INSTRUCTION AND DATA CACHE MODELING FOR TIMING ANALYSIS IN REAL-TIME SYSTEMS

YANHUI LI

NATIONAL UNIVERSITY OF SINGAPORE

2008

INSTRUCTION AND DATA CACHE MODELING FOR TIMING ANALYSIS IN REAL-TIME SYSTEMS

YANHUI LI

(B.Eng., NORTHEASTERN UNIVERISTY)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

NUS GRADUATE SCHOOL FOR INTEGRATIVE

SCIENCES AND ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2008

ABSTRACT

Caches in Embedded Systems improve average case performance, but they are a source of unpredictability, especially in the worst case software timing analysis with the consideration of data caches. This is a critical problem in real-time systems, where tight Worst Case Execution Time (WCET) is required for their schedulability analysis. Several works have studied the data cache impacts on the WCET of programs, but they can only handle programs with no input-dependent data accesses.

To provide an efficient and accurate analysis for input-dependent data caches, we develop classified cache architecture and a WCET framework for the architecture. Our work classifies predictable and unpredictable accesses, then allocates them into predictable caches and unpredictable caches accordingly, and uses CME (Cache Miss Equations) and our reuse-distance-based algorithm for their timing analysis respectively. Compared with simulation, our analysis framework produces a very good WCET tightness, and our architecture creates almost no hardware overhead or performance degradation. In addition, we examine NP-completeness for theoretical support and proved WCET analysis is NP-complete. We also explore data allocation techniques to improve system performance, and our algorithm improves cache hit ratios efficiently according to our experimental results.

Keywords:

WCET, Cache, Real-time System, input-dependent data access, NP-completeness, System Performance

ACKNOWLEDGMENTS

This work is mainly carried out in Signal Processing and VLSI Lab, Department of Electrical and Computer Engineering, National University of Singapore, under research scholarship from National University of Singapore. Chapter 4 is mainly carried out in Computer Engineering and Networks Laboratory, ETH Zurich, under scholarship from NNCR MICS[1].

I would like to express my deep gratitude to my advisors: Dr. Ha Yajun and Dr. Tay Teng Tiow. I sincerely thank them for many insightful discussions, continuous motivation and constant patient guidance on my research. I am actually guilty that I occupied much discussion time of Prof. Dr. Ha. Thank you!

Many colleagues and groups have contributed to this work through discussions, reviews, questions or other support. I benefited a lot from the weekly seminars of our research group under guidance of Dr. Ha Yajun. It has been an essential discussion forum for us to exchange ideas and get inspired. I have practiced and learned much from presenting my work and meditating on the talks from our group members. I would like to take the chance to thank: Chen Xiaolei, Yu Heng, Shakith Devinda Fernando, Zhang Wenjuan, Lok Weiting, Akash Kumar, Dong Bo, Pu Yu and many more. I also would like to thank the embedded systems research group under guidance of Dr. Samarjit Chakraborty and Prof. P.S. Thiagarajan from School of Computing, National University of Singapore. I

[1] NNCR MICS: http://www.mics.org/

attended their seminars every week, which are both useful and interesting, I would certainly also miss this great forum.

I would like to take this opportunity to thank the National University of Singapore for funding me with research scholarship and for providing such an excellent environment and services. My thanks also go to the administrative and support staff in Department of Electrical and Computer Engineering, NUS.

I thank all my friends for having a joyful time and their constant support: Zou Xiaodan, Tian Feng, Tian Xiaohua, Wei Ying, Jiang Jing, Chen Jianzhong, Lahlou Kitane Driss, Wu Liqun, Yu Rui, Hu Yingping, Xu Xiaoyuan, Cheng Xiang, Tan Jun, Zhang Fei, He Lin, Benoit Mortgat, Suybeng Voreak, and many other friends. Apart from being good friends, they were extremely supportive and were always there with me when I need accompany. I would also thank many friends for keeping in touch, sharing daily chaos, motivating me and pretending to be interested in my research: Zhao Yi, Wang Fumin, Hao Lijie, Yang Zhaohui, Zhou Haihua, Wu Xinhui and many others.

I would like to thank my parents, Wenwu and Guiyun, and my brother Li Lei for their love and support. I am always grateful for their encouragement and support for my academic choice and during my entire education and life.

And finally thanks to anyone who knows deep research, never underestimate the power of your encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

1. BB, B ..... Basic Block
2. CFG ..... Control Flow Graph
3. ILP ..... Integer Linear Programming
4. WCET ..... Worst Case Execution Time
5. WCTA ..... Worst Case Timing Abstraction
6. CME ..... Cache Miss Equations
7. NP-Complete ..... Nondeterministic Polynomial Complete
8. IPET ..... Implicit Path Enumeration Technique
9. CSTG ..... Cache State Transition Graphs
10. AST ..... Abstract Syntax Tree
11. X3C ..... Exact Cover By 3-SETS
12. 3DM ..... 3-Dimensional Matching
13. CNF: ..... Conjunctive Normal Form
14. SAT ..... Satisfiability

SUMMARY

Caches are small memories embedded close to processors that can improve their average system performance. However, caches incur the timing predictability problem in real-time systems, where the tight WCET (Worst Case Execution time) is required for their schedulability analysis. As it is unaffordable to obtain WCET of a program by exhaustive simulations, the design of efficient and accurate static WCET analysis has been a heated research topic.

To obtain an accurate WCET, past efforts have been made in two directions. One direction tries to develop more predictable new architectures, such as using cache partitioning and cache locking. These approaches require large caches and have not solved the general problem of predicting cache behavior if all tasks shared the cache. The other tries to develop more accurate analysis techniques. Previous analysis techniques focus on modeling instruction caches, and have obtained fruitful results to the extent where instruction cache timing behavior can be accurately modeled and analyzed. However, it remains tough for modeling and analyzing data caches. Very few works have studied data cache impacts on WCET of programs, and they can only handle programs without input-dependent data accesses.

To solve this problem, we develop a new cache-classified architecture. This architecture classifies predictable and unpredictable accesses, and allocates them into the classified predictable cache and the unpredictable cache respectively. For this architecture, we build a WCET analysis framework. Our framework analyses these two classified data cache accesses independently: it uses the CME-based analyzer for analyzing predictable

accesses and our reuse-distance-based analyzer for analyzing unpredictable accesses. Combining the results from the above two parallel analyzers, our framework obtains the desired WCET results. Compared with simulation results, our analysis framework shows that:

(1) Our analysis is conservative, i.e. it gives a safe upper bound of execution times, and has a low time complexity;

(2) Our analysis framework outputs an appreciable WCET tightness;

(3) Our new architecture gives little overhead in terms of complexity and execution time.

To support the WCET analysis theoretically, we explore NP (Nondeterministic Polynomial)-completeness proving techniques. We prove that the WCET analysis is NP-complete.

In addition, we examine techniques to improve system performance by allocating data in programs to increase their locality. Our goal is to apply data management technology to minimize compulsory and conflict misses in direct-mapped caches. We first cluster data accesses into memory lines based on their space locality, in the way that accesses with high space locality get allocated in the same cache line so that cache compulsory misses get minimized; and then we map these memory lines to cache lines, in the way that memory lines with minimized conflicts share the same cache lines so that cache conflict misses are minimized. Our algorithm demonstrates its good ability in improving the cache hit ratio, especially for codes with much conflicts on the given architecture.

In summary, this thesis provides a comprehensive framework to analyze instruction and data caches for single-task real-time systems with input data dependency, explores the

NP-completeness as a theoretical support, and designs a cache conscious data allocation technique to improve the cache performance.

C H A P T E R . 1
I N T R O D U C T I O N

Nowadays, there is an increasing demand of computing devices, with a large portion of them serving as components of other systems. These devices are called *embedded systems*, deployed for the purpose of data processing, control or communication. An embedded system[2] is a special-purpose computer system designed to perform one or a few dedicated functions, often with real-time computing constraints. It is usually *embedded* as part of a complete device including hardware and mechanical parts. In contrast, a general-purpose computer, such as a personal computer, can do many different tasks depending on programming. Embedded systems control many of the common devices in use today.

Embedded systems are becoming more and more prevalent in our society and this trend promises to continue in the near future 0. We need not go far to seek examples (Figure 1.1) of embedded systems applications: mission critical systems such as airplane jets, power plant monitoring systems and vehicle engine controllers, communication devices such as cellular phones and consumer electronics such as mobile phones, mp3 players, pacemaker, set-top boxes, to name but a few. Let's have a look at a market forecast (Figure 1.2) to comprehend this trend and also the development trend of embedded systems.

[2] http://en.wikipedia.org/wiki/Embedded_system

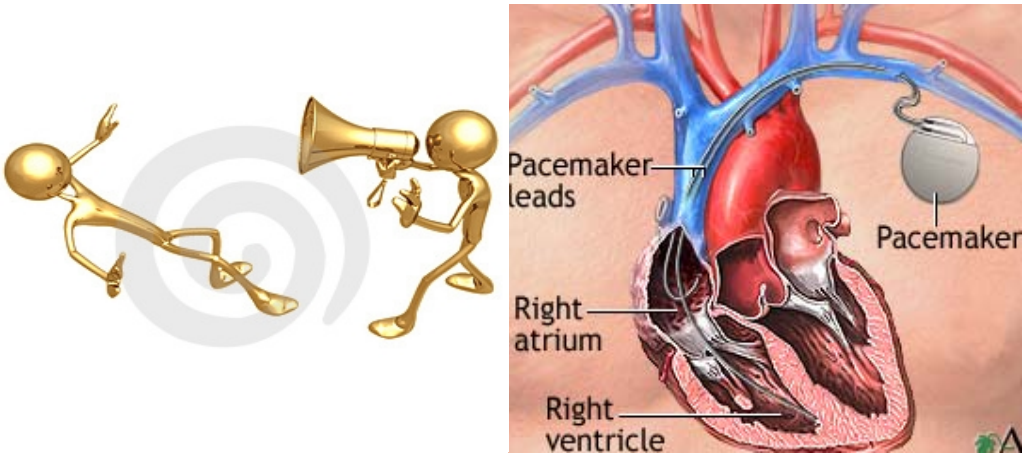

Figure 1.1a Voice Communication System, Figure 1.1b Pacemaker

Figure 1.1c Engine Controllers; Figure 1.1d Airplane Monitor and Control Systems
Source: http://images.google.com
Figure 1.1 Examples of Embedded System Applications

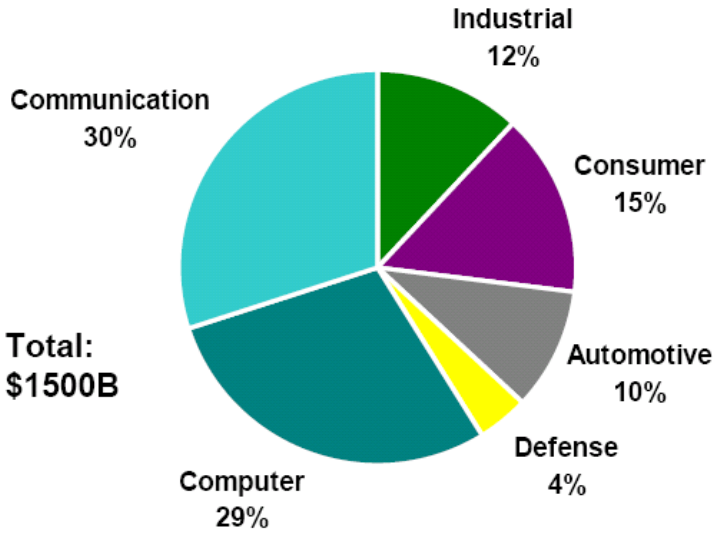

Figure 1.2 Structure of Worldwide Electronic Production in 2010 [2]

Figure 1.2 shows the market forecast [2] of the structure of worldwide electronic production in 2010. It tells that 60% of electronic production will be about embedded applications, among which communications occupies the most (30%), while consumer (15%), industrial (12%) and automotive and defense (14%) have similar percentages. These figures demonstrates that the market share for embedded systems is growing and a high portion of industrial research and development activities will be dominated by embedded systems. In order to meet numerous demanding requirements of functional, timing and low power sides, the complexity of embedded systems will inevitably increase.

Because of the steadily increasing number of functional requirements, embedded hardware and software architectures are becoming more and more complex. Usually embedded applications not only need to generate correct results but also have to achieve the results within a given time period. Timing behavior is essential if the application has to react to signals from the environment. And therefore to safely and tightly analyze timing behavior is very important and also challenging for today's complex embedded designs.

1.1 Real-time Embedded Systems

Embedded systems have to satisfy an increasing number of requirements, including both functional and non-functional requirements.

1.1.1 Functional Behavior Requirements

1) Correctness: this is a fundamental requirement of an embedded system. Functional behavior of embedded applications has to be guaranteed in any case.
2) Reliability: this requirement depends on the embedded applications. For example, while a system breakdown of a cellular phone is tolerable once a year, a similar rate for safety critical systems like aerospace applications would be disastrous.
3) Flexibility: it has two different senses—configurability and reconfigurability. A configurable system can enable the manufacturer to simplify the development for a variety of product lines while a reconfigurable one gives the customer the ability to use a device for different applications. For example, a firmware update is much less expensive than the exchange of hardware components in case software errors are detected.

1.1.2 Timing Behavior Requirements

For most embedded applications, a result has not only to be correct but has to be rendered before a specified deadline. This is the timing behavior requirement, i.e. the system has to interact with the environment in a timely fashion. In literature, these systems are called *real-time systems* and this timely response property is named timing behavior which denotes the time delay a software task takes to finish.

Real-time systems can be further classified into: soft real-time systems and hard real-time systems.

In soft real-time systems, timing behavior is an important aspect but is not essential to correct functional behavior. Occasional misses of deadlines can be tolerated. This kind of deadline-misses would not affect the functional correctness but as a price the quality can be reduced. In other words, the software task could be switched to a different processing mode, which takes shorter time to render a less accurate result. For example, voice communication systems (Figure 1.1a) or multimedia streaming applications can tolerate the loss or delay of a few frames.

However in hard real-time systems, a pre-defined deadline has to be obeyed by the application in order to guarantee the functional correctness. The term deadline denotes the longest acceptable time the computation can take to finish. Any failure to meet the deadlines would violate the timing behavior requirement and affect the functional correctness, and may even cause catastrophic results. This hard real-time system is usually mission-critical systems, such as engine control software (Figure 1.1c, Figure 1.1d) in automotive, flight control software in avionics systems automated manufacturing and sophisticated medical devices such as pacemakers (Figure 1.1b).

1.1.3 Other Requirements

There are also many other requirements of the embedded systems, such as low power consumption, size, weight and design fashion.

Note that in this thesis, we are concerned only with hard real-time systems and focus on timing behaviors of advanced embedded hardware—especially caches.

1.2 Memory Hierarchy in Real-time Embedded Systems

Let's first have a brief look at the properties of memory hierarchy in modern embedded architectures. And then we discuss its impact on the timing behavior of software applications.

Memory hierarchy is a structure that uses multiple levels of memories; as the distance from the CPU increases, the size of the memories and the access time both increase [3]. We have a large variety of storage devices in a computer system, which are organized in a hierarchy (see Figure 1.3) according to either their speed or their cost.

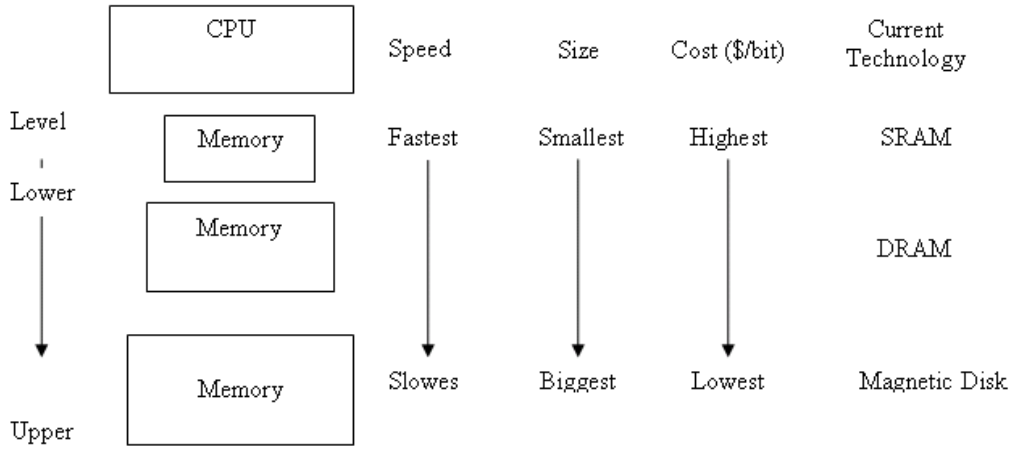

Figure 1.3 Basic Structure of a Memory Hierarchy

Note that in this thesis, we number the memory level lower to higher as distance from CPU increases.

1.2.1 Why Cache Memories and Memory Hierarchy?

Thanks to increasing clock frequencies and the exploitation of parallelism at different levels, microprocessors processing speed continues to increase exponentially at the speed of about 60% per year [4]. However, relatively slow memory access makes processor speed increase meaningless and has become a bottleneck in a general system's performance. Though memory access speed is also increasing exponentially, it is at a much lower increase rate, to be more exact, at increase rate of less than 10% per year for the last twenty years [4].

This leads to the widening gap between processor speed and main memory access speed. The overall effect is called in literature: anti-law of Moore[3] which illustrates that the speed between processor and main memory doubles every two years. The difference between diverging exponentials also grows exponentially. Thus, although the disparity between processor and memory speed is already an issue, downstream someplace it will be a more serious one. Therefore memory has become a primary obstacle in improving system performance, and how to avoid its limiting data transfer between memory and processor has become an intense interest for research works [5].

A dream processor design would be a system consisting of high processor speed and a memory with high capacity and fast speed at lower cost. To achieve this goal, we need to exploit the properties of hardware and software. Besides the gap between CPU speed and main memory speed is widening, there are another two fundamental properties of hardware and software [6]:

(1) Fast storage technologies cost more per byte and have less capacity.

---

[3] Moore's Law: http://en.wikipedia.org/wiki/Moore's_Law

(2) Well-written programs tend to exhibit good locality.

These three fundamental properties complement each other beautifully and they suggest a memory hierarchy approach for organizing memory and storage systems. That's why cache is adopted. *Cache* is a small, fast storage device that acts as a staging area for a subset of the data in a larger, slower device. To achieve the low cost as main memory and to achieve fast access speed as cache, caches are embedded between CPU and main memory forming a memory hierarchy.

The fundamental idea of a memory hierarchy is that for each level k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1. The reasons why memory hierarchy architecture can render system an illusion memory of large size, fast speed and cheep price are as follows:

(1) Programs tend to access the data at level k more often than the data at level k+1.

(2) Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.

The main property that makes the memory hierarchy work well in the general case is called *locality*, which states that programs access a relatively small portion of their address space. There are two types of locality: temporal locality and spatial locality.

(1) *Temporal locality*: a data location tends to be referenced again soon after referenced recently;

(2) *Spatial locality*: addresses nearby a recently referenced data location tend to be referenced in the near future.

Therefore, memory architecture with high speed, low cost and large storage of the system can be achieved using this memory hierarchy idea consisting of multiple levels memories with increasing size and access time as their distance from the CPU increases.

By implementing the memory system as a hierarchy as in Figure 1.2, the system gives the user an illusion that the user obtains a memory as large as the largest memory in the largest level of the hierarchy, but as if at the access speed of the fastest memory in the smallest level of the hierarchy.

1.2.2 Cache Memories

Cache memories are smaller and faster expensive storage device that acts as a staging area for a subset of the data in a larger, slower device. They are situated between processor and main memory to store the frequently used memory blocks in order to bridge the gap between fast processor speed and slow main memory [6].

Cache Terminology:

(1) *Cache blocks*:

Caches are built in the unit of cache blocks or cache lines.

In a memory hierarchy of multiple levels, data is copied between two adjacent levels at a time, the lower level (closer to the processor) is smaller and faster than the upper level (farther away from the processor). The minimum unit of information that can be transferred between two adjacent levels is called *a (memory) block or a line*.

Usually the size of a cache block is the same as the main memory block. Typically in embedded systems, a block size consists of 4 memory words, 8 memory words or 16 memory words. Storing several memory words rather than one in a block is to make advantage of the spatial locality of program accesses (Figure 1.4).

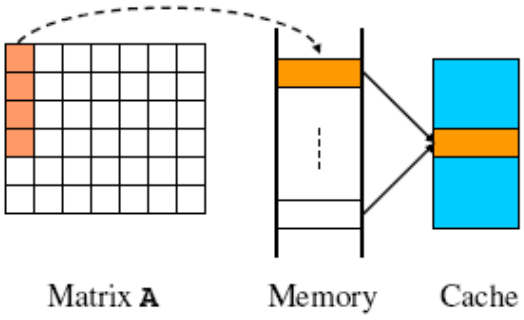

Figure 1.4 How Is Data Stored in Main Memory and Cache

(2) *Cache associativity*:

According to the number of cache blocks in a cache set, there are three cache structures: direct-mapped cache, set associative cache and fully associative cache.

A *cache set* is a set of continuous cache blocks where one main memory block maps to. A memory block can be placed in any block in the cache set it is mapped to.

1) For an *n-way associative cache*, n cache blocks are organized in a cache set.
2) In a *direct mapped cache* each cache set has exactly one cache block, e.g. the associativity is one.
3) For *fully associative cache*, the whole cache is one cache set.

The relationship between the memory block number, the number of sets in a cache and the set number the memory block can be represented as an equation as follows:

Mapped Set Number = (Block Number) Modulo (Number of Sets)

(3) *Cache hit and Cache miss*:

The memory access has two types of operations: read and write. When the processor requests data (either read or write), the corresponding address of the data is compared from the lower level to higher level to see if there is a match. If the data requested by the processor appears in some block in the lower level, this level has *a hit*. Otherwise, we

have a miss in this level, and the inquiry of the data goes to its next higher level. The hit ratio or hit rate of a cache level is the fraction of memory accesses found in it over in its adjacent upper level. The miss rate is (1 - hit rate), i.e. the fraction of memory accesses not found in the cache.

For a cache miss, we have two types: cold (compulsory and capacity) miss and conflict miss [3].

1) *Compulsory misses*—when a memory word is accessed for the first time;
2) *Capacity misses*—when cache data that would be needed in the future are displaced due to the working data set being larger in size than the cache;
3) *Conflict misses*—when cache data are replaced by other data, in spite of the presence of usable cache space.

   For example, caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k. Conflict misses occur when multiple data objects all map to the same level k block.

On a cache miss, the needed data is to be placed in the cache by copying from its adjacent upper level. If the placement of the data is to be set in a cache set which is not empty, a cache block may need to be replaced out. But which cache block in the set needs to be replaced out? This also depends on the cache structure.

1) For direct mapped cache, it's quite easy to implement, just replace out the cache block to which the requested data is mapped to.
2) For set associative or fully associative, there are two block replacement policy: random and LRU (Least Recently Used).
   a) *Random replacement* policy is to randomly replace out one cache block in the cache set the requested data is mapped to;

b) *LRU replacement* policy is to replace out the cache block which is least recently used among the cache set the requested data is mapped to.

1.2.3 Difficulty in Analyzing Timing Behavior of Caches and Systems

Cache memories hide in part the speed gap between processor and main memory, and to some degree offset the widening speed gap by making use of program locality and by rendering data at fast speed. Therefore, they improve the average performance. Unfortunately meanwhile they complicate timing analysis and incur unpredictable problems of system performance.

The main reason is rooted in caches' probabilistic nature as well as the speed difference between cache and main memory. The time to access data in the cache from the processor is very fast, e.g. a single clock cycle, if the requested data is available in the cache. Otherwise, the cache controller has to request the data from main memory, which takes much longer, typically 20-100 or more clock cycles (as also can been seen in Figure 1.3). As caches are taking advantage of the temporal and spatial locality of programs, caching scheme attempts to capture in cache memory the data and instructions that are most likely to be accessed in the near future. This leads to the probabilistic nature of cache's hit ratio and thus causing some (possibly transient) uncertainties in the execution times of the tasks. Because of this, caches have become a hindrance in usage and in the timing analysis of real times systems.

Behind this probabilistic nature rooted uncertainty is the very basic: the cache resources are too limited compared to main memory blocks. This resource limitation causes two types of interference.

1) *Intra-cache interference*: interferences between a task's memory addresses in a single tasking environment;
2) *Inter-task interference*: interferences between multi-programmed tasks in multi-tasking preemptive environment.

These interference lead to uncertainty of a cache block's content and thus resulting in the unpredictable cache hit or miss.

Therefore, timing behavior of caches gets unpredictable.

In addition, the demanding requirements for embedded systems and the complexity of the software and hardware architecture of embedded systems adds to the challenging performance analysis in real-time systems. And there is great difficulty in estimating the WCET accurately and scheduling tasks properly to meet the system timing requirements.

1.2.4 Importance of Analyzing Timing Behavior of Caches

To cushion the speed gap between slow memory access speed and high processor frequency, caches have been popular in today's embedded systems.

Caches are used to improve the average case timing behavior, but could not guarantee the worst case timing behavior and they also complicate worst case timing analysis. There is one typical saying in literature to represent the importance of worst case performance over average case performance in hard real-time systems: an adult got drown in a lake whose average depth is 0.5 meters.

However, it is essential to be able to accurately measure the timing feature of applications, in presence of cache memories, so as to schedule tasks properly to meet the system timing requirements. There are many computer applications in which computations must satisfy stringent timing constraints, i.e. it must guarantee that those computations must be completed before specified deadlines. Failure to meet the specified deadlines in such applications can lead to intolerable system degradation, and can, in some applications, result in catastrophic loss of life or property. Examples of these systems include embedded tactical systems for military applications, flight mission control, production control, robotics, etc. It is essential to guarantee that critical timing constraints will be met. A safe WCET analysis is essential in guaranteeing real-time performance. To enable caches to be largely adopted in real-time systems, the capacity to analyze their timing behavior has become urgently indispensable.

A conservative approach is to assume all memory accesses being cache misses. This is clearly overly pessimistic; in fact, some of the memory accesses could be determined whether they would be cache miss or hit before running the program in absence of input data. Too pessimistic WCET estimation would cause a waste of hardware resources.

So modeling data cache timing behavior accurately has become a critical issue in WCET analysis.

In the Harvard architecture[4], a cache is separated for instructions and data, i.e. the architecture would have split instruction caches and data caches.

1) *Instruction caches* hold only the program code and are only read from the

[4] http://en.wikipedia.org/wiki/Harvard_architecture

processor.

2) *Data caches* hold the data, which are read and written from the processor.

While many researches have been carried out to model the timing behavior of instruction caches effectively, predicting timing behavior of caches remains as a tough job. It's especially difficult to model data cache behavior. Few works have attempted model data caches' timing behavior, but only limited to no input dependent accesses involved. It still remains as one open problem to design an efficient and effective method to model data cache behavior considering input dependent accesses.

Our work is trying to improve this situation and give attempts to model data caches with input dependent accesses.

1.3 WCET Analysis

A hard real-time system usually is a collection of tasks running on a set of hardware resources. Each task can be characterized by a release time, a deadline, and a computation time and every task can repeat periodically or stochastically. WCET analysis is important for the schedulability analysis of real-time embedded systems, which have stringent application constraints compared to other embedded systems in needing the timing correctness on top of functional correctness. This requires that all tasks be scheduled such that all task does not miss their deadlines. This requirement necessitates schedulability analysis which checks whether there exists a schedule for the tasks so that they all finish executions within their deadlines for every release (ready to execute). As a consequence, priori knowledge of the computation time for each task is required. Furthermore, to guarantee that the deadline is met in any circumstance, the WCET should be used as input for the schedulability analysis, making WCET

estimation. To estimate a priori WCET of a given program on a given architecture is called *WCET analysis* in the literature.

It's often quite hard to know an exact WCET of a task and a conservative estimate is often used. However, tight WCET estimates are essential for schedulability analysis as they reduce the waste of hardware resources for a successful schedule.

In this thesis, we study efficient methods for WCET estimations. The WCET to be studied in this thesis is defined as the maximum possible memory access time of a task running on a hardware platform.

There are several points for this definition to be noted.

1) This thesis deals with single task real-time systems where one task is executed without interruption. While a task may be interrupted by a higher priority task in multi-tasking real-time systems, it is beyond the focus of our research scope.
2) WCET is the longest execution time of a task among all possible sets of data input.
3) WCET of a task depends on the underlying hardware architecture.

There are two general approaches to estimate the WCET of a task or program. As we would be dealing with a single task context of WCET determination, we will use the term program instead of task from now on.

1) Exhaustive simulation: to obtain the WCET by actually running the program on the target hardware over all sets of possible data input or on a processor simulator using stimuli data. However, this approach is impossible to guarantee the essential requirement of WCET analysis-safety. Because it's unaffordable to carry out an exhaustive simulation over all possible sets of data input which is

tremendous, but if not simulating all the possibilities the estimated WCET may be underestimated, i.e. estimated WCET does not cover the worst case and therefore is not safe.

2) Static analysis: to examine the program, derive its timing properties and makes estimation on the WCET without actually running the program. Static WCET analysis is expected to be:
    a) Conservative (safe): Safety is essential because underestimated WCET may lead to smaller execution time assignment to a task, and thus gets blamed for failure to meet the deadline of the task as opposed to be predicted.
    b) Tight: close to the actual WCET so as to save computation and other resources
    c) Efficient: in both time and space occupation.

Our WCET estimation is the static analysis approach. Please note the first property is compulsory and the latter two are desirable. Our work obeys the three properties. Readers may refer to Section 2.1 in Chapter 2 for more details on WCET estimation.

1.4 Data Memory Management

Data allocation is possible because that Code generation in embedded systems can be tuned to the given cache configuration (cache line size, cache size etc). Therefore cache performance can be improved by allocating data in the program in the way that program locality is increased. There have been researches on code placement for improving cache performance [7, 8]. However, [8] only deals with scalar variables, and [7] does not differentiate between scalar and array variables, which may cause failure to exploit array placement based on their index and expression. Our data management strategy

introduces placing scalar variables first and then gives an extension to place array by analyzing the array reference indices.

1.5 Contributions

This thesis is dedicated to obtain tight WCET in the presence of caches in real-time system. This thesis makes several contributions to modeling instruction and data cache timing behavior. Especially we provide a platform that can analyze data cache behavior with input dependency.

We develop a new architecture, which has parallel predictable caches and unpredictable caches. The architecture classifies input-independent and input-dependent accesses and allocates them into predictable caches and unpredictable caches respectively. On top of this architecture, we build our WCET analysis framework. It has two analyzers: one for analyzing input-independent accesses, and another for analyzing input-dependent accesses. For the analyzer of input-independent accesses, we use CME (cache miss equations) for timing analysis; for the analyzer of input-dependent accesses, we develop an algorithm based on the accesses' reuse distances for timing analysis.

We lead the simulation and compare the simulation results with our framework analysis results. Comparison shows that:

1) Our analysis is conservative, e.g. determines a safe upper bound of the execution time, and has a low time complexity;
2) Our analysis framework produces an appreciable WCET tightness compared with simulation;
3) Our architecture caused ignorable overhead in hardware complexity and almost no difference in execution time.

To support our WCET timing analysis theoretically, we examined NP-completeness proving techniques: restriction and local replacement. We present WCET analysis NP-completeness proof using restriction technique.

In addition, we examine data management and devise an algorithm with low complexity to increase program locality to reduce cache misses and improve cache and system performance. We target at minimizing compulsory and conflict misses in direct-mapped cache using data management technology. We cluster data accesses into memory lines in the way that accesses with high spatial locality get allocated in the same line—this is to minimize compulsory cache misses; we then map these memory lines to cache lines in the way that memory lines with minimized conflicts share same cache lines—this is to minimize cache conflict misses. Our experiment shows that we achieved good performance improvement especially for programs with poor locality.

In summary, this thesis provides a sophisticated framework to analyze effects of instruction and data caches for single task real-time systems considering input dependent accesses. It also explores NP-completeness for theoretical supports and data allocation techniques to improve cache performances.

1.6 Organization of Thesis

The rest of the thesis is organized as follows. Chapter 2 gives background information and a literature survey on cache memories, cache timing analysis and WCET analysis techniques. Chapter 3 presents our WCET analysis framework for single tasking real-time systems in presence of instruction and data caches with input dependent accesses. Chapter 4 provides NP-completeness proving technique and proof for WCET analysis

problem. Chapter 5 renders a data allocation algorithm to improve cache and system performance. Chapter 6 concludes the thesis work and points out possible future works.

CHAPTER.2
LITERATURE REVIEW

Many embedded systems are safety critical (e.g., automotive controller) and have timing constraints on the execution time of embedded software. *Schedulability analysis* is to verify whether all timing constraints are satisfied for all possible runs of a system, by taking into account the timing constraints and application's parameters such as execution times, deadlines and periods. This would require the input of the execution times of different tasks, thus the accurate estimation of execution time becomes indispensable. As a result, WCET of a program is essential for the schedulability analysis in real-time embedded systems.

WCET analysis computes an upper bound on programs' execution time on particular hardware architecture for all possible inputs. It involves both program path analysis and modeling the timing effects of processor micro-architectural features such as caches, pipelines and branch-predictors.

This chapter surveys the literature of WCET analysis in presence of cache memories. In Section 2.1, we introduce WCET simulation, static analysis and their properties. In Section 2.2, we present three phrases of WCET analysis, among which we focus on the WCET calculation phase and this phase's three approaches (Tree-based approach Path-based approach Implicit path enumeration approach). In Section 2.3, we present related works on static WCET analysis considering instruction caches, data caches without/with input dependent accesses. In Section 2.4, we introduce cache timing analysis. In Section 2.5, we present overview of our WCET analysis framework. And f in Section 2.6, we describes our architecture of the underlying model.

2.1 WCET Introduction

From Chapter 1, we are sure that estimating WCET is essential for schedulability analysis for real-time systems. Actually the importance of WCET analysis has been recognized by the real-time community and substantial progress has been made over the past twenty years. And we have two approaches for estimating WCET: exhaustive simulation and static-analysis.

2.1.1 Exhaustive Simulation Approach

Simulation is state-of-the-art in industry to determine the timing behavior of a program on a given architecture. Though typical timing behavior can be obtained this way, execution time guarantees cannot be made as it is unaffordable to cover all relevant execution scenarios using simulation. This is also unsafe because typical timing behavior can not guarantee covering the worst case as not all program paths can be covered by the limited simulation. A full coverage would require an exponential number of test data and would be too time consuming. Therefore, only a subset of all program paths is tested. However, safety is essential for WCET estimation in order to guarantee a sufficient time assignment to a task so that all tasks can meet their deadlines and a successful schedule can be obtained.

Therefore, static-analysis of WCET has gained much popularity in the literature researches.

2.1.2 Static WCET Analysis Approach

WCET analysis computes an upper bound on the program's execution time on particular hardware architecture for all possible inputs.

Properties of WCET Analysis

Let's restate the properties of a desired static-analysis of WCET more detailed in order to understand the WCET analysis criteria better:

1) Conservative: The analysis should not underestimate the actual WCET; otherwise a supposed 'successful' schedule may assign a task a computation time above the estimated WCET but not sufficient for the actual worst case. This leads to the task missing its deadline and the system failing the requirements in some circumstances.
2) Tight: The analysis should be reasonably close to the actual WCET; otherwise the task will require an unnecessarily long computation time as it will be assigned a computation time no less than the estimated WCET. This much overestimation would obviously cause a waste of hardware resources in order to guarantee a successful schedule, which is undesirable especially as the resources are stringent and expensive in embedded systems.

(3) Efficient: The static analysis should be efficient in both time and space consumption.

2.2 Three Phases of WCET Analysis

To achieve these properties of a WCET analysis, we have to examine what affects the execution time of a program so that efforts can be directed to these factors for pursuit of a good WCET analysis.

Factors Affecting Execution Time

There are two factors:

1) Data input for the program: different input sets may have different execution paths of a program, for example, taking different branches and resulting in different execution times
2) The hardware architecture on which the program is running: hardware resources have a great influence on how long an instruction would execute.

WCET Analysis Phases

Correspondingly, static WCET analysis can be divided into three phases [9]:

1) *Flow analysis*: to extract the dynamic behavior of the program. This includes information as: identify loop bounds, how many times loops iterate, which functions get called and exclude infeasible program paths from the source program or the compiled code of the program. If there are dependencies between if-statements, the information much be a safe (over) approximation including all possible program execution paths. The information can be obtained by *manual annotations* (integrated in the programming language [10] or provided separately [11], [12]), or by *automatic flow analysis* methods[13], [14], [15].

   The more infeasible paths get excluded, the more accurate a WCET can be obtained from the analysis and also with more efficiency. This step is also called *program path analysis*.
2) Micro-architecture modeling: to determine the timing behavior of instuctions given the architectural feature of the target system. For modern processors, it is especially important to study the effects of various performance enhancing hardware components such as pipeline, cache, branch prediction etc[16], [17], [18], [19]. While improving the average system performance, these advanced

hardware features cause difficulties for instruction timing prediction. They make instruction execution time not constant any more, for example a cache hit may take a much shorter execution time than a cache miss. Furthermore, the instruction execution time can be history dependent, for example a cache access resulting in a cache hit or miss depends on if that to-be-accessed data is already in the cache or not.

This step studies these impacts and provides instruction timing information so that the execution times of different execution paths can be obtained by counting in each this step's timing schema.

3) WCET calculation: to calculate an upper bound on the WCET. With the results of flow and timing information from the first two steps, the costs of feasible program paths are evaluated, and the maximum one will be taken as the estimated WCET.

Relationship between WCET Analysis Properties and Three Phases

Let's examine the relationship between WCET analysis properties and these three phases.

1) Conservativeness: it relies on the first two WCET phases to guarantee no underestimation of the WCET. During flow analysis phase, it should be made sure that no feasible paths are excluded, otherwise the worst case execution path which is feasible might be excluded out and lead to an underestimation of WCET (failure of WCET analysis); During micro-architecture modeling phase, instruction timing estimation should also be conservative in order that the overall cost of a program path (for every feasible path) will not be

underestimated.

2) Tightness: it also depends on the first two phases. As we mentioned above in the three phases, the more infeasible paths get excluded, the more accurate the estimation of WCET can be obtained. This is because infeasible paths counted out might have longer execution times, excluding these execution times can give a smaller estimation of WCET which is tighter. And the more accurate the instruction timing from step two, the tighter the estimation of execution time for the paths can be obtained as a path execution time is the sum evaluation of the instruction timing for all timing nodes along this path.

Also from the three phases, we can see that static WCET analysis is more efficient than simulation as the latter individually analyzes every possible program path, while the former considers a much smaller set of paths, simultaneously.

Recall that static WCET analysis consists of three phase: flow analysis or program path analysis, micro-architecture modeling and WCET calculation. Since WCET calculation step directly concerns the aim of the program's WCET analysis and the other two steps are performed for a better WCET calculation in WCET calculation step, we will start from WCET calculation methods, then the rest two topics.

2.2.1 WCET Calculation Approaches

There are primarily three WCET calculation methods: timing schema, path-based calculation, and Implicit Path Enumeration Technique (IPET). They are classified based on the way program paths are evaluated and the way instruction timing information is used. Let’s have a look how they differ in these two perspectives.

1) Tree-based approach: it estimates builds a syntax tree of the program and

traversing the tree from bottom to up meanwhile applying timing rules at the nodes of the tree (called "timing schema") and the estimated WCET is obtained at the root of the tree (top node). This approach is quite simple to implement while being efficient. However, it is difficult to exploit infeasible paths this way because the timing rules (timing schema) are limited to local program statements, which implies that finding timing conflicts in the execution paths are not easy.

2) Path-based approach: it computes the execution time for every feasible path in the flow graph of the program, and then searches for the largest one as the estimated WCET. While path-based approach can handle various flow information, it enumerates a large amount of paths which leads to inefficiency and large space consumption. Recent works have sought to reduce this expensive path enumeration by removing infeasible paths from the flow graph [20].

(3) Implicit path enumeration Technique (IPET): it uses arithmetical constraints to model the program flow and low-level execution times: first it builds linear equations or inequalities (constraints) to represent the program flows and the execution time of the program is represented using variables in these constraints, then maximizes the execution time equation, and that maximization value is the estimated WCET [11], [15], [21]. This maximization calculation is normally done using ILP (Integer Linear Programming) solver. Unfortunately it is prohibitively computing expensive to solve ILP problem, which actually was proven to be NP-hard [22], [7]. A second concern with this approach is that the constraints count in all the program paths including all the false ones, this would incur two results: a huge number of equations/inequalities to solve and the estimated WCET not tight.

We will discuss in detail these three approaches in the related work on WCET calculation approaches in Section 2.3.3.

2.2.2 Microarchitecture Modeling

Micro-architectural features, such as pipelining, caching and branch prediction, improve the average system performance a lot and have caught a lot of attention on their modeling for accurate WCET analysis. We review the various Micro-architecture modeling techniques in this Section and focus on cache modeling.

Pipelining

The execution of an instruction can be divided into several stages. MIPS instructions classically take five steps: fetch instruction from memory, decode instruction and read registers, execute the operation or calculate an address, access an operand in data memory, and write the result into a register. Instead of executing instructions one after another, we can overlap the execution of multiple instructions where each one of the instructions is at a particular execution stage at a time. This approach is called pipelining. It improves the throughput by executing several instructions in parallel. If each stage of the instructions takes about the same amount of time and there are enough instructions to carry out, the speedup due to pipelining is equal to the number of stages in the pipeline. However, this ideal speedup of pipelined execution normally is not achieved because of some events preventing the instructions from preceding though the pipeline smoothly. These events are called hazards in the literature [6], which are classified into three types: Structural hazards, Data hazards and Control hazards.

Branch Prediction

An instruction must be fetched at every clock cycle to sustain the pipeline, yet the decision about whether to branch does not occur until the 4th stage of instruction execution-access an operand in data memory. This delay in determining the proper instruction to fetch is called a control hazard or branch hazard [6]. Branch prediction is to address control hazards, by computing as early as possible the address of the subsequent instruction to be executed to decide whether the conditional branch is taken or not taken. If prediction is correct, we can save the processor idle wait time between the start of the branch instruction and its production of the outcome (which is called a branch penalty); if prediction is wrong, the wrong path instruction effects must be undone and correct path gets resumed (the time in between is called *misprediction penalty*, which is usually equal to or slightly higher than the branch penalty).

Caching

Caching is a mechanism to cushion the speed gap between processor and main memory, by embedding in between small and fast cache memories. There are two types of architecture: *Von Neumann architecture*[5] and *Harvard architecture*. The former has unified cache (program instructions and data are stored in a single storage) and the latter has split instruction cache and data cache to store instructions and data separately. The Harvard architecture is widely used in embedded systems and it enables analyzing instruction cache and data cache separately. In the thesis, we deal with Harvard architecture, and focus on data cache timing analysis which is more challenging and has not had an efficient solution yet.

[5] http://en.wikipedia.org/wiki/Von_Neumann_architecture

2.2.3 Flow Analysis

To estimate WCET of a program analytically, all possible paths must be analyzed. Program flow analysis is to determine the possible paths through a program. Thereafter, for each generated path, the execution time can be estimated for a particular architecture.

Definition 2.1 ***Path***: a path from node u to node v in the control flow graph of a program is a sequence of directed edges $n_0, n_1, \ldots, n_k$ such that $n_0 = u$, $n_k = v$ and (n_i, n_{i+1}) is an edge in the graph.

Flow analysis yields information about loop bounds, which functions are called and exclude infeasible program paths from the source program or the compiled code of the program. Approximation during computation is necessary to reduce path explosion, for example a simple loop with an if-then-else statement that iterates 100 times would generate 2^{100} possible paths. [23] and [24]carry out this approximation to reduce the number of paths to be analyzed by removing infeasible paths whenever it is possible. Another approach is to merge paths where a path enumeration is needed [25], [26], [27], [28]. This approach is to further reduce the path explosion and merging is applied to points of data-dependent conditionals, loop with multiple paths inside and loops with unknown loop bounds.

2.3 Related Works on Static WCET Analysis

WCET is the upper bound on the execution time of a program over all possible data inputs on a specific hardware platform. WCET of a task is an essential input to the schedulability analysis of hard real-time systems. It is difficult to estimate the WCET through simulation for any non-trivial program due to the very large number of possible

inputs. Thus static analysis techniques are employed to derive an upper bound on the WCET of a program.

Let's have a review of related works/achievements on performing timing analysis of embedded software through static analysis.

2.3.1 Related Works on Micro-architecture Modeling

One important yet difficult problem for static timing analysis is to model the timing effects of complex micro-architectural features present in modern processors. There are some other techniques on the modeling instruction caches and data caches. There are also some works on the modeling of other Micro-architecture features such as branch prediction, pipelining, prefecting etc.

Integer Linear Programming (ILP)

ILP is an established method to find the worst case execution path in timing analysis [29], [30]. Based on the control flow graph of a program, a linear optimization problem is constructed that maximizes the flow through the program. Li et al. [31], [32], [33] used ILP for instruction cache modeling and integrated it with their ILP-based WCET calculation framework. Their Micro-architecture modeling tool models direct-mapped instruction cache using a set of graphs called Cache Conflict Graphs (CCG), which models flow transfer information (instruction sequence in a basic block) among memory blocks mapping to the same cache line. Cache misses are captured by flow transfer between conflicting memory blocks. For set associative instruction caches, they introduced an extra set of graphs called Cache State Transition Graphs (CSTG) to model the more complicated behaviors of set-associative instruction caches. This ILP-based instruction cache modeling utilizes more detailed flow information and obtains good accuracies; however, its tight integration with WCET calculation results in an increase in

analysis time, especially for set associative caches. Li [34] models out-of-order pipeline execution, branch prediction and instruction cache and their interactions for WCET analysis in a framework named 'Chronos[6]'. Timing information for basic blocks of the program subject to some execution contexts and constraints on the occurrences of execution contexts (instruction cache state, branch prediction information, etc.) are generated from the modeling tool and combined with the flow constraints and user constraints to form a complete ILP problem.

Abstract Interpretation

Ferdinand [35], Theiling [36] used abstract interpretation [37], [38] for instruction cache analysis. After abstract cache states get collected at program points, program flow is traversed to update abstract cache state for each cache access and combine abstract cache states at program joints. Then memory references are classified by the abstract cache states into four categories: always hit, always miss, persistent and not classified, which will be combined with cache information for subsequent analysis.

Timing Schema

Timing schema [39]was extended by [40], [41], [42] to model pipelining and caching. The extended timing schema uses a data structure called worst case timing abstraction (WCTA) for the time-bound for a program construct. To model instruction cache effects, it divides memory accesses in a path of a construct into three groups: first/last/other references to the cache lines. Cache hits/misses of the first references uses execution information preceding the path and the last references are needed by paths succeeding it, thus they are remembered by augmenting the WCTA. At concatenating two paths across program constructs, the last references of the earlier path are used to resolve some of the

[6] http://www.comp.nus.edu.sg/~rpembed/chronos/

hits/misses in the first references in the later path, and the first/last references of the concatenated path will be computed from the first/last references of the two concatenating paths. To combine pipelining and caching effects, it simply superimposes cache miss penalties to the execution obtained from the pure pipeline analysis.

Many other static timing analysis approaches have also been proposed, such as symbolic simulation [43], implicit path enumeration [37], timing analysis language [39], and Engblom [44] provides a comprehensive study of various pipelines for WCET analysis in his doctoral dissertation.

2.3.2 Related Works on Flow Analysis

WCET calculation evaluates the costs of the program paths and takes the maximum one as the WCET. In contrast to simulation, where each program path is evaluated separately (the major drawback of the simulation approach), WCET analysis evaluates multiple program paths simultaneously. The key problem is how the program paths are grouped for evaluation, which is what flow analysis concerns about.

Flow analysis technique is used in optimizing compilers to statically analyze the properties of the source code for exploration of common sub-expressions or dead-code elimination [45]. It has been applied to single task analysis by [46].

In [48], [49], **Error! Reference source not found.**, flow analysis techniques are used in optimizing compilers to analyze timing behavior of pipelines and instruction caches.
(1) First, they use a static cache simulator [50], [51] to analyze instruction caches. The simulator examines the program control flow and categorizes instructions into four classes: always hit, always miss, first hit, and first miss.

(2) Next, pipeline analysis is carried out by using the cache category information.

(3) Last, the timing analyzer predicts the WCET of the program by using the worst case execution times of the code segments containing loops, function calls etc from bottom to up. [52]adopts data flow analysis for data cache analysis. After code generation and all optimizations, the address range of data references is obtained on low-level representation. Unknown data references are not considered and array ranges should be manually annotated.

Let's compare approaches in Section 2.3.1 (micro-architecture modeling) and 2.3.2 (flow analysis).When only concerning instruction cache analysis, the flow analysis approach and the abstract interpretation approach perform instruction cache analysis before WCET calculation phase; while in the extended timing schema, ILP and symbolic simulation approaches, instruction cache analysis is integrated with WCET calculation. The latter integrated approaches can be more accurate results at the price of using more program path information (i.e. a higher computation cost).

2.3.3 Related Works on WCET Calculation

There are primarily three WCET calculation methods: timing schema, path-based calculation, and Implicit Path Enumeration Technique (IPET). They are classified based on how program paths are evaluated and how instruction timing information is used.

1) Timing Schema

A tree-based approach [53], [54], [55], [56] is proposed to estimate the WCET of a program through a bottom-up traversal of its syntax tree and applying different timing rules at the nodes (called "*timing schema*"). Once the times of lower level constructs have been obtained, the time of the higher level construct containing them can be

estimated. This approach is very simple and efficient. However, it's difficult to exploit infeasible paths in this approach, because the local estimation in timing schema cannot account for infeasible paths which are defined by constraints across multiple constructs.

2) Path-based Calculation

This approach estimates the WCET by computing execution time for the feasible paths in the program and searching for the largest one as the estimated WCET. The infeasible paths are determined and excluded by exploiting the correlations of different program parts [43], [47], [48], [49], [57]. Arnold et al. [58] and Healy et al. [47], [49], [57] search the longest loop path in each loop-nesting level. Infeasible loop paths found by program path analysis are disregarded.

Definition 2.2 ***a loop path***: is a control-flow connected sequence of blocks in a loop which starts with the loop header and terminates at a block with a transition either to the loop head or out of the loop.

This path-based WCET calculation approach can handle various flow information, but it enumerates a huge number of paths which would result in high timing-consuming computing complexity. Recent works [59] try to reduce this expensive path enumeration by removing infeasible paths from the flow graph of the program.

3) Implicit Path Enumeration Technique (IPET)

Li and Malik [60] proposed a technique to represent all paths implicitly by using ILP. The program flows are represented as linear equations or constraints and ILP solver is used to maximize the execution time of the program under these constraints. Suppose the cost of each basic block B_i is known, denoted as $\cos t_i$, and denote its execution times as v_i, then the execution time of a complete program with N basic blocks can be

expressed as $\sum_{i=1}^{N} \cos t_i * v_i$. The goal is to maximize the value of this function over all valid combinations of v_i and $\cos t_i$ where v_i is bounded by the control flow of the program and some extra flow information. It's simple and efficient for path enumeration, in addition, ILP solvers are easy to access, therefore the IPET approach has been adopted by some researches for WCET calculation [36], [61], [62], [63], [64], [65], [66].

2.4 Cache Timing Analysis

While significantly reducing the average memory access time and leading to a total shorter execution time, caches complicate the timing behavior of the system and make it difficult to predict. In real-time systems (soft and hard) where timing correctness is on top of functional correctness, timing guarantees are especially essential in order to verify the functional behavior as well as to efficiently use hardware resources. Timing behavior of programs on a given architecture in presence of caches is important and challenging to be obtained for schedulability analysis in embedded systems. WCET analysis is to estimate such an upper bound of execution time of a program on a given architecture, and the estimated WCET would be used as input for schedulability analysis.

As we mentioned in Section 2.1, we have two options to obtain the WCET estimation: simulation and analysis. Cache simulation is unaffordable to guarantee analysis safety as explained in Section 2.1.1, efficient static timing analysis is desired to deliver safe and accurate WCET bounds.

2.4.1 Instruction Cache Analysis

In a single task context, the timing behavior for instruction caches has been extensively studied [29], [46]. Instruction cache timing analysis has achieved fruitful results and it is

now possible to obtain an accurate estimate of WCET in presence of instruction caches for non-preemptive systems [48], [49], [67]. These results can be generalized to preemptive systems [68], [69], [70], [71], [72], [73], [74], [75].

2.4.2 Data Cache Analysis Progress and Current Situation

While previous analysis techniques focus on modeling instruction caches, few works [76], [77], [78] have studied the data cache impacts on the WCET of programs. And data cache behavior has often been restricted in static worst case timing analysis. Data cache behavior is more difficult to predict, because it depends on both the control flow of the program and on the input data. While instruction addresses are fixed, a single instruction can access several data memory addresses, for example operations on an array. And to make data cache modeling more difficult, the data memory access addresses can be unpredictable as they may depend on the specific input data.

Let's first look at a terminology: predictable and unpredictable memory accesses.

Definition 2.3 ***Predictable and Unpredictable Memory Accesses:*** An unpredictable memory access is a load or store access whose reference address is unknown during estimation of the WCET. Conversely, a predictable memory access is a load or store access whose reference address is known during the estimation of the WCET [1].

Note that in this thesis we do not differentiate the terms between predictable (unpredictable) memory accesses and input independent (dependent) memory accesses as they essentially mean the same[7].

[7] For a scalar variable, memory access is predictable as address is constant. Some only use 'predictable/unpredictable', also based on whether it is an array element with index expression and whether it is dependent on input.

2.4.3 Input Independent Data Cache Analysis

The main open problem in data cache analysis is the timing behavior of unpredictable memory accesses. Simulation based approach is unaffordable, and while few researches [76], [77], [78] have been performed on WCET analysis accounting for data caches, they unfortunately either oversimplify analysis of input-dependent accesses (please refer to Section 2.4.4) or exclude input dependency such as CME frameworks[79], [80], [81] and several more as follows.

Many approaches [26], [67], [82], [83] analyze data caches timing behavior, but they are only limited to memory references for scalar variables and fail to study codes with dynamic references (i.e., arrays and pointers). There are further works on modeling data caches considering dynamic references but only limited to programs without input dependent accesses. White et al [84] provide a static simulation based timing analysis method to analyze array accesses for direct-mapped caches, but it can only analyze accesses whose address can be computed at compile time. Few more researches have been performed on WCET analysis for programs with only predictable data accesses. Wolfe et al [22] proposed to use an ILP formulation for direct mapped data caches for software timing analysis. Mueller et al [85] use data flow analysis techniques on data cache analysis. Unknown data references are not considered and array ranges would have to be annotated by the user. Gosh et al [79]proposed to use CME. These CMEs compute re-use vectors to calculate cache accesses within loops. It is arguably the most accurate analytical models for data cache behavior, but it imposes quite stringent requirements on loops to be analyzable.

2.4.4 Input Dependent Data Cache Analysis

In the field of WCET analysis, extensive researches [79], [22], [85] have been performed on WCET analysis for programs with only predictable data accesses.

Difficulties of Analyzing Input-Dependent Memory Accesses

The difficulty in analyzing input-dependent memory accesses is the inability to obtain the knowledge of the reference address of an input-dependent item. This is due to its dependency on some unknown input data. The lack knowledge of the exact cache block the referenced address would be mapped to (resulting from the unknown reference address) would add difficulty in predicting a memory access as a cache hit or a cache miss as you do not even know which specific cache block you are looking at. At most a range of this reference address can be obtained by analyzing the program and data definition. An example program exchangesort is given in Figure 2.1, which illustrates the motivation for this analysis for both predictable memory access and unpredictable memory access. The array (int a []) to be sorted is the input to the program. Exchange of array elements for sorting is based on pos_min (line 14 and 15) variable which depends on input array a. (line 9 and 10). Array access a[pos_min] in line 14 and line 15 becomes unpredictable at analysis time. But array access a[i] in line 13 and line 14 is predictable at analysis time.

```
1 int main()
2 { int MAX 1215;
3   int a [1215];
4   int i, j, c, temp, pos_min;
5   for (i = 0; i < MAX; i++)
6   {pos_min=i;
7    for (j = i+1; j < MAX; j++)
8    {
9     if (a[pos_min] > a[j])
10   {pos_min = j;
11   }
12   }
13  temp = a[i];
14  a[i] = a[pos_min];
15  a[pos_min] = temp;
16  }
17 }
```

Figure 2.1 Example with Input Dependent Access (ExchangeSort)

Some data-cache analysis frameworks consider programs with both predictable and unpredictable data accesses, but they sadly oversimplify the analysis [77], [27] of the unpredictable data accesses. The oversimplification has two forms:

1) Classify all input dependent memory accesses as non-cacheable [86], by considering them as always in the main memory (not allowed to go into the cache for the analysis though in fact it's not the case).
2) Assume this single reference address would access all the cache blocks of the referenced data's array. In [72], it's assumed that all elements of an array are loaded to the cache for each array access. However, the impact of unknown data

accesses on the cache contents is simplified as only one array element will be loaded to the cache.

Both assumptions 1) and 2) would cause WCET pessimism because:

1) The former would consider every unpredictable memory access as a cache miss which is surely far from the real case. This would obviously overestimate the execution time as some of them do go into caches, especially when a large unpredictable array access is involved, this overestimation can be large.
2) The latter tries to narrow the analysis address range of this unpredictable memory reference from the complete program address range to the address range of the complete array to which this memory reference belongs. However, it's still an overestimation of cache misses when the array occupies much more than one block, which is an unfortunate truth in most application programs. While in fact this memory reference under consideration can cause at most one cache miss, this assumption would assume it causing the number of cache misses the same as the number of blocks this array occupies. The pessimism is large when the array size is large, which is typically true.

Other researches have been performed on WCET analysis for programs with both predictable and unpredictable data accesses. Ferdinand et al [77] introduced an abstract interpretation to predict data cache behavior. A persistence analysis determines the maximum number of cache blocks that remain in cache. For an unpredictable array access, it is assumed that all cache blocks of an array are accessed, which replace many other cache blocks. However, the method is only described in theory without providing experimental results. Lundqvist et al [76] introduced a symbolic simulation technique to classify predictable and unpredictable memory accesses. Unpredictable data structures are tagged as non-cacheable and consequently always require a cache miss. The main

drawback is that input dependent memory accesses are classified as non-cacheable and are treated as always cache miss. Staschulat et al [78] provides a theoretical framework that classifies data access predictable and unpredictable in a single direct mapped data cache. For predictable accesses, local cache simulation and data flow techniques are applied, for unpredictable accesses, they use a cache miss counter updated on examining useful cache blocks (dependent access influence on independent ones) and persistent cache block analysis, i.e. examine which array elements are never replaced by other memory blocks (dependent access results in a cache hit or miss). And analysis of the two access patterns is combined using ILP formulation. No function calls or cache associativity is considered in the work. While trying to reduce the computation and space complexity of ILP programming, they bound the number of possible considered paths under a constant parameter and whenever more paths occur, a path merge algorithm is applied, i.e. compromises are made between accuracy and complexity.

2.5 Our WCET Analysis Framework Overview

Our contribution is a worst case timing analysis for data caches that classifies memory accesses as predictable and unpredictable. We have developed a new architecture which classifies input-independent and input-dependent accesses and allocates them into predictable caches and unpredictable caches respectively. The advantage of this new architecture is to allow advancing analyzing these two different data cache accesses separately. Each scalar variable is predictable, because the memory location is fixed. We classify array accesses as predictable and unpredictable memory accesses based on if the index expression can be statically computed or not. An unpredictable memory access has two influences on cache behavior. First, it has an impact on the current cache contents by possibly replacing some other cache block. Second, the access itself requires an additional cache miss, if the element is not in the cache. We masterly avoid this

complicated influence calculation by making full use of reuse distance [82] of input-dependent memory references using a fully associative cache configuration. The advantage of a fully associative cache configuration skips the concern of calculating (narrowing) the specific address (range) of the requested data as the data item can goes into any block of the fully-associative cache.

For predictable memory accesses, we adopt CME techniques [79], [78], [8]. For unpredictable memory accesses we propose a novel timing analysis algorithm that tightly predicts the cache timing behavior in the term of number of cache hits and misses. The framework can allow adopting advanced cache timing analysis techniques separately for the predictable and unpredictable memory accesses as their interferences are eliminated thanks to our novel cache-partitioned architecture. This architecture is similar to cache partitioning, to allocate the predictable memory accesses into the predictable cache part and allocate the unpredictable memory accesses into the unpredictable cache part. The difference between our architecture and traditional cache partitioning is that the two cache parts' configurations do not need to be the same. For the unpredictable cache part, we make it fully associative in order to make full play of reuse distance property with lack of knowledge of the memory reference addresses. For the predictable cache part, it can be of any cache structure. Let's set it to be k-way associative for the general case, where direct-mapped and fully-associative caches are two special cases of k-way associative structure as if k=1 it's a direct-mapped cache part, if k is equal to the number of cache blocks it's a fully-associative cache part.

The flow of our analysis framework is given as show in Figure 2.2.

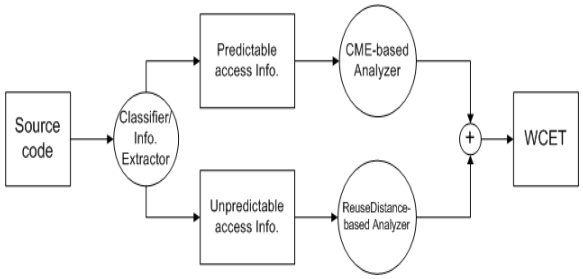

Figure 2.2 Flow Diagram of the Analysis Framework

From the C source programs, our desired information for further analysis of both unpredictable and predictable memory accesses is extracted using our designed parser and extractor. Then CME framework is used to analyze our predictable data cache behavior for predictable memory accesses, and our proposed reuse distance algorithm is employed to deal with the unpredictable accesses for our predictable data cache. Finally, we combine results from these two independent parts to yield overall cache misses and our WCET.

Methodology: To evaluate the accuracy of our analysis, the estimated result should be compared against some reference result. Ideally, it should be the actual worst case.

However, as explained earlier, it is often impossible to know the actual worst case.

As an alternative, we can use an approximate to the actual worst case by doing an in-exhaustive simulation over some sets of data input which are likely to produce the worst case. But this is too time-consuming, some may even take days to simulate. Here we use manual resource to examine the benchmark source programs and find the worst case input pattern as bad as possible. We inspect the important parts of a program (the inner loops) to see how the executions of these important parts are affected by the data input, then we try a set of data input which maximizes their execution (luckily this is not too hard as typical benchmarks are quite straight forward to find the worst case input pattern).Then set the values as stimuli for the set of input data. The simulation using this chosen input data set is called the observed worst case. Correspondingly, the result

produced by our analysis is called the estimated worst case. The relationships of the three values are: observed WCET $\leq$ actual WCET $\leq$ estimated WCET.

We denote as the WCET the time to access the two caches. The core execution time of the processor is not considered for our comparison. We define overestimation to examine the accuracy of our analysis,

$$overestimation = \frac{t_{analysis} - t_{simulation}}{t_{simulation}} * 100\%$$

Where $t_{analysis}$ means the analysis WCET, $t_{simulation}$ refers the observed WCET from our simulation using selected worst-case data.

2.6 Our Underlying Architecture Model

Our architecture model uses a uniprocessor with a memory hierarchy and the Harvard architecture. We focus on one level data caches, each cache is a direct-mapped or k-way set-associative or fully-associative cache using, replacement policy is LRU replacement policy. In the case of write misses, we assume a fetch-on-write policy so that writes and reads are modeled identically (read miss penalty is the same as the write miss penalty, we take it as 10 cycles while a hit would take 1 cycle). Our current analysis assumes a memory hierarchy indexed either by virtual or physical addresses.

Environment: We run all the experiments on a 3 GHz Pentium IV machine with 1-GB main memory. The operating system is Linux-2.4.18 and Windows XP. The parameters of the processor components will be reported together with the experimental results in the chapter 3.

CHAPTER.3
CACHE TIMING ANALYSIS

Caches in Embedded Systems improve the average case performance, but they, especially data caches, are a source of unpredictability in worst case software timing analysis. This is a critical problem in real-time systems, where tight WCET is required for their schedulability analysis. Few works [76], [77], [78] have studied the data cache impacts on the WCET of programs, but they can only handle programs without input dependent data accesses. To solve this problem, we have developed a novel architecture and a WCET analysis framework for this architecture. The architecture classifies predictable and unpredictable accesses and allocates them into predictable caches and unpredictable caches. Based on this new architecture, we build a framework to analyze these two different data cache accesses separately, using the CME and the reuse-distance-based algorithm respectively. This framework produces an appreciable WCET tightness compared with simulation results from other state-of-the-art related work, and our architecture also shows very low hardware complexity.

The remainder of Chapter 3 is organized as follows. Section 3.1 presents the related background on designing more predictable architecture and introduces CME framework. Section 3.2 describes the theory and in detail the workflow of our analysis framework (data dependency analysis; reference data access extraction; input independent analysis and input dependent analysis). Section 3.3 presents our experimental setup, results and analysis. Finally Section 3.4 concludes our WCET work and discusses future research potentials.

3.1 Principle of Cache-Aware WCET Analysis

To obtain an accurate WCET, past efforts have been made in two directions. One direction tries to develop more predictable new architectures. The other tries to develop new analysis approaches to obtain more accurate WCET results. In our work, we try to combine the two approaches and develop a classified cache architecture and enable some new analysis techniques for programs with data dependent accesses.

3.1.1 More Predictable Cache Architectures

There are several approaches to make caches more predictable and efficient.

1) Cache partitioning [74], [87] is a mechanism developed to reserve blocks of cache for individual tasks such that cache hits becomes predictable. But this has significant impact on the execution performance, because of fragmented address space. The address space becomes nonlinear such that multiple tasks can be mapped same cache partition.
2) Cache locking: to lock frequently used cache blocks [75, 88, 89]. Selected data is loaded into cache and locked in place so that it may not be replaced until the cache is explicitly unlocked. The disadvantage is locking and unlocking mechanism introduces an overhead. There could be performance loss if data is too big for the locked cache, then the whole cache must be unloaded it to be predictable. In addition, it increase area and power cost as this approach requires larger caches and main memory to become effective (which is undesired especially in embedded systems where resources are expensive and limited). While cache partition and lock strategies are certainly a very useful add-on to improve cache predictability and efficiency, they do not solve the general cache analysis problem in which all tasks share the entire cache.

3) Scratch-pad SRAM [90] also has been added to hold frequently used cache blocks to make it more predictable. But this is at significant area and power cost. This also requires compiler support for additional address space and context switching.

Our proposed classified cache architecture is similar to cache partitioning, but with minimal hardware complexity overhead and no loss of performance or memory overhead.

3.1.2 Cache Miss Equations (CME)

Extensive researches have been performed on WCET analysis for programs limited to predictable data accesses and some considered unpredictable data accesses but oversimplified the timing analysis. Refer to Sections 2.4.3 and 2.4.4 for details.
Below we would have a closer look at CME [79], which we would deploy in our framework.

CMEs are to describe the iteration points where reuse is not realized, using a set of equations formed by deriving exact hit/miss patterns for every reference in the loop nest.

Definition 3.1 ***Iteration Point***: is to represent every iteration of a loop nest, for example, in n-dimensional loop, iteration point can be represented as $\vec{i} = (i_1, i_2, ..., i_n) \in \mathbb{Z}^n, n \in \mathbb{R}$.

Definition 3.2 ***Iteration Space*** of n-dimensional loop nest: the polytope bounded by the n enclosing loops.

Definition 3.3Reuse: when the data item is referenced multiple times where a memory reference is a static instruction read or write operation. Data reuse is essential to predict

cache behavior because a datum is only in the cache if its cache line/block was referenced before.

Definition 3.4 ***Reuse Vector: R*** is a memory reference such that it has the same memory line for two iteration points $\vec{i}_1$ and $\vec{i}_2$, where $\vec{i}_1 \prec \vec{i}_2$, then $\vec{r} = \vec{i}_2 - \vec{i}_1$ is a reuse vector.

Definition 3.5 Reuse vector space: a concrete mathematical representation showing the shape of iterations that reuse the same data, which describes the direction and distance of the reuse in a methodical way.

Refer to an example (Figure 3.1) to comprehend the concepts. In the example in Figure 3.1, iteration point is $\vec{i} = (i,k,j) \mid 0 \le i < N, 0 \le k < N, 0 \le j < N$, iteration space is $\{(i,k,j) \mid 0 \le i < N, 0 \le k < N, 0 \le j < N\}$.

```
1 double A[N][N], B[N][N], C[N][N];
2 for (i=0; i<N; i++)
3 for (k=0; k<N; k++)
4  for (j=0; j<N; j++)
5   C[i][k]+=A[i][j] * B[j][k]
```

Figure 3.1 Illustration Example of Iteration Point and Iteration Space

Recall reference B[j][k] from our running example (Figurte 3.1). At iteration point $\vec{i}_0 = (1,2,3)$, it accesses the array element B[3][2], which is the same array element accessed at iteration point $\vec{i}_1 = (3,2,3)$. Therefore, we have a reuse vector $\vec{r} = \vec{i}_1 - \vec{i}_0 = (3,2,3) - (1,2,3) = (2,0,0)$. And a reuse vector for A[i][j] can be $(0,1,0)$.

There are three steps to obtain the cache hit/miss information for every reference using CMEs.

1) Compute reuse vector: Wolf and Lam's reuse framework [91] is based on polyhedral theory to model the reuse distance using polytopes and Presburger

formula (array rows aligned at cache line boundaries) and the reuse space is obtained. Cross-row reuse vector is added if not aligned.

2) Build Cache Miss Equations

   a) Cold Miss Equations: Build equations for iteration points where the reuse did not hold because one of the following reasons.

      - Reuse from point outside the iteration space

   b) Reuse from data that is mapped to a different cache lines (cache-aligned) Replacement Miss Equations: Build equations for iteration points where the reuse did not hold because:

      - Multiple references (include self-interference) map to the same cache set

3) Solve CMEs: to find the total number of misses of a loop nest

   - Each equation represents a convex polyhedron in $\mathbb{R}^n$, with the integer points inside representing potential cache misses.
   - The solution set of the CMEs represents the cache misses of a reference and its volume represents the number of misses.

   a) Two methods of solving the CMEs, one is to solve the equations analytically (only applies to direct-mapped caches), the other is to traverse the iteration space and check whether a point is solution or not of the equations that define the polyhedron (work for all cache architecture). We used the latter because the former also has high computation complexity (NP-Hard problem).

   b) Two theorems are exploited to computer the overall cache miss number:

   - **Theorem 3.1**: *The set of all misses of a reference along a reuse vector is given by the union of all the solution sets of the equations corresponding to that reuse vector.*

This can be easily understood because for a given reference and a reuse vector, an iteration point can produce a miss if it is either a compulsory miss or a replacement miss (Figure 3.2).

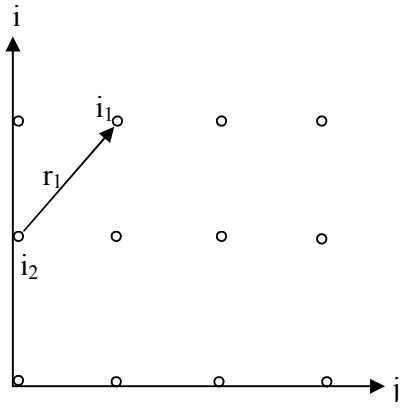

Figure 3.2 Cache Misses of a Reference along a Reuse Vector

- **Theorem 3.2**: *The set of all miss instances of a reference is given by the intersection of all the miss-instance sets along the reuse vectors.*

  This can be understood this way: given a reference, an iteration point results in a hit if it exploits the locality of at least one of the reuse vectors. Refer to Figure 3.3.

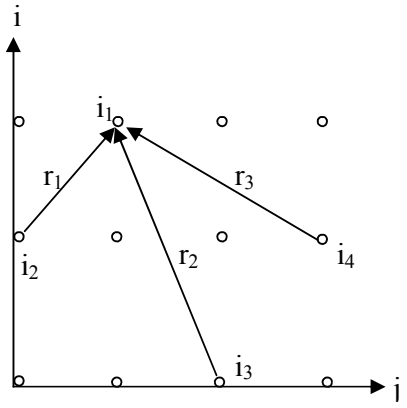

Figure 3.3 All Cache Misses of a Reference

After the traverse, we obtain the output: the set of iteration points whose corresponding references are the potential misses.

3.2 Our WCET Analysis Framework

Few works [76], [77], [78] have studied the data cache impacts on the WCET of programs, but they can only handle programs without input dependent data accesses. To solve this problem, we have developed a novel architecture and a WCET analysis framework for this architecture. We propose a new architecture, where data access is classified and analyzed separately. In other words, the architecture classifies predictable and unpredictable accesses and allocates them into predictable caches and unpredictable caches. Our framework is based on this new architecture and it analyzes these two different data cache accesses separately. For predictable accesses, the CME framework [79], [92] is employed for WCET analysis. For unpredictable accesses, we calculate array element reuse distance, which is independent of reference memory access, to examine detailed unpredictable data cache behaviors, thus making the estimated WCET more accurate under this in-depth exploration.

The flow of our analysis framework is shown as in Figure 3.4 (which was refined from Figure 2.1 in chapter 2).

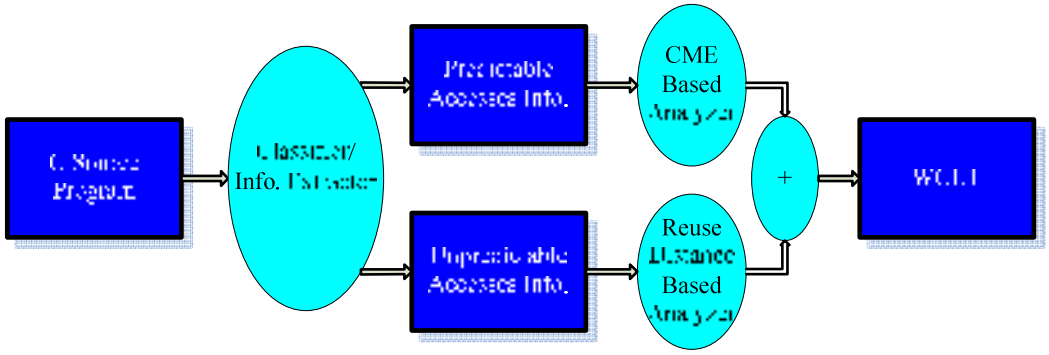

Figure 3.4 Refined Flow Diagram of Our Analysis Framework

We present these parts as follows. Section 3.2.1 introduces the parser and extractor. Section 3.2.2 introduces our classification and reuse distance extraction implementation, Section 3.2.3 uses CME to analyze the cache timing behavior for input independent

memory accesses. Section 3.2.4 presents our reuse-distance based algorithm to obtain the cache timing behavior for input dependent memory accesses, and a small example is given.

3.2.1 Proposed Cache-Classified Memory Architecture

Traditional cache memory hierarchies perform well on the average case behavior by taking the advantage of spatial and temporal locality. But they complicate system timing analysis. So modeling data cache timing behavior accurately has become a critical issue in WCET analysis. Among other difficulties in modeling data cache behavior, there is one open problem that it's nearly impossible to design an efficient and effective method to model data cache behavior considering input dependent accesses.

Currently recent works restrict these input dependent accesses as un-cacheable and thus eliminating the interface between these accesses and the predictable data accesses and simplifying the analysis. An alternative practice is to assume the input dependent accesses spanning the whole data caches and analyze them as input independent accesses. These models simplify the approach at the cost of WCET tightness and accuracy.

We propose a new cache-classified architecture where the interface between input dependent accesses and input independent accesses is eliminated. With compiler support, the data accesses are mapped into two classified caches: input dependent cache and input independent cache based on the property of the data references-input dependent or not. And data accesses are fed into the processor from the two caches independently assisted by annotating the load and store instructions. For example, if an annotated load instruction is executed; the processor would try to fetch the data in this instruction from the input dependent data cache. The benefit from this novel architecture

is that the input dependent data accesses and input independent data accesses do not compete against each other for cache blocks.

And to minimize the potential hardware overhead such that area overhead is theoretical zero, we maintain the same size cache size (noted as C_s) of overall data caches as traditional architecture, and divide this size between input dependent data cache and input independent data cache.

In our framework, we classify data memory accesses as predictable and unpredictable, depending on whether their addresses can be statically calculated or not. We proposed a new architecture to store classified data access (predictable and unpredictable memory accesses) separately. In other words, the architecture classifies predictable and unpredictable accesses and allocates them into predictable caches and unpredictable caches. Our WCET analysis framework is based on this new architecture and analyses these two different data cache accesses separately. For predictable accesses, the CME framework [79] is employed for WCET analysis. For unpredictable accesses, we calculate array element reuse distance to examine detailed unpredictable data cache behaviors, thus making the estimated WCET more accurate under this in-depth exploration. This cache-classified architecture smartly enables to analyze input dependent memory accesses efficiently and permits separate analysis technique development for input independent access and input dependent access analysis.

This new architecture was implemented by extending the Sim-outorder Model in SimpleScalar [93] to include two parallel level data caches. SimpleScalar is an open source micro architectural simulator [94]. The implementation uses SimpleScalar version 3d available from [92]. Unpredictable data accesses are put as annotations in the 64 bit PISA instruction set [93]. Original benchmark source is converted to assembly,

then manually annotated with unpredictable data access and converted to SimpleScalar executable to be run on the new architecture using the Simplescalar GCC [92].

From the C source programs, our desired information for further analysis of both unpredictable and predictable memory accesses is extracted using our parser and extractor. Then CME framework is used to analyze our predictable data cache behavior for predictable memory accesses, and our proposed reuse distance algorithm is employed to deal with the unpredictable accesses for our predictable data cache. Finally, we combine results from these two independent parts to yield overall cache misses and the estimated WCET.

3.2.2 Classification and Reuse Distance Extraction

In our framework, data memory accesses are classified as predictable and unpredictable, depending on whether their addresses can be statically calculated or not. Based on this criterion, we design the classification algorithm as shown in Figure 3.5.

By determining input-dependency of array memory accesses, unpredictable memory accesses are identified. Then the unpredictable array element reference order is able to be extracted by simulating the program once (because though their respective address could not be decided at compile time, the relative order of unpredictable accesses is often determined, which is luckily true for typical programs as it's quite rare to have complicated input dependent function calls inside others). Then reuse distances of unpredictable array elements can be calculated for unpredictable data cache behavior analysis.

```
1  Memory accesses by scalar variables are predictable (as memory address is constant)
2  Memory accesses by predefined array accesses are classified as predictable.
3  Look at the index expression of other arrays
4 If all the variables in it are input independent
5 {
6  memory accesses by this array is predictable
7 }
```

Figure 3.5 Memory Access Classification as Predictable/Unpredictable

Definition 3.4 ***Reuse Ddistance****:* the reuse distance of an array element is the number of unique unpredictable array elements referenced since the last access to that array element. And we will elaborate its usage in later Sections.

3.2.3 Cache Analysis for Input Independent References

In this Section, we further describe CMEs platform which builds miss equations to provide the number of misses for each reference in a set of nested loops, and we also demonstrate how to obtain the WCET for input-independent references.

Gosh et al [79] proposed in 1997 a mathematical method to characterize data cache behavior using a set of linear Diophantine equations, whose solutions are representing the possible misses for each reference and therefore the data cache miss/hit behavior can be predicted for each reference by solving these Diophantine equations. This is the well-known CME analysis method, which obtains an analytical and precise description of the cache memory behavior for loop-oriented codes. A major drawback of the original CME framework is its timing complexity of directly solving cache miss equations due to its NP-hard nature.

To solve the issues in the original CME, X. Vera et al [95] proposed an analytical method based on CMEs to produce cache misses efficiently to predict data cache behavior, during which reducing computing complexity is focused on and statistical sampling technology is used.

In our predictable memory access modeling, we reuse Coyote framework [96]. To estimate the miss number of predictable references in loops, we gather related information about loops, variables and references in loops, and use this information as input to Coyote. Coyote estimates the miss number of each predictable reference for us.

3.2.4 Cache Analysis for Input Dependent References

This Section presents reuse-distance and our designed algorithm that we use to obtain the cache timing behavior of input-dependent memory accesses. Refer to *definition 3.4* for the concept of reuse distance.

Reuse distance is a metric for measuring program's cache behavior [80]. Larger reuse distance indicates higher probability of cache misses. Beyls et al[97] explored the method of reducing the reuse distance and cache misses based on reuse distance visualization. It has been observed that in a fully associative LRU cache with n cache lines, a reference will hit if the reuse distance d is smaller than n, otherwise it would be a cache miss. Further, as listed in Table 1, reuse distance may help identify the type of a cache miss for a data access, based on relationship between reuse distance d and the number of cache lines n , according to Beyls et al[97].

Cache Miss Type	Relation between d and n
Conflict miss	$d < n$
Capacity miss	$n \leq d \leq \infty$
Cold miss	$d = \infty$

Table 3.1: Relationship between Cache Miss Types and Reuse Distance

It was proved by experimental results in [80] that reuse distances predict cache behavior exactly for fully associative caches. Based on the above findings in Table 3.1, we developed an algorithm for the timing analysis of input-dependent memory accesses.

With classified input-dependent reference and extracted reuse distance (r.d.) information from Section 3.2.1, the estimated number of unpredictable data cache misses can be generated by our algorithm in Figure 3.6. As illustrated in Figure 3.6, we keep a record of all the reuse distances of cache lines in the input dependent data cache at the current state. Each cache line's reuse distance is the minimum reuse distance of all data elements in this line. And when a new reference comes, we first check if this reference is already in the current data cache, if so we have a cache hit, otherwise it would be a cache miss and we need to decide if this data item would go into the input dependent data cache. We first check if its reuse distance is smaller than the maximum reuse distance of all the current cache lines. If true, the cache line containing the maximum reuse distance gets replaced out with the cache line that reference lies in., and the reuse distance of the cache is updated accordingly.

```
1  cache_miss=0;
2  while not end_of_file
3  {
4    get array element;
5    get reuse distance (r.d.);
6
7    if array element in cache
8   {
9      if array element r.d. < its cache line r.d.
10     {
11     its cache line r.d. = array element r.d.;
12      }
13    }
14    else
15   {
16    cache_miss++;
17       {
18      //replace out the cache line with largest reuse distance and update information
19        cache line (with largest r.d. in cache) = its cache line;
20        its cache line r.d. = array element r.d.;
21        cache r.d. = max(cache line r.d.);
22        }
23    }
24  }
```

Figure 3.6 Reuse-Distance Algorithm for Input-Dependent Access Analysis

We illustrate this pseudo code using the following example consisting of a sequence of five unpredictable memory accesses, in Table 3.2. Access sequence is: a[1], a[2], a[3], a[4], a[1], a[4]. Assuming the cache we used here is fully associative with the cache line

size of 4 bytes, and the number of cache sets n is 3. We compute first each data item's reuse distance and then output the miss condition for each reference as in the 3rd column of Table 3.2.

Reference	Reuse distance (d)	Hit/miss type
a[1]	∞	cold miss
a[2]	∞	cold miss
a[3]	∞	cold miss
a[4]	∞	cold miss (replace a[1])
a[1]	3	replacement miss
a[4]	1	Cache Hit

Table 3.2: Reuse Distance and Cache Miss Type Example

3.3 Experiments and Results

This Section presents the results of our analysis framework for predictable cache and unpredictable caches. We expect that our analysis is tight compared to simulation result.

3.3.1 Experimental Setup

We conduct experiments to evaluate our instruction cache and data cache timing analyses. The experiments share some commonalities, such as the benchmarks used, the methodology, and the experimental environment.

Our experimental results are using different benchmarks which are described in Table 3.4. This describes the benchmarks with number of unpredictable data access and no of loop iterations of the unpredictable data access. If the loop time is not determined, we describe it using both instruction memory (code) and data memory (data) size (Bytes).

Benchmark	No. of instructions with unpredictable accesses per iteration	No. of loop iterations/ Code/data size (Bytes)
Exchangesort	2	737505
Smallexample	1	300
Quart	1	1000
Cover	1	2000
Count	2	1000
FFT	8	1852(code), 256(data)
FIRFilter	1	240(code), 80(data)

Table 3.3 WCET Analysis Benchmark Profile

These programs have been used by other researchers for WCET analysis. Exchangesort was obtained from Staschulat et al [78] and Quart, Cover and Count were obtained from Mälardalen WCET Benchmarks[8], FFT (Fast Fourier Transform) and FIRFilter (Finite impulse response filter in signal processing) are obtained from [73]. And they were slightly modified to create input data dependency.

In Table 3.3, column " No. of loop iterations" gives the total number of loop iterations of each benchmark program in their object code. Column "No. of instructions with unpredictable accesses per iteration" gives the number of instructions with unpredictable memory accesses in an iteration. This Table is to give the overall idea of the benchmarks in the terms of locality (loop times) and the basic amount of instructions with unpredictable memory references in a loop.

[8] Mälardalen WCET Benchmarks: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

3.3.2 Experimental Results and Analysis

Then for each benchmark we set the suitable cache configuration for predictable cache and unpredictable cache. We set this configuration manually at this stage. To observe the necessary cache misses the following sets of configurations were set by experimentation. For exchangesort, small, quart benchmarks the total data cache was set as 8KB with predictable data cache 4KB and unpredictable data cache 4KB. For cover, predictable: 512B and unpredictable: 512B for a 1KB total. For count it is set as predictable: 4KB and unpredictable: 512B for a 4.5KB in total. For FFT and FIRfilter, the predictable and unpredictable caches are direct-mapped and fully associative respectively and each is 512B large.

We denote as the WCET the time to access the two caches. The core execution time of the processor is not considered for our comparison. We define,

$$overestimation = \frac{t_{analysis} - t_{simulation}}{t_{simulation}} * 100\%$$

Where $t_{analysis}$ means the analysis WCET, $t_{simulation}$ refers the observed WCET from our simulation using human observed worst possible input set.

From the results Table 3.4, we can see that the classified analysis works quite well, it's quite tight. Please note that the only exception FFT, the main overestimation comes from the limitation of CME where the loops should be well designed (bounded iteration time, index expression is affine function of the variables) and it's improved a lot than previous data at around 40% overestimation.

Benchmark	$t_{analysis}$	t_{sim}	Overestimation (%)
Exchangesort	1597536	1546740	3.28
Small	8632	8074	6.91
Quart	18749	18065	3.79
Cover	18654	18609	0.24
Count	16045	16009	0.22
FFT	254027	205136	23.83
FIRfilter	60793	56028	8.50

Table 3.4: Our Analysis Framework Experiment Results

To further define our architecture performance (to see the performance degrade from a traditional architecture with perfect cache—no cache miss), we define the deviation from perfect cache as:

$$PerformanceDegradition = \frac{\text{Our platform simulation - Chronos simulation}}{\text{Chronos simulation}} * 100\%$$

To compare our architecture performance with perfect performance architecture of zero data cache miss, we used Chronos architecture for comparison, because it does not have data cache and considers main memory access speed as cache/CPU (1 cycle). For example of exchangesort benchmark, the observed performance deviation/degrade is 0.621% in simulation WCET compared to Our new architecture is nearly perfectly cached; close to no cache miss case in Chronos. And the cache size is reasonably small (1KB instruction cache, 4KB predictable cache, 4KB unpredictable cache). Therefore, the system performance comparison is fair. Also our architecture is easy to implemente: one can simply realize it by extending the Sim-outorder Model in open source SimpleScalar [19] to include two parallel level data caches. We can comfortably say that this new architecture has ignorable hardware complexity overhead and almost no loss of performance compared to perfect cache architecture.

3.4 Conclusions and Future Works

The WCET analysis is to estimate a priori (before execution) the WCET of a given program on a given architecture. It is important for the schedulability analysis of real-time systems, which requires timing correctness on top of functional correctness. Cache memories are small memories between CPUs and main memories to reduce the average memory access time remarkably by making the use of program localities. But they have created unpredictability issues in real-time systems, because of their probable nature: they only copy a small portion of most likely to-be-accessed data from main memory to bridge growing access speed difference between CPU and main memory. And therefore, accurately modeling and analyzing caches are extremely important.

In this Chapter 3, we develop novel cached memory architecture and build a new WCET analysis framework on top of it to analyze WCET for data dependent programs. WCET analysis is considered, in the literature, to be accurate if the analysis result is less than 10% overestimation compared to WCET simulation result, where the WCET simulation should all the input sets possibilities. We use observed worst case input set to obtain this WCET simulation result. To our best knowledge, our work is among the earliest to render very good WCET tightness compared with simulations for data dependent programs with low computing complexity and almost no hardware overhead.

Our goal is to analyze memory references being cache hit or miss (i.e. the required data is in cache or not) for a given architecture in the worst case, and sum up total data access time based on the knowledge of cache and main memory access time.

Previous researches focus on WCET for programs without input-dependent data access

for cached memories, because it is much more difficult to analyze input-dependent data reference being cache hit or miss for its unpredictable memory address at analysis time. The unpredictable memory address would cause its memory block mapping to an unknown cache block, thus resulting in intractability of cache status, let alone WCET.

To solve this problem, we develop novel memory architecture having parallel predictable cache and unpredictable cache. The architecture first classifies predictable and unpredictable accesses using parser and extractor, and then allocates them into predictable and unpredictable caches accordingly. Based on this new architecture, we build an associated WCET analysis framework as in Figure 3.4 to model the timing effects of input-independent and input-dependent data accesses. The platform mainly consists of two parts: existing CME (Cache Miss Equations) based analyzer and our reuse-distance based analyzer to analyze predictable and unpredictable memory accesses respectively.

Initial experimental results show quite small WCET overestimations ranging from 0.22% to 6.91% compared to our simulation results with almost zero hardware area overhead, with exception to FFT benchmark of overestimation of 23.83% whose overestimation comes mainly from CME part.

As a side strongpoint, our architecture also opens the opportunity of developing techniques of analyzing input independent memory accesses and techniques of analyzing input dependent memory accesses separately. Other current timing analysis methods can be applied to the predictable data cache or the unpredictable data cache accordingly.

The smartness of our designed reuse-distance based algorithm lies in that we only need to obtain the memory reference order rather than the unpredictable memory address at

compile time for determining a memory access being a cache hit or miss [99]. For analysis safety, we need to determine the worst case memory access order for the given program. This can often be easily done by observing the memory access pattern of the given program and simulating once the observed worst-case data input set. From worst case memory access order, our algorithm calculates the reuse distances of these memory accesses. The reuse distance of a memory access is the number of unique memory data referenced since the last access to that data. Our reuse-distance-based algorithm determines each reference to be a cache hi/miss based on the following observation: in a fully associative LRU (Least Recently Used) cache with *n* cache lines, a reference will hit only if the reuse distance *d* is smaller than *n*. Then the number of cache hits and misses from unpredictable memory accesses are obtained. Combining the analysis results from CME and reuse-distanced based analyzer, our analysis framework produces the final WCET. Our analysis results are very tight compared with simulations on our architecture, and the overestimation comes mainly from the existing CME part.

For our future work, we would like to:

- Automate the division size between the predictable and unpredictable cache. For example, based on the reference number distribution among the two classes: If the reference number ratio $r = \frac{no.Input - Independent\_Data\_Accesses}{no.Input - Independent\_Data\_Accesses}$: the number of input dependent data accesses and input independent data accesses, then allocation $\frac{r}{1+r} * C_s$ to predictable data cache and $\frac{1}{1+r} * C_s$ to unpredictable cache. Or maybe more appropriate allocation can be found after research exploration and experiment.
- Extend our framework to analyze deeper interaction between data cache and micro-architectural components like pipelining, branch prediction and instruction cache.
- Integrate timing analysis with the compiler

CHAPTER.4
PROVING NP-COMPLETENESS

In this chapter, we present the proofs for the NP-completeness of our WCET analysis problem presented in Chapter 3.

First of all, we introduce the basic terminology in the research field of problems' hardness.

Definition 4.1 ***Decision problem***: a problem whose solution is either 'yes' or 'no'

Definition 4.2 ***P problem***: if a decision problem is solvable by a polynomial time deterministic algorithm, then it belongs to class P.

Definition 4.3 ***NP problem***: if a decision problem is solvable in polynomial time by a nondeterministic algorithm, then it belongs to class NP. Obviously $P \subseteq NP$, because that any deterministic algorithm can be used to check a nondeterministic algorithm for a problem.

Definition 4.4 ***Polynomial time reducibility and Polynomial Transformation***: reduction for which the required transformation can be executed by a polynomial time algorithm. Such a reduction is called polynomial transformation.

Its significance is that if we have a polynomial time reduction from one problem (or language as any problem can be described in a language) L_1 to another L_2, we can be sure that:

1) any polynomial time algorithm for the second problem L_2 can be converted into a corresponding polynomial time algorithm for the first problem L_1;
2) $L_1 \notin P \Rightarrow L_2 \notin P$

3) $L_2 \in P \Rightarrow L_1 \in P$

4) Polynomial Time Reducibility is transitive

Definition 4.5 ***NP-complete problem***: a decision problem is in NP and every problem in NP can be polynomial reducible to this problem, i.e. it is in NP and it is NP-hard.

Definition 4.6 ***NP-hard problem***: a problem, to which one NP-complete problem can be transformed, is NP-hard, and it has the property that it can not be solved in polynomial time unless P=NP. NP-hard problem can be in NP or not.

Meaning of Proving NP-completeness for a Problem

Given a new problem, naturally our first question is: can it be solved with a polynomial time algorithm (is it a P problem)? As if the answer is obviously 'yes', we only need to focus our efforts on trying to find a polynomial algorithm as efficient as possible. However, if no polynomial time algorithm is apparent, the second natural question to ask is: 'is the problem NP-complete'.

Assume that we are given a decision problem in NP, just as it might have been obvious that it is polynomially solvable, it might be obvious that it is NP-complete. If so, then it cannot be solved with a polynomial time algorithm and efforts can be concentrated on finding an efficient approximation algorithm with small approximation error. Therefore, proving a problem belongs to NP-complete or NP-hard is very meaningful.

The structure of this chapter is as following. In Section 4.1 we introduce two general techniques with examples for proving NP-completeness: restriction, local placement where restriction technique will be deployed in Section 4.2. In Section 4.2, we prove that WCET analysis problem is NP-complete using restriction technique. In Section 4.3 we conclude the Chapter and direct future works.

4.1 Techniques for Proving NP-completeness

We can simply proving a problem being NP-complete from another proven NP-complete problem using the following process. Given a decision problem Π, to prove its NP-completeness, the following four steps will be the process of devising its NP-Completeness proof:

(1) Showing that Π is in NP

(2) Selecting a known NP-Complete problem Π'

(3) Constructing a transformation f from Π' to Π, and

(4) Proving that f is a polynomial transformation.

In this chapter 4, we skip the first step that the given problem is in NP, as each of the problems considered is easily seen to be solvable in polynomial time given a nondeterministic algorithm solution.

Techniques for Proving NP-Completeness

There are three general proof types that can provide a suggestive framework for deciding how to attempt to prove a new problem is NP-Complete: (1) restriction, (2) local replacement, (3) component design [100].

We first briefly introduce component design, and then in detail describe restriction and local replacement. Component design is a little complicated, it is to use the constituents of the target problem instances to design certain components that can be combined to realize instances of a known NP-complete problem. There are two types of components: 'making choice' and 'testing properties'. For example, selecting vertices and choosing truth values (true or false) for variables belong to 'making choice' components; checking that each edge is covered and checking that each clause is satisfied belong to 'testing

properties' components. For the detail and examples of implementing this technique, please refer to Chapter 3.2 of [100].

We focus on the first two general proof types with detailed examples. The restriction technique will be used to prove NP-completeness of our problems in Section 4.2.

4.1.1 Restriction Technique

Proof by restriction is the simplest and most frequently applicable. An NP-Completeness proof by restriction for a given problem $\Pi \in NP$ is simply to demonstrate that Π contains a known NP-Complete problem Π' as a special case.

The essence is the specification of additional restrictions to be placed on the instances of II so that the resulting restricted problem will be identical to II'. The key lies in giving one-to-one correspondence between their instances that preserves 'yes' and 'no' answers. This one-to-one correspondence of the transformation from II' to II is usually apparent and simple.

For example, Exact Cover By 3-SETS (X3C) can be shown to be NP-complete by restriction technique to known NP-Complete 3-Dimensional Matching (3DM).

Definition 4.7 ***3-Dimensional Matching (3DM)*****:** given 3 different sets W, X, and Y, each triple in M corresponds to a 3-way bond that would be acceptable to all three participants, the question is that is it possible to arrange n bonds so that each element appears only in one bond and that every element has a bond?

Definition 4.8 ***Exact Cover by 3-SETS (X3C)*****:** for a finite sex X with $|X| = 3q$ and a collection C of 3-element subsets of X, does C contain an exact cover for X, i.e. a sub-collection $C' \subseteq C$ such that every element of X occurs in exactly one member of C'?

Proof:

X3C can restrict its instances to 3-sets that contain one element from a set W, one from a set X, and one from a set Y, where W, X, and Y are disjoint sets having the same cardinality, thereby obtaining a problem as 3DM problem. □

To make use of this technique, we focus on the problem to be proven and attempt to restrict away its non-essential aspects till a known NP-Complete problem appears. The proof technique in Section 4.2 is restriction.

4.1.2 Local Replacement Technique

NP-completeness by local replacement is to pick some aspect of a known NP-complete problem instance to make up a collection of basic units, and then, to replace each basic unit, in a uniform way, with a different structure to obtain the corresponding instance of the target problem.

For example, 3SAT can be shown to be NP-complete by local replacement technique from SAT.

Definition 4.9 ***Literal*****:** either a variable or its negation is a literal, e.g. u_1 is positive literal, $not(u_2)$ or $\overline{u_2}$ is negative literal.

Definition 4.10 ***Clause*****:** literals connected using OR within parentheses is a clause

Definition 4.11 ***Conjunctive Normal Form (CNF)***: formulae that are a conjunction (AND) of clauses

Definition 4.12 ***Satisfiability (SAT)*:** for a set $U = \{u_1, u_2, ..., u_n\}$ of Boolean variables and a set $C = \{c_1, c_2, ..., c_m\}$ of clauses over U, is there a satisfying truth assignment for C, i.e. can the variables of the given Boolean formula (with only AND, OR, NOT, variables and parentheses) assigned to make the formula TRUE?

Definition 4.13 ***3-Satisfiability (3SAT)*:** for a collection $C = \{c_1, c_2, ..., c_m\}$ of clauses on a finite set U of variables such that $|c_i| = 3 \; for \; 1 \le i \le m$, question is that does there exist a truth assignment for U that satisfies a formula where all the clauses in C are CNF?

***Proof*:**

The guiding idea is that we need to transform SAT to 3SAT, and during the transformation, we need to obey the uniform-way transform to units of SAT instance to obtain an instance of 3SAT. The basic units of an instance of SAT, which are the clauses, are replaced by a collection of clauses according to the same general rule (transform in a uniform way). Each replacement constitutes only local modification of structure.

Let $U = \{u_1, u_2, ..., u_n\}$ be a set of variables and $C = \{c_1, c_2, ..., c_n\}$ be a set of clauses making an arbitrary instance of SAT. The goal is to construct a collection of C' of three-literal clauses on a set U' of variables such that C' is satisfiable if and only if C is satisfiable.

Set $U' = U \cup \left(\bigcup_{j=1}^{m} U_j' \right)$ and $C' = \bigcup_{j=1}^{m} C_j'$

We only need to show how C_j' and U_i' can be constructed from c_j.

Let c_j be given by $\{x_1, x_2, ..., x_k\}$ where x_i 's are all literals derived from the variables in U. The way of constructing $C_i^{'}$ and $U_j^{'}$ depends on the value of k.

Case 1. k=1, $U_j^{'} = \{y_j^1, y_j^2\}, C_j^{'} = \left\{\{x_1, y_j^1, y_j^2\}, \{x_1, \overline{y_j^1}, y_j^2\}, \{x_1, y_j^1, \overline{y_j^2}\}, \{x_1, \overline{y_j^1}, \overline{y_j^2}\}\right\}$

Case 2. k=2, $U_j^{'} = \{y_j^1\}, C_j^{'} = \left\{\{x_1, x_2, y_j^1\}, \{x_1, x_2, \overline{y_j^1}\}\right\}$

Case 3. k=3, $U_j^{'} = \Phi, C_j^{'} = \{\{c_j\}\}$

Case 4. k>3, $U_j^{'} = \{y_j^{'} : 1 \le j \le k-3\}$

$$C_j^{'} = \{\{x_1, x_2, y_j^1\}\} \cup \left\{\{x_{i+2}, \overline{y_j^i}, y_j^{i+1}\} \mid 1 \le i \le k-4\right\} \cup \{\{x_{k-1}, x_k, \overline{y_j^{k-3}}\}\}$$

Then we need to show that the C' of clauses is satisfiable if and only if C is.

Suppose that $f : U \to \{T, F\}$ is a truth assignment satisfying C, we are going to show that f can be extended to a truth assignment $f' : U' \to \{T, F\}$ satisfying C'. Since the variables in $U' - U$ are partitioned into sets $U_j^{'}$ and variables in each $U_j^{'}$ occur only in clauses belonging to $C_j^{'}$ one at a time, we only need to show how f can be extended to sets $U_j^{'}$ one at a time and in each case to verify that all the clauses in the corresponding $C_j^{'}$ are satisfied.

1) If $U_j^{'}$ was constructed under case 1 and case 2, then the clauses in $C_j^{'}$ are already satisfied by f, so we can extend f arbitrarily to $U_j^{'}$, for example by setting $f^{'}(y) = T$ for all $y \in U_j^{'}$.

2) If $U_j^{'}$ was constructed under case 3, then $U_j^{'}$ is empty and the single clause in $C_j^{'}$ is already satisfied by f.

3) If $U_j^{'}$ was constructed under case 4: since f is a satisfying truth assignment for C, there must be a least integer I such that the literal x_I is set true under f. If I is either 1

or 2, then we set $f'(y_j^i) = F$ for $1 \le i \le k-3$; If I is either k-1 or k, then we set $f'(y_j^i) = T$ for $1 \le i \le k-3$; Otherwise we set $f'(y_j^i) = T$ for $1 \le i \le I-3$ and $f'(y_j^i) = F$ for $I \le i \le k-3$.

It's easy to verify that these options will ensure that all the clauses in C_j' will be satisfied, so all the clauses in C' will be satisfied by f'. Conversely, if f' is a satisfying truth assignment for C', it is easy to verify that the restriction of f' to the variables in U must be a satisfying truth assignment for C.

Thus C' is satisfiable if and only if C is.

And finally we need to prove this transformation is polynomial time. Observe that the number of three-literal clauses in C' is bounded by a polynomial in mn . Therefore the size of the 3SAT instance is bounded by a polynomial function of the size of the SAT instance. And obviously it's easy to verify that this is a polynomial transformation. □

4.2 Proving WCET Analysis is NP-complete

In this Section 4.2, we prove that WCET analysis problem is NP-complete by using restriction technique.

The WCET of any block in an un-cached program remains unchanged in different runs. On the other hand, the WCET of a block in a cached program may change between iterations, depending on cache contents. Note that although $WCET_{uncached} \geq WCET_{cached}$ holds for most practical systems, using $WCET_{uncached}$ usually implies significant performance loss, it is desirable to find a tighter bound for $WCET_{cached}$.

Theorem 4.1: *The analysis complexity of the WCET of cached/un-cached programs is NP-complete [101].*

Proof:

We assume that an Assembly program is transformed into a control graph G = (V, E), where G denotes a directed graph. Each vertex in V represents a basic block consisting of data manipulation instructions, and each edge in E represents the execution dependencies between blocks. Each vertex is assigned a non-negative value (weight) to represent its execution time. A basic block can have sequential and forward-branch instructions, but not any backward-branch instructions.

We assume that the control graph has exactly one input (root) vertex and one output vertex, where the input (output) vertex is the first (last) basic block to be executed in the construct. We assume that there exists at least one directed path from the root vertex to any vertices in the graph. The WCET of any block in an uncached program remains unchanged in different runs. On the other hand, the WCET of a block in a cached program may change between iterations, depending on the cache contents.

To analyze the execution time of a program based on its control-flow graph model, we first need to get the execution time of each block.

On one hand, WCET of any block in an un-cached program remains unchanged in different runs. Deriving the WCET of an un-cached program is equivalent to finding the path with the highest total cost in the directed control-flow graph. Since the execution cost of a block does not change with its execution sequence, we just need to find the worst case execution path in one iteration to find the WCET of the block. In a general

graph G, finding the simple path with no cycles, with the maximum weight between two vertices is equivalent to finding a simple path of length K or longer between two vertices [99]. Since this problem is NP-complete for a directed graph, the time complexity of finding the maximum-cost path in a control-flow graph G is NP-complete.

It should be noted, however, that the problem of finding the maximum-cost path can be solved in polynomial time for an acyclic directed graph [102].

On the other hand, the WCET of a block in a cached program may change between iterations, depending on the cache contents. Although $WCET_{uncached} \geq WCET_{cached}$ holds for most practical systems, using $WCET_{uncached}$ for $WCET_{cached}$ usually implies significant performance loss, it is desirable to find a tighter bound for WCET cached. For WCET analysis of cached program, the execution time of a control-flow construct of the program, which may consist of several blocks to be executed for several iterations, may change with the cache contents. It is a more complicated version of WCET analysis of un-cached program. If an instruction (construct) to be executed resides in the cache memory, then its execution time will be reduced by the time difference of accessing main memory and cache. In order to solve this problem of WCET, all (implicitly assumed) schedules would have to be tested which is an NP-complete optimization problem and exponential in number of tasks. □

That's why researchers need to accept the overestimation in favor of a reduced analysis complexity for WCET analysis of cached programs. But in the meantime the tightness of static WCET analysis is also essential, otherwise there would be hardware resources waste.

4.3 Summary

In this chapter 4, we provide theoretic background in proving NP-completeness, and we prove that WCET analysis problem is NP-complete.

For future research work, we would examine efficient approximation algorithms and techniques. This is for the purpose of guiding and assisting improvement for our WCET analysis platform, and also for approximating other coming-up NP-complete and NP-hard problems accurately and efficiently. In addition, we would like to extend our single task environment to preemption environment, like the work in [103].

CHAPTER.5
DATA ALLOCATION

Compared to general-purpose processors, embedded processors are application-specific, which means that they have only a single application running and allow longer analysis and compilation time for applications. These two features of embedded processors provide us the opportunity to improve the performance of application by exploiting the longer permitted compilation time to optimize performance. Techniques include efficient code generation [46], [102], [98], architecture exploitations [104], [105], [106], [107], and retargetable code generation [108], [109].

5.1 Problem Description

In order to improve system performance in embedded systems, we examine the techniques to allocate data in main memory in the way such that data cache performance gets improved during the execution of code. We improve cache performance by allocating data in the program to increase their locality. This data allocation is possible because that code generation in embedded systems can be tuned to the given cache configuration (cache line size, cache size etc). We target at direct-mapped caches because that it's easier and more direct to analyze and also because it is fastest to access and most commonly used as best choice for most applications among direct-mapped, set-associative and fully-associative caches. Our goal is to minimize compulsory and conflict misses in direct-mapped cache by data management technology. The focus is because of the observation that compulsory and conflict misses constitute approximately 50% of cache misses for typical programs [94], and we would be glad that if we can devise an algorithm with low complexity to deduce these two

types of cache misses as much as possible to increase the program locality and thus the cache and system performance.

Our underlying idea is simple: first cluster data accesses into memory lines based on their space locality such that accesses with high space locality can be allocated in the same cache line such that cache compulsory misses are minimized; second map these memory lines to cache lines in the way that memory lines with minimized conflicts share same cache lines such that cache conflict misses are minimized. And our algorithm improves much in cache hit ratio according to our premiere experimental results, especially well for codes with high conflicts on a given architecture.

5.2 Data Memory Allocation for Scalar Variables

1) Step 1.1: We build a vicinity graph to represent the locality among data: higher vicinity weight distance stands for higher space locality. This vicinity graph is to be used for step 1.2: cluster data into memory lines. This vicinity graph has a vertex for each scalar variable, and an edge between every two variables. Every edge has a vicinity weight which stands for the locality between the edge's two endpoints, and which initially has weight 0. For every vertex i, traverse the whole access sequence with the window size of memory line size, for every instance of another vertex j falling in the window, add the weight of the edge between the two vertices by the probability of the program flowing along $i \leftrightarrow j$, Then we have the vicinity graph after all traverses have been done for every vertex.

2) Step 1.2: We cluster vicinity graph vertices into memory lines such that the total vicinity weights are maximized (space locality is maximized). We design a greedy algorithm as in Figure 5.1.

```
1  Initialize X=vertex set of the vicinity graph, memory line M as empty;
2  While X is non-empty
3  {
4    Choose a vertex u with max sum of all edge weights in X;
5    Put u into M, update X=X-{u};
6    While size of M is smaller than size of memory line
7     {
8       Choose a vertex v with max sum of edge weights to the vertices inside M
9       Put u into M, update X=X-{u};
10    } //one memory line is compacted
11  }//every variable is allocated into one memory line
```

Figure 5.1 Cluster Scalar Variables into Memory Lines

3) Step 1.3: We build cluster interference graph to represent cache conflict among memory lines. Consider every cluster M from step 1.2 as a node, examine the access sequence again to calculate how many times every two nodes alternate along the sequence and assign this value as the edge weight between the two nodes.
4) Step 1.4: We assign memory location to every variable. This step is to map the cluster nodes (memory lines) to cache lines such that the total cache conflicts among memory lines are minimized, i.e. partition cluster nodes into cache-line-number clusters so that the total edge weight inside clusters from step 1.3 is minimized. We design a greedy algorithm as in Figure 5.2.

```
1  Initialize X=vertex set of cluster interference graph (one vertex is one memory line);
2  While X is non-empty
3  {
4    Create a new page P in memory;
5    While (size of P is smaller than number of cache lines N)&&(X is non-empty)
6    {
7      choose a vertex v with min sum of edge weights to all the vertices the same cache
       line i (i=0...N-1), assign v to cache line i of page P
8      } //one memory page is full
9  }//every memory line is in a memory page, with map to a cache line
```

Figure 5.2 Assign Memory Locations to Scalar Variables

After Step 1.4, we have an assignment of all scalar variables to memory location in the way that compulsory and conflict misses in the data cache would be minimized when the variables are accessed during program execution.

5.3 Data Memory Allocation for Array Variables

1) Step 2.1: We build interference graph for the set of arrays accessed noted by X in the program. Examine the access sequence to calculate the innermost loop's loop bound number for any two array variables accessed inside the loop and assign this value as the edge weight between the two array nodes.
2) Steps 2.2.1: We calculate cost of assigning array u to addresses starting at Addr. as in Figure 5.3.
3) Step 2.2.2: We assign memory locations to array variables as in Figure 5.4.

```
1  For every assigned array node v that have an edge with array u
2  {
3    For each loop l where both u and v are accessed, loop bound B
4   {
5    calculate n as the number of times u and v alternate to be in the same cache line
6    cost=cost+ n*B
7     }
8  }
9  Return cost
```

Figure 5.3 Cost of Assigning the Start Address of Array u to Addr

```
1  For i=0 to n-1
2  { cost =∞;  min_cost=0;
3  //min_cost records cache line number with min mapping cost for each array variable
5    For j=0 to N-1 (where N is the number of cache lines)
6     {
7      If cost(v, Addr+j)<cost
8       {
9       cost=cost(v, Addr+j)
10     Min_cost=j
11       }
12    } //array u start address is chosen s.t. cost(u,Addr) from step 2.2.1 is minimized
13   Assign address (Addr+min_cost) to 1st element of array u
14   //which has (i+1)th largest sum of incident edge weights
15  Addr = Addr + size of array u  + min_cost
16 }
```

Figure 5.4 Assign Memory Locations to Array Variables

5.4 Experiment Results and Conclusion

Then we lead the experiment with experiment setup as follows:

We use direct-mapped cache with write-through policy, cache line size is 4-words, data cache size is 256B, and instruction cache size is 1KB. We simulate the original (unoptimized) benchmarks using Simplescalar on Pentium 4 CPU 3.0GHz to estimate data cache hit ratio for comparison with the cache performance after allocating the data using our algorithm. Our performance measure is the data cache hit ratio.

The profile of benchmarks we used is given in Table 5.1. Experiment results of the above six benchmarks are given in Table 5.2.

Benchmark	Scalars No.	Arrays No.	Array Access No.	Description
Matrix_add	2	3	3	matrix add
Laplace	2	2	10	Laplace transform
Dequant	7	5	5	from MPEG decoder
Idct	20	3	9	inverse discrete cosine transform
Inner_prod	2	3	2	matrix inner product
FFT	20	4	20	fast Fourier transform
Diag_elim	2	3	4	Tri-diagonal elimination

Table 5.1: Data Allocation Benchmark Profile

Benchmark	data cache hit ratio unoptimized	Data cache hit ratio optimized
Matrix_add	9.59%	75.51%
Laplace	95.92%	95.92%
dequant	37.35%	81.84
Idct	28.37%	56.12%
Inner_prod	6.53%	73.46%
FFT	2.02%	23.94%
Diag_elim	3.97%	73.89%

Table 5.2: Data Allocation Experiment Results

Our algorithm for scalar and array variables allocation in memory considers both cache line size and cache size, and it can significantly improve performance in execution of embedded code. It works especially well for codes with high conflict misses on the given data cache architecture.

In the future, we would like to automate the algorithm and extend the work from direct-mapped caches to limited set-associative caches.

CHAPTER.6
CONCLUSION

In this thesis, we have presented methods to analyze the timing behavior of caches with input-dependent accesses. We propose a new architecture consisting of parallel predictable and unpredictable caches. It classifies data access as predictable or unpredictable accesses and then analyzes these two kinds of accesses separately. The proposed classified-cache architecture is to differentiate and analyze the timing effects of the predictable cache accesses and the unpredictable cache accesses separately. Based on this classified architecture, we build our WCET analysis framework. For predictable accesses, the CME framework [92] is employed for the WCET analysis. For unpredictable accesses, we calculate the array element reuse distance to examine the detailed unpredictable data cache behaviors, thus making the estimated WCET more accurate under this in-depth exploration. Compared with simulation results of other state-of-the-art related work, our analysis framework shows that: our analysis is conservative (safe), reasonably tight, and it has very little hardware complexity overhead.

We have also provided the theoretic background and the proof of the NP-completeness of the WCET analysis problem.

In addition, we have examined techniques to improve system performance by allocating data (both scalar and array variables are considered) in programs to increase their locality. Taking into account cache configurations (cache line size, cache size etc), our technique is aimed to minimize compulsory and conflict misses in direct-mapped cache using data management technology. We first cluster data accesses into memory lines to

keep space locality accesses in the same cache line such that cache compulsory misses get minimized; and then we map these memory lines to cache lines in the way that memory lines with minimized conflicts share same cache lines such that cache conflict misses get minimized. Our algorithm demonstrates its good ability in improving cache hit ratio, especially for codes with much conflicts on the given architecture.

In summary, this thesis provides a comprehensive framework to analyze instruction and data cache effects for single task real-time systems with input data dependency. The framework also introduces a data allocation technique to improve the cache performance.

Future Work

For our future work on the WCET analysis framework, we would like to:

- Automate the division size between the predictable and unpredictable cache, for example, based on the reference number distribution among the two classes: If the reference number ratio r is the no of input dependent data accesses and input independent data accesses, then allocation r/(1+r)*Cs to data dependent data cache and 1/(1+r)*Cs to data independent data cache.
- Extend our framework to analyze the interaction between the data cache and micro-architectural components like pipelining, branch prediction and instruction cache.
- Integrate timing analyzer with the compiler.
- Extend our framework from single task environment to multitask environment considering preemption, by studying and incorporating cache preemption related delay analysis.

For future research work on algorithms and complexity, efficient approximation algorithms and techniques are to be explored for NP-complete and NP-hard problems.

For our future work on the data allocation technique to improve cache performance, besides automation, we would like to:

- Extend the work from direct-mapped caches to limited set-associative caches and examine its corresponding computation complexity;
- Consider other data layout techniques, such as loop tiling.

BIBLIOGRAPHY

[1] B. Tabbara and A. Sangiovanni-Vicentelli. *Function/Architecture Optimization and Co- design of Embedded Systems*. Kluwer Academic Publishers, 2000.

[2] Jean-Philippe Dauvin. *Semiconductor Market and Industry Mega-Trends* 2005-2010. In *MEDEA DAC*, May 2005.

[3] David A. Patterson, John L. Hennessy. *Computer Organization and Design-the Hardware/Software Interface*. Morgan Kaufmann, 2005.

[4] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: problems and solutions. Volume 5 , Issue 3es, Spring 1999.

[5] Wm. A. Wulf and Sally A McKee. Hitting the Memory wall: Implications of the Obvious. *Computer Architecture News*, 23(1), pp. 20-24, March 1995.

[6] David A. Patterson, John L. Hennessy. *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann, 1996.

[7] T.M.Austin, *Hardware and Software Mechanisms for Reducing Load Latency*. PhD thesis, University of Wisconsin-Madison, April 1996

[8] J.Rawat. *Static Analysis of Cache Performance for Real-time Programming*. Master's Thesis, Iowa State University, May 1993

[9] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper and S. Wiegratz.. Static WCET Analysis of Real-Time Task-Oriented Code in Vehicle Control Systems. Second Ineternational Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), 2006.

[10] R. Kirner and P. Puschner, "Transformation of path information for WCET analysis during compilation," in *Proc. 13*th *Euromicro* for *Conference of Real-Time Systems, (ECRTS'01)*. Delft: IEEE Computer Society Press, June 2001, pp. 29–36.

[11] A. Ermedahl, "A modular tool architecture for worst-case execution time analysis," Ph.D. dissertation, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.

[12] C. Ferdinand, R. Heckmann, and H. Theiling, "Convenient user annotations a WCET tool," in *Proc. 3*rd *International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[13] J. Gustafsson, "Analyzing execution-time of object-oriented programs using abstract interpretation," Ph.D. dissertation, Dept. of Information Technology, Uppsala University, Sweden, May 2000.

[14] C. Healy, M. Sj¨odin, V. Rustagi, and D. Whalley, "Bounding Loop Iterations for Timing Analysis," in *Proc. 4*th *IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June1998.

[15] N. Holsti, T. L°angbacka, and S. Saarinen, "Worst-case executiontime analysis for

digital signal processors," in *Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference)*, 2000.

[16] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *IEEE Proceedings on Real-Time Systems*, 2003.

[17] J. Engblom, "Analysis of the execution time unpredictability caused by dynamic branch prediction," in *Proc. 8*th *IEEE Real-Time/ Embedded Technology and Applications Symposium (RTAS'03)*, May 2003.

[18] C. Healy, R. Arnold, F. M¨uller, D. Whalley, and M. Harmon, "Bounding pipeline and instruction cache performance," *IEEE Transactions on Computers*, vol. 48, no. 1, Jan. 1999.

[19] J. Engblom, "Processor pipelines and static worst-case execution time analysis," Ph.D. dissertation, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, Apr. 2002, ISBN 91-554-5228-0.

[20] C. Ferdinand and R. Wilhelm. *"On predicting data cache behavior for real-time systems*". In *ACM SIGPLAN Workshop 1998 on Languages, Compilers, and Tools for Embedded System*, 1998.

[21] C. Ferdinand, F. Martin, and R. Wilhelm, "Applying compiler techniques to cache behavior prediction," in *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97),* 1997.

[22] Y.-T. S. Li, S. Malik, and A. Wolfe. "Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches". In *Proceedings of the IEEE Real-Time Systems Symposium,* Dec. 1996.

[23] A. Ermedahl and J. Gustafsson. *"Deriving Annotations for Tight Calculation of Execution Time". In Proceedings of Euro-Par (EUROPAR'97),* pages 1298-1307, August 1997.

[24] T. Lundqvist and P. Stenstrom. *"Integrating Path and Timing Analysis Using Instruction-Level Simulation Techniques"*. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compiler, and Tools for Embedded Systems (LCTES'98)*, Pages 1-15, June 1998.

[25] C. Ferdinand and R. Wilhelm. *"Efficient and Precise Cache Behavior Prediction for Real-Time Systems". Real-Time Systems*, 17:131-181, 1999.

[26] C. A. Healey, D. Whalley and M. Harmon. *"Integrating the Timing Analysis of Pipelining and Instruction Caching"*. In *Proceedings of 16th Rea-Time Systems Symposium (RTSS'95)*, pages 288-297, 1995.

[27] T. Lundqvist and P. Stenstrom. *"A Method to Improve the Estimated Worst-Case Performance of Data Caching"*. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99),* pages 255-262, December 1999.

[28] Xavier Vera. *Cache and Compiler Interaction-How to Analyze,* Optimize and Time Cache Behavior. PhD thesis, Malardalen University, 2003.

[29] Yau-Tsun Steven Li and Sharad Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[30] F. Wolf, R. Ernst and W. Ye. *"Associative Caches in Formal Software Timing Analysis*". Journal of Design Automation for Embedded Systems, Kluwer Academic Publishers, 7(3):271-295, 2002.

[31] Li, Y.-T. S. S. Malik and A. Wolfe. *"Efficient Micro-architecture Modeling and Path Analysis for Real-Time Software", In Proceedings of the IEEE Real-Time Systems Symposium*, 1995.

[32] Li, Y.-T. S. S. Malik and A. Wolfe. *"Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches", In Proceedings of the IEEE Real-Time Systems Symposium*, 1996.

[33] Li, Y.-T. S. S. Malik and A. Wolfe. *"Performance Estimation of Embedded Software with Instruction Cache Modeling", ACM Transactions on Design Automation of Electronic Systems,* 4(3), 1999.

[34] Li Xianfeng, *Micro-architecture Modeling for Timing Analysis of Embedded Software.* Ph.D. Thesis, National University of Singapore, 2005.

[35] C. Ferdinand and R. Wilhelm. *"Fast and Efficient Cache Behavior Prediction for Real-Time Systems", Real-Time Systems*, 17(2/3), 1999.

[36] H. Theiling, C. Ferdinand and R. Wilhelm. *"Fast and Precise WCET Prediction by Separated Cache and Path Analysis", Journal of Real Time Systems*, May 2000.

[37] P. Cousot and R. Cousot. *"Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints",* In *ACM Symposium on Principles of Programming Languages,* 1977.

[38] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PH.D. thesis, Saarlandes University, Germany, 1997.

[39] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon and C. S. Kim. *"An Accurate Worst Case Timing Analysis for RISC Processors". IEEE Transactions on Software Engineering*, 21(7):593-603, July 1995.

[40] Y. Hur, Y. H. Bae, S.-S. Lim, S.-K. Kim, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin and C. S. Kim. *"Worst Case Timing Analysis for RISC Processors:R3000/R3010 Case Study".* In *IEEE Real-Time Systems Symposium (RTSS),* 1995.

[41] J. Pierce and T. Mudge. *"Wrong-Path Instruction Prefetching",* In *ACM Intel Symposium on Micro-architectures (MICRO),* 1996.

[42] C. Pierce. *"MIPS IV Instruction Set, revision 3.1",* 1995.

[43] T. Lundqvist and P. Stenstrom. *"An Integrated Path and Timing Analysis Method Based On Cycle-Level Symbolic Execution", Journal of Real-Time Systems*, 17(2-3), 1999.

[44] A. Mok, P. Amerasinghe, M. Chen and K. Tantisirivat. *"Evaluating Tight Execution Time Bounds Of Programs By Annotations"*. In *Proceedings 6th IEEE Workshop on Real Time Operating Systems and Software,* May 1989.

[45] A. V. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, GB, 1998.

[46] G. Goosens, J. Raebaey, J. Vandewalle and H. D. Man. *"An Efficient Microcode Compiler for Application Specific DSP Processors"*. IEEE Transactions CAD/ICAS 9, 9 September, 925-937.

[47] C. Healy, R. Arnold, F. Mueller, D. Whalley and M. Harmon. "*Bounding Pipeline and Instruciton Cache Performance*". *IEEE Transactions on Computers*, 48(1), 1998.

[48] R. Arnold, F. Mueller, D. Whalley and M. Harmon. "*Bounding Worst-Case Instruciton Cache Performance*". In *Proceedings of 15th Real-Time Systems Symposium (RTSS'94)*, pages 172-181, 1994.

[49] C. A. Healey, D. Whalley and M. Harmon. *"Integrating the Timing Analysis of Pipelining and Instruction Caching"*. In *Proceedings of 16th Rea-Time Systems,)*, pages 288-297, *1995.*

[50] F. Mueller and D. B. Whalley. *"Fast Instruction Cache Analysis via Static Cache Simulation"*, In *Simulation Symposium*, 1995.

[51] F. Mueller. *Static Cache Simulation and it Applications*. PH.D. thesis, Florida State University, 1994.

[52] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley and M. G. Harmon. *"Timing Analysis for Data and Wrap-Around Fill Caches". Journal of Real-Time Systems,* Springer Netherlands, 17(2-3), 1999.

[53] A. Shaw. "*Reasoning about Time in Higher Level Language Software", IEEE Transactions on Software Engineering,* 1(2), 1989.

[54] C. Park and A. Shaw. *"Experiments with a Program Timing Tool Based On Source-Level Timing Schema", IEEE Transactions on Computers,* 24(5), 1991.

[55] C. Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PH.D. thesis, University of Washington, 1992.

[56] C. Y. Park. *"Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths"*. Real-Time Systems, 5(1):31-62, 1993.

[57] C. Healy and D. Whalley. *"Automatic Detection and Exploitation of Branch Constraints for Timing Analysis". IEEE Transactions on Software Engineering,* 28(8), 2002.

[58] F. Wolf, J. Staschulat, and R. Ernst. *"Hybrid cache analysis in running time verification of embedded software*". In *Journal of Design Automation for Embedded Systems*, 7(3):271–295, 2002.

[59] F. Stappert, A. Ermedahl and J. Engblom. *"Efficient Longest Executable*

Path Search for Programs with Complex Flows and Pipeline Effects". In *CASES*, 2001.

[60] Y.-T.S. Li and S. Malik. *"Performance Analysis of Embedded Software Using Implicit Path Enumeration Techniques"*. In *Workshop on Languages, Compilers and Tools for Real-Time Systems,* 1995.

[61] K. Chen, S. Malik and D. August. *"Retractable Static Software Timing Analysis*. In *IEEE/ACM Intl. Symposium on System Synthesis (ISSS),* 2001.

[62] T. Mitra, A. Roychoudhury and X. Li. *"Timing Analysis of Embedded Software for Speculative Processors"*. In *ACM SIGDA International Symposium on System Synthesis (ISSS),* 2002.

[63] Xianfeng Li, T. Mitra, and A. Roychoudhury. *"Accurate timing analysis by modeling caches, speculation and their interaction". 40th ACM/IEEE Design Automation Conference* (DAC), June 2003.

[64] H. Theiling and C. Ferdinand. *"Combining Abstract Interpretation and ILP for Micro-architecture Modeling and Program Path Analysis"*. In *Proceedings of the 19th IEEE Real-Time Systems Symposiums (RTSS),* 1998.

[65] X. Li, T. Mitra and A. Roychoudhury. *"Modeling Control Speculation for Timing Analysis". Journal of Real-Time Systems Symposium*, 2004.

[66] X. Li, A. Roychoudhury and T. Mitra. *"Modeling Out-of-Order Processors for Software Timing Analysis"*. In *IEEE Real-Time Systems Symposium*, 2004.

[67] M.Alt, C. Ferdinand, F. Martin and R. Wilhelm. *"Cache Behavior Prediction by Abstract Interpretation"*. In *Proceedings of Static Analysis Symposium (SAS'96)*, LNCS 1145, pages 52-66. Springer-Verlag, September 1996.

[68] S. Basumallick and K. Nielsen. *"Cache Issues in Real-Time Systems"*. In *Proceedings of ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'95)*, pages 14-24, May 1995.

[69] J. V. Busquests-Mataix, J. J. Serrano, R. Ors, P. Gil and A. Wellings. *"Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems"*. In *Proceedings of 2nd Real-Time Technology and Applications Symposium (RTAS'96)*, June 1996.

[70] J. V. Busquests-Mataix, J. J. Serrano and A. Wellings. *"Hybrid Instruction Cache Partitioning for Preemptive Real-Time Systems*. In *Proceedings of 9th Euromicro Workshop on Real-Time Systems (EUROMICRO-RTS'97)"*, June 1997.

[71] M. Carnpoy, A.P. Ivars and J. V. Busquests-Mataix. *"Static Use of Locking Caches in Multitask Preemptive Real-Time Systems"*. In *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium*, 2001.

[72] D. B. Kirk. "*SMART (Strategic Memory Allocation for Real-Time) Cache Design*. In *Proceedings of 10th Real-Time Systems Symposium (RTSS'89),*

December 1989.

[73] C. G. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee and C. S. Kim. *"Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling"*. *IEEE Transaction on Computers,* 47, 1998

[74] F. Mueller. *"Compiler Support for Software-Based Cache Partitioning"*. In *Proceedings ACM Workshop on Languages, Compilers and Tools for Real-Time Systems (LCTES'95),* June 1995.

[75] I. Puaut and D. Decotigny. *"Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems"*. In *Proceedings of 23th Real-Time Systems Symposium (RTSS'02)*, December 2002.

[76] T. Lundqvist and P. Stenström, *"A method to improve the estimated worst-case performance of data caching",* In *Intl Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 255–262, 1999.

[77] C. Ferdinand and R. Wilhelm. *"On predicting data cache behavior for real-time systems*". In *ACM SIGPLAN Workshop 1998 on Languages, Compilers, and Tools for Embedded System*, 1998.

[78] C. Ferdinand and R. Wilhelm. *"On predicting data cache behavior for real-time systems*". In *ACM SIGPLAN Workshop 1998 on Languages, Compilers, and Tools for Embedded System*, 1998.

[79] S. Ghosh, M. Martonosi, and S. Malik. *"Cache miss equations: a compiler framework for analyzing and tuning memory behavior"*. In *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[80] H. Ramaprasad and F. Mueller. *"Bounding worst-case data cache behavior by analytically deriving cache reference patterns"*. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 148–157, 2005.

[81] X. Vera, B. Lisper, and J. Xue. "*Data caches in multitasking hard real-time systems"*. In *IEEE Real-Time Systems Symposium*, 2003.

[82] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Par and C. S. Kim. *"An Accurate Worst Case Timing Analysis Technique for RISC Processors"*. In *Proceedings of 15th Real-Time Systems Symposium (RTSS'94),* pages 97-108, 1994.

[83] S. K. Kim, S. L. Min and R. Ha. *"Efficient Worst Case Timing Analysis of Data Caching"*. In *Proceedings of IEEE Real-Time Technology and Applications Symposium (RTAS'96),* 1996.

[84] R. T. White, F. Mueller, C. Healy, D. Whalley and M. Harmon. *"Timing Analysis for Data Caches and Set-Associative Caches"*. In *Proceedings of Third IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192-202, 1997.

[85] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. *"Timing analysis for data and wrap-around fill caches"*. *Real-Time Systems*, 17(2-3):209–233, 1999.

[86] T. Mitra, A. Roychoudhury and X. Li. *"Timing Analysis of Embedded Software for Speculative Processors"*. In *ACM SIGDA International Symposium on System Synthesis (ISSS),* 2002.

[87] J. Liedtke, H. H¨artig, and M. Hohmuth. *"Os-controlled cache predictability for real-time systems"*. In *IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 9-11 1997.

[88] B. Lisper and X. Vera. *"Data cache locking for higher program predictability"*. In *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, Mar. 06 2003.

[89] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. *"Static use of locking caches in multitask preemptive real-time systems"*. In *IEEE Real-Time Embedded System Workshop*, December 2001.

[90] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig. "Fast, predictable and low energy memory references through architecture-aware compilation". In *Proceedings of the 2004 Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair* (Yokohama, Japan, January 27 - 30, 2004).

[91] M. E. Wolf and M. S. Lam. *"A Data Locality Optimizing Algorithm'*. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30-44, 1991.

[92] S. Malik and Y.-T. S. Li. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[93] SimpleScalar LLC **http://www.simplescalar.com/**

[94] Todd Austin, Eric Larson, Dan Ernst, *"SimpleScalar: An Infrastructure for Computer System Modeling,"* Computer, 35(2), pages 59-67, Feb., 2002

[95] Xavier Vera etc. *"A Fast and Accurate Approach to Analyze Cache Memory Behavior". ACM Transactions on Programming Languages and Systems*, Vol.26, No.2, March 2004, Pages 263-300.

[96] Coyote Framework, N. Bermudo and X. Vera.
http://www.mrtc.mdh.se/projects/wcet/Coyote

[97] Kristof Beyls. Erik H. D'Hollander. Ghent. *"Reuse Distance as a Metric for Cache Behavior". In Proceedings of PDCS'01*, pages 617-662, Aug 2001.

[98] P. Paulin, C. Liem, T. May and S. Sutarwala. *"Flexware: A Flexible Firmware Development Environment for Embedded Systems"*. In *P. Marwedel and G. Goosens: Code Generation for Embedded Processors*, Kluwer Academic, 65-84, 1995.

[99] Yanhui LI, Shakith Devinda Fernando, Heng Yu, Xiaolei Chen, Yajun Ha, Teng Tiow Tay. *"Tighter WCET Analysis of Input-dependent Programs with Classified Cached Memory Architecture". 15th IEEE International Conference on Electrics, Circuits, and Systems (ICECS)*, Malta, 2008.

[100]Michael R. Garey, David S. Johnson. Computers and Intractability—A guide to the theory of NP-Completeness, W.H. Freeman and company, 1979.

[101]Jyh-Charn Liu, Hung-Ju Lee. "Deterministic upper bounds of the worst-case execution times of cached programs". In Real-Time Systems Symposium, 1994.

[102] E. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holts, Rinehart and Winston, 1976.

[103] J. Staschulat and R. Ernst. *"Multiple Process Execution in Cache Related Preemption Delay Analysis"*. In *ACM International Workshop on Embedded Software (EMSOFT),* pages 278–286, Italy, 2004.

[104]G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang and A. Wang.*" Challenges in Code Generation for Embedded Systems"*. In *P. Marwedel and G. Goosens: Code Generation for Embedded Processors,* Kluwer Academic, 48-64, 1995.

[105]S. Liao, S. Devadas, K. Keutzer, S. Tjiang and A.WANG. *"Storage Assignment to Decrease code size"*. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation,* 186–195, June 1995.

[106]A. Sudarsanam and S. Malik. *"Memory bank and register allocation in software synthesis for ASIPS".* In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design,* 388–392, November 1995.

[107]C. Liem, T. May and P. Paulin. *"Instruction-set matching and selection for DSP and ASIP code generation"*. In *Proceedings of the European Design and Test Conference 31-37,* March *1994.*

[108]D. Lanner, J. V. PRAET, A. KIFLI, K. SCHOOFS, W. GEURTS, F. THOEN and G. GOOSENS. *"Chess: Retargetable code generation for embedded DSP processors"*. In *P. Marwedel and G. Goosens: Code Generation for Embedded Processors*, Kluwer Academic, 65–84. 1995.

[109]W. SCHENK. "*Retargetable code generation for parallel, pipelined processor structures"*. In *P. Marwedel and G. Goosens: Code Generation for Embedded Processors*, Kluwer Academic, 119–135, 1995.