be counted as a UMB again for the subsequent preemptions of all the DABs that are in the loop region of D_i . This is because the data reference loading the UMB into the cache is in a different loop region from D_i , and the data reference will never be re-visited during the execution of all the DABs in this loop region. For example, the O-UMB $\overline{m}_4\{(D_0, \emptyset), (D_3, \emptyset)\}$ of D_2 is loaded into the cache by the data reference in D_0 which is in a different loop region from D_2 . D_0 will never be re-visited during the execution from the first visit of D_1 or D_2 to the latest visit of D_1 or D_2 . Thus, the UMB $\overline{m}_4\{(D_0, \emptyset), (D_3, \emptyset)\}$ of D_2 cannot be counted as a UMB again for D_2 and D_1 in their subsequent preemptions if it is counted as a UMB of D_2 in the previous preemption.

Assuming j I-UMBs of a DAB D_i accessed by a must data reference have been evicted from the cache, according to the definition of *Must* data reference we know that, among these j I-UMBs at most one memory block can be loaded into the cache when the program resumes execution. Thus if these j memory blocks are already counted as UMBs of D_i in its first preemption, among them, at most one memory block should be counted a UMB of D_i in its subsequent preemptions.

In summary, if the first preemption occurred at D_i , its O-UMBs can be used to precisely find whether an O-UMB of a DAB in the same loop region with D_i can be counted as a UMB again in the subsequent preemptions; its I-UMBs can be used to bound the number of I-UMBs of a DAB in the same loop region with D_i in the subsequent preemptions.

3.5 Useful Cache Block Analysis for Subsequent Preemptions

We define the loop region as our analysis unit since the DABs in it may have the same redundant UMBs. With the analysis unit, we can get an accurate UCBs of a DAB in subsequent preemptions by only performing the subsequent preemption analysis once. In this section, according to the different categories of UMBs, we compute the UCBs of each DAB for the subsequent preemptions and compute the total CRPD of the program for multiple preemptions.

Similar with the first preemption analysis, we define vectors $U^{2nd}_{D_i}$ of a DAB D_i for subsequent preemption analysis. A $U^{2nd}_{D_i}$ records all the memory blocks that should be counted as UMBs in the subsequent preemptions. Initially, we copy all the $U^{1st}_{D_i}$ of D_i and re-name them as $U^{2nd}_{D_i}$. In our method, we mainly analyze two types of UMB: (1) O-UMB: the UMBs in the subsequent preemptions that we can precisely identify. and (2) I-UMB: the UMBs in the subsequent preemptions that we cannot precisely identify but can bound their total number. In the following we will delete O-UMBs that should not be counted as UMBs in the subsequent preemptions from $U^{2nd}_{D_i}$, and compute how many I-UMBs in $U^{2nd}_{D_i}$ could be counted as UMBs in the subsequent preemptions.

3.5.1 O-UMB Analysis for Subsequent Preemptions

According to the discussion of O-UMB in Section 3.4.3, if the program is preempted at D_i before, an O-UMB of D_i should not be counted as a UMB for all the DABs in the same loop region as D_i for the subsequent preemption analysis. However, an O-UMB of D_i may not always be counted as a UMB if the program is preempted at D_i . This is because a DAB may have several $U^{1st}_{D_i}$, and we only discuss one $U^{1st}_{D_i}$ when computing the UMBs. We define an O-UMB set OA_{D_i} for a DAB D_i as:

$$OA_{D_i} = \bigcap_{\forall U \in U^{1st}_{D_i}} U(\text{O-UMB})$$

where $U(\text{O-UMB})$ denotes all the O-UMBs in U . Thus, a memory block in OA_{D_i} is always counted as a UMB of D_i in its first preemption. The equality between two O-UMBs depends

on their scopes in RCS. For simplicity, we omit the S_{lcs} of all the memory blocks in OA_{D_i} .

For the example in Fig 3.2, according to $\text{U}^{1st}_{D_2}$ shown in Section 3.4.1, we have:

$$\text{U}^{1st}_{D_2}(\text{O-UMB}) = \text{U}^{1st^2}_{D_2}(\text{O-UMB}) = \{(\overline{m_4}\{(D_0, \emptyset)\})\}$$

So OA_{D_2} is

$$\text{OA}_{D_2} = \{(\overline{m_4}\{(D_0)\})\}$$

Memory blocks in OA_{D_i} should not be counted as UMBs for all the DABs that reside in the same loop region with D_i , if the first preemption occurred at D_i .

In order to find the O-UMBs that should not be counted as UMB for all the DABs that reside in the same loop region with D_i , if the first preemption occurs at anywhere of the loop region, we define an O-UMB set for a loop region LR_i as:

$$\text{OA}_{\text{LR}_i} = \bigcap_{\forall D \in \text{LR}_i} \text{OA}_D$$

The OA_{LR_i} only contains the O-UMBs that already be counted as a UMB for first preemption occurring at anywhere in LR_i . Specifically, the memory blocks in OA_{LR_i} have already been considered to be evicted from the cache by the first preemption that occurs at anywhere of LR_i . Therefore, we have the following lemma.

Lemma 3.5.1. *Suppose D_j is a DAB in LR_i , if we delete all the memory blocks that in OA_{LR_i} from $\text{U}^{2nd}_{D_j}$, the remaining memory blocks are an superset of UMBs of D_j that will be re-loaded in to the cache, if the program is preempted at LR_i before.*

Proof. All the memory blocks in OA_{LR_i} are a subset of memory blocks that can not be re-loaded into cache if the program is preempted at LR_i before. This is because memory blocks in OA_{LR_i} are accessed by the data references that is in the different loop region with LR_i . Therefore, if deleting all the memory blocks in OA_{LR_i} from U^{2nd} of each DAB D_j in LR_i , the remaining memory blocks are an superset of UMBs of D_j that will be re-loaded in to the cache, if the program is preempted at LR_i before.

□

In our example, similar to D_2 , we have:

$$\text{OA}_{D_1} = \{(\overline{m_4}\{(D_0)\})\}$$

Since loop region LR_1 only contains D_1 and D_2 , we have:

$$\text{OA}_{\text{LR}_1} = \{(\overline{m_4}\{(D_0)\})\}$$

3.5.2 I-UMB Analysis for Subsequent Preemptions

According to the discussion of I-UMB in section 3.4.3, among all I-UMBs of a DAB D_i accessed by a *Must* data reference, at most one memory block should be counted as a UMB of the DABs in the same loop region with D_i for each subsequent preemption, if the first preemption occurred at D_i . Similar with the O-UMB analysis, we define an I-UMB set IA_{D_i} for a DAB D_i as :

$$\text{IA}_{D_i} = \bigcap_{\forall U \in \text{U}^{\text{1st}}_{D_i}} \text{U(I-UMB)}$$

where U(I-UMB) denotes all the I-UMBs in U . All the I-UMBs in IA_{D_i} must be counted as UMB of D_i in its first preemption analysis. Since different DABs may have different I-UMBs, we define the I-UMB set IA_{LR_i} for a loop region LR_i as:

$$\text{IA}_{\text{LR}_i} = \bigcap_{\forall D \in \text{LR}_i} \text{IA}_D$$

Memory blocks in IA_{LR_i} are all counted as UMBs by the first preemption analysis if the first preemption occurred at anyplace of LR_i . By IA_{LR_i} we have the following lemma.

Lemma 3.5.2. *Assuming memory blocks in IA_{LR_i} are accessed by l different *Must* data reference. If all the memory blocks in IA_{LR_i} are evicted from the cache by a preemption that occurred at LR_i , at most l memory blocks can be re-loaded into the cache during the following execution of LR_i .*

Proof. If memory blocks accessed by a must data reference are evicted from the cache, at most one of these memory blocks can be re-loaded into the cache after the program resumes execution. Therefore, among the evicted memory blocks accessed by l different *Must* data references, at most l memory blocks can be re-loaded into the cache during the following execution. \square

Among all the memory blocks in \mathbf{IA}_{LR_i} , we can get that at most l memory blocks should be counted as UMBs in a subsequent preemption occurring at a DAB in LR_i , but can not decide which memory block should be counted as a UMB in subsequent preemptions. This is because, a UMB in \mathbf{IA}_{LR_i} may be evicted from the cache by the first preemption or the subsequent preemptions. Therefore, when computing which UMB may be evicted from the cache in the subsequent preemptions, we consider all of them to be UMBs, but only l memory blocks can be evicted from the cache.

Since LR_1 only contains D_1 and D_2 , and \mathbf{IA}_{D_1} equals \mathbf{IA}_{D_2} , we have:

$$\begin{aligned} \mathbf{IA}_{LR_1} = \mathbf{IA}_{D_1} = \mathbf{IA}_{D_2} = & \{(\overline{m}_0\{(D_2, L_1(0, 7))\}), \\ & (\overline{m}_1\{(D_2, L_1(8, 15))\}), (\overline{m}_2\{(D_2, L_1(16, 23))\}), \\ & (\overline{m}_3\{(D_2, L_1(24, 31))\}), (\overline{m}_5\{(D_1, L_1(0, 31))\}) \} \end{aligned}$$

3.5.3 UCB Calculation for Subsequent Preemptions

In this subsection we discuss how to compute U^{2nd} of each DAB using \mathbf{IA}_{LR_i} and \mathbf{OA}_{LR_i} . The U^{2nd} of each DAB is initially set to be the same as U^{1st} . According to lemma 3.5.1, all the memory blocks in \mathbf{OA}_{LR_i} should not be counted as UMBs anymore for subsequent preemptions of all the DABs in LR_i . For all the UMBs in \mathbf{OA}_{LR_i} , we delete all the corresponding UMBs from U^{2nd} s of all the DABs in LR_i . By Lemma 3.5.2, we can compute the number of I-UMBs that could be counted as UMB in the subsequent preemptions. For all the UMBs in \mathbf{IA}_{LR_i} , we label the corresponding UMBs in U^{2nd} s of all the DABs in LR_i with I (UMB^I).

We define ECB as the set of cache blocks that will be visited by all the preempting tasks. For each DAB D_i , we compute the number of cache blocks that may be evicted from the cache by the first preemption occurring at D_i and denote the number as $UCB_{D_i}^{1st}$. For each U^{1st} of D_i , we count the number of $U^{1st}[x]$ s ($0 \leq x < n$) in U^{1st} which satisfies:

$$U^{1st}_{D_i}[x] \neq \perp \wedge c_x \in ECB$$

Among all the U^{1st} of D_i , $UCB_{D_i}^{1st}$ equals the maximum one.

For each DAB D_i , we also compute the number of cache blocks that may be evicted from the cache by the subsequent preemptions occurring at D_i and denote the number as $UCB_{D_i}^{2nd}$. For each $U^{2nd^j}_{D_i}$ of D_i , we initially set $UCB_{D_i}^{2nd^j}$ equals to 0. For all the $UCB^{2nd^j}_{D_i}[x]$ of $UCB^{2nd^j}_{D_i}$ that contains a UMB not labeled with I , we count the number of $UCB^{2nd^j}_{D_i}[x]$ s which satisfies:

$$U^{2nd}_{D_i}[x] \neq \perp \wedge c_x \in ECB$$

and add the count to $UCB^{2nd^j}_{D_i}$. For all the $U^{2nd}_{D_i}[x_1, x_2, \dots, x_p]$ s that only contain UMB^I s loaded into the cache by same *Must* data reference, if

$$\exists l \in [0, p], c_{x_l} \in ECB$$

$UCB^{2nd^j}_{D_i}$ adds by one. When all the $UCB^{2nd^j}_{D_i}$ are computed, we set $UCB_{D_i}^{2nd}$ equal to the largest $UCB^{2nd^j}_{D_i}$. For the UCB^{2nd} we have the following lemma.

Lemma 3.5.3. $UCB_{D_i}^{2nd}$ is safe to bound the number of UCBs for the preemption occurring at D_i if there was a preemption occurring at the same loop region with D_i before.

Proof. If there was a preemption occurring at the loop region of D_i . By lemma 3.5.1, the remaining memory blocks in $U_{D_i}^{2nd}$ is a superset of UMBs that can be counted as UMB in the subsequent preemptions. By lemma 3.5.2, among all the I-UMBs in $U_{D_i}^{2nd}$, only l (the number of must data references of these I-UMBs) I-UMBs be counted as UMB in the subsequent

preemptions. So the $\text{UCB}_{D_i}^{2\text{nd}}$ is safe to bound the number of UCBs of D_i if there was a preemption occurring at anyplace in its loop region . \square

For the running example, we have IA_{LR_1} and OA_{LR_1} as presented in Section 3.5.2 and Section 3.5.1 respectively. Then we can get $U^{2\text{nd}}_{D_2}$:

$$\begin{aligned}
\mathbf{U}^{2\text{nd}^1}_{D_2} = & \{(\overline{m}_0^I \{(D_2, L_1(0, 7)), (D_2, L_1(0, 7))\}), \\
& (\overline{m}_1^I \{(D_2, L_1(8, 15)), D_2, L_1(8, 15)\}), \\
& (\overline{m}_2^I \{(D_2, L_1(16, 23)), D_2, L_1(16, 23)\}), \\
& (\overline{m}_3^I \{(D_2, L_1(24, 31)), D_2, L_1(24, 31)\}), \\
& , \perp, (\overline{m}_5^I \{(D_1, L_1(0, 31)), D_1, L_1(0, 31)\}), \perp, \perp\} \\
\mathbf{U}^{2\text{nd}^2}_{D_2} = & \{(\overline{m}_0^I \{(D_2, L_1(0, 7)), (D_4, L_2(0, 7))\}), \\
& (\overline{m}_1^I \{(D_2, L_1(8, 15)), D_4, L_2(8, 15)\}), \\
& (\overline{m}_2^I \{(D_2, L_1(16, 23)), D_4, L_2(16, 23)\}), \\
& (\overline{m}_3^I \{(D_2, L_1(24, 31)), D_4, L_2(24, 31)\}), \\
& \perp, (\overline{m}_5^I \{(D_1, L_1(0, 31)), D_5, L_2(0, 31)\}), \perp, \perp\}
\end{aligned}$$

In the example, all the UMBs in $\mathbf{U}^{1\text{st}^1}_{D_2}$ and $\mathbf{U}^{2\text{nd}^1}_{D_2}$ are labeled with I , and loaded into the cache by two different *Must* data references, A in D_2 and b in D_1 . Therefore, $\text{UCB}_{D_2}^{2\text{nd}}$ is 2. Assuming all the UMBs will be evicted from the cache by the preempting task, the CRPD of D_2 in each of its subsequent preemption is $2 \times t_{\text{reload}}$, which is much less than $6 \times t_{\text{reload}}$ produced by [71].

3.5.4 Total CRPD Analysis

In this subsection we introduce an algorithm to compute the total CRPD of k -preemptions with $\text{UCB}^{1\text{st}}$ and $\text{UCB}^{2\text{nd}}$ of each DAB. Similar with the state-of-the-art [7, 37], we still pick the k -largest number of UCBs to compute the total CRPD, but in our work the number of UCBs of each DAB is not fixed.

When selecting the k -largest number of UCBs, we first choose $\text{UCB}^{1\text{st}}$ of every DAB. After a $\text{UCB}_{D_i}^{1\text{st}}$ is selected, we adopt the $\text{UCB}^{2\text{nd}}$ of all the DABs that is in the same loop region as D_i . Iteratively, we select k UCBs for k preemptions. A DAB residing in a loop should be taken into account as often as the loop iterates. In our algorithm, we use T_{D_i} to denote the number of times that D_i should be taken into account. Obviously, T_{D_i} is no less than 1. The algorithm is defined as follows.

Algorithm 2 $K - \text{CRPD}$

- 1: For each DAB, set its UCB equal to its $\text{UCB}^{1\text{st}}$
 - 2: **for** $i=0 ; i < k ; i++$ **do**
 - 3: Chose the largest UCB of all the DABs D_i .
 - 4: **if** $T_{D_i} \neq 0$ **then**
 - 5: $T_{D_i} --$;
 - 6: The i -th largest UCB is set as UCB_{D_i}
 - 7: **else**
 - 8: $i-1$
 - 9: **end if**
 - 10: For the selected DAB, the UCB of the DABs which have the same loop region with it is set to $\text{UCB}^{2\text{nd}}$.
 - 11: **end for**
-

The selected k -largest UCBs by Algorithm 2 may belong to DABs of different LRs. For each LR, there must exist one $\text{UCB}^{1\text{st}}$ and several $\text{UCB}^{2\text{nd}}$ s. The $\text{UCB}^{1\text{st}}$ must no less than any $\text{UCB}^{1\text{st}}$ of a DAB in the LR. These $\text{UCB}^{2\text{nd}}$ s must be no less than $\text{UCB}^{2\text{nd}}$ s of other DABs in the LR. Then we have the following proof.

Proof. For the k selected UCBs:

1. The largest UCB must be selected from a $\text{UCB}_{D_i}^{1\text{st}}$. It is safe to bound the number of UCBs if only one preemption occur.

2. If the second largest UCB is $UCB_{D_j}^{2nd}$, D_i and D_j must be in the same loop region. $UCB_{D_j}^{2nd}$ must be no less than UCB^{2nd} s of all the DABs in the same loop region with D_j . So if exist one preemption occurring in different loop region with D_i , the two selected UCBs are safe. This is because both $UCB_{D_i}^{1st}$ and $UCB_{D_j}^{2nd}$ are no less than UCB^{1st} of DABs that in different loop region with D_i . Also, $UCB_{D_j}^{2nd}$ must be the largest UCB^{2nd} among all the DABs that is in the loop region of D_i . By lemma 3.5.3, if the two preemptions both occurring in the loop region of D_i , the two selected UCBs are safe.
3. If the second largest UCB is $UCB_{D_j}^{1st}$, D_j must be in the different loop region with D_i . $UCB_{D_j}^{1st}$ must be no less than UCB^{1st} s of all the DABs in the different loop region with D_i . So if exist one preemption occurring in the different loop region of D_i , the two selected UCB are safe. Also the $UCB_{D_j}^{1st}$ must be no less than the largest UCB^{2nd} s of all the DABs in the same loop region with D_i . Therefore, the two selected UCBs are safe if the two preemptions are both occurring at the loop region of D_i .
4. By 2 and 3, if there are two preemptions, the two selected UCB are safe to bound the total number of UCBs. Repeating the above reasoning iteratively, we know the k selected UCBs can bound the total UCBs for k preemptions.

□

UCB^{1st} and UCB^{2nd} of each DAB for the example program in Fig 3.2 shown in the following Table.

According to algorithm 2, we first select the largest UCB^{1st} (i.e., 6), which belongs to D_2 and D_3 . Since D_2 and D_3 both reside in the same loop region, no matter which one is selected, we will adopt their UCB^{2nd} in the following. Thus, the second largest number of

DAB	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
UCB ^{1st}	1	6	6	5	5	5	5	0
UCB ^{2nd}	1	2	2	0	2	2	2	0
Execution counts	1	32	32	1	32	32	32	1

UCBs is 5 which belongs to D_4 , D_5 and D_6 . If selecting D_4 as the second preemption point, the third largest number of UCBs should be selected from D_5 or D_6 (i.e., 5), because D_4 can only be selected once. If selecting D_5 or D_6 as the second preemption point, we adopt their UCB^{2nd} in the following. Since UCB^{2nd} of D_5 and D_6 are less than the UCB^{1st} of D_4 , the third largest number of UCB should be selected from D_4 (i.e., 5). If the program in this example will be preempted 8 times, the total preemption cost is:

$$t_{\text{reload}} \times (6 + 5 + 5 + 2 \times 5) = 26 \times t_{\text{reload}}$$

However, the total CRPD for 8 preemptions estimated by the state-of-the-art methods [7, 37] is:

$$t_{\text{reload}} \times 6 \times 8 = 48 \times t_{\text{reload}}$$

Compared with the state-of-the-art method, our method approximately reduces half of total CRPD for 8 preemptions, which can greatly improve the system-level analysis precision.

3.5.5 Discussion

The complexity of the proposed method is bounded by $O((S * L * M)^E * D + M * D + D^2)$, where S is the number of cache sets, L is the maximal level of loop nesting, M is the number of memory blocks, D is the number of DABs, and E is the number of conditional branches in program. Our multiple preemption CPRD analysis includes three steps. First, we adopt [71] to do the first preemption analysis for each DAB, the computational complexity

of which is $O((S * L * M)^E * D)$ as presented in [71]. In the second step, we classify all the UMBs of each DAB as O-UMBs and I-UMBs, which complexity is $O(M * D)$. Third, we select k -largest $UCBs$ from $2 * D$ $UCBs$, the complexity is $O(D^2)$. The state-of-the-art method only perform the first preemption analysis for each DAB, and its computation complexity is $O((S * L * M)^E * D + D^2)$. Compared with it, the additional computation complexity of our approach is $O(D * M)$, which is in general much less than the state-of-the-art method.

3.6 Evaluation

In this section, we evaluate the proposed multiple preemption CRPD analysis approach with 10 WCET benchmarks which are selected from Mälardalen WCET benchmark [2]. The Mälardalen WCET benchmark is the one of the most widely used benchmarks for WCET analysis and CRPD analysis. For the general interests of data cache analysis, among 20 programs in the benchmark suit that have data memory accesses, we (i) exclude programs with very limited number of data memory accesses (e.g., "qurt", "minver"); and (ii) select representatives for programs with identical data memory access behaviors (e.g., among various bubble sort or selection sort based programs). These 10 benchmarks used in our evaluation explore different program structures and memory access patterns, including nested loops, non-rectangular loop nests, row/column-based matrix accesses, data/input dependent branches, etc. Table 3.1 shows the benchmark name, a brief description and the input array size for each benchmark.

In the experiment, we first follow [71] to get the UCB^{1st} of each DAB. Then, we compute the UCB^{2nd} of each DAB using the proposed method. With UCB^{1st} and UCB^{2nd} of each DAB, the total number of UCBs for multiple preemptions can be computed by Algorithm 2. Assume that each benchmark executes on a SimpleScalar processor with 5-stage pipeline, in-order execution, perfect branch prediction, and separate L1 instruction and data caches. In order to get an sufficient evaluation, we use two different cache configurations, the data cache with 512 cache blocks and 256 cache blocks, the instruction cache has the

Table 3.1. Benchmark descriptions and array sizes

Benchmark	Benchmark description	Array size
Adpcm	Adaptive pulse code modulation	2048
Lms	adaptive signal enhancement	1024
Matmult	Matrix multiplication	24×24
Cnt	Count non-negative in matrix	64×64
Bsort100	Bubblesort program	1024
Edn	(FIR) filter calculations	1024
Ns	Search multi-dimensional array	32×64
St	Statistics program	1024
Fir	Finite impulse response filter	720
Insertsort	Insertion sort	1024

same size with the data cache. Both instruction and data caches are direct-mapped, and each block has block size $32B$.

The number of UCBs is widely used to bound the CRPD of preempted tasks if the preempting tasks are nondeterministic. We compare the total UCBs produced by our approach and the state-of-the-art method with different numbers of preemptions. The state-of-the-art method sums the k -largest number of UCBs of different program points produced by [71] as the total UCBs of k preemptions. The analysis result of different approaches are shown in Fig. 3.3. The legend entries shown in the bottom right corner represent the approach and corresponding cache configuration. For example, the "our approach 512" means that our approach is performed with a data cache containing 512 blocks. According to the

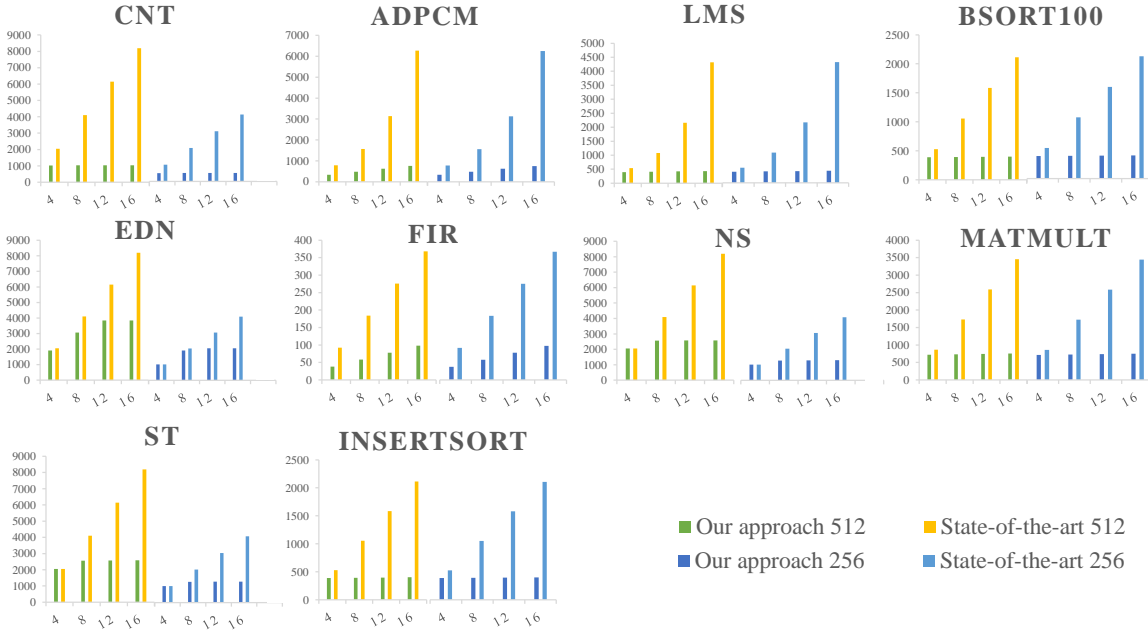


Figure 3.3. Total number of UCBs of different benchmarks with different numbers of preemptions

experimental result, we can conclude that, no matter how much the cache size is, our approach can significantly reduce the total number of UCBs of each benchmark with different number of preemptions.

Our approach has a large improvement on benchmarks "cnt", "bsort100", "insert-sort", "fir" and "ns" no matter how many preemptions there are. In these five benchmarks, the largest number of UCBs all belongs to the DAB in a loop. So the state-of-the-art method computes the total number of UCB by multiplying the largest UCB^{1st} and the number of preemptions. Since these programs only contain *Must* data references, in our method, the UCB^{2nd} of the DABs that reside in a loop equals the number of *Must* data references in its loop region. Therefore, UCB^{2nd} of each DAB is much less than its UCB^{1st} . Thus, the total number of UCBs produced by our approach is much less than the state-of-the-art method.

Our approach has a small improvement on "Edn" when the number of preemptions is less than 8. Benchmark "Edn" contains 13 loop regions and the largest 8 UCB^{1st} of different loop regions are not much different with each other. Therefore, the sum of the largest 8 UCB^{1st} of different loop regions is not much different with the largest UCB^{1st}

Table 3.2. Total CRPD of tasks preempted by different preempting tasks

preempted task	preempting tasks	our approach	state-of-the-art
Fir	Bsort100,St,Insertsort,Lms	3800ns	9200ns
Adpcm	Matmult,Ns,Edn,Cnt	33800ns	52400ns
Cnt	Insertsort,St,Ns,Edn	102600ns	204800ns
Insertsort	Matmult,St,Lms,Cnt	38900ns	52800ns
Lms	Insertsort,St,Ns,Cnt	39600ns	54000ns

multiplied by 8. However our method can also get a large improvement when the number of preemptions increase, because with the increase of the number of preemptions we will select more UCBs from UCB^{2nd} which is much less than UCB^{1st} . Benchmark "Adpcm" has a *May* data reference. Meanwhile, it accesses two *Must* data references in the same loop region as the *May* data reference. Although our method can not identify whether a UMB accessed by a *May* data reference can be counted as a UMB in the subsequent preemptions. Due to the *Must* data references, we can still exclude a certain number of redundant UMBs from UCB^{2nd} . As a result, our method can still get a significant improvement on benchmark "Adpcm".

Among all the benchmarks, "cnt", "st" "ns" and "edn" access more than 256 memory blocks during its execution. Since the number of UCBs must be less than the number of total cache blocks, the number of UCBs decreases when the cache size decreases from 512 to 256. However whether a UMB is a O-UMB or a I-UMB has no relevant with the cache size. So our method can also get significant improvement with different cache sizes.

We also evaluate our method with concrete preempting tasks. In this part, the number of cache blocks of data cache is 512. The preempted task and the preempting tasks are shown

in Table 3.2. Assuming the preempted task will be preempted 4 times in one period and the cache block reloading time is $100ns$, the total CRPD of 4 preemptions computed by our approach and the state-of-the-art method are shown in Table 3.2. When the preempting tasks are deterministic, we can get a more accurate CRPD analysis result. This is because, if a UCB of a preempted task is never accessed by the preempting tasks, it will never contribute to the CRPD. According to the analysis result shown in Fig 3.2, our method can still get a significant improvement when the preempting tasks are deterministic.

According to the analysis result, we can conclude that, no matter how many preemptions there are, our method can averagely reduce more than 50 percent of total UCBs compared with the state-of-the-art method for most benchmarks. For a small part of benchmarks, our method may have less improvement when the number of preemptions is small. However, when the number of preemptions is larger than a certain number, our method can get a substantial improvement, and the certain number is not as large as unreasonable. Thus, our work can also perform a more precise total CRPD analysis with concrete preempting tasks.

3.7 Conclusion

Extending the existing total instruction CRPD method to total data CRPD analysis may lead to a significant overestimation. In this chapter, we produce a new method to precisely compute the total data CRPD for multiple preemptions. Specifically, we distinguish which memory blocks can be counted as UMB again in the subsequent preemptions based on the first preemption analysis result, and further compute a tight CRPD bound of multiple preemptions. With the proposed approach, we can perform a more accurate timing analysis for the real-time tasks in preemptive systems.

CHAPTER 4

SHARED CACHE CONTENTION ANALYSIS FOR WCET ESTIMATION ON MULTI-CORES

4.1 Introduction

Embedded real-time systems are shifting to multi-core platforms in order to meet both high-performance requirements and strict thermal and power constraints [26,46]. Multi-core processors typically contain a shared cache which is accessed by different cores. Different tasks that execute in parallel on different cores may access the same shared cache line, thus causing shared cache contentions with each other. Similarly to preemptions, shared cache contentions between parallel executing tasks may also cause additional cache misses, and additional execution time. Shared cache makes the WCET analysis problem on multi-cores significantly more difficult than the single-core case [14].

To estimate the WCET of a task on multi-cores, one needs to bound the number of extra cache misses caused by inter-core contention on the shared cache. To guarantee the soundness of the estimated WCET bound, the state-of-the-art method [39] considers that all the potential shared cache contentions between parallel executing tasks happen together, which is too conservative. However, since the access of memory references of a task follows a certain order defined by its control flow graph, not all the potential shared cache contentions can happen together in reality. Moreover, since the execution counts of different memory references are different, when a memory reference (the *interfered* memory reference) contends with other tasks, the shared cache contention may not happen every time the memory reference is accessed. Therefore, the state-of-the-art method, which considers all the access of all the conflicted memory references as shared cache misses is pessimistic.

A detailed discussion on the above-mentioned pessimism with illustrative examples will be given in section 4.4.

Challenges and Contributions. In practice, due to the access order constraints, some potential shared cache contentions never happen together. We call a combination of shared cache contentions a *feasible combination*. In order to derive a safe and tight WCET bound in the presence of shared cache contentions, we should find the feasible combination that causes the maximum shared cache misses (henceforth called *worst-case feasible combination*). Between parallel executing tasks, a huge number of potential shared cache contentions between memory references may be caused, and much more feasible combinations may be further constructed. Therefore, identifying the feasible combinations and further finding the worst-case feasible combination is challenging, which intuitively requires enumerating all the possible combinations. This thesis proposes a novel analysis model that can efficiently find the worst-case feasible combination. In addition, we precisely predict the maximum extra shared cache misses caused by each potential shared cache contention rather than simply considering all the memory accesses of effective memory references as extra shared cache misses. Totally, the maximum extra shared cache misses caused by shared cache contentions are precisely predicted. In addition, we believe that the proposed model can be extended to other shared resource contention analysis. Experimentally, our method significantly tightens the estimated WCET bound over the state-of-the-art with most benchmark programs without compromising efficiency.

4.2 Related Work

The WCET analysis problem was initially studied for single-core processors [68], for which the Abstract Interpretation (*AI*) based method [22] demonstrated to be successful solutions and form the common foundation for later research in cache analysis for WCET estimation. This approach typically includes two phases: first computing abstract cache states of program points by a fixed-point iteration, and second classifying memory references as cache hit or miss according to the abstract cache states. Later, the *AI*-based method is extended and

refined to deal with data caches [31, 60], cache-related preemption delay (CRPD) [5, 71, 72], multi-level caches [28, 61, 74].

The aforementioned works all assumed that the program under analysis owns the entire cache. However, on multi-core processors, this assumption is impractical as co-running tasks on different cores contend with each other on the shared cache. [69] is the first work to consider the shared cache contentions in the context of WCET analysis. It simply reclassifies all the memory references that access the same cache line with other parallel executing tasks as shared cache miss. Later, Liang et al. proposed a more precise method [39] by excluding the shared cache contentions between tasks whose lifetime does not overlap. In addition, their work can also deal with shared set-associative caches while work in [69] can only handle the direct-mapped cache. Liang's work represents the state-of-the-art method as it has the best accuracy when dealing with shared set-associative instruction cache contentions so far [45]. Later, [15] studied the WCET analysis on a timing-anomaly architecture.

All existing works used the similar method to compute extra shared cache misses caused by shared cache contentions, which can be summarized as the following two steps: (1) Assume a single task environment and use multi-level cache analysis techniques [28] to get the cache behavior of memory references at different levels. (2) According to the analysis result of step (1), find the memory references that access the same shared cache lines and change their cache behavior accordingly. Note that in step (2), both [39] and [69] assume all potential shared cache contentions would happen. However, in reality, a large portion of potential shared cache contentions are mutually exclusive and thus should not be counted at the same time. Therefore, [39] and [69] both lead to a pessimistic estimated WCET bound. We will discuss their pessimism in detail in Section 4.4.

[73] pointed that not all the potential shared cache contentions would happen together, and adopted an enumeration-based method to explore all the feasible combinations of shared cache contentions. Due to its low efficiency, this approach is hard to deal with systems that contain more than two cores. Moreover, it is restricted to deal with direct-mapped caches, while we consider set-associative caches.

4.3 Preliminary

This section gives the assumed processor architecture—the platform which the applications run on, and the Control Flow Graph (CFG)—the program model used to describe the analyzed applications. As the necessary background knowledge, the state-of-the-art method [39] is also briefly introduced.

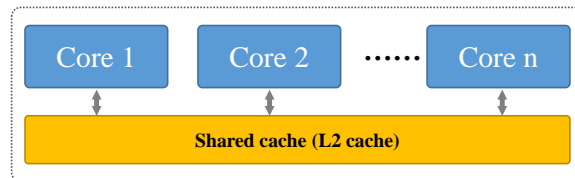


Figure 4.1. A multi-core processor with shared cache

4.3.1 System Model

Since our work focuses on the shared cache analysis, to simplify the presentation, we assume a cache architecture that only contains a single-level shared instruction cache as shown in Figure 4.1. Note, our work can also deal with multi-level caches through a straightforward extension shown in Section 4.6. Real-time systems are generally deployed on a microprocessor which has a simple structure. Similar with most relative works [28, 45], we make the following reasonable assumption:

- the assumed multi-core processor is an in-order processor,
- the relative memory address of each memory reference can be obtained at compile time,
- the processor is a timing-anomaly-free architecture,
- the shared cache is a k -way set-associative cache with least recently used (LRU) replacement policy.

- the data memory references do not interfere with the shared instruction caches¹.

In the set-associative cache, the age of all the memory blocks are assumed to be initialized with k . When a memory block is accessed, its age is changed to 0, and the age of memory blocks with smaller age than it increases by one. Once the age of a memory block increases to k , the memory block is evicted from the cache.

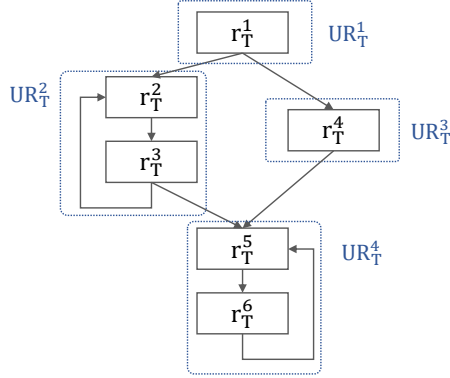


Figure 4.2. Control flow graph (CFG) of task T

This work focuses on the program level analysis rather than system level analysis. We assume that whether tasks are executed in parallel are known in advance, as it can be solved by existing method [39].

We use the Control Flow Graph (CFG) composed by memory references to model each task, where a memory reference corresponds to an instruction at the assembly code level. We show an example of CFG in Figure 4.2. In the example, task T contains 6 memory references. Memory references r_T^2 and r_T^3 reside in the same loop. Memory references r_T^5 and r_T^6 reside in a nested loop, where r_T^5 resides in the inner loop of r_T^6 . Memory references r_T^1 and r_T^4 do not reside in any loop. We assume the loop bounds of each loop are known in advance, and the maximum execution counts of each memory reference can be obtained at compile time. Some notations that will be used are summarized as follows.

¹Similar to [39, 69, 73], this can be serviced from a separate data shared cache or a perfect private data cache.

- r_T^i : A memory reference of task T.
- $C(r_T^i)$: Cache set accessed by r_T^i .
- $x(r_T^i)$: Maximum execution counts of r_T^i .
- k : Associativity of shared cache.
- T_{con} : Set of tasks that may execute with T in parallel.
- c^{hit} : Latency of shared cache hit.
- c^{miss} : Latency of shared cache miss.

4.3.2 The State-of-the-Art Method

Intra-core analysis. The WCET analysis with shared caches includes two steps, intra-core analysis and inter-core analysis. The intra-core analysis computes the WCET of a task T if it owns the whole cache, and its major work is to classify memory references as the following categories:

- **Always Hit (AH)**: The memory block accessed by the memory reference is always in the cache when the memory reference is accessed.
- **Always Miss (AM)**: The memory block accessed by the memory reference is always not in the cache when the memory reference is accessed.
- **Persistent (PS)**: The memory block accessed by the memory reference is always in the cache when the memory reference is accessed except when the memory reference is first accessed.
- **Not classified (NC)**: The memory reference cannot be classified as either AH, AM or PS.

After finishing the classification, we further classify a persistent (PS) memory reference as AM for its first access and AH for the rest of the accesses when computing the WCET. With the analyzed cache behavior of each memory reference, the longest path which leads to the maximum execution time and the execution counts of memory references on the path can be obtained [64]. In the timing-anomaly-free architecture, c^{hit} must be less than c^{miss} . Existing methods generally treat NC memory references as cache miss when computing the WCET bound. Therefore, the intra-core analysis result of task T denoted as $\text{WCET}_{\text{intra-T}}$ is:

$$\text{WCET}_{\text{intra-T}} = x_{\text{PathIntra}}^{\text{hit}} * c^{\text{hit}} + x_{\text{PathIntra}}^{\text{miss}} * c^{\text{miss}} \quad (4.1)$$

where $x_{\text{PathIntra}}^{\text{hit}}$ (resp. $x_{\text{PathIntra}}^{\text{miss}}$) denotes the total execution counts of memory references classified as AH (resp. AM or NC) on the longest path.

Inter-core analysis. Since different tasks executed on different cores may contend with each other on shared cache, the cache behavior analyzed by intra-core analysis should be modified accordingly. The state-of-the-art method performs shared cache contention analysis by reclassifying *conflicting memory references* as NC, where conflicting memory reference is defined as:

Definition 4.3.1 (Conflicting Memory Reference). *A memory reference r_T^i of task T is a conflicting memory reference iff:*

1. *The memory reference is AH in intra-core analysis.*
2. *The sum of the maximum age of the memory reference in intra-core analysis and the counts of memory references that access the same cache set with it is less than the associativity k.*

Based on the new cache behaviors, the longest path in inter-core analysis can also be obtained. Denoting the total number of execution counts of memory references classified as AH (resp. AM or NC) on this longest path as $x_{\text{PathInter}}^{\text{hit}}$ (resp. $x_{\text{PathInter}}^{\text{miss}}$). The WCET bound

can be computed:

$$WCET_T = X_{PathInter}^{hit} * C^{hit} + X_{PathInter}^{miss} * C^{miss} \quad (4.2)$$

By subtracting $WCET_{intra-T}$ from $WCET_T$, the extra execution time caused by shared cache contentions computed by the state-of-the-art method can be obtained:

$$WCET_{inter-T} = (X_{PathInter}^{hit} * -X_{PathIntra}^{hit}) * C^{hit} + (X_{PathInter}^{miss} - X_{PathIntra}^{miss}) * C^{miss} \quad (4.3)$$

Section 4.4 shows that $WCET_{inter-T}$ computed by the state-of-the-art method is pessimistic and section 4.5 presents a new method to obtain a tighter $WCET_{inter-T}$.

4.4 Motivation

This section discloses the pessimism of the state-of-the-art method [39] with a concrete example shown in Figure 4.3. This example shows two parallel executing tasks, T and T'. Task T contains four memory references, r_T^1 , r_T^2 , r_T^3 and r_T^4 . Among them, r_T^1 and r_T^2 reside in one loop, while r_T^3 and r_T^4 reside in another loop. Both these two loops iterate for 10 times. Task T' contains two memory references, $r_{T'}^1$ and $r_{T'}^2$. The mapping among memory blocks, cache sets and memory references is shown in Table 4.1.

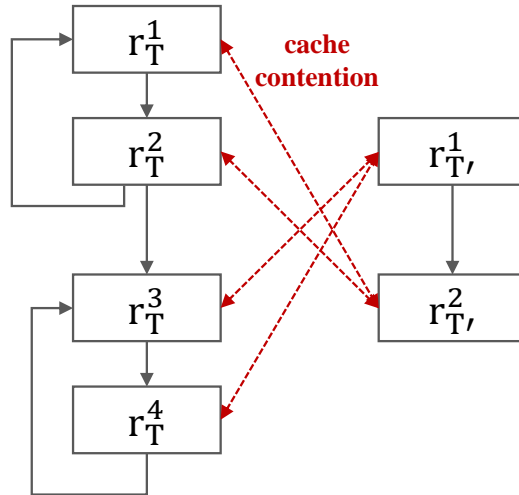


Figure 4.3. CFGs of two parallel tasks T and T'

Table 4.1. Mapping among memory blocks, cache sets and memory references of T and T'

cache set	memory block	memory reference
C_1	m_1	r_T^1
	m_2	r_T^2
	m_3	$r_{T'}^2$
C_2	m_4	r_T^3
	m_5	r_T^4
	m_6	$r_{T'}^1$

Assuming the associativity of the shared cache is 2, the state-of-the-art method classifies r_T^1 , r_T^2 , r_T^3 and r_T^4 as PS in the intra-core analysis. We further classify each of them as AM for its first access and AH for the rest of the accesses. In the following, we use $(r_T^1)'$, $(r_T^2)'$, $(r_T^3)'$ and $(r_T^4)'$ to denote the parts classified as AH respectively. Therefore,

$$WCET_{\text{intra-T}} = 4 * c^{\text{miss}} + 36 * c^{\text{hit}}.$$

Since $r_{T'}^1$ accesses the same cache set with r_T^3 and r_T^4 , and $r_{T'}^2$ accesses the same cache set with r_T^1 and r_T^2 , $(r_T^1)'$, $(r_T^2)'$, $(r_T^3)'$ and $(r_T^4)'$ are all conflicting memory references and all of them are reclassified as NC. Therefore,

$$WCET_T = 40 * c^{\text{miss}}$$

By subtracting $WCET_{\text{intra-T}}$ from $WCET_T$, the extra execution time caused by shared cache contentions is:

$$WCET_{\text{inter-T}} = 36 * (c^{\text{miss}} - c^{\text{hit}}).$$

4.4.1 Overestimation I

As mentioned above, the state-of-the-art method modifies the cache behavior of a conflicting memory reference as NC. If the conflicting memory reference is accessed several times (i.e., reside in a loop), its every memory access is reclassified as NC and considered as shared cache miss in the WCET computation. However, the shared cache contention may not happen every time the conflicting memory reference is accessed, and some accesses of it may still be shared cache hit. For the example in Figure 4.3, $(r_T^4)'$ is a conflicting memory reference and contends with r_T^1 . The state-of-the-art method reclassifies all the 9 accesses of $(r_T^4)'$ as NC. However, r_T^1 is only accessed by T' once, and only contends with $(r_T^4)'$ once. Assuming r_T^1 is accessed just after the 2nd access of r_T^4 , the cache states of C_2 are shown in Figure 4.4.

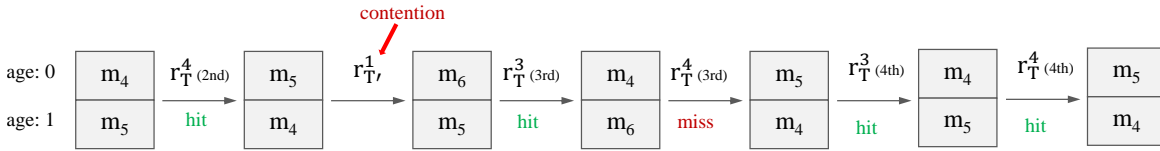


Figure 4.4. Cache states of C_1

According to the cache states, shared cache contention causes the 3rd access of r_T^4 to miss the cache, and the following accesses of r_T^4 are still cache hit. Thus, we have the following observation.

Observation 3. *Shared cache contention may not happen every time a conflicting memory reference is accessed. Thus, considering all the accesses of a conflicting memory reference as shared cache miss is pessimistic.*

4.4.2 Overestimation II

The state-of-the-art method reclassifies all the conflicting memory references as NC. More specifically, between two tasks, it considers that all the potential shared cache contentions

between memory references can happen together. However, due to the certain access order among memory references defined by the control flow graph, some potential shared cache contentions never happen together in practice. For the example in Figure 4.3, the state-of-the-art method reclassifies both $(r_T^1)'$ and $(r_T^3)'$ as NC. Assuming shared cache contention between $(r_T^3)'$ and $r_{T'}^1$ happens first, T must execute to r_T^3 and r_T^1 will never be accessed. Therefore, shared cache contention between $(r_T^1)'$ and $r_{T'}^2$ will not happen. Similarly, if shared cache contention between $(r_T^1)'$ and $r_{T'}^2$ happened, shared cache contention between $(r_T^3)'$ and $r_{T'}^1$ never happen. To summarize, shared cache contention between $(r_T^1)'$ and $r_{T'}^2$ and shared cache contention between $(r_T^3)'$ and $r_{T'}^1$ can not happen together.

By examining all the four potential shared cache contentions shown in Fig. 4.3, we can easily conclude that at most two of them can happen in reality (i.e., either shared cache contention between $(r_T^1)'$ and $r_{T'}^2$ and shared cache contention between $(r_T^2)'$ and $r_{T'}^2$, or shared cache contention between $(r_T^3)'$ and $r_{T'}^1$ and shared cache contention $(r_T^4)'$ and $r_{T'}^1$). Therefore, we have the following observation:

Observation 4. *Some shared cache contentions between memory references of parallel tasks never happen together. The method that reclassifies all the conflicting memory references as NC may overestimate the WCET.*

4.4.3 Summary and Discussion

Above observations show that, the pessimism of the state-of-the-art method is twofold. First, it overestimates the extra number of shared cache misses caused by each potential shared cache contention. For the example in Figure 4.3, shared cache contentions cause at most one access of each memory reference of T to miss the shared cache, but the state-of-the-art method reclassifies all its 9 memory accesses as NC. Second, it considers that all the potential shared cache contentions happen together, thus counting some infeasible combinations of shared cache contentions. For the example, among these four potential shared cache contentions between T and T', the state-of-the-art method considers that all of them happen

together, and sums the extra execution time caused by them as $WCET_{inter-T}$. However, at most two potential shared cache contention can happen in reality, and the maximum shared cache misses caused by shared cache contentions is 2. Therefore:

$$WCET_{inter-T} = 2 * (c^{miss} - c^{hit})$$

which is significantly less than $36 * (c^{miss} - c^{hit})$ analyzed by the state-of-the-art method.

To find the worst-case feasible combination for a tight WCET estimation, two major problems should be solved. 1. How to efficiently identify whether a combination of shared cache contentions is a feasible combination. 2. How to efficiently find the worst-case feasible combination. We present a novel method in Section 4.5 to solve these two problems.

4.5 Methodology

This section presents a novel WCET analysis framework to solve the pessimism disclosed in Section 4.4. Firstly, we define a new analysis unit in section 4.5.1. Then, section 4.5.2 shows how to precisely compute the maximum shared cache misses caused by each potential shared cache contention, while section 4.5.3 proposes a novel algorithm to efficiently find the worst-case feasible combination.

4.5.1 Access Order Analysis

According to Observation 4, some shared cache contentions can not happen together because the access to memory references should follow a certain order. In Figure 4.3, shared cache contention between $(r_T^3)'$ and r_T^1 , and shared cache contention between $(r_T^1)'$ and r_T^2 , can not happen together as $(r_T^3)'$ *must be accessed after* $(r_T^1)'$ and r_T^1 *must be accessed before* r_T^2 . In order to identify whether two shared cache contentions can happen together, the access order among effective memory references therefore should be precisely obtained. However, in real programs, due to the structure of loops and branches, sometimes it is impossible to get a constant access order among memory references.

Assuming two memory references r_{τ}^A and r_{τ}^B are in the same loop, the access of r_{τ}^A in the former iterations must be accessed before the access of r_{τ}^B in the later iterations, and the access of r_{τ}^A in the later iterations must be accessed after the access of r_{τ}^B in the former iterations. Therefore, the access order between memory references in the same loop is unfixed. Let's assume two shared cache contentions, shared cache contention between r_{τ}^A and $r_{\tau'}^i$, and shared cache contention between r_{τ}^B and $r_{\tau'}^j$, where r_{τ}^A and r_{τ}^B are in the same loop. Since r_{τ}^A may be accessed before and after r_{τ}^B , these two shared cache contentions can always happen together no matter what access order between $r_{\tau'}^i$ and $r_{\tau'}^j$ is. Therefore, the unfixed access order is useless for feasible combination analysis. In order to efficiently identify whether two shared cache contentions can happen together, we group all the memory references in the same loop together as the new analysis unit, Unordered Region (UR).

Definition 4.5.1 (Unordered Region, UR). *A UR is a set of memory references. The set either contains only one memory reference that does not reside in any loop, or all the memory references that reside in an outmost loop.*

Moreover, due to the structure of branches in a program, the access order between URs may be nonexistent. Assuming two URs UR^i and UR^j are in the different paths of a branch, UR^i is neither accessed before nor accessed after than UR^j . In order to precisely capture the access order between URs, we enumerate all the paths from the start UR to the end UR, where the path is called *Constant Execution Order Path* (CEOP). Since there are no loops after using UR as the analysis unit, the number of CEOPs of a program is generally small. According to the number of CEOPs of the programs of the most popular WCET benchmark [27] shown in Table 4.3, the number of CEOPs of most benchmark programs is no larger than 5.

We denote a CEOP of task T as P_{τ} , and the x -th UR in P_{τ} as $P_{\tau}(x)$. For the example in Figure 4.2, memory references r_{τ}^2 and r_{τ}^3 are grouped as UR_{τ}^2 , while memory references r_{τ}^5 and r_{τ}^6 are grouped as UR_{τ}^4 . Other memory references not in any loop are grouped as a UR individually. Finally we get the CFG composed by URs, and denote it as *UR-CFG*. Then we

enumerate all the CEOPs from the start UR (UR_T^1) to the end UR (UR_T^4):

$$P_T^A : UR_T^1 \rightarrow UR_T^2 \rightarrow UR_T^4$$

$$P_T^B : UR_T^1 \rightarrow UR_T^3 \rightarrow UR_T^4$$

The x -th UR in P_T is denoted as $P_T(x)$. For example:

$$P_T^A(3) = UR_T^4$$

Therefore, assuming two URs in P_T , $P_T(x)$ and $P_T(y)$, if $x < y$, $P_T(x)$ *must be accessed before* $P_T(y)$, and $P_T(y)$ *must be accessed after* $P_T(x)$. In the following of this section, when we mention the shared cache contention, we refer to the shared cache contention between URs. With two CEOPs of different tasks, we can identify whether two shared cache contentions can happen together through the following lemma.

Lemma 4.5.1. *Let's assume two CEOPs of two parallel tasks T and T' , P_T and $P_{T'}$. $P_T(x_1)$ and $P_T(x_2)$ are two different URs of P_T . $P_{T'}(y_1)$ and $P_{T'}(y_2)$ are two different URs of $P_{T'}$. If $x_1 < x_2$ and $y_1 < y_2$, shared cache contention between $P_T(x_1)$ and $P_{T'}(y_2)$ can not happen together with shared cache contention between $P_T(x_2)$ and $P_{T'}(y_1)$.*

Proof. Since $x_1 < x_2$, and $y_1 < y_2$, $P_T(x_1)$ must be accessed before $P_T(x_2)$, and $P_{T'}(y_1)$ must be accessed before $P_{T'}(y_2)$.

- Supposing shared cache contention between $P_T(x_1)$ and $P_{T'}(y_2)$ happens first, T' must execute to $P_{T'}(y_2)$ and has no chance to access $P_{T'}(y_1)$. Therefore, the shared cache contention between $P_T(x_2)$ and $P_{T'}(y_1)$ does not happen.
- Similarly, if shared cache contention between $P_T(x_2)$ and $P_{T'}(y_1)$ happens first, shared cache contention between $P_T(x_1)$ and $P_{T'}(y_2)$ does not happen.

Therefore, these two shared cache contentions can not happen together. □

In order to precisely predict the access orders among different memory references, we define UR as the analysis unit. Moreover, we have lemma 4.5.1 to formally identify whether two shared cache contentions can happen together. Based on the new analysis unit and lemma 4.5.1, we develop an algorithm which can efficiently find the worst case feasible combination in Section 4.5.3.

4.5.2 Precise Shared Cache Contention Analysis Between URs

According to Observation 3, when a conflicting memory reference (r_T^A) contends with another memory reference ($r_{T'}^B$), not all the accesses of r_T^A are changed to shared cache miss. To precisely predict the maximum extra shared cache misses caused by each potential shared cache contention, we have the following lemma.

Lemma 4.5.2. *Assuming a conflicting memory reference (denoted as r_T^A) contends with a memory reference (denoted as $r_{T'}^B$) on the shared cache, each memory access of $r_{T'}^B$ causes at most one memory access of r_T^A to miss the cache.*

Proof. According to the definition of conflicting memory reference, r_T^A is always shared cache hit if no tasks are executing in parallel. Taking shared cache contentions into consideration, when T' executes to $r_{T'}^B$, memory block accessed by r_T^A is evicted from the cache. Then the next access of r_T^A misses the cache. Consequently, the age of the memory block accessed by r_T^A is updated with 0 which is no larger than its original age when it is always hit. Therefore, the following accesses of r_T^A must be cache hit until the next access of $r_{T'}^B$. \square

By lemma 4.5.2, we can conclude that the extra number of shared cache misses of r_T^A caused by shared cache contentions does not exceed the execution counts of $r_{T'}^B$. A conflicting memory reference may contend with different memory references, and therefore the extra number of shared cache misses of a conflicting memory reference does not exceed the

execution counts of itself and the sum of the execution counts of all the memory references that contend with it.

We define the function f to compute the maximum extra execution time caused by the shared cache contention between the new analysis units, URs. Assuming two URs, UR_T^i and $UR_{T'}^j$, the extra execution time of T due to shared cache contention between UR_T^i and $UR_{T'}^j$ can be computed as:

$$f(UR_T^i, UR_{T'}^j) = \sum_{\forall r_T^i \in R_{\text{con}}(UR_T^i)} \left\{ \min(x(r_T^i), \sum_{\forall r_{T'}^j \in UR_{T'}^j} x(r_{T'}^j)) \times c^{\text{miss}} \mid (C(r_T^i) = C(r_{T'}^j)) \right\}$$

where $R_{\text{con}}(UR_T^i)$ denotes the set of conflicting memory references in UR_T^i , $\sum x(r_{T'}^j)$ returns the sum of the execution counts of all the memory references of $UR_{T'}^j$ that access the same cache set with r_T^i (i.e., a conflicting memory reference). Therefore, the extra number of shared cache misses of r_T^i does not exceed the maximum execution counts of itself (i.e., $x(r_T^i)$) and $\sum x(r_{T'}^j)$. Function f computes the maximum extra number of shared cache misses of each conflicting memory reference in UR_T^i individually and sums them together to get the maximum extra execution time caused by shared cache contentions between UR_T^i and $UR_{T'}^j$.

For the example shown in Figure 4.3, we use UR_T^1 and $UR_{T'}^2$ as the target URs to illustrate function f . UR_T^1 contains two conflicting memory references, $(r_T^1)'$ and $(r_T^2)'$, and they both access the same cache set with $r_{T'}^2$. Therefore, we have:

$$f(UR_T^1, UR_{T'}^2) = \left[\min(x((r_T^1)'), x(r_{T'}^2)) + \min(x((r_T^2)'), x(r_{T'}^2)) \right] * c^{\text{miss}}$$

The maximum execution counts of $(r_T^1)'$ and $(r_T^2)'$ are both 9, and that of $r_{T'}^2$ is 1. As a result,

$$f(UR_T^1, UR_{T'}^2) = (1 + 1) * c^{\text{miss}} = 2 * c^{\text{miss}}$$

4.5.3 Precise Shared Cache Contention Analysis Between Tasks

According to Observation 4, in order to obtain a precise estimated WCET bound, the worst-case feasible combination of shared cache contentions should be found out. This section

details the proposed efficient and precise method.

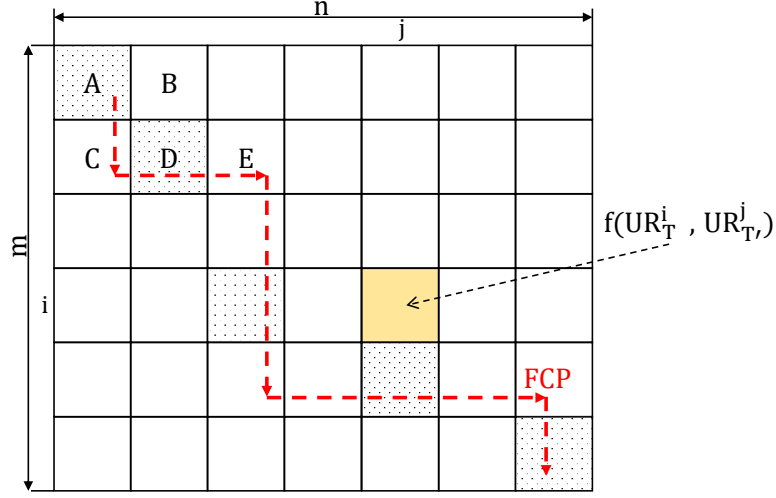


Figure 4.5. $CM_{(P_T^A, P_{T'}^B)}$

Our method is performed with every pair of CEOPs of two parallel tasks. When T executes P_T^A and T' executes $P_{T'}^B$, we build a matrix of elements as shown in Figure 4.5. In the figure, a square represents an element. We call the matrix as *Contention Matrix* of P_T^A and $P_{T'}^B$, and denote it as $CM_{P_T^A, P_{T'}^B}$. The size of $CM_{P_T^A, P_{T'}^B}$ is $m * n$, where m and n are the number of URs of P_T^A and $P_{T'}^B$ respectively. In $CM_{P_T^A, P_{T'}^B}$, element $CM_{P_T^A, P_{T'}^B}[i][j]$ represents the shared cache contention between $P_T^A(i)$ and $P_{T'}^B(j)$, and it has a value:

$$CM_{P_T^A, P_{T'}^B}[i][j] = f(P_T^A(i), P_{T'}^B(j))$$

$CM_{P_T^A, P_{T'}^B}$ contains all the potential shared cache contentions between P_T^A and $P_{T'}^B$. Some of them can not happen together as mentioned above. Assuming two shared cache contentions in $CM_{P_T^A, P_{T'}^B}$, $CM_{P_T^A, P_{T'}^B}[i_1][j_1]$ and $CM_{P_T^A, P_{T'}^B}[i_2][j_2]$, by Lemma 4.5.1, if

$$i_1 < i_2 \wedge j_1 > j_2$$

they can not happen together. Specifically, if one element is in the *upper-right or the lower-left* of another, they can not happen together. In a combination of elements, if there exists an element in the upper-right or lower-left of another, the combination is infeasible. Therefore,

a combination is a feasible combination if no element is in the upper-right or lower-left of others. We define *Feasible Contention Path* (FCP) and prove that elements of any feasible combination can be connected by a FCP.

Definition 4.5.2 (Feasible Contention Path, FCP). *FCP is a loop free directed path. It is composed by a set of sub-paths whose directions are either from upper to lower or from left to right. From the sub-path in the left-upper to the sub-path in the right-lower, the destination of a sub-path is the starting point of the next sub-path. The starting point of the FCP is the start point of the first sub-path and the destination of the FCP is the destination of the last sub-path.*

An example FCP is shown in Figure 4.5. The first sub-path is from **A** to **C**, and its destination (i.e., **C**) is the starting point of the next sub-path (i.e., **C** to **E**). Next, we prove that any feasible combination can be covered by a FCP.

Lemma 4.5.3. *All the elements of a feasible combination of shared cache contentions can be connected by a FCP.*

Proof. We first prove that, for any element if there exist elements in the same row with it there does not exist elements in the same column with it and vice-verse. We use the example **CM** in Figure 4.5 to prove this. Treating **A** as the effective element, **B** is in the same row with it and **C** is in the same column with it. Therefore, **B** is in the upper-right of **C**, and the combination of **A**, **B** and **C** is infeasible.

We select one element as the start element. Then we prove this lemma from the start element step by step. For each step, there are three cases:

1. If an element **O** is in the same row with the start element, we use a path from right

to left to connect them. Then, among all the unconnected elements, no one is in the upper row or left column of \mathbf{O} .

2. If an element \mathbf{O} is in the same column with the start element, we use a path from upper to lower to connect them. Then, among all the unconnected elements, no one is in the upper row or left column of \mathbf{O} .
3. For other cases, between the start element's left neighbouring element and lower neighbouring element, we randomly choose one \mathbf{O} and connect them. Then, among all the unconnected elements, no one is in the upper row or left column of \mathbf{O} .

Comparing with all the unconnected elements, \mathbf{O} is in the left-most and upper-most. Then, we treat \mathbf{O} as the start element and repeat (1) (2) (3) until the element in the right-most and lower-most is connected. Therefore, all the elements of a feasible combination are connected by a set of paths whose directions are either from upper to lower or from left to right. A FCP is constructed by connecting all these sub-paths end to end and all the elements in the feasible combination are connected by the FCP. \square

A feasible combination of shared cache contentions can construct a FCP, and the extra execution time caused by all the shared cache contentions in it is the sum of the values of all the elements covered by the FCP. Denoting the sum of values of all the elements connected by a FCP as V_{FCP} , the maximum extra execution time caused by shared cache contentions between two CEOPs is the maximum V_{FCP} among all the FCPs in the CM. We propose algorithm 3 to find the FCP with the maximum V_{FCP} , denoted as FCP_{max} , for a CM.

Algorithm 3 Maximum V_{FCP}

- 1: define matrix $\text{ArvVal}[m][n]$
- 2: **for** $i=1$; $i < m$; $i++$ **do**

```

3:   ArvVal[i][0] = CM[i][0] + CM[i-1][0]
4: end for
5: for i=1 ; i<n ; i++ do
6:   ArvVal[0][i] = CM[0][i] + CM[0][i-1]
7: end for
8: for i=1 ; i<m ; i++ do
9:   for j=1 ; j<n ; j++ do
10:    ArvVal[i][j] = max(ArvVal[i-1][j], ArvVal[i][j-1]) + CM[i][j]
11:   end for
12: end for

```

Since the values of all the elements in a CM are no less than 0, FCP_{max} must start from $CM[0][0]$ and end with $CM[m-1][n-1]$. In the algorithm, we compute the arriving value of all the elements if it is on FCP_{max} . The arriving value of an element is the sum of values of all its preceding elements and itself, and the arriving value can be furthermore used to compute its succeeding elements' arriving value. In algorithm 3, we store the arriving values in $ArvVal[][]$ (line 1). According to the direction of FCP, when a FCP arrives an element $CM[i][j]$, the FCP comes from either its left neighboring element (i.e., $CM[i-1][j]$) or upper neighboring element (i.e., $CM[i][j-1]$). From the point $CM[0][0]$, we first consider two special cases, the elements in the first row and the elements in the first column. If an element in the first row is on FCP_{max} , FCP_{max} can only come from its left neighboring element. Therefore, the arriving value of it is the sum of the values of all the elements in its left and itself (line 3). Similarly, the arriving value of elements in the first column is the sum of the values of all the elements in its upper and itself (line 6). For the remaining elements (e.g., $CM[i][j]$), we only need to choose the one with the larger arriving value as its preceding element between its two possible preceding elements (e.g., $CM[i-1][j]$ and $CM[i][j-1]$), and its arriving value is the sum of the larger arriving value and the value of itself (line 10). Finally, the arriving value of all elements can be computed, and the maximum V_{FCP} is the arriving value of $ArvVal[m-1][n-1]$.

For given P_T^A and $P_{T'}^B$, the maximum extra execution time caused by shared cache contentions denoted as $W(P_T^A, P_{T'}^B)$, can be computed. For the analyzed task (i.e., T) and its parallel executing task (i.e., T'), we compute the $W(P_T^A, P_{T'}^B)$ for each pair of CEOPs separately, and find the maximum $W(P_T^A, P_{T'}^B)$. Therefore, $WCET_{inter}(T, T')$ can be computed by:

$$WCET_{inter}(T, T') = \max \{ \forall (P_T \in T, P_{T'} \in T'), W(P_T, P_{T'}) \}$$

A task may have more than one parallel executing tasks. We compute the additional execution time caused by shared cache contention with each parallel executing task individually and sum them as the total additional execution time. Thus, $WCET_{inter-T}$ can be computed by:

$$WCET_{inter-T} = \sum_{T' \in T_{con}} WCET_{inter}(T, T')$$

where T_{con} contains all the tasks that execute with T in parallel no matter which core it executes on. Note that, since a task T' may execute several times during the execution of T , T_{con} contains l T' , where l denotes the times that T' may execute during the execution of T . Specifically, for each execution of T' , we compute the $WCET_{inter}$ individually and then sum them up. Last, $WCET_T$ is the sum of $WCET_{inter-T}$ and the $WCET_{intra-T}$ analyzed by the intra-core analysis. Since the $WCET_T$ computed by the proposed method may not always be less than the state-of-the-art method. We also use the state-of-the-art method to compute the $WCET_T$ and treat the smaller one as the final result. Note, by the evaluation, the estimated WCET bound analyzed by the proposed method is tighter in most cases.

The computational complexity of the proposed method is $N * (x^y)^2$, where N denotes the number of tasks that execute with T in parallel, x denotes the maximum number of paths of branches in *UR-CFG*, and y denotes the number of branches in *UR-CFG*. Specifically, (x^y) denotes the number of CEOP of tasks. Since *UR-CFG* does not contain any loop, the number of CEOP is generally small as shown in Table 4.3. The timing complexity of the proposed method increases linearly with the increase of the number of parallel executing tasks. Although the theoretical worst case complexity is exponential, real world programs do not involve too many nested branches and hence the proposed method is effective.

4.6 Multi-Level Cache Analysis

To simplify the presentation, the above sections assume a simple cache architecture which only contains a single level shared instruction cache. However, our method can be easily extended to the processors with non-inclusive multi-level caches, a more general case, by adopting the existing intra-core multi-level cache analysis technique [28]. We assume a standard architecture of the multi-level caches as shown in Figure 4.6. Each core has a private level 1 (L1) instruction and data cache, different cores share a level 2 (L2) cache.

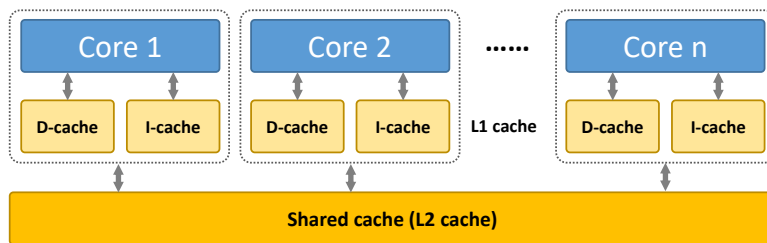


Figure 4.6. A multi-level cache architecture

We adopt the standard multi-level cache analysis technique [28] as the intra-core analysis to get the cache behaviors of memory references at different levels of cache. The intra-core analysis is first performed at the L1 cache, and classifies memory references as always hit (L1_AH), always miss (L1_AM), persistent (L1_PS) or non-classified (L1_NC). Eliminating the memory references that are hit in L1 cache by a filter function, it uses the same method to get the L2 cache behavior of each memory reference. The filter function is defined as:

L_i cache behavior	L_{i+1} access behavior
L1_AH	Never (N) access L_{i+1} cache
L1_AM	Always (A) access L_{i+1} cache
L1_NC	Uncertain (U) (may or may not access L_{i+1} cache)

Iteratively, the intra-core analysis is performed level by level, until the last level. Finally, the cache behavior of memory references at different level are produced. With the analyzed cache behavior, the longest path $Path_{Intra}$ can be found out.

Accordingly, the equation to compute the $WCET_{intra-T}$ should be changed depending on the number of levels. Assuming the number of levels is 2: the $WCET_{intra-T}$ is:

$$WCET_{intra-T} = X_{PathIntra}^{L1_hit} * c^{L1_hit} + X_{PathIntra}^{L1_miss} * c^{L1_miss} + X_{PathIntra}^{L2_hit} * c^{L2_hit} + X_{PathIntra}^{L2_miss} * c^{L2_miss}$$

where $X_{PathIntra}^{L1_hit}$, $X_{PathIntra}^{L1_miss}$, $X_{PathIntra}^{L2_hit}$ and $X_{PathIntra}^{L2_miss}$ denote the total number of execution counts of memory references classified as L1_AH, L1_AM or L1_NC, L2_AH and L2_AM or L2_NC respectively. c^{L1_miss} , c^{L1_hit} , c^{L2_miss} and c^{L2_hit} denote the latency of L1 cache miss, L1 cache hit, L2 cache miss and L2 cache hit respectively.

When extending our method to multi-level caches, the only additional step is to replace the intra-core analysis with the standard multi-level intra-core analysis [28]. According to definition 4.3.1, all the conflicting memory references can be obtained. Also the *UR-CFG* of each task and the maximum extra execution time caused by each potential shared cache contention can also be computed by the method presented in section 4.5.1 and section 4.5.2 respectively. Subsequently, following the method presented in section 4.5.3, the worst-case feasible set of shared cache contentions can be computed. Further, we can get the $WCET_{inter-T}$. $WCET_T$ can be finally computed by summing $WCET_{intra-T}$ and $WCET_{inter-T}$.

4.7 Evaluation

This section evaluates the precision and the efficiency of the proposed approach through a comparison with the state-of-the-art method [39]. In the experiment, we assume a dual-core processor. Each core has a private L1 instruction cache and L1 data cache, while two cores share a L2 instruction cache. The size of L1 instruction cache and L1 data cache are both 512B. Since in hard real time systems, the shared cache size is relatively small, we set the L2 instruction cache size as 4K which is same with the experiment setting of the state-of-the-art method. Both the L1 cache and the L2 cache are 2 way set-associative, and the block size of them are both 32-byte. We assume the latency of L1 cache hit, L2 cache hit and L2 cache miss are 1 cycle, 6 cycles and 106 cycles respectively.

Table 4.2. Reduced estimated WCET in percentage of our method with different benchmark

programs

T \ T'	fir	mat	fdct	cnt	expint	qurt	end	ludcmp	ns	adpcm	st	ndes	bs	avg.
mat	0.1	0	0.1	0.1	0	0	0	0.1	0.1	0	0	0	0.1	0.05
cnt	0	0	0.1	0	0.3	0.1	0	0.3	0.4	0	0	0	0.3	0.12
fir	33.5	16.7	0.1	16.8	0.1	0	16.7	33.6	0.1	0	0.1	0	16.8	10.35
fdct	54.4	51.6	33.5	51.6	48.8	16.7	2.7	41.8	65.5	0	19.5	0	61.3	34.42
expint	6.2	3.7	9.9	7.4	0	0	0	3.7	6.2	0	0	0	5	3.24
qurt	28.2	35.2	5.5	33.8	21.5	0	29.2	34.7	28.3	0	15.5	17.3	17.3	20.55
edn	31.3	27.5	28.3	30	25.2	0	9.2	24.9	40.9	0	7.9	0	37.3	20.19
ludcmp	0	0.8	0.8	0	0	0.8	0	0	0.8	0	0	0	0.8	0.31
ns	0	0	0	0	0	0	0	0	0	0	0	0.6	0	0.05
ndes	39.3	25.2	30.3	45.8	40	0	0	54	52.2	0	14.3	0	39.8	26.22
bs	6.4	0	12.8	6.4	0	0	12.6	6.4	0	0	0	0	0	3.43
adpcm	49.6	50.9	44.5	50.5	51.1	16.2	23.7	42.5	62.5	0	27.3	8.4	58.2	37.33
st	1.3	1.3	1.2	1.2	1.3	0.1	0.5	0.9	0.2	0	0.8	0.5	1.9	0.86

Our method is evaluated with the most widely used benchmark program suite in static timing analysis, Mälardalen WCET benchmark suite [27]. The benchmark programs used in our experiment are shown in Table 4.2. Our evaluation is performed with every pair of benchmark programs as shown in Table 4.2. We assume task T (i.e., the first column of Table 4.2) executes on a core, while task T' (i.e., the first row of Table 4.2) executes on the other core in parallel. Both of them execute only once. We conduct the estimated WCET bound of task T with different T' using our method denoted as **OUR** and the state-of-the-art method denoted as **OTHER** separately. Compared with the state-of-the-art method, The ratios between **OTHER** – **OUR** and **OTHER** with different pairs of T and T' are shown in Table 4.2. Note that, the higher the number in Table 4.2 is, the tighter the estimated WCET bound of our method is. For example, when T is *fir* and T' is *edn*, the estimated WCET bound

produced by our method is 16.7 percentage points less than that produced by the state-of-the-art method.

Our method is implemented based on an existing WCET analysis tool Chronos [38]. Originally, Chronos can get the cache access behavior of memory references and the CFG of the analyzed task. Based on the original function of Chronos, we implement the proposed method to get the conflicting memory references analysis, *UR-CFG*, and further the extra execution time caused by shared cache contentions.

According to the analysis results shown in Table 4.2, when T is *matmult*, *cnt*, *ludcmp*, *ns* or *st*, our method gets slight improvement in accuracy. This is because, among these benchmarks, most of the memory references are L1_AH. For example, program *matmult* contains 504 instruction memory references, among which 468 references are L1_AH. Therefore, the time used to access the L2 cache is only a small portion of its overall execution time. That is, the L2 cache behavior has slight effect on the program's execution time. Hence, although our method performs more accurate analysis for L2 cache, the improvement is also only a small portion of the estimated WCET. Therefore, the reduced estimated WCET for these benchmarks are relatively slight.

When task T' is *expint*, *edn*, *adpcm* or *ndes*, the gained improvement in accuracy of our method is less. This is because these benchmarks all contain a large number of memory references that access the L2 cache, and these memory references cause a large number of shared contentions. More importantly, a conflicting memory reference may contend with several memory references. By our method, shared cache contentions between a conflicting memory reference and some interfering memory references may be excluded, but the conflicting memory reference may still contend with other interfering memory references causing the conflicting memory reference to miss the cache. For example, *ndes* contains 234 memory references accessing the L2 cache. Thus, under these situations, the state-of-the-art method that considers that all the conflicting memory references should be changed to L2_NC is relatively accurate. As a result, the gained improvement in accuracy of our method is relatively less.

For other pairs of T and T' , our method significantly tightens the estimated WCET bound compared with the state-of-the-art method. In these cases, the analyzed task (i.e., T) contains lots of conflicting memory references. The state-of-the-art method re-classifies all the conflicting memory references as L2.NC, and considers all of them as shared cache miss when computing the WCET. However, in our method, through a precise shared cache contention analysis, a large number of shared cache contentions are excluded, and therefore a large number of accesses of conflicting memory references are still considered as shared cache hit when computing the WCET. Therefore, our method gains significant improvement in accuracy as the estimated WCET bound is much tighter. For example, when T is *fdct* and T' is *ns*, *fdct* contains 57 conflicting memory references. The state-of-the-art method considers that all of them are reclassified as L2.NC. Since total execution counts of all the 57 conflicting memory references are 456, the state-of-the-art method considers that shared cache contentions between *fdct* and *ns* introduce 45600 cycles latency. However, since not all the 57 conflicting memory references will be re-classified as L2.NC together, the maximum latency caused by shared cache contentions analyzed by our method is only 8000 cycles. Adding the latency caused by the shared cache contentions to the intra-core analysis result, totally our method gains a 65.5% improvement in accuracy.

Averagely, our work can significantly tighten the estimated WCET bound comparing with the state-of-the-art method. Meanwhile, as shown in Table 4.3, our method consumes almost the same analysis time with it. The proposed shared cache contention analysis is performed with each pair of GEOPs of parallel tasks. In contrast, the state-of-the-art method considers all the GEOPs of a program to be together. As shown in Table 4.3, after using UR as our new analysis unit, the number of GEOPs of most benchmark programs is no larger than 2. Therefore, the additional pairs of paths that need to be analyzed are not much. More importantly, most of the analysis time is consumed by the intra-core analysis, which exists both in our method and the state-of-the-art method. Therefore, the additional time that our method consumes is quite a small part of the total. The exceptional benchmark programs are *qurt* and *ndes*. They contain more GEOPs, and our method consumes more analysis time. But in these experiments, our method achieves much more improvement in accuracy.

Table 4.3. Average analysis time and the number of CEOPs of each benchmark

Benchmarks	Analysis time(ms)		Number of CEOPs
	State-of-the-art	Our method	
matmult	53.07	57.99	1
cnt	36.64	51.16	1
fir	28.29	32.04	2
fdct	42.69	46.46	1
expint	41.44	45.63	1
qurt	106.17	204.81	27
edn	104.73	113.96	1
ludcmp	51.79	55.36	2
ns	41.75	44.91	1
ndes	137.02	163.16	4
bsort100	26.69	29.81	1
adpcm	359.39	526.39	1
st	45.48	49.47	1

4.8 Conclusion

Producing a tight estimated WCET bound for systems with shared caches is crucial for modern hard real-time systems. Therefore, the maximum shared cache misses caused by shared cache contentions should be precisely predicted. However, the state-of-the-art method is

pessimistic in shared cache contention analysis. It not only overestimates the number of shared cache misses caused by each potential shared cache contention, but also counts some infeasible shared cache contentions. The proposed method can improve the state-of-the-art method in the above-mentioned two aspects, and further produce a much tighter and safe estimated WCET bound. As a result, the real-time system test and design can be benefited from the tighter estimated WCET bound.

CHAPTER 5

LATICS: A LOW-OVERHEAD ADAPTIVE TASK-BASED INTERMITTENT COMPUTING SYSTEM

5.1 Introduction

It is predicted that the number of Internet-of-Things (IoT) devices will exceed 20 billion by 2025 [3]. A challenge is how to power such a huge amount of IoT devices. As such devices are typically deployed in complex working environments, it is almost impossible to charge or change their batteries. Energy harvesting is a promising approach which allows a device to rely on energy harvested from the ambient environment. As energy output of harvesters is typically weak and unstable, the computing system must ensure software programs will make progress in the presence of frequent power failures. Therefore, energy-harvesting systems must be able to timely store program states and keep making progress in the presence of frequent power failures.

To this end, checkpointing, a widely used technique in fault-tolerant computing, has been applied to energy-harvesting systems [10, 11, 18, 30, 32, 33, 35, 43, 49, 56, 66]. When program execution reaches a checkpoint, system states are saved to non-volatile memory (NVM) for future restoration. However, checkpointing is generally expensive, as all volatile states, including processor context and data on volatile memory (VM), have to be saved to NVM.

The *task-based paradigm* [20, 29, 48, 58, 70] offers a lighter-weight alternative to checkpointing. Take the state-of-the-art task-based intermittent computing systems InK [70] for example, an application is programmed as a collection of atomic execution blocks called

tasks. Each task is written as a function by the programmer, and the control flow between tasks are also explicitly defined by the programmer. System states are saved at task boundaries, and when recovering from a power failure occurred during the execution of a task, the system resumes execution from the beginning of that task. At task boundaries, the system only needs to store the *global data*, i.e., the variables whose lifetimes cross task boundaries and shared by multiple tasks, but not the local variables accessed inside each task. Essentially, task boundaries play a similar role as checkpoints. However, as the programmer has full control to decide how to group related functionalities and variables into a task, typically, much fewer data need to be saved at task boundaries than the standard checkpointing approach.

Many modern ultra-low-power processors are equipped with NVM, such as FRAM, which has good enough read/write speed and lifetime to serve as the main memory [1]. Therefore, the system can allocate data on NVM and programs can directly address them, which is more efficient than copying them back and forth between VM and NVM. However, directly operating data on NVM may cause memory inconsistency due to the so-called *Write-After-Read (WAR)* problem [43] in the presence of power failure (detailed in Section 5.2). To solve the problem, the system should maintain a backup version for the WAR variables. When recovering from a power failure occurred in some task, the system restores the WAR variables from the backup version to main memory, and then correctly resumes execution from the beginning of the aborted task.

In the task-based paradigm, task granularity plays an important role to system performance. In general, the larger size are the tasks, the higher amount of incomplete execution is aborted at power failures. In the extreme case, if a task is too large to be finished by the energy harvested until the next power failure, the system will trap in the re-execution of this task and never make any progress (the so-called *Sisyphus effect*). On the other hand, if tasks are too small, the execution flow will cross task boundaries and thus save program states frequently, which leads to high overhead. To address this dilemma, some systems adopt an adaptive execution approach, which can skip state saving at task boundaries under good energy conditions.

Skipping state saving at task boundaries makes ensuring memory consistency a complex problem. On one hand, the overhead is reduced as state saving is done less frequently; on the other hand, skipping state saving may introduce new WAR variables. To keep the memory consistent, newly introduced WAR variables have to be identified and saved, which may adversely increase state saving overhead. A related work, Coala [51], addressed the problem by a very conservative approach: copying used data pages on-demand to VM in case operating the variables in the pages may cause WAR problem. During state saving, Coala pessimistically saves all modified data pages to NVM. As shown in our experiments (Sec. 5.5), this approach makes program execution very inefficient.

In this work, we present LATICS, a low-overhead adaptive task-based intermittent computing system that saves much fewer states at task boundaries than existing approaches. It is generally believed that skipping state saving at task boundaries is always beneficial, so related systems try to skip as many as possible if the energy condition allows. However, our work discloses that skipping state saving at one task boundary may cause the system to save more data at other places, and thus leads to higher overall overhead. In light of this observation, we propose a novel task-based intermittent computing system that enforces mandatory state saving at certain task boundaries, regardless of the current energy conditions, and conducts analysis to decide the breaking points and the states to be saved to ensure correct execution. Experiments conducted on an MSP430 platform show that LATICS drastically reduces the amount of states to save at task transitions and improves execution efficiency, compared with existing solutions.

5.2 Overview

The main objective of LATICS is to reduce state saving overhead and improve execution efficiency of adaptive task-based intermittent computing systems.

At run-time, LATICS will coalesce the execution of several consecutive tasks into a large execution block called *transactional execution block* (TEB). The first task in the TEB is called the *leading task*. System states are only saved at the beginning of the leading task, but

not at any other task boundary inside the TEB. If a power failure occurs and aborts a TEB, the system will resume from its leading task. If the TEB successfully finishes execution, a new TEB will be decided. The above process repeats until the program is completed. TEB is now the atomic execution block of the system. LATICS tends to use a larger TEB and thus skip state saving on more task boundaries if it has high confidence that the current energy condition is good and the possibility of power failures is low. A unique feature of LATICS is that, LATICS also forces TEBs to end at certain task boundaries and conduct state saving, in order to reduce the overall state saving overhead. To see its benefit, we first explain what kind of states must be saved at the beginning of a TEB.

We begin with explaining the WAR problem [43]. Fig. 5.1(a) shows three sequentially executed tasks, T_1, T_2, T_3 . Each task executes as an individual TEB, i.e., if power fails during the execution of a task, the system resumes execution at the beginning of that task. u, x, y, z are variables allocated on NVM, a is a variable allocated on VM. Suppose power fails after “ $z = a + 1$ ”, the system will resume execution from the beginning of T_2 . The correct read to z at $a = z$ should return -1 (assigned in T_1). However, value 0 is actually returned as a result of the partial execution before the power failure. This is the so-called *WAR problem*. We say z is a WAR variable, and the read and write to z have WAR dependency. Note that y is a WAR variable in T_3 ($y = y + 1$ means y 's value is first read, then incremented by 1, and finally written back to y).

To avoid the WAR problem, a system should save all WAR variables at the beginning of each task. As shown in Fig. 5.1(b), if the value of z is saved at the beginning of T_2 , when the system recovers from a power failure, z 's correct value can be restored and then T_2 continues to execute consistently. Saving the WAR variables at task boundaries is a major overhead in task-based intermittent computing systems.

Now we consider an adaptive task-based intermittent computing system that only saves states at the beginning of a TEB and skips state saving at the task boundaries inside the TEB. In Fig. 5.1(c), assume at run-time T_2 and T_3 are coalesced into a TEB, so the state saving between T_2 and T_3 is skipped. As power may fail anywhere in T_2 and T_3 ,

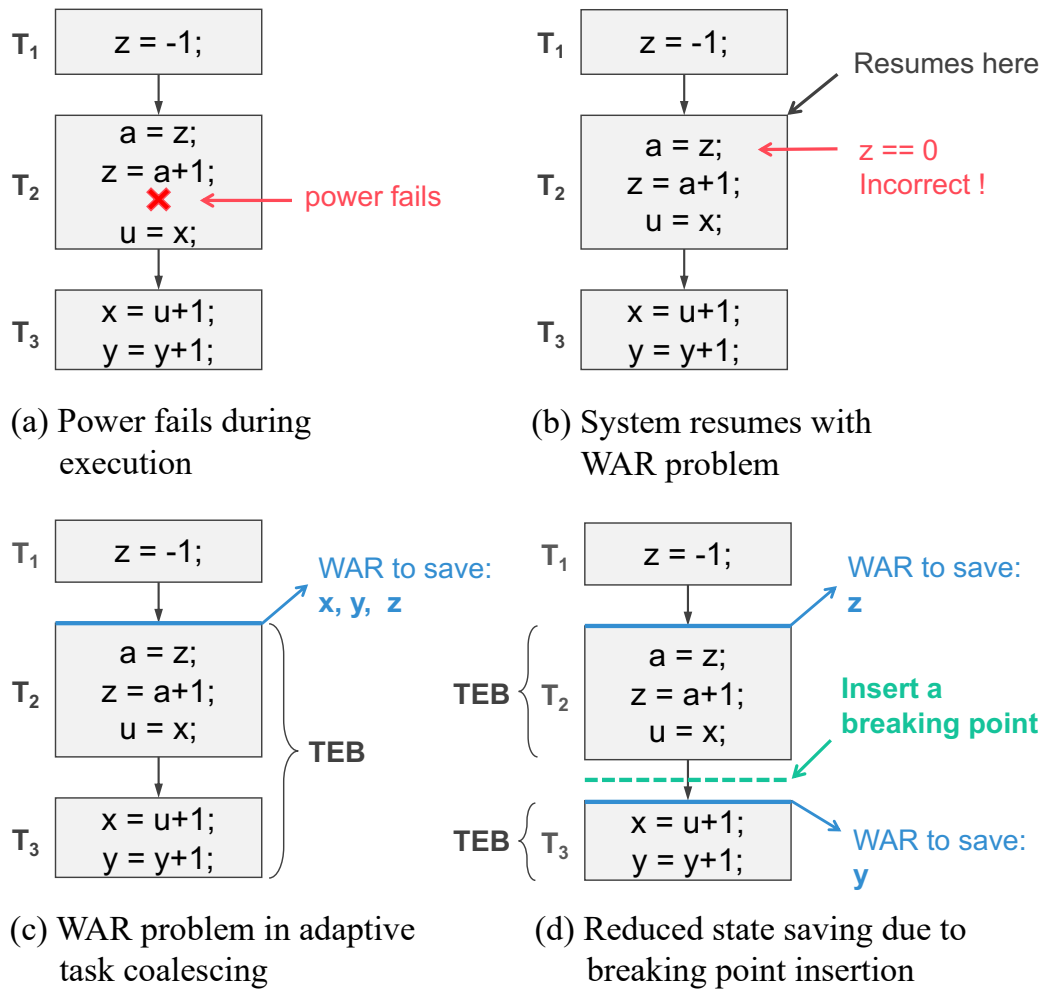


Figure 5.1. The WAR problem and breaking point insertion

WAR variables inside both tasks have to be saved at the beginning of the TEB. Notably, x now becomes a new WAR variable as a result of coalescing T_2 and T_3 . If power fails after $x = u + 1$ and the system resumes execution from T_2 , reading x at $u = x$ may be incorrect as a result of $x = u + 1$ in the incomplete execution. Saving x introduces new overhead, a result of task coalescing.

Reduce state saving by breaking point insertion. Assume x is a large-size array, the benefits of skipping state saving between T_2 and T_3 may well be covered by the extra overhead of saving x . To address this problem, LATICs inserts *breaking points* into the program on task boundaries. When program execution reaches a breaking point, state saving

must be conducted immediately. In the above example, LATICS inserts a breaking point between $T2$ and $T3$ and breaks the read and write dependency of the WAR variable x . As a result, x is excluded from the state saving at the beginning of the TEB.

The rest of this chapter details the two main components of LATICS: 1) a static analysis to decide the breaking points and compute a set of variables to save for each task if it leads a TEB; 2) a run-time system which decides how to group tasks into a TEB, manages state saving and restoration, and most importantly enforces the breaking point mechanism.

5.3 Analysis

Saving states at TEB boundaries guarantees memory consistency for adaptive task-based intermittent computing systems. The main objective of this work is to minimize the state saving overhead during the execution of TEBs that are decided at run-time. This section presents our static analysis to achieve this goal. First, we show it is more precise to compute a specific set of breaking points and a set of variables to save *for each task that leads a TEB* than for the whole program; second, we present an algorithm to efficiently decide the breaking points and compute the variables to save at each leading task.

5.3.1 Rationale

To tightly identify a set of variables to save at the leading task of a TEB requires two levels of information: (1) the WAR variables that need to be saved, and (2) which tasks are grouped into a TEB.

To precisely identify the WAR variables in a TEB, we need to consider the following cases. First, a WAR variable in a task remains a WAR variable in the TEB that includes the task [49,51]. Second, WAR variables introduced by coalescing tasks, such as x in Fig. 5.1(d), should be saved [49,51]. Third, there is no need to save the variables which are only read or written, as power failures do not make the values of such variables inconsistent. Last but not least, coalescing multiple tasks may as well exclude some WAR variables by destroying

the WAR dependency of their reads and writes. Suppose a TEB groups task T_1 and T_2 , and v is a WAR variable local to T_2 . If there is a write operation to v in T_1 , v is no longer a WAR variable in the TEB. When a power failure occurs in the TEB, the system will resume execution from T_1 (the leading task of the TEB). The write to v in T_1 overwrites any value of v written in the TEB's incomplete execution before the power fails, thus we do not have to save v at the beginning of the TEB.

In adaptive task-based intermittent computing systems, each TEB is decided only at run-time. The variables that need to be saved depends on which tasks are grouped together, which is affected by the current power condition and future program paths. Adaptively deciding TEBs brings difficulties in precise identification of the variables to be included in state saving.

Computing the states to save at the leading task of a TEB.

Due to the dynamic feature on how TEBs are decided, to compute a fixed set of variables to save for a whole program will be too pessimistic. So in this work, we propose to compute an individual set of variables *for each task*. If a TEB starts, only the subset of variables associated with its leading task need to be saved. This approach helps to exclude unnecessary state saving. Furthermore, for a given leading task, the decision on the scope of a TEB may vary according to different power conditions, so the computed set of variables should cover TEBs with all possible scopes.

Deciding breaking points for the leading task of a TEB. A major observation in our study is that, forcing the TEB to end before a task and conducting an in-situ state saving may reduce the overall amount of states to save, as shown in Fig. 5.1(d). Now, we have two options: to decide a set of breaking points for the whole program, or to decide an individual set of breaking points for each leading task. LATICS chooses the latter option, the reason of which is given below.

Consider an example in Fig. 5.2 with three tasks T_1 , T_2 and T_3 . $WAR(a)$ in T_1 means a is a WAR variable local to T_1 . The $R(d)$ in T_1 and T_2 are read operations to variable d , and $W(d)$ in T_3 is a write operation to d . Consider coalescing T_1 and T_3 into TEB_1 , T_2 and T_3

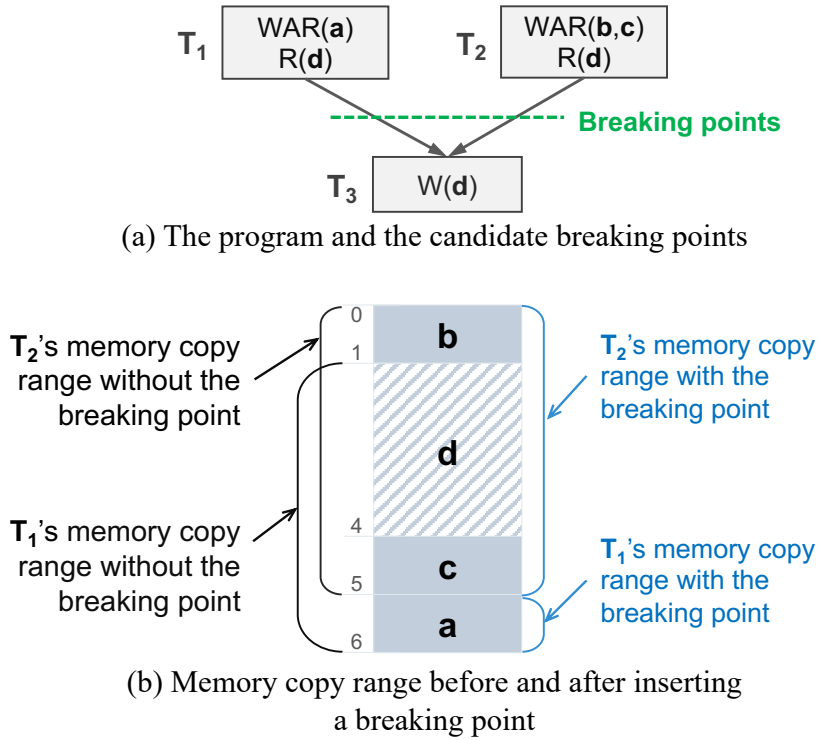


Figure 5.2. Deciding breaking points for each leading task

into TEB_2 . We can compute the sets of variables involved in the state saving at the beginning of TEB_1 and TEB_2 , which are $\{a, d\}$ and $\{b, c, d\}$, respectively. In LATICs, we associate with each leading task a *memory copy range* which is the minimal address region that covers the state variables. For example, given the memory layout in Fig. 5.2(b), the memory copy ranges for T_1 and T_2 are $[1, 6]$ and $[0, 5]$ respectively. The main objective is to leverage the performance benefit of bulk-copying (further discussed in Sec. 5.5.4).

In order to reduce state saving overhead for T_1 and T_2 , let's consider inserting breaking points on the incoming edges to T_3 . As a result, d is no longer a WAR variable of either TEB_1 or TEB_2 . The variables to save at the beginning of T_1 is changed to $\{a\}$ with corresponding memory copy range $[5, 6]$. For TEB_2 , the variables to save at the beginning of T_2 is now $\{b, c\}$, but the memory copy range that covers $\{b, c\}$ remains to be $[0, 5]$. Therefore, inserting a breaking point for TEB_2 is pointless in reducing the total amount of data copying. Motivated by this observation, LATICs chooses to decide an individual set of breaking

points for each task.

To summarize, to reduce state saving overhead, we propose to decide an individual set of breaking points for each leading task and also compute the variables to be saved at the leading task. Analysis is detailed in the following sub-sections.

5.3.2 Analysis Target and the Definition of Breaking Point

The analysis target are the WAR variables that are either local to a task or introduced by task coalescing (inserting breaking points may exclude WAR variables). A program is a directed graph, denoted by G , where each node is a task and the edges are control flows among tasks. Each task T can be further modeled by a control flow graph CFG_T , comprising basic blocks (denoted by BB) connected by control flow edges.

The WAR variables in a task can be defined and identified over the CFG of the task.

Definition 5.3.1 (WAR variable in a task). *If a variable x is a WAR variable in task T_i , there must exist a basic block BB_R which reads x , such that:*

- *Condition 1: BB_R is reachable from the starting basic block of CFG_{T_i} ,*
- *Condition 2: along one of the paths satisfying Condition 1 there is no basic block that writes x ,*
- *Condition 3: there exists a basic block BB_W that writes x and BB_W is reachable from BB_R in CFG_{T_i} .*

By the above definition, WAR variables in a task can be obtained by analyzing the reachability between the basic blocks in CFG_T . A widely used polynomial time algorithm to find reachability between the nodes on a directed graph is to compute the graph's transitive closure. We also apply this algorithm in our approach to find the WAR variables in a task T_i and represent them with a set WAR_{T_i} .

As discussed above, given a task T_i , TEBs with different scopes can be decided with T_i as the leading task, and each TEB may require a different set of variables to save at its beginning. To overcome this uncertainty, we propose to compute a set of variables for state saving considering TEBs with all possible scopes and led by T_i . We call the set of variables the *Possible WAR Set* of task T_i , denoted by PWS_{T_i} .

Definition 5.3.2 (Possible WAR Set). *If variable $x \in PWS_{T_i}$, there must exist a basic block BB_R in some task that reads x , such that:*

- *Condition 1: BB_R is reachable from the starting basic block of T_i in G ,*
- *Condition 2: along one of the paths satisfying Condition 1 there is no basic block of any task that writes x ,*
- *Condition 3: there exists a basic block BB_W in some task that writes x and BB_W is reachable from BB_R on G .*

Note that PWS_{T_i} is defined over G , so BB_R and BB_W in Definition 5.3.2 can reside in tasks other than T_i .

Definition 5.3.3 (Breaking point¹). *A breaking point is associated to an edge on G , such that when the execution of a TEB reaches a breaking point, the PWS for the outgoing task of the edge must be saved.*

An important feature is, inserting breaking points does not introduce new WAR variables. First, the WAR variables in the tasks before the breaking point are not changed and should be saved at the beginning of the TEB. Second, the WAR variables in the tasks after the breaking point are not changed, and should be saved at the leading task after the breaking

¹A breaking point is essentially a checkpoint if this approach is applied to checkpoint-based design. Inserting a breaking point means conducting a mandatory checkpoint at a specified program point.

point. Third, the WAR variables, whose read occurs before the breaking point and write occurs after the breaking point, need not to be saved, as their WAR dependency is destroyed.

5.3.3 Computing PWS and Inserting Breaking Points

Now we present an algorithm to compute the PWS for each leading task, the complexity of which comes from breaking point insertion. As a breaking point may destroy the WAR dependency of some WAR variables, inserting a new breaking point to G may cause a re-computation of PWS for all leading tasks. Finding the optimal breaking point assignment will be computationally intractable, as one needs to explore all possible edge combinations. For efficiency, we present a heuristic algorithm which *incrementally computes the PWS for a leading task and inserts breaking points on-the-fly*. For a leading task T_i , the algorithm starts a traversal of an induced sub-graph of G whose nodes are all reachable from T_i . During each step, the algorithm collects read and write operations to data allocated on NVM and identifies WAR variables. At each step to process a node, an evaluation is performed to decide whether breaking point(s) should be inserted on its incoming edge(s). If an insertion is decided, its impact on the PWS computation will take effect immediately.

– Program transformation

If program graph G has loop structures, the result of PWS depends on the order of node traversal, and it is in general unclear when the traversal would terminate. To address this problem, we propose to transform G into G' by the following unroll approach and compute PWS over G' . We build a new construct for a loop structure which contains two copies of the loop body serially connected, and replace the original loop structure with the new one. We say the first and second loop bodies are at *level 1* and *level 2* respectively. Each loop structure in G will be replaced by the unrolled structure to form G' . If there are nested loops, the unroll is conducted from the inner-most level to the outer-most level. The resulting G' is now a directed acyclic graph which is much easier to process for PWS computation. By the unroll operation, nodes in the same loop body may be replicated multiple times. If the nodes in G' are generated from the same replication, we say they are in the same unroll level, e.g.,

T_1 and T_2 are in the same unroll level in the example of Fig. 5.3.

As for the structure of task-based programs, there can be doubts that “goto” style control flows may be produced as programmers are allowed to freely specify the control flow among tasks by task transition directives. But this situation will rarely occur. A program is a set of instructions to implement the functionality of an application, regardless of whether the program progresses continuously or intermittently. Task is only a construct allowing a programmer to specify places where system states are saved to survive power failures. Thus, a task-based program should be naturally programmed by inserting task boundaries into a correctly implemented ordinary program. As most programs are well-structured, the task-based version obtained as above will remain well-structured.

Definition 5.3.4 (Image and image set). *In G' , each replica of a node/edge v is an image of v , and all the images of node/edge v , including v , form an image set of v , denoted by IS_v .*

For example, T_1 and T'_1 in Fig. 5.3 form an image set IS_{T_1} .

From now on, we compute the PWS for each leading task and insert breaking point over G' instead of G . We will prove any WAR variables in G will be definitely identified in G' .

Definition 5.3.5 (Reachability over image set). *In G' , an image set IS_B is reachable from another image set IS_A , if there is at least one node y from IS_B and one node x from IS_A , such that y is reachable from x in G' .*

Lemma 5.3.1 (Reachability maintenance). *If node B is reachable from node A in G , IS_B is reachable from IS_A in G' .*

Proof. We prove this lemma by distinguishing two cases:

Case 1: If A and B are in the same loop body in G , there are two sub cases. a) B is reachable from A by a path that does not include the back edge of the loop body, this path

belongs to the loop body and is always retained in G' according to our unroll method. b) B is reachable from A by a path that include the back edge, then there must be a path from A (at level 1) to B' (at level 2) in G' . So we can conclude IS_B is reachable from IS_A in G' .

Case 2: If A and B are in different loop bodies in G , without loss of generality, we assume A is in the outer loop and B is in the inner loop. There must be a path p_1 from A to the entry node N_B of B 's loop body and a path p_2 from N_B to B . Note that p_1 belongs to the outer loop and p_2 belongs to the inner loop. Both paths are retained in G' , so IS_B is reachable from IS_A . If otherwise A is in the inner loop and B is in the outer loop, then A 's exit node will be used to bridge the reachability from A to B . The above property can be extended to nodes that span across more than two levels of loop structures, by repeating the above reasoning for adjacent loop levels. \square

Lemma 5.3.2 (Reachability maintenance with node removal). *By removing node X from G and at the same time removing IS_X from G' , if node B is reachable from node A in $G \setminus \{X\}$, then IS_B is reachable from IS_A in $G' \setminus \{IS_X\}$.*

Proof. If node B is reachable from node A in $G \setminus \{X\}$, there must exist a path p in G from A to B and p does not contain X . By the construction of G' , p also exists in G' , making IS_B reachable from IS_A in G' by path(s) in IS_p . Removing IS_X from G' does not destroy IS_p , as X is not on p in G . So IS_B must be reachable from IS_A in $G' \setminus \{X\}$. \square

Theorem 5.3.1 (PWS maintenance). *Any variable which is in PWS_T over G is in PWS_T over G' .*

Proof. If a variable $x \in PWS_T$, x must satisfy three conditions. Note that conditions 1 and 3 in the definition of **PWS** are essentially reachability between nodes; condition 2 can be

checked by removing all write nodes to x from G and then explore reachability between nodes. By Lemma 1 and Lemma 2, if x satisfies all three conditions over G , x must satisfy all the conditions over G' , which means x is also in PWS_T if it is evaluated over G' . \square

Algorithm 4 Compute PWS and insert breaking points for a leading task

Input: Program graph G' , Leading task T_i

Output: PWS for T_i ; BPS, the set of breaking points for T_i

```

1: PWS =  $\emptyset$ , BPS =  $\emptyset$ , Ready queue RdyQ =  $\{T_i\}$ ,  $R_T = \emptyset$ ,  $W_T = \emptyset$ 
2: while RdyQ is not empty do
3:   T = Dequeue(RdyQ) /* to evaluate task T */
4:   for each incoming edge E of T do
5:     let  $T_E$  be the starting node of E
6:      $PWS' = PWS \cup (WAR_T - W_{T_E}) \cup (R_{T_E} \cap WIN_T)$ 
7:     if  $\text{Range}(PWS') - \text{Range}(PWS) > \alpha \times \text{Cost}$  then
8:       Insert a dummy task  $T_{BE}$  on each edge in  $IS_E$ 
9:       BPS = BPS  $\cup T_{BE}$ 
10:    else
11:      PWS = PWS'
12:    end if
13:  end for
14:   $R_T = (\bigcup_{p=\text{pre}(T)} R_p - WB_T) \cup (RT_T - \bigcap_{p=\text{pre}(T)} W_p)$ 
15:   $W_T = (\bigcap_{p=\text{pre}(T)} W_p) \cup WB_T$ 
16:  mark T as VISITED
17:  for each node  $T_n$  NOT VISITED do
18:    if all its direct preceding nodes are VISITED then
19:      Enqueue(RdyQ,  $T_n$ )
20:    end if
21:  end for
22: end while

```

– *PWS computation and breaking point insertion*

Before introducing the algorithm to compute PWS, some definitions and notions used in the analysis are given here. We use WIN_T to denote the set of variables written in task T . Furthermore, we define *read-through variable* and *write-break variable* for task T used in the analysis.

Definition 5.3.6 (Read-through variable and write-break variable). *A variable x is a read-through variable of task T , if there exists a path from the starting BB to the end BB of T , such that there is a BB_R on the path that read x , and there is no write to x on the sub-path from the starting point of T to BB_R . The set of read-through variables of T is denoted by RT_T . A variable x is a write-break variable of task T , if for all paths from the starting BB to the end BB of T , there is a write to x on the path. The set of write-break variables of T is denoted by WB_T .*

Computing PWS. Algorithm 4 gives the proposed analysis which takes the graph G' as input, computes the PWS for leading task T_i and decides a set of edges to insert breaking points for T_i . The analysis starts with node T_i (line 4), and iteratively explores all the task nodes reachable from T_i (line 5, 18-20). Note that before computing PWS, the WAR_T , RT_T , WB_T , and WIN_T are pre-computed for each task T .

In the evaluation to each task T , the PWS for T_i is updated by adding the newly identified WAR variables (line 9), which includes two cases:

- ($\text{WAR}_T - \text{W}_{T_E}$): If a variable is already a WAR variable in T , then it is potentially a WAR variable in a TEB led by T_i . If the variable is also a write-break variable in the task preceding T , the variable is no longer a WAR variable in the TEB, which has been explained in Sec. 5.3.1².

²In principle, a WAR variable x in T is excluded only if x is a write-break variable in all incoming edges,

- $R_{T_E} \cap WIN_T$: If there is a read to a variable in the preceding node of T and a write to the variable in T , a new WAR variable resulted by coalescing tasks is identified.

The above evaluation is performed for all the incoming edges of T (line 7).

Inserting breaking points. Once the PWS for T_i is updated (the new PWS'), a breaking point decision will be made immediately (line 10-14). Here, $Range(PWS)$ refers to the memory copy range for the set of variables in PWS. If the increase in the amount of memory copy exceeds a threshold, $\alpha \times Cost$, inserting a breaking point here is considered beneficial to reduce state saving overhead and thus a breaking point is inserted. If a breaking point is inserted on edge E , T_i 's PWS will not be updated, because 1) the breaking point breaks the WAR dependency of the variables that fall into the set $R_{T_E} \cap WIN_T$, and 2) a state saving is forced at the beginning of T , so T_i 's local WAR variables WAR_T need not to be saved at the beginning of T_i . The result of the new breaking point to the computation of T_i 's PWS will take effect right away and be carried into the computation in later iterations.

Note that when a breaking point is decided, it is inserted on all edges in the image set of E on G' (line 11). This is because if we allow breaking points to be inserted in only one level of the loop in G' , the run-time system has to distinguish which iteration the loop is executing to implement such a breaking point decision, which will introduce considerable run-time overhead. For run-time efficiency, we choose not to distinguish loop iterations when inserting breaking points.

The breaking point operation may bring a new issue. Consider an edge E from A to B in G . It is possible a breaking point is decided when evaluating E' , which means in the earlier evaluation to E , the effect of inserting the breaking point on E is not considered.

i.e., a write operation to x is guaranteed to exist before T is reached. The WAR variable can be prematurely excluded in the evaluation of an individual edge. However, in the evaluation of other edges, if x is not a write-break variable in the incoming node, x will be included back into PWS as a set union is conducted in PWS update.

Now we prove not considering the breaking point effect when evaluating E will not cause an actual WAR variable to be missed in the computed PWS.

Not considering a breaking point for an edge E is equivalent to including a set of new paths starting from E and the nodes along these paths in PWS computation. So we prove including the new paths does not cause WAR variables to be missed in the PWS computation, by Lemma 5.3.3.

Lemma 5.3.3. *If a variable x is a WAR variable in graph G , then x is still a WAR variable in graph G_N , where G_N is produced by adding new edges and nodes into G .*

Proof. By the definition of PWS, there must exist a path p_1 satisfying condition 1 and 2 and a path p_2 satisfying condition 3 to make x a WAR variable. No matter what read and write operations may occur on a new path p_n , adding p_n will not destroy p_1 or p_2 in G_N , so all the conditions are still met and x is a WAR variable in G_N . \square

Then we discuss cost evaluation. Whether to insert a breaking point is decided by comparing the benefit of reducing state saving and the cost of conducting a state saving operation.

The benefit is reduced memory copying by the breaking point, i.e., $\text{Range}(\text{PWS}') - \text{Range}(\text{PWS})$. The state saving operation cost, denoted by $Cost$, is the overhead of the common operations to prepare the execution environment of the next TEB, such as updating the scope of the next TEB, and setting the pointer of the leading task for the run-time system. Note that an important feature of breaking points is that inserting a breaking point will not introduce new WAR variables. In LATICS, we multiply a factor α on the cost side, meaning that a breaking point will be inserted if the benefit is larger enough than $Cost$. The value of α should be set by the system designer, and different values of α may have different results in inserting breaking points. In this work, we set α to 5. Further exploration of other α settings is left to future work.

In our implementation, we add a dummy task T_{BE} on the edge with a breaking point to avoid run-time maintenance of program paths. Once T_{BE} is reached, the PWS of the immediate succeeding task of T_{BE} will be saved.

Updating read and write sets. After the PWS is computed, R_T and W_T are updated for task T (line 15-16). The purpose of maintaining R_T and W_T is to transfer the read and write behaviors of the data on NVM to T 's succeeding nodes to be involved in identifying new WAR variables in future iterations.

R_T first includes the reading set for all incoming nodes ($\bigcup_{p=\text{pre}(T)} R_p$), but if a write to a variable x in R_T is guaranteed to occur for all paths in T , x is excluded, as the read to x no longer cause WAR dependency in future nodes to be explored. The read-through variables of T are also included into R_T . But if a write to a read-through variable is guaranteed at all possible incoming edges, the variable will not cause WAR dependency in the future and is thus excluded. ($R_T - \bigcap_{p=\text{pre}(T)} W_p$) implements the latter case.

The update of W_T is relatively simple. If a write-breaking variable appears in all incoming nodes, it is included into W_T . The write-breaking variables in T are also included into W_T and carried to the evaluation of future nodes.

The algorithm iterates until all nodes reachable from T_i are explored. The computational complexity of the proposed algorithm is $O(N)$, where N is the number of nodes in G' .

– A running example

To illustrate how Algorithm 4 works, we use an example in Fig. 5.3 to show the computation of PWS and breaking point insertion. For simplicity, we omit the evaluation to T_1 . After T_1 is evaluated, we have:

$$\text{PWS} = \{a\}$$

$$\text{RdyQ} = \{T_2, T_3\}, \quad R_{T_1} = \{b\}, \quad W_{T_1} = \emptyset$$

Step 1. Dequeue T_2 and compute PWS' :

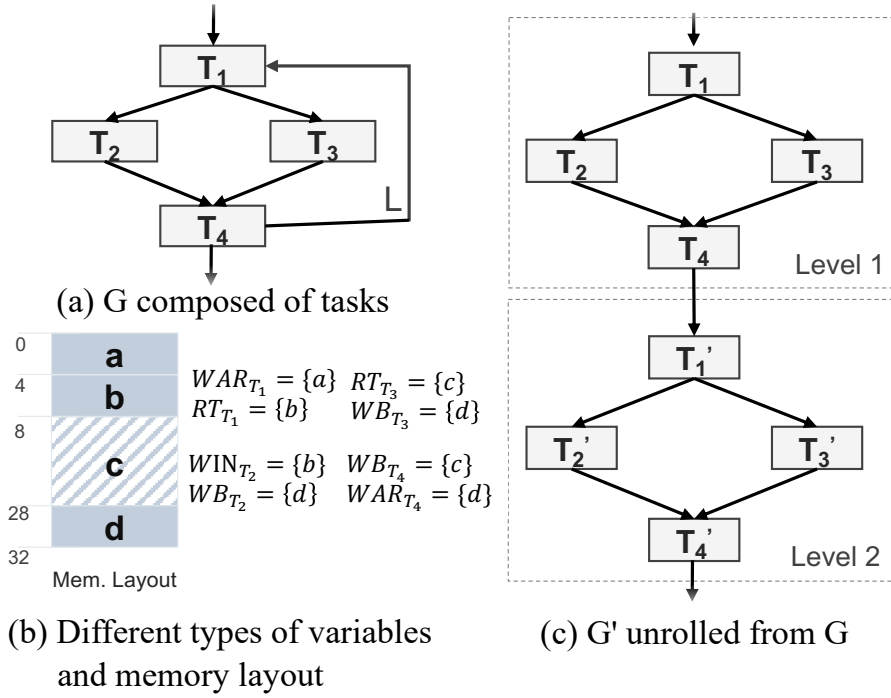


Figure 5.3. An example to explain PWS computation and breaking points insertion

$$PWS = \{a\}, \quad PWS' = \{a, b\}$$

$$RdyQ = \{T_3\}, \quad R_{T_2} = \{b\}, \quad W_{T_2} = \{d\}$$

In this step, there is a path from T_1 to T_2 on which T_1 reads b and T_2 writes b , so b is a new WAR variable and is included in PWS' . Given $\alpha = 5$ and $Cost = 1$, $Range(PWS') - Range(PWS) < 5$, so no breaking point is inserted, and PWS is updated as PWS' . Then R_{T_2} and W_{T_2} are also updated with the information in Fig. 5.3(b).

Step 2. Dequeue T_3 and compute PWS' :

$$PWS = \{a, b\}, \quad PWS' = \{a, b\}$$

$$RdyQ = \{T_4\}, \quad R_{T_3} = \{b, c\}, \quad W_{T_3} = \{d\}$$

Similar to step 1, R_{T_3} , W_{T_3} and PWS are updated. After the analysis of T_3 , T_4 is inserted into $RdyQ$ as all of its preceding nodes (T_2 and T_3) have been visited.

Step 3. Dequeue T_4 and evaluate the edge from T_2 to T_4 :

$$PWS = \{a, b\}, \quad PWS' = \{a, b\}$$

Then evaluate the edge from T_3 to T_4 :

$$PWS = \{a, b\}, \quad PWS' = \{a, b, c\}$$

Since $\text{Range}(PWS') - \text{Range}(PWS) = 20 > 5$, we insert a breaking point on this edge, and do not update PWS. R_{T_4} , W_{T_4} and RdyQ are updated as follows:

$$\text{RdyQ} = \{T_1'\}, \quad R_{T_4} = \{b\}, \quad W_{T_4} = \{c, d\}$$

After all the nodes in G' have been explored, the final PWS_{T_1} is $\{a, b\}$. A breaking point is inserted on the edge from T_3 to T_4 . Each time this breaking point is reached at run-time, PWS_{T_4} will be saved.

5.4 The Run-Time System

We build a run-time system to implement the core functionalities of the proposed approach. The run-time system makes dynamic decisions on task coalescing, manages data buffers, conducts state saving and restoration, and most importantly enforces mandatory state saving at breaking points. We will first give the work flow of LATICs' run-time system and then detail its main components.

5.4.1 The Main Work Flow

Algorithm 5 The run-time system

- 1: Let $currTEB$ be a newly created TEB
- 2: Let T_0 be the leading task of $currTEB$
- 3: **if** recovered from power failure **then**
- 4: RESTORE(T_0)
- 5: **end if**
- 6: $currTEB = \text{DECISION}()$;
- 7: **while** not end of the program **do**
- 8: $T_i = T_0$

```

9:   if no power failure then
10:     SAVE( $T_i$ )
11:   end if
12:   while  $currTEB$  is not finished do
13:     if  $T_i$  is a breaking task then
14:        $T_0 = T_i$ 
15:       SAVE( $T_i$ )
16:     end if
17:     EXECUTE( $T_i$ )
18:     UPDATE_HISTORY()
19:      $T_i = NEXT(T_i)$ 
20:   end while
21:    $currTEB = DECISION()$ 
22: end while

```

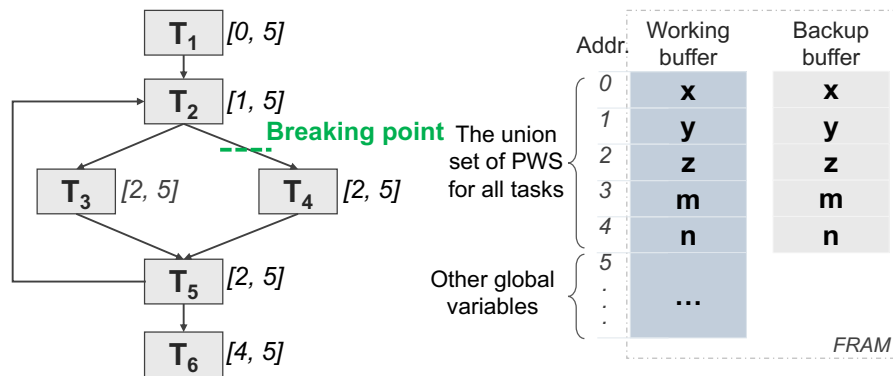


Figure 5.4. A running example of the run-time system

The behavior of the run-time system is specified by Algorithm 5, and we use the example in Fig. 5.4 to explain how it works. A possible execution trace may be as follows.

- The program starts and the run-time system decides a TEB led by task T_1 and containing 3 tasks.

- System states are saved by copying the data in memory copy range $[0, 5]$ covering PWS_{T_1} from the working buffer to the backup buffer.
- T_1 , T_2 and T_3 execute until T_3 is finished.
- As the current TEB is done, the run-time system creates a new TEB at this point, and makes a decision to let the TEB coalesce 6 tasks.
- As T_5 is the leading task of the new TEB, data in memory copy range $[2, 5]$ covering PWS_{T_5} are copied from the working buffer to the backup buffer.
- A power failure occurs in the execution of T_2 (after T_5).
- After recovery, the run-time system identifies the power failure, therefore, it restores the system states at the beginning of the aborted TEB, by copying data in memory copy range $[2, 5]$ from the backup buffer to the working buffer.
- The run-time system creates a new TEB the TEB coalesces 4 tasks, and execution resumes from T_5 . The new TEB continues and the program takes the right branch.
- After T_2 is finished, the run-time system encounters a breaking point on the incoming edge to T_4 . Although the current TEB is not finished yet, a state saving is forced at this point, saving the data in memory copy range $[2, 5]$ covering PWS_{T_4} to the backup buffer. As a result, if a power failure occurs later, the system will resume execution from T_4 instead of the original leading task T_5 .
- T_5 is finished, and a new TEB starting from T_2 will be decided by the run-time system.
- Execution continues until the program is completed.

5.4.2 Core Components

LATICS allocates local variables that are only accessed in a single task on SRAM, and allocates the variables shared among tasks on FRAM. LATICS maintains two buffers for the shared data: a *working buffer* and a *backup buffer*. The working buffer serves as part of the

main memory and stores all the shared data. The backup buffer stores a backup copy for the set union of the PWS for all leading tasks, which is a subset of the shared data. Note that the PWS for each leading task is computed from the shared data.

The SAVE() function conducts state saving. It is invoked when the program starts for the first time, and when a TEB successfully finishes execution and the run-time system decides a new TEB. The main operation is to save the variables in the memory address range covering the PWS of the leading task from the working buffer to the backup buffer.

The RESTORE() function restores states and is only invoked after the system recovers from a power failure. States saved at the beginning of the aborted TEB are restored. The main operation is to copy the variables in the memory copy range covering the PWS of the leading task from the backup buffer to the working buffer. SAVE() and RESTORE() are the key operations to ensure memory consistency for the system.

DECISION() is the function to realize adaptive execution by deciding the TEBs. The function is invoked either after a TEB successfully finishes or after the system recovers from a power failure. In LATICS, DECISION() predicts the available energy considering the execution history recorded by UPDATE_HISTORY(), and then makes a decision on the number of tasks to be coalesced into the next TEB.

Enforcing the breaking point mechanism. During execution, if the run-time system identifies a breaking point before a task T, it ends the current TEB and conducts state saving by invoking SAVE(T). From now on, the rest of the original TEB forms a new TEB and continues execution. If power fails in the new TEB, the system will later resume from T, instead of the leading task of the original TEB.

Note that the state save/restore mechanism in LATICS differs from the state-of-the-art task-based system InK [70]. In both LATICS and InK, task execution operates on the working buffer. After a task is finished, InK changes the role of its working buffer and backup buffer by a pointer swap, and right before starting the next task, the run-time system copies all data in the new backup buffer to the new working buffer for memory consistency. While LATICS maintains only a subset of the data (the set union of the PWS for all leading

tasks) in the backup buffer. By this means, LATICS reduces memory usage and avoids unnecessary state saving.

5.5 Experiments and Evaluation

5.5.1 Experimental Setup

LATICS provides a solution for efficient adaptive execution of task-based intermittent computing systems. Currently, LATICS is implemented based on the state-of-the-art task-based system InK [70]. In principle it can be applied to other task-based systems as well. The LATICS run-time system and the application program are compiled into the binary executable using the TI v18.12.4 compiler by TI Code Composer Studio.

We deploy LATICS on TI's MSP430FR5994 launchpad equipped with a microcontroller, 8KB volatile memory (SRAM) and 256KB non-volatile memory (FRAM) all clocked at their highest speed. We allocate local variables of each task on SRAM, and the data that are shared by tasks on FRAM. Bulk-copying between memories uses DMA. The hardware board is powered by a programmable power supplier by which we can generate different power traces to evaluate LATICS under different power conditions.

LATICS is compared against two state-of-the-art task-based intermittent computing systems: InK [70] and Coala [51]. InK is a task-based system which saves states at all task boundaries. A working buffer and a backup buffer are implemented to manage the data allocated on NVM. LATICS differs with InK, in that LATICS is an adaptive task-based intermittent computing system which may skip state saving when power supply is sufficient, and more importantly, with the support of static analysis, LATICS saves only the PWS of a task that leads a TEB which is a small subset of the data on NVM. Coala is a recently proposed adaptive task-based intermittent computing system. At run-time, Coala dynamically predicts the number of tasks to coalesce from execution history. Coala implements a paging system and copies used data pages on-demand to VM, and at the end of a TEB, saves all modified data pages to a backup buffer. Comparably, LATICS adopts the breaking point mechanism

to reduce unnecessary state saving, and only saves PWS for the leading task of a TEB.

We use 8 benchmark programs adopted in InK and Coala to evaluate system performance. The benchmarks are Cold-chain Equipment Monitoring (CEM), Cuckoo Filter (CK), Cyclic Redundancy Check (CRC), Bitcount (BC), Dijkstra shortest path algorithm (DIJ), Selection sort algorithm (SRT), RSA encryption (RSA) and Activity Recognition (AR).

In the benchmarks, the read/write to the global variables shared among tasks are explicitly programmed by two macros, `__GET(v)` and `__SET(v)` where v is a global variable. The transitions between tasks are programmed by the `NEXT(T_i)` macro indicating the next task is T_i . The CFG for each task is extracted from the task's source code and the program graph G is constructed by connecting the tasks' CFG with the task transitions. WAR dependency is obtained by exploring G and tracking the read/write operations.

5.5.2 Evaluation Methods

LATICS' performance is evaluated and compared with InK and Coala with the above benchmarks. We run each benchmark program 20 times and report the averaged execution time for a single run. The *total execution time* for each benchmark is recorded from the start to the end of its execution and is further divided into *state saving time* and *task execution time*. The state saving time is the time spent on state saving at the beginning of the TEBs. The task execution time refers to the time to execute the task code. Both parts include the partial execution in the aborted TEBs resulted by power failure.

For task coalescing, we adopt Coala's *Weighted Energy-Guided Coalescing Strategy* for fair comparison. The strategy essentially collects the execution history to obtain an energy capacity indicating how many tasks can be coalesced within the energy capacity. Once a TEB successfully finishes, the capacity will be halved, and once a power failure occurs, the capacity will be updated according to the past execution history. All the parameters are set to the same with Coala.

To test the compared systems under different power supplies, we conduct experi-

ments with *periodic power traces* in which power failure occurs periodically during system execution with a given period called *power cycle*. Tab. 5.2 reports the results under periodic power traces, where the second column is the power cycle. Experiments are conducted for eight power cycle configurations $1ms, 2ms, 3ms, \dots, 8ms$, which are in the typical range of power cycles in intermittent systems powered by capacitors [48]. Due to limited space, only the results for power cycle $1ms, 2ms, 4ms, 8ms$ are listed in Tab. 5.2. We also conducted experiments on *random power traces* that reflect real-life scenarios. In these traces, the time gap between two consecutive power failures are randomly chosen from range $[1ms, 8ms]$. Fig. 5.5 reports the experimental results under random power traces.

We also further discuss: (1) the effectiveness of our proposed approach by comparing it with an exhaustive search method; (2) the impact of α to breaking point decision; (3) performance of different state copying methods.

5.5.3 Empirical results and evaluation

For all experiments under all power traces, LATICS outperforms InK and Coala w.r.t. the state saving time and the total execution time, which shows that LATICS has low state saving overhead and high execution efficiency. We now give detailed evaluation to the results.

First, we investigate how much state saving can be reduced by our proposed approach. Tab. 5.1 lists the results of InK and LATICS. The numbers for InK are the total state saving size for all tasks, i.e., a product of the size of system states and the number of tasks in the program. For LATICS, we sum up the sizes of the memory copy ranges covering the PWS for all tasks (assuming each task can serve as a leading task) computed by our proposed method. The results show that, by inserting breaking points and by the computation of PWS, very few data need to be saved at the leading tasks of TEBs, so state saving overhead is significantly reduced in LATICS.

Second, the performance of LATICS under random power traces is evaluated, the results of which are shown in Fig. 5.5. The figure illustrates the total execution time for LATICS, InK and Coala, normalized to that of InK. The grey part in each bar refers to the

Table 5.1. State saving size for all tasks (in bytes)

	CEM	CRC	BC	CK	AR	DIJ	RSA	SRT
InK	41904	16	220	4260	1804	3570	4200	824
LATICS	260	8	170	3452	324	140	1624	216

state saving time, and the slashed part refers to the task execution time. For all benchmarks, LATICS has the smallest total execution time. InK consumes a large portion of time on state saving, as it pessimistically saves all system states at all task boundaries. Coala saves at the end of a TEB the data pages that have been modified since the beginning of the TEB. Coala's state saving is not efficient, since (1) writes to non-WAR variables may also cause a data page to be saved; (2) as Coala conducts memory copy in pages, the actual amount of data copied in state saving is larger than the total size of all modified variables. LATICS outperforms InK and Coala, because LATICS adaptively group tasks into TEBs and by adopting the breaking point mechanism and the PWS computation, unnecessary state saving at the beginning of the TEB is avoided as much as possible. Similar comparison results can also be witnessed in the experiments on periodic power traces, shown in Tab. 5.2.

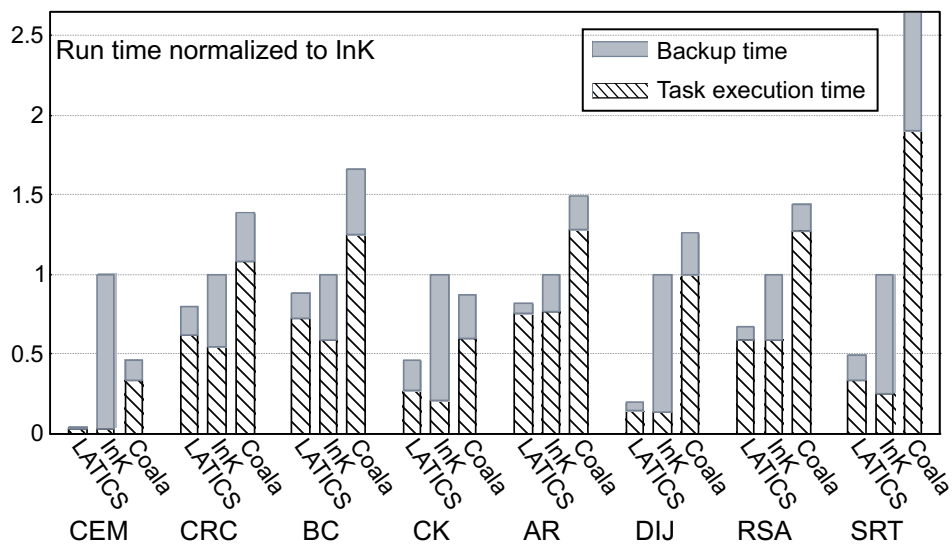


Figure 5.5. Execution time results under random power traces (normalized to InK's total execution time)

To further explain the results, we explore in detail the CEM benchmark which has significant performance improvement. The program graph of CEM (tasks' CFG omitted, only showing inter-task transitions) is given in Fig. 5.6 which contains 12 tasks. There are read operations to a large-size array A in T_5 , T_6 and T_7 and write operations to A in T_8 . Breaking points were inserted on the incoming edges to T_8 , which breaks the WAR dependency of A and thus considerably reduces state saving. Some benchmarks, such as RSA, do not exhibit significant improvement in Fig. 5.5, because they spend execution time in loops that do not access the variables that are in InK's states but excluded by the breaking points in LATICS.

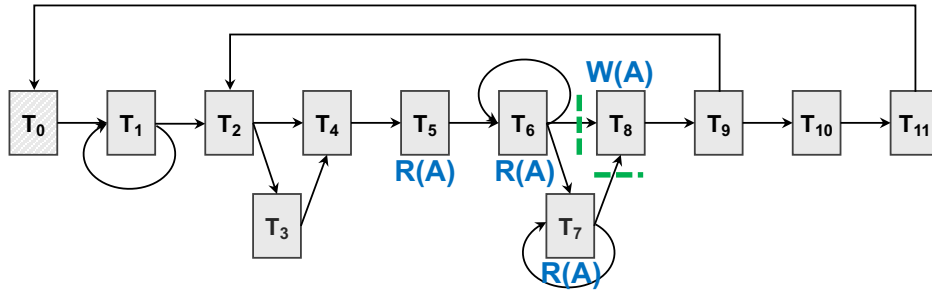


Figure 5.6. The program graph of CEM

Third, experimental results for periodic power traces are shown in Tab. 5.2 (Dashes in the table mean the program failed to progress). Regarding task execution time, LATICS provides comparable performance to InK, only slightly larger in some benchmarks. As tasks in LATICS are coalesced into larger TEBs, in the presence of power failures, LATICS may undergo more re-execution time with larger atomic execution blocks (i.e., TEBs). Note that this is a general phenomenon for all adaptive task-based intermittent computing systems. Coala performs worst in task execution time. To avoid the WAR problem, Coala adopts an inefficient paging system which copies the requested data page on-demand to VM. Coala's paging system considerably sacrifices execution efficiency. Also shown in Tab. 5.2, when power cycle increases, LATICS performs better with less state saving overhead, in that the benefit of skipping state saving within the TEBs increases with larger power cycles. We also

list the number of tasks that are executed in one power cycle for all benchmarks in Tab. 5.3. LATICS executes more tasks in a power cycle than InK and Coala, which also shows that LATICS has less state saving overhead and thus generally makes better progress.

Table 5.2. Execution time results under periodic power traces with different power cycles

		Coala		InK		LATICS	
Apps	Power cycle (ms)	Backup (ms)	Total (ms)	Backup (ms)	Total (ms)	Backup (ms)	Total (ms)
CEM	1	--	--	777.39	800.33	2.26	25.66
	2	76.51	259.37	617.30	640.25	1.60	24.73
	4	52.19	258.84	604.71	627.21	1.23	23.79
	8	30.91	225.44	586.84	609.79	1.08	24.10
CRC	1	0.73	3.15	0.92	2.14	0.07	1.28
	2	0.37	2.80	0.93	2.16	0.04	1.26
	4	0.25	2.68	1.01	2.20	0.03	1.27
	8	0.36	2.80	0.92	2.14	0.02	1.25
BC	1	7.44	32.85	8.46	20.33	0.79	12.85
	2	4.19	29.01	8.36	20.17	0.53	12.62
	4	2.58	27.97	8.15	19.76	0.33	12.07
	8	1.79	27.13	8.30	20.09	0.17	12.03
CK	1	8.85	25.83	20.10	25.23	1.27	6.41
	2	5.78	22.84	19.74	24.89	0.80	6.08
	4	3.78	21.7	19.26	24.29	0.34	5.37
	8	2.24	20.23	19.59	24.72	0.23	5.37
AR	1	38.11	199.76	29.81	144.15	4.19	99.73
	2	31.24	187.17	27.66	119.75	2.29	96.68
	4	16.34	170.24	26.63	114.65	1.77	88.11
	8	8.45	118.60	25.49	113.06	0.63	88.74
DIJ	1	98.58	384.15	224.31	258.76	5.08	38.86
	2	79.77	366.07	212.71	247.19	3.10	37.01
	4	52.53	329.01	207.78	241.51	2.23	36.26
	8	34.39	313.66	210.49	244.97	1.67	35.81
RSA	1	14.72	101.53	30.21	72.33	2.58	48.00
	2	9.77	83.28	29.00	70.29	2.64	48.00
	4	6.33	77.64	28.85	70.13	2.34	45.72
	8	3.05	61.35	29.19	70.67	2.85	44.38
SRT	1	216.16	741.1	173.65	231.04	10.78	68.27
	2	99.27	640.23	169.17	226.52	5.84	63.30
	4	95.33	539.46	169.00	226.47	3.18	60.54
	8	92.72	500.23	171.54	230.10	1.76	59.15

Table 5.3. The number of tasks finished in each power cycle

Power cycle (ms)	Methods	CEM	CRC	BC	CK	AR	DIJ	RSA	SRT
1	Coala	–	39	27	21	4	8	5	7
	InK	2	49	41	21	8	10	10	25
	LATICS	64	96	66	76	12	65	17	85
2	Coala	13	91	59	44	7	18	10	15
	InK	5	98	82	43	18	21	22	51
	LATICS	131	199	136	161	24	137	34	182
4	Coala	27	190	115	91	16	37	19	32
	InK	10	200	167	86	36	42	44	103
	LATCIS	267	402	275	335	49	280	71	380
8	Coala	55	385	236	186	26	77	38	68
	InK	21	401	344	174	74	85	89	207
	LATICS	551	803	605	675	98	563	142	794

5.5.4 Further Discussion

– *Effectiveness of the breaking point insertion approach*

To justify the effectiveness of our proposed breaking point insertion approach (Algorithm 1), we implemented an exhaustive search method (abbreviated ES) to find the “optimal” solution and compared it with our approach.

In fact, the optimal breaking point locations depend on the program paths and the power traces, and an optimal solution only exists with known program path and power trace. As the executed program path and the actual power trace is only known at run time, a clairvoyant method is required to search the optimal solution.

To this end, we run each benchmark program once with a periodic power trace (1ms power cycle) and obtain an execution history. Then an induced program graph is generated by excluding the paths not taken in the execution history. An exhaustive search is conducted

on the induced program graph by enumerating whether a breaking point is inserted on each edge of the graph. Note that as data inputs for all benchmarks are fixed in our experiment, the program is guaranteed to take the same path each time it is executed. In each experiment, we run the benchmark program with the same power trace and record the total run time. If the search does not finish after 20 hours, we stop the search and report the best solution so far. Table 5.4 gives the results for ES and our approach.

Table 5.4. The minimal execution times (in *ms*) obtained by ES and LATICS under periodic power trace with *1ms* power cycle

	CEM	CRC	BC	CK	AR	DIJ	RSA	SRT
ES	49.02	1.27	13.01	6.13	97.84	35.29	41.93	64.04
LATICS	25.66	1.28	12.85	6.41	99.73	38.86	48.00	68.27

We compare the minimal total run time obtained by ES and the total run time by LATICS. The exhaustive search for CRC, DIJ, and SRT were finished and the results obtained are optimal. It can be seen that the total run times of LATICS for the three benchmarks are slightly larger than those of ES. The reasons are two fold: first, ES is clairvoyant while LATICS is not; second, our approach, i.e. Algorithm 1, is a heuristic method which only considers a subset of the problem state space. The exhaustive search for the other 5 benchmarks did not finish within 20 hours. This is because even the program path is known, searching for an optimal breaking point assignment still needs to explore a huge state space. Note that 20 hours are still not enough for ES to find sufficiently good results so the execution times obtained by ES are larger than those of LATICS for some benchmarks.

– The impact of α to breaking point insertion

Parameter α is involved in deciding whether a breaking point will be inserted when Algorithm 1 explores the unrolled graph (line 10). Only when the difference in the size of memory copy range exceeds $\alpha \times Cost$, a breaking point is inserted. We conducted experiments with $\alpha = 2, 5, 10$ under a power trace with *1ms* power cycle to explore how α will affect the results.

Table 5.5. Results with different α values under periodic power trace with 1ms power cycle

	$\alpha = 2$		$\alpha = 5$		$\alpha = 10$	
Apps	Backup (ms)	Total (ms)	Backup (ms)	Total (ms)	Backup (ms)	Total (ms)
CEM	7.54	27.27	2.26	25.66	2.26	25.66
CRC	0.07	1.28	0.07	1.28	0.07	1.28
BC	3.49	15.58	0.79	12.85	0.79	12.85
CK	2.67	7.14	1.27	6.41	1.27	6.41
AR	4.19	99.73	4.19	99.73	5.90	111.77
DIJ	5.08	38.86	5.08	38.86	5.08	38.86
RSA	2.98	48.24	2.58	48.00	4.55	49.60
SRT	10.78	68.27	10.78	68.27	10.78	68.27

The total run time and backup time are shown in Tab. 5.5. First, the total run times for all three configurations are very close, which shows that the values of α within the explored range do not significantly affect the results for the benchmarks. The total run time is the lowest for the $\alpha = 5$ configuration. If the value of α is too small, e.g., $\alpha = 2$, too many breaking points will be inserted, so the benefit of task coalescing is decreased. If the value of α is too large, e.g., $\alpha = 10$, too few breaking points will be inserted, which compromises the benefit of breaking point.

– Performance of state copying

In LATICS, the PWS is copied to a backup buffer at the beginning of a leading task. Instead of copying the exact PWS, we copy a memory copy range that covers the PWS. The main motivation is that bulk-copying by DMA is much faster than copying the variables one-by-one. As the variables of a PWS may not be stored in continuous addresses in the memory, a memory copy range may contain variables other than the PWS to save at a point.

Even though, copying the data in the memory copy range in bulk is in most cases more efficient.

In Tab. 5.6, we report the time overhead of three copying methods running on our MSP430 board, bulk-copying by DMA (DMA bulk), variable-copying by DMA (DMA copy), and variable-copying by CPU (implemented by `memcpy()`, CPU copy), for different data sizes. We provide results for copying 2B to 512B data. For variable-copying, we set the variable size to be 2B, the size of an integer on MSP430.

For variable-copying, the time overhead is linearly proportionally to the copied data size, with DMA slightly faster than CPU. For bulk-copying by DMA, the copy speed continues to increase from 2B to 512B. From 16-32B data size, bulk-copying by DMA is one order of magnitude faster than variable-copying approaches. 70 memory copy ranges exist in all benchmarks, 13 of them are sized 2-15B, 43 of them are sized 16-64B, and 14 of them are sized 65-284B. So in most cases, conducting bulk-copying is more efficient. The results explain why we choose bulk-copying in LATICS even if extra variables may be included in a memory copy range. In general, large programs with large state copying size will benefit more from bulk-copying.

Table 5.6. Time cost (in μs) for different state copying methods under different data sizes (in bytes)

Methods	2	4	8	16	32	64	128	256	512
CPU copy	7.5	15.8	31.5	61.8	123.5	247	494	988	1976
DMA copy	7.2	14.4	28.8	57.5	115	230	460	920	1840
DMA bulk	7.2	7.4	7.5	8.3	10	13.3	19.8	32.8	58.5

The actual memory layout of the state variables may impact the size of the memory copy range and thus the state copying overhead. In general, there does not exist an optimal layout in reality, as the efficiency of a memory layout is related to the actual occurrence of leading tasks which is further affected by the actual power traces. We leave systematic investigation in memory layout optimization to future work.

5.6 Conclusion

This chapter presents LATICS, a low-overhead adaptive task-based intermittent computing system. The major challenge addressed by LATICS is how to reduce the amount of saved states in the presence of dynamic decision of how many tasks are grouped together for atomic execution. We observed that it is not always beneficial to skip state saving at task boundaries as it may sometime cause higher amount of saved states at other places and thus leads to higher overall run-time overhead. LATICS is implemented on top of the state-of-the-art task-based intermittent computing system InK [70], and evaluated on a hardware based on MSP430. Experimental results show that LATICS significantly reduces state saving overhead and improves execution efficiency compared to the state-of-the-art.

CHAPTER 6

INTERMITTENT COMPUTING WITH EFFICIENT STATE BACKUP BY ASYNCHRONOUS DMA

6.1 Introduction

Energy harvesting is a promising approach which allows a device to rely on energy harvested from the ambient environment. As energy output of harvesters is typically weak and unstable, the computing system must ensure software programs will make progress in the presence of frequent power failures. To this end, a new computing paradigm called intermittent computing is proposed [25]. In intermittent computing systems, a program progresses incrementally with the granularity of program segment. At the end of each program segment, the program states are backed up to non-volatile memory (NVM), such as FLASH or FRAM, and then the system continues to execute the next program segment. Once a power failure occurs and later the system recovers, the backup states on NVM are reloaded so that the execution can resume from the beginning of the failed segment, instead of conducting a complete restart from the very beginning of the program. State backup is very time-consuming and is frequently conducted during the progress of a program. Although Direct Memory Access (DMA) is used in most systems to accelerate data copying, the time overhead of state backup is still very large.

To the best of our knowledge, all existing intermittent systems, with documented method on state backup, conduct state backup sequentially with program execution, i.e., during state backup the program has to stop and wait. However, if state backup can be conducted in parallel with program execution, then the state backup latency will be hidden

in program execution. As a result, the program has a much smaller total execution time, and thus can make better progress.

In this work, we seek to parallelize state backup and program execution by asynchronous DMA. The main challenge is data racing may cause incorrect program states that are polluted by program execution to be backed up into NVM. In our solution, we parallel state backup with program execution, and at the end of the state backup detect whether an error has occurred. Moreover, even if an error is detected, we do not immediately handle the error, but let the program continue to execute, as it is highly probable that the erroneous backup will be covered by a future correct backup. We have implemented an intermittent system based on the proposed approach. Experiments conducted on an STM32F7-based platform show that the proposed method can efficiently detect and cover state backup errors, and by parallelizing state backup with program execution, system performance is considerably improved.

6.2 Related Work

Intermittent computing [42,47] is recently proposed to enable a system to keep progress and produce correct computation in the presence of frequent power failures. The main principle is to let the system timely back up program states to NVM so that when the system recovers from a power failure, it can resume from the most recent backup states instead of experiencing a complete restart. With the increasing number of energy harvesting IoT devices [3], intermittent computing is an enabler for such devices to provide sustainable services.

As state backup is a major overhead in intermittent computing, we did a survey of existing intermittent systems on their state backup methods. A classification of state backup methods is given in Table 6.1 (For intermittent systems that are not named, we use the surname of the first author to represent the proposed system or method). All systems with documented state backup method adopt either CPU or DMA to conduct state backup. First, in all systems that use CPU, state backup is sequentially conducted with program execution. Second, systems that use DMA also conduct state backup sequentially with program exe-

Table 6.1. A survey of state backup methods

Methods	Intermittent systems
CPU	Cotai [58], Chain [20], Mementos [56], Alpaca [48], QUICKRECALL [32], Ratchet [66], Chinchilla [49]
DMA	InK [70], Daulby [23], Chen [17], TICS [36], Coala [51], CoSpec [19], ELASTIN [18], CatNap [50]
Unknown	Hibernus++ [10], HarvOS [11]

cution. In CoSpec [19], a specialized hardware, which is not widely available in embedded MCUs, is required in state backup. This chapter towards reducing state backup overhead by parallelizing state backup and program execution. Note that in intermittent systems that adopt Just-in-Time checkpointing [10], system execution is firstly stopped and then the system states are saved to NVM, so state backup is always serially conducted with system execution.

6.3 The Parallel State Backup Problem

This section shows the correctness problem caused by parallel state backup and program execution. In sequential state backup using either CPU or DMA, a program segment is allowed to start only after the program states at its starting point are totally copied to NVM. Now consider to conduct state backup simultaneously with the execution of a program segment. We will show by the example in Fig. 6.1 that a critical correctness problem may occur.

Assume A is one of the variables to back up. The execution of program segment T operates A during execution, and state backup copies state variables including A to NVM. A correct state backup at the beginning of T should copy A 's value 1 to NVM. The execution

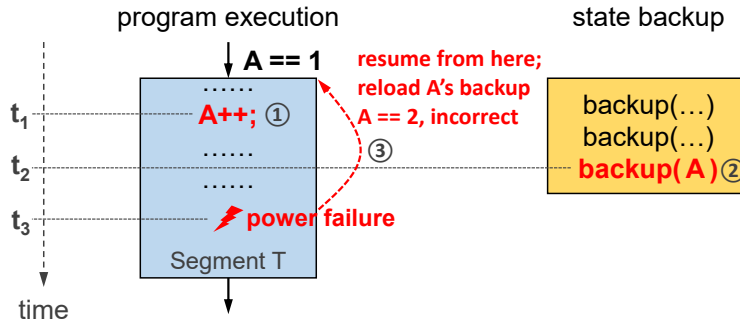


Figure 6.1. An example of the parallel state backup problem

of T modifies A 's value to 2 by an increment operation at time t_1 , while the backup of A is conducted at a later time t_2 . As a result, the NVM stores an incorrect value of A (i.e., 2). If a power failure occurs at time t_3 , after the system recovers, the program resumes from the beginning of T by reloading the backup states from NVM. Now an incorrect value of A is reloaded. We call this phenomenon the “*parallel state backup problem*”. The problem will further result in incorrect computation of T.

Due to the lack of techniques to solve the parallel backup problem exemplified by Fig. 6.1, to the best of our knowledge, existing intermittent computing systems pessimistically serialize task execution and state backup (by either CPU or DMA). The consequence is that a large portion of the system execution time is spent on frequent state backup [59].

6.4 Overview of Our Approach

The objective of this work is to parallelize state backup and program execution for intermittent systems without destroying logical correctness. Such a design allows to hide state backup latency in task execution time, and thus can improve system performance. In this section, we will give an overview of the main idea. Key design details are provided in the next section.

The main idea is carried by the work flow shown in Fig. 6.2. The system behavior mainly contains a loop to execute program segments one after another until the program is

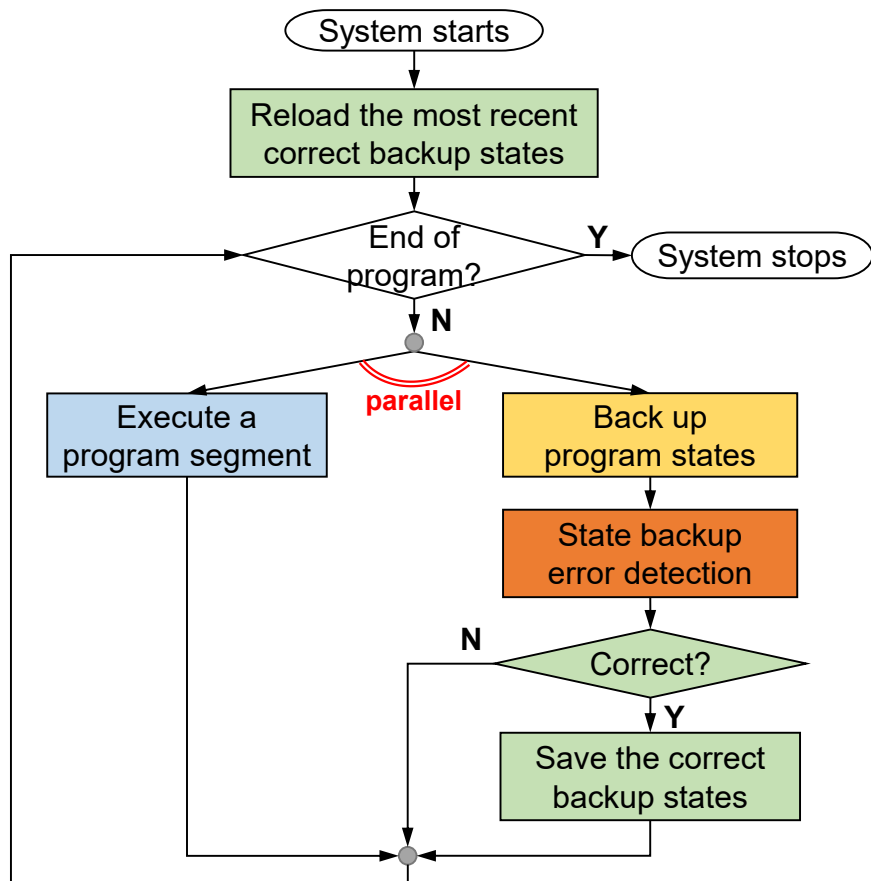


Figure 6.2. Overview of the proposed state backup approach

finished. In our approach, we conduct state backup simultaneously with the execution of a program segment. As this may cause the problem shown in Fig. 6.1, we propose a technique to detect potential backup errors at the end of the state backup (The error detection technique will be detailed in Sec. 6.5).

Once a state backup error is detected, we know the current backup data is not trustworthy and can not be used for system restoration. A common method is to completely restart the system. However, a complete restart will incur too much re-execution overhead, so we try to avoid it. Note that state backup is performed timely and frequently in intermittent systems. It is highly probable that an incorrect backup can be overwritten by a future correct backup. Motivated by this observation, we propose *fault-tolerant backup management* to allow the system to continue execution even if the backup states are temporarily incorrect. We will explain the idea with the example in Fig. 6.3.

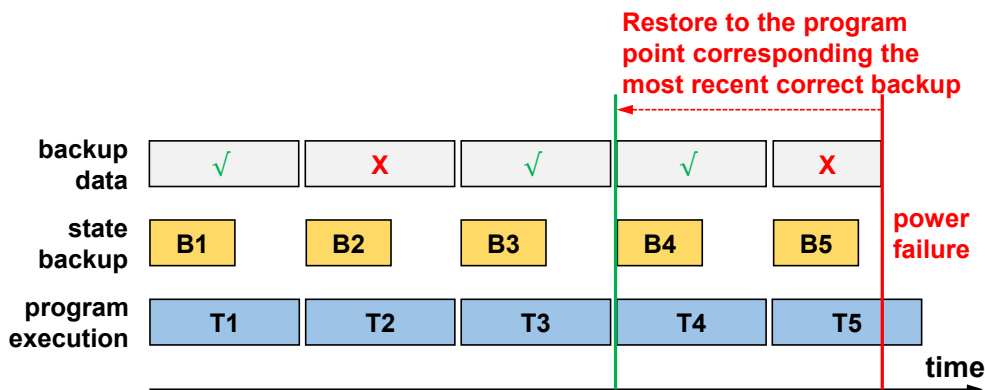


Figure 6.3. Fault-tolerant backup management

In Fig. 6.3, T_i represents program segments, and B_i represents the state backup conducted in parallel with T_i . When backup B_2 is done, a backup error is detected, leading the backup data to an inconsistent state. However, we do not immediately solve the problem, and simply allow the system to continue executing T_3 . As the next state backup B_3 is correct, the incorrect backup states are covered by new correct backup states that can be used for system restoration. Note that state backup B_5 is also incorrect, and unfortunately during the execution of T_5 , a power failure occurs. As the backup states now are incorrect, the system

is not able to restore to the beginning of T_5 . Since backup B_4 is correct, the system can roll back to the beginning of T_4 , i.e., the program point corresponding the most recent correct backup. The fault-tolerant backup management, as part of the whole solution, is depicted by the green blocks in Fig. 6.2.

The next section will present the detailed design for state backup error detection and fault-tolerant backup management.

6.5 Design

In this section, we first introduce the task-based software model [70] which is used to explain the proposed techniques. Then state backup error detection is presented in detail. At last, we present a heuristic to explore how memory layout can affect the occurrence of state backup errors.

6.5.1 Software Model

By task-based model, a program is coded as a collection of *tasks*, where each task is essentially a program segment implemented as a function. The tasks are connected by the control flows specified by the programmer. During the execution, state backup is conducted at the beginning of each task. If a power failure occurs, and later the system recovers, it first restores the program states from the backup, and then continues to execute the last unfinished task before power failure. Program states in task-based model refer to the set of variables that are shared by multiple tasks and whose lifetimes span across task boundaries, but not the local variables allocated on the stack and accessed inside the task. To distinguish data copies, we say task execution accesses task-shared variables stored in a *working buffer* in main memory, and in state backup, task-shared variables are copied from the working buffer to a *backup buffer* on NVM using DMA.

6.5.2 State Backup Error Detection

Target for Error Detection

In the general sense, any task-shared variable in the working buffer may experience the problem in Fig. 6.1, if state backup is conducted in parallel with task execution. However, not all backup errors will eventually lead to incorrect program execution. Error detection only needs to observe those backup errors that affect the correctness of task execution.

Whether the backup error of a variable will cause incorrect execution depends on the access behavior of the variable. Without loss of generality, we consider a task-shared variable A accessed in task T . The behaviors of accessing A can be classified into the following cases which have different results on the correctness of task execution.

- *CASE 1: A is only read in T .* No backup error will occur as the execution of T does not change A 's value.
- *CASE 2: A is only written in T .* A write to A may cause incorrect state backup and assume such a case occurs. If power fails in T , the system resumes from the beginning of T . Although an incorrect value of A will be reloaded, the incorrect value will be overwritten by the write operation to A in T , so task execution remains correct. Otherwise, if power failures do not occur, at the end of T , the correct value of A remains in the working buffer. The execution of the next task will execute on the working buffer and the task sees a correct value of A .
- *CASE 3: A is first written and then read in T .* This case is the same as CASE 2.
- *CASE 4: A is first read and then written in T .* T 's execution will be incorrect if an incorrect value of A is backed up and power fails in T . After system resumes, the incorrect backup value of A is reloaded from the backup buffer. The first read to A loads the incorrect value, which may cause incorrect computation. Variable A in Fig. 6.1 is an example of this case.

With the above analysis, an error detection method only needs to observe the backup results for those variables that fall into *CASE 4*. The access properties of variables are evaluated within each task. A variable, evaluated to be in *CASE 4* in any task, will be observed. *In the next sub-sections, when talking about variables, we mean the variables of CASE 4.*

Error Detection

To detect the backup error for a variable, we need to observe the timing relation between the write to the variable by program execution and the read to the variable by state backup. To this end, we instrument both the task code and the backup data with flag variables to record the progress of both CPU and DMA, so as to provide information for error detection at the end of state backup. We will use the examples in Fig. 6.4 to explain our method.

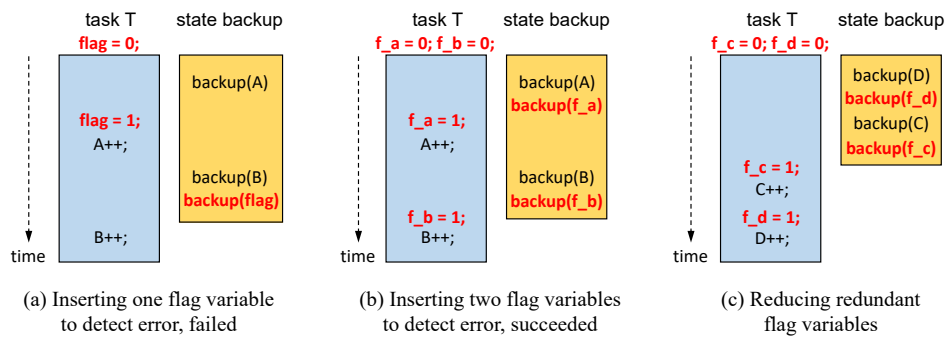


Figure 6.4. State backup error detection exemplified (principle and optimization)

Task T modifies variables *A* and *B* during its execution. We need to know whether before such modifications the backup to the two variables is finished or not. We introduce a flag variable “*flag*” and initialize it to 0 before T starts. In task code, we insert a flag instruction $FI(flag)$ before the first write to *A* or *B* in the task. For example, in Fig. 6.4(a), $flag = 1$ is a flag instruction. On the state backup side, the variable *flag* is also backed up, immediately after *B*. When state backup is finished, we check the value of *flag* in the backup buffer to detect error. If the value of *flag* in the backup buffer is 0, it is clear that after both *A* and *B* are backed up, none of the modifications to *A* or *B* is executed, then we can safely conclude that no backup error occurred. Otherwise, if the value of *flag* in the

backup buffer is 1, it indicates at the time when A and B finish backup, task execution has passed $flag=1$. In such case, it is possible that the task has executed $A++$ or even $B++$. Thus, we are not sure whether an error actually occurred during state backup, and have to report an error for safety. In the example in Fig. 6.4(a), the value of $flag$ in the backup buffer is 1 as $backup(flag)$ executed after $flag=1$, so a backup error is reported.

Actually, in this example, there is no backup error, since the backup operation to A did occur before the modification to A (by $A++$), and so does B . This behavior can not be captured with only one flag variable. Then we introduce two flag variables f_a and f_b to observe the access conflicts on A and B , respectively. The code instrumentation is shown in Fig. 6.4(b). This time, when we check the values of f_a at the end of state backup, we find that f_a 's value in the backup buffer is 0 as $backup(f_a)$ occurs before $f_a=1$, so A is correctly backed up. Similarly, $f_b = 0$, so B is also correctly backed up. Only if the value of *all* flag variables in the backup buffer are 0, we can safely conclude that there is no backup error caused by the execution of the task. By inserting more flag variables in a finer-grained way, we are able to find out correct state backups that would be classified incorrect by a coarser flag insertion.

Removing redundant flags

To maximize the precision of error detection, one can choose to use an independent flag for every variable. However, this will insert redundant flags. We will show what is a redundant flag with the example in Fig. 6.4(c).

Let us consider two variables C and D observed by two flag variables f_c and f_d respectively. We use the notion $e1 \Rightarrow e2$ to represent event $e1$ occurs earlier than event $e2$. In this example, if at the end of state backup f_c 's value in the backup buffer is 0, i.e., C is correctly backed up, we know that $backup(f_c) \Rightarrow f_c = 1$. Now we can conclude that D is also correctly backed up, because $backup(f_d) \Rightarrow backup(f_c) \Rightarrow f_c = 1 \Rightarrow f_d = 1$. Otherwise, if C is not correctly backed up, of course one can still leverage f_d to observe whether D is correctly backed up. Note that we can conclude the state backup is correct only

if all flag variables are 0 in the backup buffer. Even if D is found to be correctly backed up, as an error is already detected for C , the detection result for D does not affect the final conclusion. Thus, we can safely remove flag variable f_d without sacrificing the precision of error detection. We say f_d is a redundant flag. The property described above is formulated with Lemma 6.5.1. As the proof is straightforward, we omit the proof.

Lemma 6.5.1. *Suppose variable A has a higher address than variable B in the buffers, which means $\text{backup}(B) \Rightarrow \text{backup}(A)$, and in a task the first write to A occurs before that to B , i.e., $\text{write}(A) \Rightarrow \text{write}(B)$. If the backup of A is correct, the backup of B must be correct.*

Assume flag variables and corresponding flag instructions have been inserted to observe all variables. Redundant flag variables that satisfy *Lemma 1* can be found by exploring the flag instructions on the control flow graph (CFG) of a task according to *Definition 1*. Any identified redundant flag variable and its flag instructions will be removed.

Definition 6.5.1. *A flag instruction $\text{FI}(f)$ is a redundant flag instruction of task \mathbb{T} iff: on every path from the starting node of the CFG of \mathbb{T} to $\text{FI}(f)$, there is a flag instruction operating a flag whose memory address is no lower than f .*

To summarize, our proposed state backup error detection technique works as follows:

1. Off-line instrumentation.

- (a) Identify task-shared variables that needs observation.
- (b) For each such variable, insert a flag into the backup data behind the variable it observes, and insert a corresponding flag instruction into the task before the first write on the variable.
- (c) Remove all redundant flags from the backup data and corresponding flag instructions in the program.

2. **Run-time Support.** All the flag variables are set to 0 before each task starts execution. After state backup is finished, if all the flag variables in the backup buffer are 0, the backup is concluded correct; otherwise, the backup is considered incorrect.

6.5.3 Buffer Design for Fault-tolerant Backup Management

In Sec. 6.4 we introduced fault-tolerant backup management which allows the system to continue execution even if the backup states are temporarily incorrect. The technique is implemented by the buffer design.

For now, we have a working buffer for program execution and a backup buffer to store backup states. As the backup buffer can be polluted by incorrect state backup, we introduce a new buffer called *safe buffer* to maintain the most recent correct backup copy. When state backup is finished, the correct backup data is in the backup buffer. We do not copy the data from the backup buffer to the safe buffer. Instead, we swap the role of the two buffers. In the implementation, there is a pointer to each buffer. The role swap can be done by a pointer swap. If the backup buffer is polluted by an erroneous backup, the safe buffer still maintains correct program states corresponding an earlier program point. If a power failure occurs, the system will always resume by reloading the states in the safe buffer and roll back to the most recent consistent program point. A power failure may also occur during the data copying from the working buffer to the backup buffer, making the backup buffer inconsistent. For such a case, the safe buffer still enables the system to restore to a most recent consistent program point.

6.5.4 Reorganizing Memory Layout to Reduce Backup Errors

In state backup, the variable at a lower address will be copied earlier and thus has a lower probability to experience backup error. So, changing the layout of the variables in the buffers may affect the occurrence of backup errors. However, an optimal memory layout does not generally exist. First, different tasks of a program may access different variables and they

compete to put their own variables in low memory address, in the layout optimization process; second, the execution frequency of different tasks is input-dependent and can not be decided offline, so it is impossible to decide which task should have higher priority in competing lower addresses. Therefore, we developed a heuristic method to re-organize memory layout to investigate its impact on backup errors.

The main idea is to put the “most frequently” accessed variables in the lowest memory address. We compute a weight, W_v , for each variable to model its access frequency by equation (6.1), where v is a variable, and $C_{T_i}(v)$ is the execution count of task T_i that accesses v . In our exploration, we run each program a sufficiently large number of times to measure the average execution count of each task.

$$W_v = \forall_{v \in \text{CASE 4}} \sum C_{T_i}(v) \quad (6.1)$$

6.6 Experiments and Evaluation

6.6.1 Experimental Setup

We designed a task-based intermittent system to implement the proposed approach, and run the system on a STM32F7-based development board. 8 benchmark programs from related work [51, 70] are used for evaluation¹. The sizes of shared variables in the benchmarks range from 18 bytes to 622 bytes. A programmable power supply is adopted to generate power traces. In the experiments, we assume that the system suffers a power failure around every 5ms.

New NVM devices are increasingly adopted in new processors [16]. They vary in multiple features including most importantly the access speed. To evaluate the performance of our approach for different NVM devices, we use SRAM to simulate NVM at different speeds. This is implemented by evenly inserting dummy variables into the variables to be

¹Available at <https://github.com/IntermittentComputing/TaskBased>

backed up. We denote the access speed of SRAM by v_s . For instance, to simulate an NVM with half the speed of SRAM, denoted by $v_s/2$, we insert a dummy variable before each shared variable with the same size. In the experiments, we simulated four NVM speeds, $v_s/1$, $v_s/2$, $v_s/3$ and $v_s/4$.

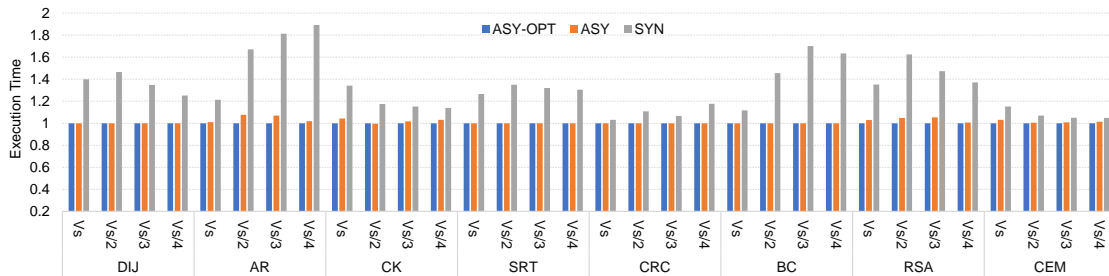


Figure 6.5. Execution times of different benchmarks under different NVM speeds (Normalized to ASY-OPT)

6.6.2 Results and Evaluation

To evaluate the performance of the proposed approach, and also how it can be affected by different memory layouts, we measure the execution time of the benchmark programs executed in three different settings:

- ASY-OPT: asynchronous DMA on the memory layout produced by the technique in Sec. 6.5.4
- ASY: asynchronous DMA with initial memory layout
- SYN: sequential state backup (by DMA) and program execution with initial memory layout

The results are given in Fig. 6.5 with all execution times normalized to ASY-OPT. The performance of the proposed approach (ASY-OPT and ASY) is considerably improved compared to existing approaches (SYN). The performance gain is comparably lower in programs such as CRC and CEM. The reason lies in the ratio between the average execution

time of a task and the average latency of state backup. Take CEM for example, as the states to backup is very large, the backup latency is much larger than the average task execution time. Parallelizing state backup and program execution does not help much in reducing total execution time. Program CRC is on the other extreme. CRC has a very small state size, and the backup latency is much smaller than the average task execution time. In such a case, the space for asynchronous DMA to improve performance is reduced. The inserted flag variables cause an average increase in the sizes of backup states by 3%.

We analyze the results regarding different NVM speeds. There is not an identical trend for all programs. In essence, different NVM speeds indicate different backup latency, and thus affect the ratio between the average task execution time and the backup latency. If the ratio becomes too small or too large, as discussed before, the space for performance improvement by asynchronous DMA will be small.

Table 6.2. Average numbers of uncovered incorrect backups (**UIB**), incorrect backups (**IB**) and total backups (**B**)

		DIJ	AR	CK	SRT	CRC	BC	RSA	CEM
ASY-OPT	UIB	0	0	0	0	0	0	0	0
	IB	0.5	3	24.7	0	0	5.5	8.5	0
	B	167.5	152	260	402	102	87	81	540.5
ASY	UIB	0	0	0.5	0	0	0	0	2.2
	IB	0.5	13.5	49.5	0	0	5.5	10.2	214.2
	B	167.5	152	260.5	402	102	87	81	541.2

At last, we evaluate the impact of memory layout to the proposed approach. In Table 6.2, for each program, we list the average numbers of total backups (**B**), incorrect backups (**IB**) and uncovered incorrect backups (**UIB**) under different NVM speeds, and compare the

results between ASY-OPT and ASY. The results show that the proposed heuristic can reduce the number of incorrect backups, especially for AR, CK and CEM. If we look at the execution time under ASY-OPT and ASY in Fig. 6.5, only AR is comparably more sensitive to the memory layout. There are two main reasons. First, even if the better memory layout reduces the number of incorrect state backups, almost all incorrect state backups can be efficiently covered by the proposed fault-tolerant backup management technique regardless of the memory layout (see the UIB row for the two layouts in Table 6.2). Second, in most programs, the state backups are finished before conflicts between state backup and program execution occur, since the write operations to the variables in a task are separated by computation steps.

6.7 Conclusion

This chapter presents an approach to improve the performance of intermittent systems by enabling parallel state backup and program execution, and at the same time ensure the system still produces correct computation. The main techniques are an error detection method to precisely identify state backup errors and a backup management method that allows the system to tolerate state backup errors at run time. Experimental results show that the proposed approach can considerably improve execution performance compared to the existing approaches with sequential state backup and program execution.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Embedded system is generally subject to resource constraints, thus bringing difficulties in both design and analysis. This thesis considers two resource constraints, timing constraints and energy constraints, and proposes several methods to improve the analysis accuracy and system performance.

The worst-case execution time of real-time tasks should be precisely computed to guarantee the timing constraints can be satisfied. However, inter-task interference may cause more execution time, and the existing timing analysis method can not safely and tightly bound the additional execution time. We in this thesis consider two common inter-task interference, shared cache contention and preemption, and precisely compute the worst-case additional execution time. Experimental results show that the worst-case execution time bound caused by inter-task interference can be significantly tightened.

Energy harvesting system is the promising technology to power a huge number of IoT devices in the future. But the state backup in energy harvesting systems costs a lot of energy and time, significantly impacting the system's performance. In this thesis, we present two methods to reduce the state backup overhead. In the first method, we propose an adaptive task-based intermittent computing system, which can skip some unnecessary state backup. Moreover, we precisely compute the minimum set of data that need to be backed up instead of backing up all the system state. In the second method, we, for the first time, propose to parallel the system execution and state backup. To ensure system correctness in the case of data racing, we propose a backup error detection method. Once a backup error

is detected, a fault-tolerant resumption mechanism can guarantee the system's correctness. Experimentally, our method can significantly reduce the task's execution time.

7.2 Future Work

This thesis has studied the design and analysis for embedded systems under timing constraints and energy constraints, which can be extended to different directions in the future. First, for real-time systems, based on the static analysis result presented in this thesis, we can improve the real-time tasks' worst-case performance, i.e., reduce real-time tasks' worst-case execution time. Some existing methods, like cache locking or cache partition, can be used to improve the worst-case performance. The static analysis result presented in this thesis can provide some hints for us to design new cache locking algorithms or cache partition mechanisms to improve real-time tasks' worst-case performance. Second, for energy harvesting systems, a hardware and software co-design can be used to further reduce the system overhead while achieving better system performance. Some hardware can be configured to balance the energy consumption and the computational accuracy. For instance, one can configure the resolution of the analog-to-digital converter (ADC) to balance the computational accuracy and overhead; When erasing a flash memory, users can adaptively change the erase time/voltage based on the 0/1-bit ratio of the erased data to tradeoff the energy/time consumption and the longevity of the stored information. By utilizing such configurable hardware components, the system can achieve better performance while satisfying the constraints.

REFERENCES

- [1] <http://www.ti.com/microcontrollers.html>.
- [2] WCET Benchmarks, <http://mrtc.mdh.se/projects/wcet.html>.
- [3] <https://iot-analytics.com/>, referenced in Sep. 2020.
- [4] Sebastian Altmeyer and Claire Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS'09, 2009.*, pages 109–118.
- [5] Sebastian Altmeyer and Claire Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *ECRTS'09. 21st Euromicro Conference on Real-Time Systems*.
- [6] Sebastian Altmeyer and Claire Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57(7):707–719, 2011.
- [7] Sebastian Altmeyer, Robert I Davis, and Claire Maiza. Improved cache related preemption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5).
- [8] Sebastian Altmeyer, Robert I Davis, and Claire Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 261–271. IEEE, 2011.

- [9] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: tightening the crpd bound for set-associative caches. In *ACM Sigplan Notices*, volume 45, pages 153–162. ACM, 2010.
- [10] Domenico Balsamo, Alex S. Weddell, and et al. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.
- [11] Naveed Anwar Bhatti and Luca Mottola. Harvos: efficient code instrumentation for transiently-powered embedded sensing. In *ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2017.
- [12] Reinder J Bril, Sebastian Altmeyer, Martijn MHP Van Heuvel, Robert Davis, Moris Behnam, et al. Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with pre-emption thresholds. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 161–172.
- [13] Siddhartha Chatterjee, Erin Parker, Philip J Hanlon, and Alvin R Lebeck. Exact analysis of the cache behavior of nested loops. *programming language design and implementation*, 36(5):286–297, 2001.
- [14] S Chattopadhyay and A Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. 2011.
- [15] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified wcet analysis framework for multicore platforms. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):124, 2014.
- [16] An Chen. A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 2016.
- [17] Wei-Ming Chen, Pi-Cheng Hsiu, and Tei-Wei Kuo. Enabling failure-resilient intermittently-powered systems without runtime checkpointing. In *ACM/IEEE Design Automation Conference, DAC*, 2019.

- [18] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *IEEE Real-Time and Embedded Technology and Applications Symposium ,RTAS*, 2019.
- [19] Jongouk Choi, Qingrui Liu, and Changhee Jung. Cospec: Compiler directed speculative intermittent computation. In *Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2019.
- [20] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2016.
- [21] David Cormie. The arm11 microarchitecture. Retrieved July, 21:2004, 2002.
- [22] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
- [23] Timothy Daulby, Anand Savanth, Geoff Merrett, and Alex S Weddell. Improving the forward progress of transient systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.
- [24] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, 1999.
- [25] Graham Gobieski, Amolak Nagi, and et al. Manic: A vector-dataflow architecture for ultra-low-power embedded systems. In *Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2019.
- [26] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. *Emsoft*, 2010.

- [27] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *OASICS-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.
- [28] Damien Hardy and Isabelle Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS, 2008, IEEE*.
- [29] Josiah D. Hester, Kevin M. Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th SenSys*, 2017.
- [30] Matthew Hicks. Clank: Architectural support for intermittent computation. In *Proceedings of the 44th ISCA*, 2017.
- [31] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, 2011.
- [32] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. QUICKRECALL: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *International Conference on VLSI Design*, 2014.
- [33] Maeng Kiwan and Lucia Brandon. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th PLDI*, 2019.
- [34] Jan C Kleinsorge, Heiko Falk, and Peter Marwedel. A synergetic approach to accurate analysis of cache-related preemption delay. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 329–338. ACM, 2011.
- [35] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemyslaw Pawelczak. Time-sensitive intermittent computing meets legacy software. In *the 25th ASPLOS*, 2020.
- [36] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software.

In *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2020.

- [37] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.
- [38] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007.
- [39] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems*, 48(6):638–680, 2012.
- [40] Fang Liu and Yan Solihin. Understanding the behavior and implications of context switch misses. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):21, 2010.
- [41] Songran Liu, Wei Zhang, Mingsong Lv, Qiulin Chen, and Nan Guan. Latics: A low-overhead adaptive task-based intermittent computing system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.
- [42] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent computing: Challenges and opportunities. In *Summit on Advances in Programming Languages, SNAPL*, 2017.
- [43] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th PLDI*, 2015.
- [44] Will Lunniss, Sebastian Altmeyer, Claire Maiza, Robert Davis, et al. Integrating cache related pre-emption delay analysis into EDF scheduling. In *Real-Time and Embedded*

Technology and Applications Symposium (RTAS), 2013 IEEE 19th, pages 75–84. IEEE, 2013.

- [45] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1, 2016.
- [46] Mingsong Lv, Yi Wang, Guan Nan, and Yu Ge. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 - December 3, 2010*, 2010.
- [47] Kaisheng Ma, Yang Zheng, and et al. Architecture exploration for ambient energy harvesting nonvolatile processors. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.
- [48] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent Execution without Checkpoints. In *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2017.
- [49] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2018.
- [50] Kiwan Maeng and Brandon Lucia. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *ACM Conference on Programming Language Design and Implementation, PLDI*, 2020.
- [51] Amjad Yousef Majid, Carlo Delle Donne, and et al. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks*, *TOSN*, 2020.
- [52] Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP inter-*

- national conference on Hardware/software codesign and system synthesis*, pages 201–206. ACM, 2003.
- [53] Harini Ramaprasad and Frank Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2006*, pages 71–80.
- [54] Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemption points. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 212–224.
- [55] Harini Ramaprasad and Frank Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 148–157, 2005.
- [56] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: system support for long-running computation on rfid-scale devices. In *International conference on Architectural support for programming languages and operating systems, ASPLOS*, 2011.
- [57] Jan Reineke, Sebastian Altmeyer, Daniel Grund, Seungyong Hahn, and Claire Maiza. Selfish-lru: Preemption-aware caching for predictability and performance. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 135–144.
- [58] Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *ACM Conference on Programming Language Design and Implementation, PLDI*, 2019.
- [59] Joshua San Miguel, Ganesan, and et al. The eh model: early design space exploration of intermittent processor architectures. In *IEEE/ACM International Symposium on Microarchitecture ,MICRO*, 2018.

- [60] Rathijit Sen and YN Srikant. Wcet estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 203–212. ACM, 2007.
- [61] Tyler Sondag and Hriday Rajan. A more precise abstract domain for multi-level caches for tighter wcet analysis. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 395–404.
- [62] Jan Staschulat and Rolf Ernst. Worst case timing analysis of input dependent data cache behavior. pages 227–236, 2006.
- [63] Jan Staschulat and Rolf Ernst. Scalable precision cache analysis for real-time software. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(4):25, 2007.
- [64] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 358–363. ACM, 2006.
- [65] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(1):7, 2007.
- [66] Joel van der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2016.
- [67] Xavier Vera and Jingling Xue. Let’s study whole-program cache behaviour analytically. In *International Symposium on High-Performance Computer Architecture*, pages 175–186, 2002.
- [68] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and

- survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.
- [69] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS. IEEE*, pages 80–89.
- [70] Kasim Sinan Yildirim, Amjad Yousef Majid, and et al. Ink: Reactive kernel for tiny batteryless sensors. In *ACM Conference on Embedded Networked Sensor Systems, SenSys, 2018*.
- [71] Wei Zhang, Fan Gong, Lei Ju, Nan Guan, and Zhiping Jia. Scope-aware useful cache block analysis for data cache related preemption delay. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*.
- [72] Wei Zhang, Nan Guan, Lei Ju, and Weichen Liu. Analyzing data cache related preemption delay with multiple preemptions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2255–2265, 2018.
- [73] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA.*, pages 455–463.
- [74] Zhenkai Zhang and Xenofon Koutsoukos. Precise multi-level inclusive cache analysis for wcet estimation. In *IEEE Real-time Systems Symposium, 2015*.