PRECISE YET SCALABLE RESOURCE ANALYSIS VIA SYMBOLIC EXECUTION

RASOOL MAGHAREH

(B.Sc.(Hons.) Shiraz University)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2017

Declaration

Rasool Maghareh

Friday 10th February, 2017

Dedicated to my grandmother Mina (1941 - 2013)

Sadly missed, dearly remembered, forever loved.

Acknowledgments

In the name of Allah, God, the Creator and Nurturer. I can never express all my gratitude and appreciations for all his favors. I am sincerely thankful for his each and every blessings in the past few years.

I would like to thank my wife Narjes for her love and support. Without her support, I would not be able to write this thesis. I also like to thank my parents and my wife's parents Mohammad, Zohreh, Ali and Farahnaz and all my family members including our grandparents Morteza, Maryam, Fatemeh, Zahra, my siblings and their families Hossein, Sadegh, Ghazaleh, my brother and sister in laws and their families Motahareh, Ali, Mohammad, Hengameh, Ahmad and Bita . Thank you all for your love, support, pray and bearing with our absence during the course of my PhD studies. You are the true reason I am here today. In the last year of my PhD my wife and I were blessed to have our son Mahdi. The love and joy he brought to our life was a great motivation towards the end of my PhD journey.

I would like to extend my sincere gratitude to my thesis advisor, Professor Joxan Jaffar for his guidance and professional insights throughout my PhD. It was only by his support that I embarked on this PhD journey. Thank you Prof. Jaffar for your valuable advices that helped me to be an independent researcher. Besides, I truly appreciate Dr. Duc-Hiep Chu, for all technical help and advices during these years. It was a pleasure to work with him. This thesis would never have come together without Prof. Jaffar and Dr. Chu's continuous guidance and support. I am very much grateful to my lab mates and my friends from School of

Computing at NUS, for all the wonderful time that we had with each other.

Finally, I would like to thank National University of Singapore and Agency for Science, Technology and Research (A*STAR) and specifically, Singapore International Graduate Award (SINGA) program and Lee Foundation for providing the financial support for this research.

I also feel a deep appreciation for my friends in Singapore, who have made experience of living in Singapore and studying in NUS a unique and unforgettable one: Hossein Nejati, Shima Bayat, Saeid Montazeri, Narges Nourieh, Mohammadreza Chamanbaz, Faezeh Emami, Mohammad Sadegh Tavallali, Mehdy Ghaeminia, Sogand Zarei, Safoura Sameni, Ahmadreza Shehabinia, Siavash Sakhavi, Sajjad Seifozzakerini, Amin Kianinejad, Mohammad Akbari, Reihaneh Eslami, Ahmadreza Pourghaderi, Fatemeh Jamshidian, Ali Khalili, Mona Khalighi, Seyyed Mohammad Majedi, Samaneh Tavakolinia, Hamed Mirabolghasemi, Fatemeh Safari, Hamed Sepehr, Zahra Barikbin, Saeid Arabnejad, Mohammad Nouri, Amin Torabi Jahromi, Alireza Partovi, Asad Norouzi, Fahimeh Masoumian, Alireza Rezvanpour, Shima Najafi Nobar, Ehsan Rismani, Khatereh Hajizadeh, Davood Afshari, Fatemeh Movahednia, Meghdad Attatzadeh, Elham Jahanshahi and other Iranian friends in Singapore as well as Nadim, Fatemeh, Mohamed, Simi, Rahim, Aeylia and Yati all my dear friends in AYC.

Contents

Summary

The expanding complexity of embedded systems has emerged the need of techniques to analyze different non-functional properties of programs such as performance, resource consumption or security. Three of the important non-functional properties for embedded systems are *execution time*, *consumed memory* and *consumed energy.*

Both dynamic (measurement-based) and static resource analysis methods can be employed to estimate such non-functional properties of programs. While dynamic methods are unable to generate safe upper bounds for these properties, static methods used for resource analysis, have been proven to generate safe upper bounds on the highest resource consumption for a program. In general, due to scalability reasons, resource analysis is usually performed in two separate high-level and low-level phases. Such approaches are named as modular approaches. Contrary to these modular approaches, integrated methods perform resource analysis in a single phase. To the best of our knowledge, integrated methods were not scalable up to now.

In this thesis, we present a novel integrated framework for resource analysis where micro-architectural modeling and systematic path-sensitivity, are synergized. This would give us a very high precision for resource analysis, but at the same time, it is a huge challenge for scalability. Our contribution is then a dynamic programming algorithm with a powerful concept of *reuse.* Reuse, in turn, depends on the core concepts of *interpolation* and *dominance.* While interpolation-based methods have been used in program verification for the purpose of pruning the search space of symbolic execution, our setting is novel not just because we are performing analysis instead of verification, but because our interpolation with dominance covers reuse under an environment where resource consumption of program paths is *dynamic* and/or *symbolic*.

We should highlight that since we are systematically path-sensitive, our algorithm is more precise. The important point, however, is that it also can scale to a reasonable level. Our realistic benchmarks will show both aspects: that systematic path-sensitivity, in fact, brings significant accuracy gains; and, likewise, the algorithm scales well.

The thesis makes several contributions in the area of *worst-case resource consumption analysis*. First, an integrated analysis framework is presented in Chapter 2. Next, the analysis framework is customized for the following three analyses:

1. **Worst-case Execution Time (WCET) Analysis:** The analysis of the execution time of an embedded system becomes necessary in hard real-time systems. Worst-case execution time can be used to ensure the temporal correctness of hard real-time systems. However, the worst-case input of a program is not known in general and is hard to derive. As a result, an upper bound on the worst-case execution time is generated by utilizing WCET analysis.

   Our worst-case resource analysis framework is customized for WCET analysis in Chapter 3. The main contribution of the analysis presented in Chapter 3 is performing a precise integrated WCET analysis *down to instruction and data caches*. The key challenge, scalability, is obtained by using a notion of reuse in an environment where the contribution of each basic block to the overall WCET is *dynamic*. In realistic benchmarks, it is shown that the extreme attempt at precision, in fact, pays off because there is a significant increase in precision, and this is obtained in a reasonable time.

2. **Memory High-watermark (MHW) Analysis:** In Chapter 4, our analysis framework is customized for symbolic MHW analysis. Memory high-watermark refers to the highest amount of memory that a program can acquire in its executions. It can be compared to WCET, except that WCET represents the longest execution time of a program in all its executions, while MHW represents the highest amount of memory usage of a program (along

all its possible executions). We will explain how MHW analysis is needed to ensure the reliability of safety-critical embedded systems.

The main contributions of the Memory high-watermark analysis are (1) it is performed on a non-cumulative resource analysis where the resource of interest can be consumed or released and (2) the concept of reuse with interpolation and dominance is performed in the presence of *symbolic bounds.*

3. **Worst-case Energy Consumption (WCEC) Analysis:** WCEC analysis has gained interest in the recent years and it is important for embedded systems with limited access to energy. The aim of this analysis is to ensure that a hard-real time system would have access to enough energy to perform its assigned tasks. Due to the complex micro-architectural features in modern systems, the path resulting in the worst-case execution time can be different from the worst-case energy consuming path.

   In Chapter 5, we customize our analysis framework for WCEC analysis in the presence of cache and in-order pipelines. Energy is data dependent, and our integrated approach can generate more precise bounds compared to other analyses. In this chapter, we extend reuse to be performed in the presence of *in-order pipeline.* Finally, Chapter 5 finishes by presenting an extension of our WCEC analysis framework to out-of-order pipeline.

List of Tables

List of Figures

List of Abbreviations

ACS	Abstract Cache State
AI	Abstract interpretation
ALU	Arithmetic Logic Unit
CFG	Control flow graph
CM	Commit
D-cache	Data cache
ELB	Estimating Lower-bound
EUB	Estimating Upper-bound
EX	Instruction Execute
FIFO	First In First Out
FPU	Floating-point Unit
I-cache	Instruction cache
ID	Instruction Decode & Dispatch
IF	Instruction Fetch
ILP	Integer linear programming

IPET	Implicit path enumeration technique
LRU	Least Recently Used
LD/ST	Load/Store Unit
LLVM IR	LLVM Intermediate representation
MULTU	Multiplication Unit
MHW	Memory High-watermark
MRU	Most Recently Used
PLRU	Pseudo-LRU
RCSP	Resource-constrained Shortest Path
RISC	Reduced Instruction Set Computing
SSA	Static Single Assignment
VIVU	Virtual interpretation of virtual unrolled (loops)
WB	Write Back
WCEC	Worst-case Energy Consumption
WCET	Worst-case Execution Time
WCHA	Worst-case Heap Analysis
WCSA	Worst-case Stack Analysis

List of Symbols

A	Cache Associativity
C	Cache Capacity
$clock_{path}$	Clock Energy
δ	Dominating Condition
Δ_p	Abstract Transformer
Δ_m	Machine State Summary
Δ_s	Cache-set Abstract Transformer
E	Energy
$Energy_{reg}$	Energy of Registers
$Energy_{wk}$	Wakeup Logic Energy
$Energy_{FPU}$	Energy of FPU
$Energy_{MULTU}$	Energy of MULTU
$Energy_{ALU1}$	Energy of ALU1
$Energy_{ALU2}$	Energy of ALU2
$Energy_{ic}$	Energy of Instruction Cache

$Energy_{dc}$	Energy of Data Cache
FO	Set of First-order Logic Formulas
Γ	Witness
Γ_h	High-watermark Witness
Γ_n	Net-usage Witness
ins	Instruction
$instr$	Sequence of Instructions
$\overline{\Psi}$	Interpolant
ℓ_0	Initial Program Point
$leakage_{path}$	Leakage Energy
m	Machine State
m_0	Initial Machine State
mem	Memory Access
$\mathcal{M}$	Set of Pairs $\langle mem, i\rangle$
$\mathcal{L}$	Set of Program Points
Ops	Set of Operations
P	Power
$\mathcal{P}$	Transition System
π	Path Constraints Along the Witness
seq	Sequence of Memory Block Accesses
s	Symbolic State
s_0	Initial Symbolic State
$SymStates$	Set of Symbolic States
$selection_{path}$	Selection Logic Energy
$Switchoff$	Sum of the Switch-off Energy

σ	Symbolic Store
tr	Transition
t	Execution Time of Instructions
Υ	Sequence of All Memory Accesses Along a Path
Υ_h	Sequence of $alc(\pm, e)$ Depicting All Memory Allocations or Deallocations
$Vars$	All Program Variables

Chapter 1

Introduction

An *embedded system* is a system which performs a specific task and is composed of software and hardware or mechanical parts. Embedded systems have been utilized to accomplish *safety-critical tasks* in different domains, e.g., in medical devices, aircraft flight control systems or automobiles. The failure of such embedded systems may result in massive financial loss or even loss of lives. An example of a safety-critical embedded system is a heart pacemaker. Similarly, modern air crafts and automobiles rely on a network of embedded systems which some might be performing critical tasks.

Beside the functional safety of embedded systems, reasoning over the *non-functional properties* of embedded systems has been an interesting research topic for many years. Non-functional properties are referred to properties that do not influence the input-output behavior of embedded systems directly. The *execution time*, *consumed memory* and *consumed energy* are examples of important non-functional properties of embedded systems.

Analyzing non-functional properties has also become part of the safety criteria for certain groups of embedded systems. In these embedded systems, the worst-case usage of a non-functional property is estimated to make sure that the embedded system will function properly in extreme cases.

1.1 Estimating Worst-case Resource Consumption

There are two classes of methods to obtain an approximation on the resource consumption of a program:

- *Dynamic (measurement-based) methods* which use real executions of the program or parts of the program to estimate the consumed resource from the executions or the combination of the executions of the program parts.
- *Static methods* which use the program itself plus extra information such as loop bounds and an abstract model of the corresponding hardware to approximate the highest consumption of a resource.

1.1.1 Dynamic (Measurement-based) Methods

Dynamic methods for estimating resource consumption are frequently used in industry. For example, for measuring the worst-case execution time (WCET), the easiest method is to measure the execution time of a subset of the real program executions and report the longest seen execution time. Figure 1.1 illustrates the distribution of all possible resource costs of a sample program. As it can be seen in Figure 1.1 the reported result would be *the maximal observed resource consumption*, which can be different from the actual WCET.

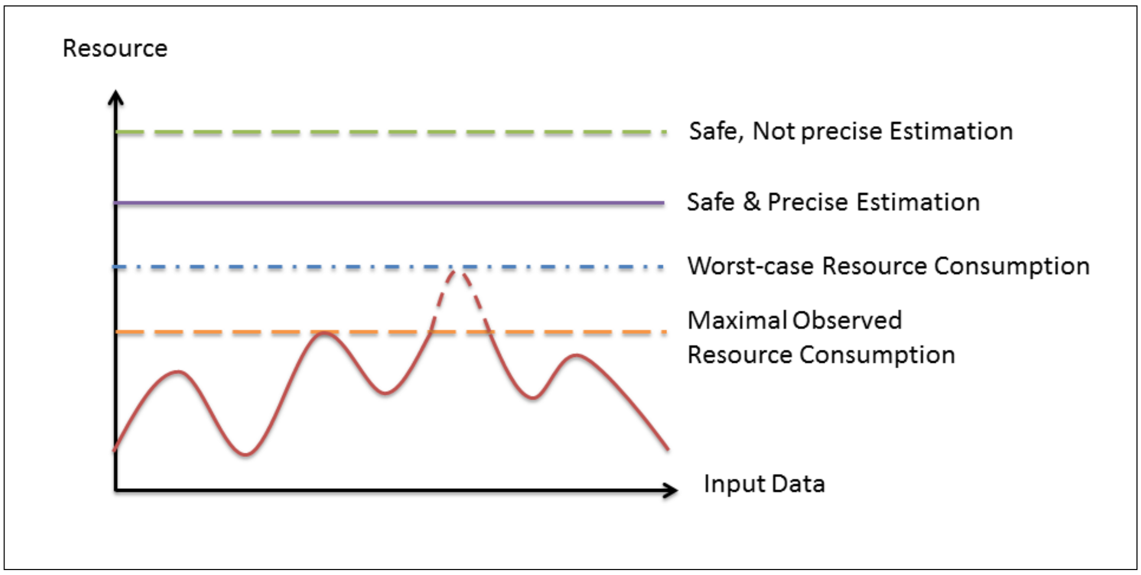


Figure 1.1: The distribution of possible execution times of a program

Another traditional method which is still being used in industry [Wil06], measures the execution time of small snippets of code plus a safety margin. The results are combined based on the control flow graph. This method helps to extend the subset of real executions measured in the program since each code snippet is measured once and used several times. The correctness of this method follows the correctness of the *combine* step. However, [The04] shows that the assumptions on the correctness of combine step can be wrong and still the measured WCET would be the maximal observed execution time (dashed line with orange color in Figure 1.1).

Dynamic methods can be used to estimate the memory usage of an embedded system too. However, similar to WCET, dynamic methods can not generate a safe upper-bound on the resource usage. In fact, the bound returned by measurement methods is the peak memory usage on a concrete execution of the software. There are many commercial and open-source memory profiling tools where some of them such as Valgrind Massif [Val] and Perfrewrite [Kru14] measure the peak memory.

Last but not least, due to the complicated nature of energy consumption analysis many methods have been developed for generating an estimation of the energy consumption of embedded systems [T^+94, Q^+00] and mobile applications [B^+14, A^+12b, M^+13].

In general, the main disadvantage of dynamic methods is that they cannot generate safe upper-bounds over the worst-case resource usage of embedded systems.

1.1.2 Static Methods

The estimation of the worst-case usage of a non-functional property is also performed using *program analysis* techniques. In general, these techniques are categorized under static methods for worst-case resource analysis. Static methods provide estimations over the resource consumption by examining the program itself with some extra information (such as the loop bounds) and without executing it directly on the hardware. Static methods can generate safe upper-bounds over the worst-case resource consumption of embedded systems.

As illustrated in Figure 1.1, the highest resource cost that a program will take to execute is referred as the *worst case resource consumption* of a program. In order to find this bound for a program the worst-case input for the program is needed. The worst-case input is not known in general and due to the large state space of all possible executions of a program, the worst-case input is difficult to derive. Therefore, an approximation on the worst-case resource consumption is returned as the upper timing bound (line with purple color in Figure 1.1).

Generally, the withdrawal of static methods is that these methods need specific models of processor behavior and micro-architectural features which enhance the performance of a processor (e.g., caches or pipelines). Additionally, it is possible that static methods generate imprecise results due to the inaccuracy of the models of the hardware or the overestimations (dashed line with green color in Figure 1.1). Static analysis methods analyze either the source code or the disassembled binary executable to determine the structure of a program.

1.2 Modular Worst-case Resource Consumption Analysis

Traditionally, modular approaches to static analysis are used for worst-case resource consumption analysis. These approaches are performed in two phases [W$^+$08]:

- *Micro-architectural Modeling Phase:* Also referred to as low-level analysis, often involves micro-architectural modeling to determine the maximum resource consumption accurately for each of the basic blocks. In this phase, the effects of performance enhancing processor features such as pipeline and caches on the consumed resource are considered.

- *Path Analysis Phase:* Also referred to as high-level analysis, this phase concerns a program level path analysis to determine loop bounds and infeasible paths in the program's control flow graph (CFG), and moreover it generates an upper-bound over the worst-case estimation.

For example, in the case of WCET analysis [LM95], the low-level analysis generates a timing for each basic block (using the timing information of the real hardware with all its specific features). By providing the information from the low-level analysis to the high-level analysis, the high-level analysis generates an upper-bound over the WCET of a given program and hardware platform.

Low-level analysis and high-level analyses are still performed separately for the reason of scalability. Throughout this thesis, we will call such approaches to static analysis as *modular approaches*.

Micro-architectural modeling has been an active research topic in resource analysis. Initial works on instruction cache modeling used integer linear programming (ILP) [LMW99]. However, the work did not scale due to a huge number of generated ILP constraints. Subsequently, the abstract interpretation (AI) [CC77] framework for micro-architectural modeling, proposed in [TFW00], made an important step towards scalability. The solution has been proved scalable, and it has also been applied in commercial WCET tools (e.g., [aiT]). For most existing WCET analyzers, AI framework has emerged to be the basic approach used for micro-architectural modeling.

On the other hand, three classes of methods are suggested for high-level analysis in modular approaches:

- **Structure-based Methods:** In structure-based methods, as used in [CP00], an upper-bound is calculated in a bottom-up traversal of the syntax tree of the program which combines the bounds computed for the structures according to the combination rules [W$^+$08].
- **Implicit Path Enumeration (IPET):** Implicit path enumeration (IPET) represents the control flow of the program in a set of linear equations and presents the resource consumption formulation in the maximization of an objective function of an ILP. The ILP is passed to an ILP solver, and the result is an upper-bound on the resource consumption of the program.

- **Path-based Methods**: Path-based techniques try to find the amount of consumed resource of a program by measuring the upper-bound for the different paths in the program.

1.2.1 Imprecision in Modular Approaches

The AI framework for micro-architectural modeling + IPET [TFW00] has been used to estimate the WCET of programs. This method combines the sets of possible states at the merging points in the control flow graph. In order to manage the potentially exponential number of states merging is inevitable. However, this may lead to significant overestimation, since some associated information will be lost during the merge step. The situation becomes worse when it is noticed that in some cases feasible paths are merged with infeasible paths which result in unnecessary loss of information. However, the analysis is still guaranteed to be safe; all possible states are considered and the result will never be smaller than the maximum consumed resource, but the results are often not precise. There are two main causes of imprecision in AI framework + IPET:

1. The analysis of the estimations of memory accesses will lose precision through joins at the control flow merge points. This results in subsequently overestimating in the micro-architectural modeling.

2. Beyond the one-iteration virtual unrolling of loops [TFW00], AI is unable to give different resource cost for a basic block executed in different iterations of a loop.

Much research has been conducted on finding infeasible paths. By detecting infeasible paths and avoiding merging them with feasible paths the effect of scenario

(1) is reduced. Some of the methods, [L+10], apply the infeasible path detection only to the *path analysis* phase and still use the AI framework for the *micro-architectural modeling*. The result in these cases will not help in decreasing the effect of scenario (1). Besides, not all techniques for finding infeasible paths are applicable to IPET-based resource analysis.

Recent approaches such as [CR11, BCR13] have made improvements on the AI framework with verification technology. These researches have demonstrated that the accuracy of WCET estimates (time as the resource of interest) can be improved by integrating some forms of *infeasible path discovery* into micro-architectural modeling. We note, however, due to the scalability issue, such additions of path-sensitivity is quite limited. Moreover, although these works show promising results as an AI framework which reduces certain overestimation resulted from scenario (1), they are unable to address systematically the imprecision caused by scenario (2). In other words, these works, while possessing some flavor of path-sensitivity, essentially still employ a fixed point computation. Therefore, the works give a worst-case resource consumption for each basic block, even though the resource consumption of a basic block in different iterations of a loop can diverge significantly. We will elaborate more on these methods in Chapter 3.

Path analysis has also been well investigated by the resource analysis research community (e.g., with focus on WCET analysis [LM95, EG97, G+05, GEL06, CJ11, CJ13]). These works aim to improve the estimate of the consumed resources at the program path level.

The most recent related works are [CJ11, CJ13]. The fully automated symbolic simulation algorithm presented in [CJ11] performs *loop unrolling* and utilizes the notion of *interpolation and reuse* to scale. We will elaborate more on these works while presenting the foundations of our resource analysis framework in Chapter 2.

1.3 Integrated Worst-case Resource Analysis

In the previous section, we reviewed the main causes of imprecision in modular approaches. Though these methods are safe, but the amount of overestimation in the analysis can be quite high. In general, the two most important scales to evaluate a resource analysis is *safety*, meaning the approximation is equal or more than the real worst-case consumed resource and *precision*, meaning if the measured bound is close to the exact worst-case consumed resource. Any resource analysis which maintains the safety of the generated resource consumption bound is called a *sound resource analysis*.

The abstract hardware model used in the low-level analysis can affect both the safety and the precision of the estimated resource consumption through the effects of the hardware components such as cache, pipeline, branch prediction, etc.

On the other hand, the existence of infeasible paths in the high-level analysis of modular approaches, though do not affect the safety of the estimated resource consumption bound, but affect the precision of the estimation. A more precise estimate is generated by ignoring the infeasible paths in the resource analysis. Moreover, the high-level analysis finds upper bounds on the number of the iterations of the loops which is a requirement for precise estimation of the resource consumption.

In contrast to modular approaches, *integrated methods* perform both the micro-architectural modeling and path analysis in one phase. An *integrated path and resource analysis* would theoretically give us ideal precision for a precise resource analysis. The first attempt to perform integrated resource analysis based on symbolic execution was Lundqvist et al. [LS99] which did not scale to realistic programs.

1.4 Thesis Contributions

In this thesis, we investigate on a scalable integrated resource analysis which integrates micro-architectural modeling with program path analysis. The objective of this thesis is to address the precision issue in resource analysis. We propose a *symbolic execution* framework where micro-architectural modeling and systematic path-sensitivity are synergized. The essence of our proposed method is that it is *fully path-sensitive modulo summarization* which leads to a more precise inspection of the consumed resource measured from the underling hardware model.

Our framework can be compared to the state-of-the-art methods based on the number of paths being disregarded due to infeasibility. The current state-of-the-art methods disregard *none of the infeasible paths* [TFW00] or disregard the infeasible paths where the reason for infeasibility *falls inside a constant number of merge points* [CR11, BCR13], while our method disregards *the maximum possible number of the infeasible paths*[1].

In our framework, loops are unrolled fully[2] and summarized (inherited from [CJ11]). As a result, our framework can detect the exact number of iterations in complicated loop patterns such as non-rectangular loops and amortized loops [GZ10].

Moreover, our framework generates an exact incoming context for each loop

[1]Limited by the only merging performed at the end of the loops and the power of the theorem solver

[2]We should note that the loop unrolling in our algorithm is done virtually and not physically, and is different from the loop unrolling in compilers.

iteration. This will enable our framework to detect the infeasible paths that were not detected due to loss of information in the fixed point computation or the infeasible paths which are infeasible in only one loop iteration. The only abstraction performed in our framework is within a loop iteration, and not across loop iterations. This leads to a precise inspection of the consumed resource by the underlying hardware model (because the micro-architectural state can be tracked across the iterations).

Based on this, we emphasize that the significance of our resource analysis framework (presented in Chapter 2) is that it exhibits the highest level of path sensitivity, so in theory produces the most precise estimates. Moreover, integrated methods have not been scalable up to now [W$^+$08] and to the best of our knowledge, our method is the *first scalable integrated resource analysis.* The following are the advantages of our framework:

- **Advantage1:** Our framework possesses maximum infeasible path detection. This results in a decrease of the overestimations in the resource analysis.
- **Advantage2:** Our framework differentiates between the incoming context of different loop iterations. As a result, the state of the micro-architecture features can be precisely tracked during the analysis, which in turn results in a more precise estimation of the analyzed resources.
- **Advantage3:** Our framework can dynamically perform complete loop unrolling and aggregate the results from each loop iteration separately. It will be able to calculate the exact number of loop iterations (inherited from [CJ11]).

We have customized our resource analysis framework for three different analyses:

1. **Worst-case Execution Time Analysis (WCET):** The analysis of the worst-case execution time in embedded systems, and especially for real-time embedded systems, has been receiving a lot of attention by researchers. Real-time embedded systems require completing their work and delivering their services on a timely basis. These systems are becoming increasingly important in everyday life. The traffic light regulator or the airport aviation monitoring system are examples of real-time embedded systems.

2. **Memory High-watermark Analysis (MHW):** Memory has also been an important resource in embedded systems. Memory consumption behavior is an important feature in the design of embedded systems using cost-efficient hardware.

3. **Worst-case Energy Consumption Analysis (WCEC):** Energy has received increasing attention in the last few years. The impact of the embedded systems and their energy consumption behavior on the environment is becoming increasingly important in recent years. Furthermore, the consumed energy in embedded systems has been a major barrier for devices which have limited access to energy.

1.5 Thesis Organization

We will present our integrated resource analysis framework in Chapter 2. Next, in Chapters 3-5, we will customize our resource framework for WCET, MHW and WCEC analysis. In this section, we will separately review each of these customized analyses. Finally, this thesis finishes in Chapter 6 with a discussion on conclusion and future works.

1.5.1 Worst-case Execution Time Analysis with Cache

The analysis of the execution time in real-time embedded systems has been a major research topic in embedded systems. Real-time systems can be classified as *hard real-time systems* or *soft real-time systems*. By definition, hard real-time systems must satisfy all deadline constraints, and a failure in satisfying any of the deadline constraints may result in a disaster. For example, the effectiveness of a car airbag system, as an example of a hard real-time system, highly depends on the timing of its ignition. In a typical frontal collision at 50 km/h against a wall, the gas generator must be ignited approximately 10 ms to 30 ms after the crash has begun. This is worse in the case of a side collision when the gas generator must be ignited only 5 ms after the crash begins [FV98].

A hard real-time system is deemed as failed unless all of its deadline constraints are satisfied. Accordingly, the correctness of a hard real-time system depends both on its logical correctness (i.e., the program implements exactly the things it is supposed to do), and its temporal correctness (i.e., all deadlines are met). In contrast, *soft real-time systems* do not require to satisfy deadline constraints strictly, but it is desirable to do so [Kop11].

In order to evaluate the temporal correctness of hard real-time systems, upper-bounds on their execution times is needed. The upper-bounds are then used to demonstrate that the execution of the hard real-time systems will finish before the deadlines. Note that, in general, it is not possible to calculate upper-bounds on the execution time of programs; otherwise, we could solve the halting problem [Sch94]. However, some restrictions, such as allowing only explicitly bounded recursion and loops, are imposed on real-time systems, which guarantees the termination of real-time programs.

Chapter 3 presents a precise WCET analysis based on our resource analysis framework. The scope of the research presented in Chapter 3 is analyzing both the instruction and data cache for WCET analysis.

1.5.2 Memory High-watermark Analysis with Symbolic Bounds

Traditionally, in safety-critical embedded systems, it was recommended not to use dynamic memory allocation because of two main reasons: (a) the allocation instructions might take longer than expected, resulting in the failure of temporal constraints in hard real-time systems; and (b) the memory fragmentation issue. As a result, the stack was the only memory that grows dynamically during execution. Worst-case stack usage was estimated by methods such as the one proposed in [KF14]; the estimate is compared with the available memory to ensure stack overflow errors does not occur.

In the past few years, there have been advances in both hardware and software of embedded systems. The drop in the hardware cost, the development of customized operating systems for embedded systems, and finally the advent of constant time memory allocation algorithms with a reasonable handling of memory fragmentation [MRC03, M+08] are among these advances. Besides these, as the embedded systems become more complex, the need to use third-party code – which might require dynamic memory allocation – becomes more inevitable. As a result, dynamic memory allocation has now been used more frequently in embedded software [A+15].

Such increased use of dynamic memory allocation will raise concerns about the reliability of embedded systems that are deployed for safety-critical tasks. Thus, there is a real need for developing *program analysis* methods to avoid both stack and heap overflows in safety-critical systems. Besides, the estimate produced by such analyzers would be useful in the design process of embedded systems to reduce hardware cost [T^+13]; it can also be presented to the programmers who are interested in dissecting the memory footprint of an embedded system.

Memory is a *non-cumulative resource*: what is acquired can later be released. As a result, unlike time and energy where the maximum consumption of an execution path is at the end of the path, the maximum memory usage of a path can be at any place in that path, e.g., right in the middle of it. Thus, many approaches developed for worst-case analysis of cumulative resources, such as WCET analysis, becomes inapplicable. More specifically, these methods often abstract away the orders between the acquires/releases, which is crucial for precise analysis of non-cumulative resource.

Memory high-watermark (MHW), refers to the highest amount of memory that a program might require in any execution. It can be compared to the WCET, except that WCET represents the longest execution time of a program in all its executions, while MHW represents the highest amount of memory usage of a program, among all its executions. Chapter 4, presents a precise MHW analysis as another application of worst-case resource analysis of embedded systems.

1.5.3 Worst-case Energy Consumption Analysis with Cache and Pipeline

In some of the embedded systems, energy is a scarce resource. Energy consumption analysis is becoming an important trend in different programming paradigms such as embedded systems and mobile applications.

One first approach in the literature has measured an average energy usage of a program. The measured value can be reported as an indicator of the average energy usage of software on a specific hardware to the customers [GGP+14].

On the other hand, a second approach in the literature has been insisting that energy consumption analysis is an important factor for energy harvesting embedded systems. The lack of access to infinite energy source can be either because the energy source of these embedded systems is not always available and cannot grantee an infinite operation (e.g., solar powered sensor nodes) or such embedded systems do not have access to rechargeable energy resources after being deployed (e.g., underwater sensor nodes). This analysis is indeed important for embedded systems which have scarce access to energy. The researchers in this group suggest that an energy consumption analysis should be performed which can guarantee that an embedded system would have access to enough energy to perform the set of tasks assigned to it [B+05, V+14, PS01].

Similar to WCET, the path resulting in the worst-case energy consumption (WCEC) of an embedded system is not known in general, and a WCEC analysis should be performed to find an upper-bound on the WCEC of an embedded system [J+06, W+15]. Note that, due to the complex micro-architectural features in modern systems, the path resulting in the worst-case execution time is not the same as the worst-case energy consuming paths. So, the path returned by the WCET analysis cannot be used directly to generate the WCEC of a program [J+06]. In Chapter 5, a framework for a precise WCEC analysis is presented. The resource analysis method presented in Chapter 5 extends our framework with resource analyses in the presence of superscalar in-order processors. Moreover, a discussion on extension of the resource analysis framework to analyzing out-of-order processors is presented in the end of Chapter 5.

Chapter 2

Integrated Resource Analysis Framework

In this chapter, we present our *symbolic execution* resource analysis framework. Symbolic execution has been quite a well-established method in program verification. However, it suffers from the path explosion problem which makes it undesirable for program analysis. In this chapter, we will present a modified version of symbolic execution empowered with notions of *reuse* and *domination* which can be used to perform precise and scalable resource analysis.

2.1 Symbolic Execution

Symbolic execution [C$^+$76, Kin76] is one of the methods used for reasoning about the programs. In symbolic execution, symbolic values are used as inputs instead of actual data. The values of the program variables are stored as symbolic expressions of the input symbolic values. Symbolic execution was first developed for program testing, but it has been subsequently used for program verification and program analysis.

The main challenges of performing symbolic execution for program verification is (1) exponential number of paths and (2) infinite-length paths in the presence of unbounded loops. In the scope of this thesis, we would assume programs containing bounded loops. This is a valid assumption for programs used in safety critical embedded systems. However, the first challenge still remains for utilizing symbolic execution.

2.1.1 Symbolic Execution for Program Verification

In the recent years symbolic execution has emerged as a successful method for program verification. The main advantages of using symbolic execution for program verification is that it avoids exploring infeasible paths[1], hence, program verification using symbolic execution is more accurate and does not result in false alarms. More specifically, the paths in the symbolic execution tree are enumerated and verified one by one. In case a path is found which contains a bug, the verification of the program has failed and the path is reported to the user. Otherwise, after enumerating all paths, the program is reported as safe.

The most relevant verification tool to our analysis, TRACER [JMNS12], is a symbolic execution-based verification tool for sequential C programs. It attempts to address the first challenge by summarizing each subtree in the symbolic execution tree after finishing traversing it. Next, it attempts to subsume other similar subtrees in the symbolic execution tree using the generated summarization. The ability to subsume portions of the symbolic execution tree by previously analyzed subtrees will enable TRACER to remain scalable while verifying a program.

In order for the subsumption to be sound, an interpolation test is performed which checks if the infeasible paths in the analyzed subtree would remain infeasible wrt. the context reaching the subsumed subtree. We will elaborate more on the concept of *summarization with interpolation* in the next section.

[1]Limited to the power of the theorem solver

2.1.2 Symbolic Execution for Program Analysis

In program analysis, symbolic execution also known as symbolic simulation, is used to simulate the execution of tasks on an abstract model of the processor. The simulation is performed without input values and it is the simulator's duty to deal with the unknown execution states such as branches.

The first attempt to make symbolic execution for program analysis scalable was [JSV08], where the concept of *summarization with interpolation* was used for analysis on the specific problem of resource-constrained shortest path (RCSP).

The problem was formulated such that the scalability could be achieved by reusing the summarizations of previously computed sub-problems. After analyzing the paths in a sub-tree, an interpolant, preserving the discovered infeasible paths in the analyzed subtree, and a witness formula preserving the (sub)-analysis of the subtree was computed and stored. In case the node was encountered in another path, such that its context entailed the previously computed interpolant and witness formula, the paths emerging from that node were not explored. This step was called reuse and the subsumed node would share the analysis results of the subsuming node. Otherwise, the symbolic execution would naturally explore all the paths emerging from that node. In other words, this was a generalized form of dynamic programming. However, reuse of the summarizations was limited to loop-free programs.

The next step was taken in [CJ11, CJ13], where reuse of summarizations was enhanced for WCET analysis of programs with loops. In this modular resource analysis approach the timings of the basic blocks could be generated from the state-of-the-art low-level analysis. In the analysis, loops were fully unrolled and the analysis was made scalable by introducing compounded summarizations, where

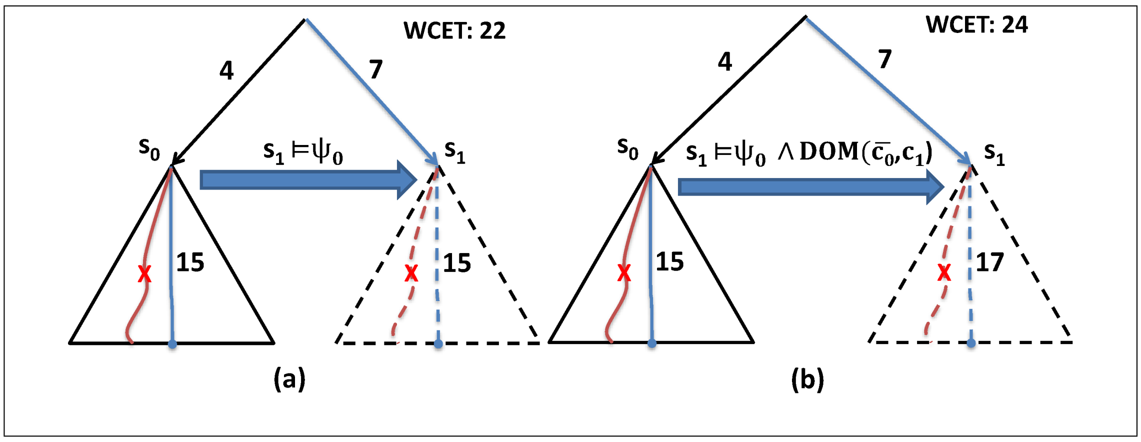


Figure 2.1: Reuse of Summarizations: (a) [CJ11] vs. (b) Our Analysis

one or more loop iterations were summarized and reused for subsuming other loop iterations.

Example 1. *Figure 2.1 (a), informally depicts a symbolic execution tree, where each triangle presents a subtree. The program contexts for the left and right subtrees, i.e., the symbolic states s_0 and s_1 respectively, are of the same program point. If we had applied the algorithm in [CJ11] on the left subtree, we would obtain two things: an* interpolant Ψ_0, *a generalization of s_0, encapsulating any context that would preserve the infeasible paths (indicated with a red cross) of the subtree. We also obtain a "representative" path, called a* witness, *indicated in blue, which gives rise to the* WCET *(15 in this case) of the subtree.*

The algorithm [CJ11] now considers the right subtree, where two tests are performed. First the context s_1 is checked if it implies the interpolant. If so, every infeasible path in the left subtree remains infeasible in the right subtree. A second test is whether the witness path is still feasible. If both tests are passed, the analysis can be reused *here, and the* WCET *of the right subtree can now be computed without traversal.*

The final analysis, at the root of the tree, can be computed by collating the

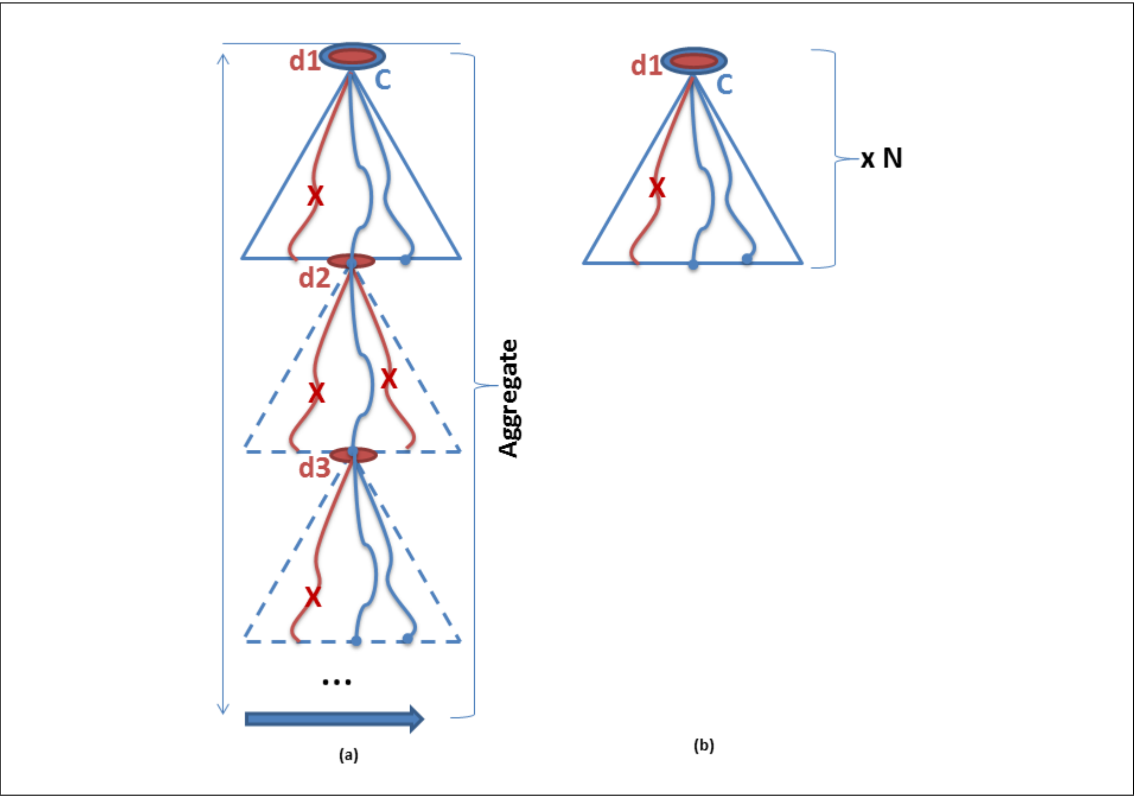


Figure 2.2: (a) Loop analysis in our framework (b) Loop analysis in AI Framework with some infeasible path detection, where C is the fixed point context of loop and d_i is the incoming context of a loop iteration

analyses of the left and right subtrees, and we can now determine its value of 22 as indicated. Note, importantly, that we never actually traversed the path that gives rise to this result; instead, we inferred its value.

2.1.3 Symbolic Execution on Loops

In [CJ11], loops are unrolled fully and summarized[2]. As a result, the exact number of iterations in complicated loop patterns can be found. In [CJ11] the loops are unfolded dynamically and for each loop iteration a different incoming context $d_1, d_2, d_3, \ldots$ is generated (Figure 2.2(a)). Note that the paths in blue are feasible

[2]Loops are unrolled gradually, meaning that loops are unrolled once then checked till unrolling is no longer feasible.

paths and the paths in red are infeasible paths. By checking the exact incoming context for each loop iteration, it can detect the infeasible paths which, due to the fixed point computation, were not detected in ILP-based methods (Figure 2.2(b)). For example, the framework in [CJ11] can detect the infeasible paths which are infeasible in only one loop iteration (e.g., the right-most path which is infeasible in only one triangle in Figure 2.2(a) and is feasible in Figure 2.2(b)). The process of loop unfolding continues till it reaches the end of the loop (blue arrow).

The method presented in [CJ11] is a powerful framework which can handle most of the challenges of utilizing symbolic execution for resource analysis. It can be generalized and used for resource analysis as long as it can be combined with a sound low-level analysis. However, the symbolic execution framework in [CJ11] performs program path analysis only and does not take into account the state of the micro-architectural features.

2.2 Overview of Our Resource Analysis Framework

In our framework, we need to go beyond program path analysis; thus we inherit or extend the fundamental concepts in [CJ11] (unrolling loops, reuse with summarization, etc.) to include micro-architectural features. This would enable our framework to inherit the benefits of the symbolic execution framework presented in [CJ11] and yet be more precise by taking into account the state of the micro-architectural features.

2.2.1 Microarchitecture-Aware Symbolic Execution

In our symbolic execution framework, each symbolic state is coupled with the internal states of all respective micro-architectural features. We refer to the internal state of such related micro-architectural features as the *implicit machine state*. The implicit machine state is the internal states of the micro-architectural features relevant to a resource analysis, which cannot be explicitly derived from the source of a program. The implicit machine state would contain enough information to perform precise resource analysis. In short we will use the term *machine state* to refer to the implicit machine state.

Not all micro-architectural features do affect all resource consumption behavior of programs. For example, in WCET analysis different micro-architectural features, ranging from cache to branch prediction, do have effect on the generated bound, while, in MHW analysis, the internal cache state does not have any effect on the amount of allocated memory. So, our proposal is to only capture the internal states of the micro-architectural features which are relevant to a resource analysis.

Our resource analysis framework is integrated in the sense that it performs high-level and low-level analysis in one phase. In summary, our framework performs resource analysis by:

1. Fully enumerating all symbolic paths in the symbolic execution tree.
2. Generating and precisely updating the machine state across the symbolic paths.
3. Using machine states to precisely capture the resource consumption behavior of the program along the symbolic paths.

2.2.2 Precision of Our Resource Analysis Framework

The three key features of our framework to perform precise resource analysis are (1) path sensitive symbolic execution, (2) microarchitecture-aware symbolic execution and (3) loop unrolling. These key features equip our framework with the means for precise resource analysis. We will demonstrate the precision of our framework in the following example.

```
for (i = 0; i < 100; i += 3) {
   if (i  %  5 == 0) { /* a */ }
   else { /* b */ }
}
```

Figure 2.3: An Academic Example

Example 2. *Let us consider the academic example in Figure 2.3. We like to measure a certain resource denoted by r, which can be affected in the presence of caches (e.g., timing or energy). We assume a direct-mapped cache, where block a and block b access the memory block m_1 and m_2 respectively, and m_1 and m_2 map to the same cache set. In other words, the memory blocks m_1 and m_2 conflict in the cache. Note that the execution of block a and block b is feasible, though in different iterations.*

A pure AI *approach such as [TFW00] will have to conservatively declare that the fixed point* must *cache at the looping point contains neither m_1 nor m_2. Thus, all the accesses are considered as misses, which affects the consumed resource r.*

On the other hand, [BCR13] might improve the analysis by ignoring some "simple" infeasible states from being considered. For example, if the conditional expression was $(i < 50)$, [BCR13] could discover that an execution of b cannot be followed by an execution of a in the next iteration. In other words, it could discover that *the access m_2 is indeed* persistent *[FW98, H$^+$11]. Specifically, only the first access to m_2 is a cache miss, the rest of m_2 accesses are cache hits. Now consider the case that the conditional expression is a bit more complicated, for example, $i \ \% \ 5 > 1$. Without knowing the precise value of i, after executing block a (or block b) in the current iteration, it is possible to either execute block a or block b in the next iteration. In such a case, a fixed point method, equipped with some form of path-sensitivity, will not be of much help to improve the analysis precision.*

In our analysis, we precisely capture the value of i throughout the analysis process. Consequently, we are able to disregard all infeasible states from consideration, thus achieving more accurate analysis result comparing to the state-of-the-art.

2.2.3 Scalability of Our Resource Analysis Framework

The precision of our analysis comes at the cost of scalability. Clearly, any framework which attempts the full exploration of the symbolic execution tree will not scale. As elaborated in Section 2.1.2, the core concept of reuse was used for scalability in [CJ11]. Reuse in [CJ11] was based on the idea that *the contribution of each basic block* in the computation of the timing (resource consumption) of a path is *constant*. However, in the presence of micro-architectural features such as caches, the contribution of each basic block is no longer constant. We define this concept as *dynamic resource consumption model*. As a result, reusing summarizations would no longer be sound. Note that we explained the safety and soundness terms for worst-case resource consumption bounds and analyses in Section 1.3.

In our framework, resource analysis is performed in the presence of the dynamic resource consumption model. The main contribution of our framework is furnishing the concept of reuse with the means to *dynamically generate the resource* *consumption behavior* of a subtree based on the stored summarization and the machine state reaching a node. For this to happen the *witness* should dominate all the other paths in the presence of the dynamic resource consumption model. As a consequence, the safety of reuse is guarded not only by the concept of *interpolation*, but also with the concept of *dominance*. By that, we mean, given a new symbolic state, it is sound to reuse a summarization if first the interpolant is satisfied which ensures all the discovered infeasible paths are maintained at the reuse point, and second a dominating condition is satisfied which ensures the machine state reaching the reuse point is similar enough to the machine state of the subsuming node. We will demonstrate this in the next example.

Example 3. *consider Figure 2.1(b) where we now focus on dynamic resource consumption, which arises because of cache configurations. A* cache state *c_0 is also part of the context s_0 of the subtree on the left. After analyzing the left subtree we obtain an interpolant Ψ_0 and a witness path (indicated in blue) as before. As explained above the reuse of this witness path is* unsound *in general. To remedy this, we now compute a* dominating condition *$\overline{c_0}$. Essentially, this is a formula which describes an abstract cache configuration which is sufficient to guarantee the witness path* remains optimal*, i.e., the worst-case path in the subtree, when encountering a new context.*

In Figure 2.1(b), suppose the dominating condition applies, that is, suppose that the cache context c_1 is covered by $\overline{c_0}$. We indicate this by the predicate $\mathsf{DOM}(\overline{c_0}, c_1)$. Now this allows us to reuse the witness path. We then need to proceed replaying *the witness path under the new cache configuration c_1. This, importantly, can lead to new* value *of the path (now 17), which is different from the original value (15). Finally, we can conclude the analysis on the whole tree with the value 24.*

Now suppose the dominating condition did not *apply. Then the path indicated by 17 may not be the worst-case path in the right subtree. For example, there could be a path of length 18 somewhere else in the subtree. If we reuse the witness path, we would now report, wrongly, a final value of 24.*

We end this section by emphasizing that reuse is the key to scalability, since with reuse our analysis avoids the full exploration of a symbolic execution tree. Reuse in the presence of the dynamic resource consumption model is only possible when path and resource analysis are performed in one integrated phase. We now present our analysis framework.

2.3 General Framework

2.3.1 Symbolic Execution with Machine State

We model a program by a transition system. A transition system $\mathcal{P}$ is a tuple $\langle \mathcal{L}, \ell_0, \longrightarrow \rangle$ where $\mathcal{L}$ is the set of program points, $\ell_0 \in \mathcal{L}$ is the unique initial program point. Let $\longrightarrow \subseteq \mathcal{L} \times \mathcal{L} \times Ops$, where Ops is the set of operations, be the transition relation that relates a state to its (possible) successors by executing the operations.

Basic operations are either assignments or "assume" operations. The set of all program variables is denoted by $Vars$. An assignment $x := e$ corresponds to assign the evaluation of the expression e to the variable x. The expression $assume(cond)$ means: if the conditional expression $cond$ evaluates to true, execution continues; otherwise it halts. We shall use $\ell \xrightarrow{op} \ell'$ to denote a transition relation from $\ell \in \mathcal{L}$ to $\ell' \in \mathcal{L}$ executing the operation $op \in Ops$[3]. Clearly a transition system is derivable

[3]Note that based on the resource of interest, basic operations can also have other forms such as memory allocations/deallocations for MHW analysis. We will customize the set of basic

from a CFG.

Definition 1 (Symbolic State). *A symbolic state s is a tuple $\langle \ell, m, \sigma, \Pi \rangle$ where $\ell \in \mathcal{L}$ is the current program point, m is the machine state the symbolic store σ is a function from program variables to terms over input symbolic variables, and finally the path condition Π is a first-order formula over the symbolic inputs.* □

Let $s_0 \overset{\text{def}}{=} \langle \ell_0, m_0, \sigma_0, \Pi_0 \rangle$ denote the unique initial symbolic state, where m_0 is the initial machine state. At s_0 each program variable is initialized to a fresh input symbolic variable. For every state $s \equiv \langle \ell, m, \sigma, \Pi \rangle$, the evaluation $[\![e]\!]_\sigma$ of an arithmetic expression e in a store σ is defined as usual: $[\![v]\!]_\sigma = \sigma(v)$, $[\![n]\!]_\sigma = n$, $[\![e{+}e']\!]_\sigma = [\![e]\!]_\sigma{+}[\![e']\!]_\sigma$, $[\![e{-}e']\!]_\sigma = [\![e]\!]_\sigma{-}[\![e']\!]_\sigma$, etc. The evaluation of the conditional expression $[\![cond]\!]_\sigma$ can be defined analogously. The set of first-order logic formulas and symbolic states are denoted by *FO* and *SymStates*, respectively.

Definition 2 (Transition Step). *Given $\langle \mathcal{L}, l_0, \longrightarrow \rangle$, a transition system, and a symbolic state $s \equiv \langle \ell, m, \sigma, \Pi \rangle \in SymStates$, the symbolic execution of transition $tr : \ell \xrightarrow{op} \ell'$ returns another symbolic state s' defined as:*

$$s' \overset{\text{def}}{=} \begin{cases} \langle \ell', m', \sigma, \Pi \wedge cond \rangle & \textit{if } op \equiv assume(cond) \\ \langle \ell', m', \sigma[x \mapsto [\![e]\!]_\sigma], \Pi \rangle & \textit{if } op \equiv x := e \end{cases}$$

where m' is the new machine state derived from m and op. m' is updated using $\mathsf{updMachineState}$ *function, where $m' \equiv \mathsf{updMachineState}(op, m)$.*

In order to be able to capture the precise machine state, we need a more low-level representation of a program. We have chosen LLVM intermediate representation (IR) [LA04], which while being expressive enough to precisely capture the machine state, preserves the overall CFG of the program. Moreover, LLVM IR

operations for each analyses in Chapters 3-5.

is both platform independent and is well-suited to perform static program analyses [S+12, ZGSV11]. For a more detailed discussion on how we generate the transition system from LLVM IR, we refer the interested reader to Appendix B.

The updMachineState function can be defined to capture the updates on the machine state based on the update criteria for the micro-architectural features. The updMachineState should be defined with regard to the resource of interest. We will customize the updMachineState function for each of the resource analyses presented in Chapters 3-5.

Abusing notation, the execution step from s to s' is denoted as $s \xrightarrow{tr} s'$ where tr is a transition. Given a symbolic state $s \equiv \langle \ell, m, \sigma, \Pi \rangle$ we also define $[\![s]\!] : \mathit{SymStates} \rightarrow \mathit{FO}$ as the projection of the formula

$$[\![\Pi]\!]_\sigma \wedge \bigwedge_{v \in \mathit{Vars}} v = [\![v]\!]_\sigma$$

onto the set of program variables Vars. The projection is performed by the elimination of existentially quantified variables.

For convenience, when there is no ambiguity, we just refer to the symbolic state s using the abbreviated tuple $\langle \ell, m, [\![s]\!] \rangle$ where ℓ and m are as before, and $[\![s]\!]$ is obtained by projecting s as described above.

Example 4 (Translation into Transition System). *Consider the C program fragment in Figure 2.4(a) and part of its respective LLVM IR in Figure 2.4(b). The program points are enclosed in angle brackets. Some of the transitions are shown in Figure 2.4(c). For instance, the transition $\langle\langle 1\rangle$,c := 0,$\langle 2\rangle\rangle$ represents that the system state switches from program point $\langle 1\rangle$ to $\langle 2\rangle$ and the constraint denotes the reset of c to* 0.

```
⟨1⟩        c = 0;
⟨2⟩        if (a > 0) ⟨3⟩t += 1;
⟨4⟩        if (b > 0) ⟨5⟩t += 2;
⟨6⟩        if (c > 0) ⟨7⟩t += 3;
⟨8⟩
```

(a) The C Program

```
⟨1⟩        store i32 0, i32* %c
           br label %2

⟨2⟩        %10 = load i32* %a
           %cmp = icmp sgt i32 %10, 0
           br i1 %cmp, label %3, label %4

⟨3⟩        %11 = load i32* %t
           %add = add nsw i32 %11, 1
           store i32 %add, i32* %t
           br label %4

⟨4⟩        %12 = load i32* %b
           %cmp1 = icmp sgt i32 %12, 0
           br i1 %cmp1, label %5, label %6
                    ...
```

(b) The LLVM IR

```
⟨⟨1⟩,   c := 0          ,⟨2⟩⟩
⟨⟨2⟩,   assume(a>0)     ,⟨3⟩⟩
⟨⟨3⟩,   t := t+1        ,⟨4⟩⟩
⟨⟨2⟩,   assume(a≤0)     ,⟨4⟩⟩
              ...
```

(c) The Transition System

Figure 2.4: From a C program to its Transition System

A path $\pi \equiv s_0 \rightarrow s_1 \rightarrow \ldots s_n$ is feasible if $s_n \equiv \langle \ell, m_n, [\![s_n]\!]\rangle$ and $[\![s_n]\!]$ is satisfiable. Otherwise, the path is called *infeasible* and s_n is called an infeasible state. Here we query a *theorem prover* for satisfiability checking on the path condition. We assume the theorem prover is sound, but not complete. If $\ell \in \mathcal{L}$ and there is no transition from ℓ to another program point, then ℓ is called the *end* *point* of the program. Under that circumstance, if s_n is feasible, then s_n is called *terminal* state.

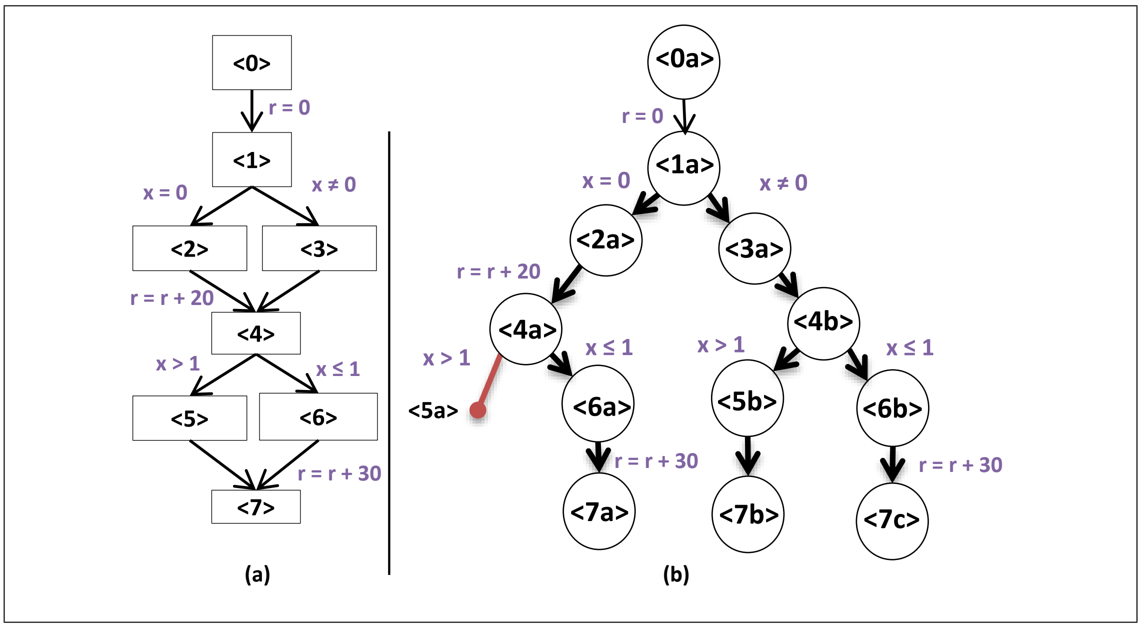


Figure 2.5: (a) a CFG and (b) Its Symbolic Execution Tree

Example 5 (Symbolic Execution Tree)**.** *Consider the CFG in Figure 2.5(a). Each node abstracts a basic block. In each basic block, a* program point *is shown. For brevity, we might use interchangeably the identifying program point when referring to a basic block. Two outgoing edges signify a branching structure, while the branch conditions are labeled beside the edges. Moreover, r is set to* 0 *in the beginning and the updates to it are also shown.*

Next, in Figure 2.5(b), we show our analysis tree*. Each node, shown as a circle, is identified by the corresponding program point, followed by a letter to distinguish between multiple visits to the same program point. Each path denotes a* symbolic execution path *of the program. Each node is associated with a symbolic state, but for simplicity we do not explicitly show any state content.*

Now assume that no basic block modifies x. At node $\langle 5a \rangle$, the projection of the path condition over program variables Vars*, namely* $[\![s_{5a}]\!]$*, is* $r = 20 \wedge x = 0 \wedge x > 1$*, which is equivalent to* false*. In other words, the leftmost path in Figure 2.5(b) is in fact* infeasible*. On the other hand, at node $\langle 7a \rangle$, the projection of the path condition over program variables* Vars*, namely* $[\![s_{7a}]\!]$*, is* $r = 50 \ \wedge \ x = 0 \ \wedge \ x \leq 1$.

The set of program variables $Vars$, also include a resource variable r. Our analysis computes a sound and accurate bound for r in the end, across all feasible paths of the program. Note that r is always initialized to 0. The allowed operations on r are defined based on the point that the resource of interest is cumulative or non-cumulative. For example, for WCET analysis the only operation allowed is concrete increment, but in MHW analysis, r can be both incremented or decremented.

Given a symbolic state $s \equiv \langle \ell, m, [\![s]\!] \rangle$ and a transition $tr : \ell \xrightarrow{op} \ell'$, the amount of change in the value of r at s by executing tr will be evaluated by a function ResEval. ResEval receives the machine state m and the current value for r as input and measures the updated value for r. ResEval is defined specifically based on the resource analyses in Chapters 3-5. The resource variable r is not used in any other way.

We stated in Section 2.2.1 that in our framework, loops are unrolled fully and summarized. We now introduce the concepts required for our loop unrolling framework. Our method benefits from the loop unrolling technology introduced in [CJ11] to find the exact number of iterations in complicated loop patterns (explained in Section 2.1.3).

We assume that each loop has only one loop head and one unique end point. This can be achieved by a preprocessing phase. For each loop, following the back edge from the end point to the loop head, we do not execute any operation. Note that our transition system is a directed graph. Moreover, since programs in safety critical systems do not contain **break** and **return** instructions in loops, the same treatment can be applied to have one loophead for loops.

Definition 3 (Loop). *Given a directed graph $G = (V, E)$ (transition system), we call a strongly connected component $S = (V_S, E_S)$ in G with $|E_S| > 0$, a loop of G.*

Definition 4 (Loop Head). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ of G, we call $\mathcal{E} \in V_L$ a loop head of L, also denoted by $\mathcal{E}(L)$, if no node in V_L, other than $\mathcal{E}$ has a direct successor outside L.*

Definition 5 (End Point of Loop Body). *Given a directed graph $G = (V, E)$, a loop $L = (V_L, E_L)$ of G and its loop head $\mathcal{E}$. We say that a node $u \in V_L$ is an end point of a loop body if there exists an edge $(u, \mathcal{E}) \in E_L$.*

Definition 6 (Same Nesting Level). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$, we say two nodes u and v are in the same nesting level if for each loop $L = (V_L, E_L)$ of G, $u \in V_L \iff v \in V_L$.*

2.3.2 Constructing Summarizations

In our framework, a "subtree" is a portion of the symbolic execution tree. Given a state s and program point ℓ_2 such that (a) state $s \equiv \langle \ell_1, m, [\![s]\!] \rangle$ appears in the tree, and (b) ℓ_2 post-dominates ℓ_1, then $subtree(s, \ell_2)$ depicts all the paths emanating from s and, if feasible, terminating at ℓ_2. (Note that ℓ_2 may not be the end point of the whole tree.) We call ℓ_1 and ℓ_2 the entry and exit points of the subtree.

A summarization of a subtree, intuitively, is a succinct description of its analysis. This is formalized as a tuple of certain important components of the analysis.

These are: the entry and exit program points, an interpolant describing infeasible paths, one or more witnesses describing the paths with highest resource consumption in the subtree, one or more dominating condition ensuring the witnesses represents the appropriate worst-case paths in the subtree, and finally an abstract transformer relating the input and output program variables and an abstract transformer relating the input and output machine states. The abstract transformers are used to generate the outgoing context at the exit point.

We start with our notion of interpolant. The idea here is to approximate at the root of a subtree, the weakest precondition in order to maintain the infeasibility of all the nodes inside[4]. In the context of program verification, an interpolant captures succinctly a condition which ensures the *safety* of the tree at hand. Adapting this to program analysis was first done in [JSV08] which formalized a generalized form of dynamic programming. The work applying interpolation for reuse in program analysis has been [CJ11]. In the context of program analysis, the problem is formulated such that the scalability can be achieved by reusing previously computed sub-problems. Since all infeasible nodes are excluded from calculating the analysis result of a subtree, in order to ensure soundness, at the point of reuse, all such infeasibility must also be maintained. Our framework adopts the efficient algorithm for computing the interpolant from [JSV09].

Next, we discuss the *witness* concept. Intuitively, it is a path that depicts the worst-case resource consuming path of a subtree and/or the path with the peak resource consumption (for non-cumulative resources). Witness is depicted by Γ. The resource cost of a witness is obtained dynamically from replaying the sequence

[4]An exact computation of the weakest precondition is in general intractable. However, the approximation of the weakest precondition generated by our framework would be in conjunction form and as a result is tractable

resource consuming items along the path under an incoming machine state m.

We say that two nodes in a symbolic execution tree are *similar* if they refer to the same program point. Thus two subtrees are similar if they share the same entry and exit program points.

We next discuss *dominating condition*, another component of our analysis of a subtree. Intuitively, this is a description of what machine state is needed in order that the witness *remains optimal* in a similar subtree. That is, in a subtree, the witness remains the most resource consuming path.

The witness and the dominating condition are customized based on the resource analysis. We will present customized witness and dominating condition for each of the resource analyses presented in Chapters 3-5.

We now discuss an abstract transformer Δ_p of a subtree from ℓ_1 to ℓ_2 which is an abstraction of all feasible paths (w.r.t. the incoming symbolic state s) from ℓ_1 to ℓ_2. Its purpose is to capture an input-output relation between the program variables. In our implementation, we adopt from [CJ11] which uses the polyhedral domain [CH].

Similarly, we also need an abstract transformer for machine state. We name it as a machine state summary Δ_m. The machine state summary is generated based on the micro-architectural features captured in the machine state. More details on the how to generate and apply machine state summaries is presented separately for each of the resource analyses in Chapters 3-5.

Definition 7. *A* summarization *of subtree*(s, ℓ_2)*, where* ℓ_1 *is the program point of* s*, is a tuple*

$$[\ell_1, \ell_2, \overline{\Psi}, \Gamma, r, \delta, \Delta_p, \Delta_m]$$

where $\overline{\Psi}$ is an interpolant, Γ is the witness, r is the resource cost of the subtree(s, ℓ_2), and δ is the dominating condition. Δ_p is an abstract transformer relating the input and output variables and finally, Δ_m is a machine state summary. □

Note that all components stored in the summarization (except abstract transformer) are in conjunctive form and these components do not contain any disjunctions.

2.3.3 Reuse with Interpolation and Dominance

We now display a key feature of our algorithm: reuse of a summarization. Suppose we have already computed a summarization $[\ell_1, \ell_2, \overline{\Psi}, \Gamma, r, \delta, \Delta_p, \Delta_m]$. Suppose we then encounter a symbolic state $s' \equiv \langle \ell_1, m, [\![s']\!] \rangle$. The summarization now can be reused if:

1. $[\![s']\!]$ implies the stored interpolant Ψ i.e., $[\![s']\!] \models \Psi$.
2. The context of s' is consistent with the witness, i.e., $[\![\Gamma]\!] \wedge [\![s']\!]$ is satisfiable.
3. The dominating condition is satisfied by the incoming machine state m, i.e., $\mathsf{DOM}(\delta, m)$ holds.

The worst-case resource cost of the subtree beneath the state s' is then derived from the witness Γ and the machine state m. Note that the resource cost of the subtree beneath s' can be different from r. Finally, the outgoing state of the subtree at ℓ_2 denoted by s'_2 is generated by $s' \xrightarrow{\Delta_p, \Delta_m} s'_2$.

We now conclude this subsection by mentioning that we only summarize at selected program points. Given entry point ℓ_1, the corresponding exit point ℓ_2 is determined as follows. It is the program point that post-dominates ℓ_1 s.t. ℓ_2 is of the same nesting level as ℓ_1 and either is (1) an end point of the program, or (2) an end point of some loop body. In other words, we only perform "merging" abstraction at loop boundaries. As ℓ_2 can always be deduced from ℓ_1, in a summarization, we omit the component about ℓ_2. So, in a short form we store a summarization as $[\ell, \overline{\Psi}, \Gamma, r, \delta, \Delta_p, \Delta_m]$.

2.4 Customizing the Analysis Framework

Before presenting the integrated resource analysis algorithm, we will review over the terms and functions which should be customized for performing any specific resource analysis with our framework. These terms have been briefly defined in this chapter and a detailed presentation of these terms and functions will be presented in Chapters 3-5.

Table 2.1 presents the list of the terms and the functions needed to be customized based on the resource of interest. By defining these terms and their respective functions in detail, our framework will be enabled to perform different resource analyses. We will now review the terms and functions one by one:

Table 2.1: Customizable Terms

	Customizable Terms	Respective Functions
1	Basic Operations & Resource Consumption Cost Model	ResEval
2	Machine State	updMachineState
3	Witness & Dominating Condition	Summarize-a-Trans
		Combine-Witness
		Merge-Witness
4	Machine State Summary	Combine-Summary
		Merge-Summary

1. **Basic Operations:** As explained in Section 2.3, based on the resource of

interest, basic operations can also have other forms such as memory allocations/deallocations for MHW analysis. We will customize the set of basic operations for each analyses.

Moreover, the resource consumption cost model should be defined. For example, in WCET analysis the resource which we are interested to estimate is time and we should define the cost model for time in the presence of micro-architectural features tracked in the machine state (cache state in this case). The ResEval function will capture the consumed resource based on the cost model.

2. **Machine State:** Defining precisely the machine state is the second step for customizing the resource analysis framework. The machine state will represent the internal state of the micro-architectural features that affect the resource analysis. For example, in the WCET analysis presented in Chapter 3, we will take into account the instruction and data cache. So the machine state will be defined such that it would contain the internal states of the data and instruction cache. Moreover, the updMachineState function which tracks the updates of the machine state during the traversal of the symbolic execution tree would be defined.

3. **Witness and Dominating Condition:** In this chapter, we defined the witness as the most resource consuming path in an analyzed subtree and the dominating condition as the set of constraints that guarantee the domination of the witness over the other paths in a subtree w.r.t. the machine state. The witness only stores the items which affect the amount of the resource consumed in the path. As a result the details of the items stored in the

witness and the details of the constraints stored in the dominating condition are defined based on the analyzed resources.

The Summarize-a-Trans function computes a summarization for a single transition *tr* at state *s*. This can be seen as a basic step in our algorithm. This function is customized based on the resource of the interest.

Finally, the Combine-Witness and Merge-Witness functions should be defined to present the basic merge and combine steps for generating witness and the dominating condition recursively.

4. **Machine State Summary:** The machine state summaries are used when reusing summarizations of loop iterations. Machine state summary can only be defined after machine state is precisely defined. The machine state summary is the last term that should be defined for customizing the resource analysis framework.

   The Combine-Summary and Merge-Summary functions are defined to present the merge and combine steps for the machine state summaries.

2.5 The Integrated Algorithm

In this section, we explain Algorithm 1 in a top-down manner. The function Analyze takes the initial symbolic state s_0 and the transition system $\mathcal{P}$ of an input program. It then invokes Summarize to generates a summarization for the whole program (line 1) and returns r as the maximum resource consumption of the program (line 2).

2.5.1 The Summarize Function

The Summarize function performs a depth-first traversal of the symbolic execution tree. During the depth-first traversal, at each node either (1) a summarization

Algorithm 1 Integrated WCET Analysis Algorithm

```
function Analyze($s_0$,$\mathcal{P}$)
     Let $s_0$ be $\langle 0, m_0, [\![s_0]\!]\rangle$
⟨1⟩ $[0, \cdot, \cdot, r, \cdot, \cdot, \cdot] :=$ Summarize($s_0, \mathcal{P}$)
⟨2⟩ return $r$

function Summarize($s$,$\mathcal{P}$)
     Let $s$ be $\langle \ell, m, [\![s]\!]\rangle$
⟨3⟩ if ($[\![s]\!] \equiv false$) return $[\ell, false, [\,], -\infty, false, [\,], [\,]]$
⟨4⟩ if (outgoing($\ell, \mathcal{P}$) $= \emptyset$) return $[\ell, false, [\,], 0, [\,], [Id(Vars, m)], [\,]]$
⟨5⟩ if (loop_end($\ell, \mathcal{P}$)) return $[\ell, false, [\,], 0, [\,], [Id(Vars, m)], [\,]]$
⟨6⟩ $S := [\ell, \overline{\Psi}, \Gamma, r, \delta, \Delta_p, \Delta_m] :=$ memoed($\ell$)
⟨7⟩ if ($[\![s]\!] \models \overline{\Psi} \ \wedge \ [\![\Gamma]\!] \ \not\equiv \ false \ \wedge$ DOM($\delta, c$)) return $S$
⟨8⟩ if (loop-head($\ell, \mathcal{P}$))
⟨9⟩     $S_1 := [\cdot, \cdot, \Gamma_1, \cdot, \cdot, \Delta_{p1}, \Delta_{m1}] :=$ TransStep($s, \mathcal{P}$, entry($\ell$,$\mathcal{P}$)))
⟨10⟩    if ($[\![\Gamma_1]\!] \equiv false$)
⟨11⟩        $\overline{S} :=$ JoinHorizontal($m, S_1$, TransStep($s, \mathcal{P}$, exit($\ell, \mathcal{P}$)))
        else
⟨12⟩        Let $tr$ be $\ell \xrightarrow{\Delta_{p1}, \Delta_{m1}} \ell'$
⟨13⟩        $s \xrightarrow{tr} s'$
⟨14⟩        $S' :=$ Summarize($s', \mathcal{P}$)
⟨15⟩        $S :=$ Compose($S_1, S'$)
⟨16⟩        $\overline{S} :=$ JoinHorizontal($m, S$, TransStep($s, \mathcal{P}$, exit($\ell, \mathcal{P}$)))
⟨17⟩ else $\overline{S} :=$ TransStep($s, \mathcal{P}$, outgoing($\ell, \mathcal{P}$))
⟨18⟩ memo and return $\overline{S}$
```

is reused, thus we do not need to expand the node; or (2) after expanding it, we compute its summarization based on the summarizations of its child nodes. We now discuss the Summarize function in detail.

Base Cases: Summarize handles 4 base cases. First, when the symbolic state s is infeasible (line 3). Note that here path-sensitivity plays a role because provably infeasible paths will be excluded from contributing to the analysis result. Thus the returned witness is $[\,]$. The abstract transformer is $[\,]$ too. Note that the empty machine state summary is also depicted by $[\,]$ for brevity. Second, s is a terminal state (line 4). Here Id refers to the identity function, which keeps the symbolic state unchanged. The end point of a loop is treated similarly in the third base case (line 5). The last base case, lines 6-7, is the case that a summarization can be reused. We have discussed this step in Section 2.3.3.

Expanding to the next programming point: Line 17 depicts the case when transitions can be taken from the current program point ℓ, and ℓ is not a loop head. We call TransStep to move recursively to next program points. TransStep considers all transitions emanating from ℓ, denoted as $\mathsf{outgoing}(\ell, \mathcal{P})$, then calls Summarize recursively and compounds the returned summarizations into a summarization of ℓ.

In more detail, for each tr in $TransSet$, TransStep extends the current state with the transition. We then call Summarize with the resulting child state (line 22). The algorithm aggregates each returned summarization into a single summarization, namely $\overline{S}$. This is achieved by first calling Compose (line 23), then calling JoinHorizontal (line 24). We use Compose and JoinHorizontal (explanation delegated to the later parts in this section) to combine vertically and merge horizontally two summarizations. Note here that we construct a summarization from a single transition before calling Compose. Since all components (except abstract transformer) stored in the summarizations are not in disjunctive form, the execution of the Compose and JoinHorizontal steps is not heavy in terms of the computation.

Handling Loops: Lines 9-16 handle the case when the current program point ℓ is a loop head. Let $\mathsf{entry}(\ell, \mathcal{P})$ denote the set of transitions going into the body of the loop, and $\mathsf{exit}(\ell, \mathcal{P})$ denote the set of transitions exiting the loop.

Upon encountering a loop, our algorithm attempts to unroll it once by calling the function TransStep to explore the entry transitions (line 9). If the returned witness formula is *false*, meaning that it is infeasible to execute another iteration, we

```
function TransStep($s, \mathcal{P}, \mathit{TransSet}$)
        Let $s$ be $\langle \ell, \cdot, \cdot, \cdot \rangle$
⟨19⟩ $\overline{S} := [\ell, \mathit{false}, [\ ], 0, [\ ], [Id(Vars, m)], [\ ]]$
⟨20⟩ foreach ($tr \in \mathit{TransSet}$) do
⟨21⟩     $s \xrightarrow{tr} s'$
⟨22⟩     $S' :=$ Summarize($s', \mathcal{P}$)
⟨23⟩     $S :=$ Compose(Summarize-a-Trans($s, tr$), $S'$)
⟨24⟩     $\overline{S} :=$ JoinHorizontal($m, \overline{S}, S$)
        endfor
⟨25⟩ return $\overline{S}$
end function

function Compose($S_1, S_2$)
        Let $S_1$ be $[\ell_1, \Psi_1, \Gamma_1, r_1, \delta_1, \Delta_{p1}, \Delta_{m1}]$
        Let $S_2$ be $[\ell_2, \Psi_2, \Gamma_2, r_2, \delta_2, \Delta_{p2}, \Delta_{m2}]$
⟨26⟩ $r := r_1 \ + \ r_2$
⟨27⟩ $\Delta_p := \Delta_{p1} \ \wedge \ \Delta_{p2}$
⟨28⟩ $\Delta_m :=$ Combine-Summary($\Delta_{m1}, \Delta_{m2}$)
⟨29⟩ $\Psi := \Psi_1 \ \wedge$ Pre-Cond($\Delta_{p1}, \Psi_2$)
⟨30⟩ $\{\Gamma, \delta\} :=$ Combine-Witnesses($\Gamma_1, \Gamma_2, \delta_1, \delta_2$)
⟨31⟩ return $[\ell_1, \overline{\Psi}, \Gamma, r, \delta, \Delta_p, \Delta_m]$
end function

function JoinHorizontal($m, S_1, S_2$)
        Let $S_1$ be $[\ell, \Psi_1, \Gamma_1, r_1, \delta_1, \Delta_{p1}, \Delta_{m1}]$
        Let $S_2$ be $[\ell, \Psi_2, \Gamma_2, r_2, \delta_2, \Delta_{p2}, \Delta_{m2}]$
⟨32⟩ $\{\Gamma, r, \delta\} \ = $ Merge-Witnesses($m, \Gamma_1, \Gamma_2, r_1, r_2, \delta_1, \delta_2$)
⟨33⟩ $\Delta_m :=$ Merge-Summary($\Delta_{m1}, \Delta_{m2}$)
⟨34⟩ $\Delta_p := \Delta_{p1} \ \vee \ \Delta_{p2}$
⟨35⟩ $\Psi := \Psi_1 \ \wedge \ \Psi_2$
⟨36⟩ return $[\ell, \Psi, \Gamma, r, \delta, \Delta_p, \Delta_m]$
end function
```

Figure 2.6: Helper Functions

thus proceed with the exit branches. The returned summarization is merged (using JoinHorizontal) with the summarization of the previous unrolling attempt (line 11). Otherwise, we first use the returned abstract transformer to produce a new continuation context, (line 12 and 13), then we continue the analysis from the next loop iteration on-wards (line 14). The returned information is then compounded with the summarization of the first iteration (line 15). Note that, importantly, compounded summarizations of the inner loop(s) can be reused in later iterations of the outer loop. In other words, the compounded summarizations generated from the iterations of an inner loop can be reused to avoid exploring part or all of the inner loop in later iterations of the outer loop.

2.5.2 Compounding Two Summarizations

Next, we will elaborate on how summarizations are compounded through the functions Compose and JoinHorizontal in Figure 2.6.

Compounding Vertically Two Summarizations: Consider $subtree(s_2, \ell_3)$ suffixes $subtree(s_1, \ell_2)$, where $s_2 \equiv \langle \ell_2, m_2, [\![s_2]\!] \rangle$ and $s_1 \equiv \langle \ell_1, m_1, [\![s_1]\!] \rangle$. In other words, a path π_1 from ℓ_1 to ℓ_2 followed by a path π_2 from ℓ_2 to ℓ_3 corresponds a path π in $subtree(s_1, \ell_3)$. The Compose function returns a summarization for $subtree(s_1, \ell_3)$ by compounding the two existing summarizations, respectively for $subtree(s_1, \ell_2)$ and $subtree(s_2, \ell_3)$.

The resource cost of $subtree(s_1, \ell_3)$ is computed as the sum of the resource cost of the subtrees (line 26), the abstract transformer Δ_p is computed as the conjunction of the input abstract transformers (line 27), with proper variable renaming. Note that in our implementation, abstract transformers are computed using polyhedral domain. We employ Δ_p to generate *one* continuation context, before proceeding the analysis with subsequent program fragments. Next, the desired interpolant must capture the infeasibility of S_1, as well as the infeasibility of S_2 given that we treat $subtree(s_1, \ell_2)$ as an abstract transition, of which the operation is Δ_p. We rely on the function Pre-Cond, which in line 29 under-approximates the weakest-precondition of the post-condition Ψ_2 w.r.t. to the transition relation

Δ_p. Finally, we use Combine-Summary to construct the overall machine state input-output relation for $subtree(s_1, \ell_3)$ (line 28) and Combine-Witnesses to compound the witnesses and the dominating conditions of the summarizations (line 30).

Compounding Horizontally Two Summarizations:

Given two summarizations rooted at two nodes which are siblings, we want to propagate the information back and compute the summarization for the parent node. While propagation can be achieved by Compose, we need JoinHorizontal (presented in Figure 2.6) to "merge" the contributions of the two children to the parent node. Note that unlike Compose, we need to select the longer path between the two witnesses of the input summarizations. Such selection depends on the current machine state. That is why the machine state m is passed as an input to JoinHorizontal, which subsequently is passed it on to Merge-Witnesses. Merge-Witnesses and Merge-Summary, which are customized based on the analysis, are used to merge witnesses, dominating conditions and machine state summary. The abstract transformer Δ_p, however, is computed straightforwardly as the disjunction of the input abstract transformers. All the infeasible paths in both substructures must be maintained, thus the desired interpolant is the conjunction of the two input interpolants. Examples on compounding summarizations vertically or horizontally are presented in the Example Analysis Sections of Chapters 3-5.

Finally, we conclude this section with a formal statement about the soundness and termination of our framework.

Theorem 1 (Sound Resource Consumption Estimation)**.** *Our algorithm always produces* safe *worst-case resource consumption estimates.*

Proof. Our algorithm performs a depth-first traversal of the symbolic execution tree. In all steps except when reuse happens, what we perform is only widening the execution contexts, not narrowing them. Because of such steps, we might over-approximate the real worst-case resource cost; but this is safe. However, reuse in our setting is different from [CJ11], so we need to check its soundness separately.

Assume that, at symbolic state $s \equiv \langle \ell, m, [\![s]\!] \rangle$, we reuse a summarization $[\ell, \overline{\Psi}, \Gamma, r, \delta, \Delta_p, \Delta_m]$ of a subtree T.

Also assume that the reuse is *unsafe*. Note that when reuse happens, we employ the abstract transformers to generate a continuation context and continue the analysis from there. This step is also a widening step, thus it is safe. As a result, there must be a feasible path in the *avoided* subtree emanating from s, of which the resource cost is more than the resource cost of the witness Γ. Let us call this path Γ'.

Because the first condition for reuse implies that all infeasible paths of T stay infeasible under the new context s, Γ' must be feasible in T as well. Obviously, in order for Γ to be reported as the witness, in T, the resource cost of Γ' must be not more than the resource cost of Γ.

The third condition for reuse ensures that the dominating condition is satisfied. This implies that the machine state at s maintains the optimality of Γ. In particular, if the resource cost of Γ (in T) is not less than the resource cost of some other feasible path (in T), it is still the case under the new context s. Consequently, under context s, the resource cost of Γ' can not be more than the resource cost of Γ. This is a contradiction. $\square$

We remark here that we do not make use of the second condition for reuse in the proof of soundness. In fact, that condition has to do with the *precision* of reuse, rather than its soundness. An important implication – which has been shown in [JSV08] – is that our algorithm produces "exact" analysis for loop-free programs.

□

Theorem 2 (Termination of Resource Analysis)**.** *Our algorithm always* terminates.

Proof. Our algorithm performs a depth-first traversal of the symbolic execution tree. Due to the assumptions on programs used in safety critical embedded systems, the programs terminate always. Thus, the symbolic execution tree is a bounded tree. Considering that all the data structures generated and utilized in our algorithm are also bounded in length, our algorithm will always terminate.

□

2.6 Summary

In this chapter, we presented our resource analysis framework which performs integrated resource analysis. We demonstrated that our framework is able to generate the most accurate bounds over the consumed resources. Moreover, the notions of *reuse* and *domination* were introduced or enhanced to help our framework remain scalable. Finally, we presented the steps needed to customize our framework. Following these steps, in the next three chapters, we will customize our framework for three different analyses: (1) WCET analysis, (2) MHW analysis and (3) WCEC analysis.

Chapter 3

Integrated Worst-case Execution Time Analysis with Cache

3.1 Introduction

In this chapter, we customize the resource analysis framework presented in Chapter 2 for WCET analysis. The main contribution of the WCET analysis presented in this chapter is that while our integrated resource analysis framework increases the precision, it also scales for a class of realistic benchmarks. Thus, our method can be employed for applications where precise WCET analysis is pivotal.

As discussed in Chapter 1, hard real-time systems need to meet *hard* deadlines. Static Worst-Case Execution Time (WCET) analysis is therefore very important in the design process of real-time systems. However, micro-architectural features in processors (e.g., caches) make WCET analysis a difficult task.

WCET analysis is usually proceeded in two phases for scalability: high-level phase and low-level phase. In low-level analysis, we need to consider timing effects of performance enhancing processor features such as pipeline and caches. This chapter focuses on caches, since the impact of the caches on the real-time behavior of programs is much more than other features [MHH02]. Consequently, the machine state will only *track the updates on the cache states* in this chapter.

Cache analysis – to be scalable – is usually accomplished by abstract interpretation (AI) [TFW00]. In other words, we need to analyze the memory accesses of the input program via an iterative fixed point computation. In Chapter 1, two reasons for imprecision of AI framework was presented. We will elaborate more on these reasons with respect to caches in Sections 3.2 and in the results presented in Section 3.5 of this chapter. We explained that most of such algorithms employ a *fixed point* computation in order to aggregate the analysis across loop iterations. Consequently, they still inherit the imprecision from an AI framework, identified as point (2) in Section 1.2.1. More specifically, a fixed point method will compute a worst-case timing for each basic block in all possible contexts, even though the timings of a basic block in different iterations of a loop can diverge significantly.

In Chapter 2, we demonstrated that our resource analysis framework is able to *systematically address* these two causes of impression. However, we left certain terms in the framework to be defined while customizing the framework for different resources. In this chapter, We will customize the resource analysis framework for WCET analysis.

3.2 Related Work

WCET analysis has been the subject of much research, and substantial progress has been made in the area (see [PB00, $W^{+}08$] for surveys of WCET). As discussed before, WCET analysis is often conducted by separating low-level analysis and high-level analysis into different phases.

3.2.1 High-level analysis

Among the works on high-level analysis, our most important related work is [CJ11] which was thoroughly investigated in Chapter 2. On the other hand, there are recent CEGAR-like methods, which start by generating a rough WCET estimate and then gradually refine it. “WCET squeezing” [KKZ13] is built on top of the Implicit Path Enumeration Technique (IPET) [LM95]. A solution to the given integer linear programming (ILP) formula corresponds to number of program traces, of which the feasibility will be checked (one-by-one) via SMT solving. If such a trace is infeasible, additional ILP constraints are added to exclude it from further consideration. Subsequently, [Č+] proposes hierarchical segment abstraction, thus allows the computation of WCET by solving a number of independent ILP problems, instead of one large *global* ILP problem. Since the abstract segment trees can store more *expressive* constraints than ILP, better refinement procedure can be implemented.

We also mention the recent work [H+14a], which also employs the concept of interpolation, but under the SMT framework, to avoid state explosion in WCET analysis. Like [JSV08], this approach is formulated for loop-free programs, and not yet suitable for analyzing programs with loops.

In summary, we can see a trend of research where recent advances in software verification are employed for WCET high-level analysis. However, it is unclear if these approaches will remain scalable when extended towards low-level analysis, under the presence of loops and/or many infeasible paths.

3.2.2 Low-level analysis

Low-level analysis, with emphasis on caches, has always been an active research topic in WCET analysis. Initial work on instruction cache modeling uses ILP [LMW99]. However, the work does not scale due to a huge number of generated ILP constraints. Subsequently, the AI framework [CC77] for low-level analysis, proposed in [TFW00], has made an important step towards scalability. The solution has also been applied in commercial WCET tools (e.g., [aiT]). For most existing WCET analyzers, AI framework has emerged to be the basic approach used for low-level analysis. Additionally, static timing analysis with data cache has been investigated in [W$^+$97, FW98, H$^+$11].

Recent approaches [CR11, BCR13] by the same research group – combining AI with verification technology – have shown some promising results. In the more recent work [BCR13], a partial path is tracked together with each micro-architectural state μ. This partial path captures a subset of the control flow edges along which the micro-architectural state μ has been propagated. If a partial path was infeasible, its associated micro-architectural state can be excluded from consideration. To be tractable, micro-architectural states are merged at appropriate sink nodes. (In fact, the partial path constraints are merged to *true*.) As a result, the approach is only effective for detecting infeasible paths whose conflicting branch conditions appeared relatively close to each other in the CFG.

In a similar spirit as [KKZ13] and [CR11], Nagar and Srikant [NS] propose the concept of *cache miss paths*. The method employs IPET formulation, using the information from the worst-case solution of the ILP problem (which corresponds to a number of program paths) to improve the precision of AI-based cache analysis. However, it is reported that for benchmarks `statemate` and `nsichneu` – which contain a large number of program paths – little improvement is obtained.

It is important to note that, in general, the above-mentioned approaches still employ a fixed-point computation in order to ensure sound analysis across loop iterations. Thus, they inherit the imprecision of AI, because the timings of a basic block in different iterations of a loop often can diverge significantly.

3.2.3 Other Related Work

We mention some orthogonal works that represent recent and interesting advances in WCET research. [L$^+$] performs loop unrolling, passing flow information from source level through the process of compiler optimization in order to help tighten the WCET estimates. At the current stage, the approach seems to be limited to single-path programs. Zolda and Kirner [ZK15] propose to incorporate the information from collected program execution traces into IPET framework to enhance the precision of calculated WCET bounds. The effectiveness of this approach seems dependent on the quality of the *collected* traces as well as the amount of infeasible paths in the given input program.

We remark that the idea of coupling low-level analysis with high-level analysis (with loop unrolling) dates back to [LS99]. However, to counter state explosion, the only solution of [LS99] is to perform merging frequently. In the end, the approach forfeits its intended precision, while at the same time, does not scale realistic benchmarks.

Finally, we remark on the issue of *timing anomaly* [R$^+$06]. In general, timing anomaly can make abstraction (and therefore AI) *unsound*. It is extremely hard to systematically address this issue. More often, custom solutions are employed. For example, [RS09a] can compute a constant bound to be added to the local worst-case path to safely handle timing anomalies, provided they are not of "domino-effect" type. This approach is also applicable to us. In this chapter we explore WCET analysis without timing anomaly.

3.2.4 Commercial WCET Tools

In this section, we will briefly mention a few of the many commercial WCET tools. Most commercial tools, such as [aiT], use AI framework for the low-level analysis. The worst-case timing for each basic block is then aggregated using the ILP formulation to give the final WCET estimate. In other words, low-level and high-level analyses are performed separately. The immediate benefit is that these tools scale impressively and are applicable to a wide range of input programs.

AbsInt timing-analysis tool (aiT)

The purpose of the AbsInt timing-analysis tool aiT [aiT] is to obtain upper bounds for the execution times of code snippets in executables. aiT works on executables in order to acquire the information on the register usage and the instruction and data addresses. The addresses are later used in cache analysis. Moreover, their tool supports memories with different timing behavior areas. aiT is one of the most powerful tools using the ILP method and collects constraints based on the abstract processor model. The team working on aiT has extended the tool with support for abstract models of different commercial processors.

Bound-T Time and Stack Analyzer

Bound-T [Bou], is another tool which was planned for the analysis of on-board software in spacecraft. The tool determines an upper bound on the execution time of subroutines, including called functions. It gets binary executable program as input with an embedded symbol table containing the debug information. The tool is able to compute upper bounds on some normal loops. For other loops the user provides annotations, called assertions in Bound-T.

Florida WCET Analysis Tool

The research prototype of Florida State and Fruman Universities [Flo14] works on hard real-time systems and energy-aware embedded systems with timing constraints. It performs path-based static analysis.

TU Vienna Analysis Tools

From the TU Vienna, the first is a prototype tool for static-timing analysis with IPET that has been integrated into a Matlab/Simulink tool chain and can analyze C code or Matlab/Simulink models. It performs timing analysis for C programs coded in WCETC (a subset of C with extensions to make annotations about (in)feasible execution paths [K$^+$02]). The second tool is a measurement based tool that uses genetic algorithms to direct input-data generation for timing measurements in the search for the worst case or long program execution times. The third tool is a hybrid tool for timing analysis that uses both measurements and static analysis with IPET to assess the timing of C code.

Chronos WCET Analysis Tool

Chronos [chr14] is developed in the National University of Singapore. It is an open-source static WCET analysis tool. Its aim is to find more close-fitting bounds on the WCET of programs executed on modern processors. Chronos receives a C program with the configuration of the target processor. First the tool performs data-flow analysis to compute loop bounds. In case of failure, user annotations have to be provided. The user can add information about infeasible-paths to improve the accuracy of the results. Chronos has supports for analyzing out-of-order pipelines, dynamic branch prediction schemes, and instruction/data caches. More details on the commercial WCET tools are available at [W$^+$08].

3.2.5 A discussion on the State-of-the-art WCET Analysis

Considering the different methods and commercial WCET tools which have been discussed in this section, finding the state-of-the-art WCET analysis is not a trivial task. The reason is that the accuracy of a WCET analysis not only depends on how well the high-level and low-level analyses are performed but also it depends on how well a hardware is simulated in the analysis. However, comparing different methods based on the accuracy of their high-level and low-level analyses can help the ultimate precision that a WCET analysis can reach to.

Based on this, we can state that the algorithm in [BCR13], which expands some path-sensitivity to low-level analysis, comprises the state-of-the-art micro-architectural modeling (AI+SAT) combined with an ILP formulation for WCET analysis. In this algorithm, must analysis and persistence analysis are used to model abstract instruction and data cache. In the rest of this chapter, we will refer to this algorithm as **`AI+SAT`**$\oplus$**`ILP`**.

On the other hand, the path based method presented in [CJ11] presents the state of the art high-level analysis. It appears that the state-of-the-art modular WCET analysis would be a *hypothetical* algorithm which benefits from combining the `AI+SAT` low-level analysis and the high-level analysis in [CJ11]. More specifically, this algorithm improves on other methods because of fully unrolling loops and an increased infeasible path detection in high level and low level analyses. In the rest of this chapter, we will refer to this algorithm as **`AI+SAT`**$\oplus$**`Unroll_s`**. We will compare our framework with these two analyses in Section 3.5.

3.3 General Framework

In this section, we will present the customized resource analysis framework for WCET analysis with emphasis on caches. As explained in Section 3.1, in this chapter, we will limit the machine state only to the data and instruction caches.

We adopt the concept of *abstract cache state* (ACS) and the *update* and *join* functions in our framework from the update and join functions of the A-way set associative cache with LRU replacement policy for the *must analysis* in the AI framework [TFW00]. The Update function of the LRU cache replacement policy in the abstract domain model the impact on the ACS of every reference inside a basic block and the Join function merges two ACS at join points. Note that in our analysis we only merge cache states in the end of loop iterations.

In Table 2.1, we indicated the list of the terms and functions needed to be defined for customizing the resource analysis framework. In the rest of this section, we will define these terms and functions for WCET analysis.

3.3.1 Basic Operations and Timing Cost Model

Recall from Chapter 2 that we model a program by a transition system. The set of basic operations, as the first customizable term, remains the same for WCET analysis.

In WCET analysis the variable of interest r models the execution time. The execution time is the summation of the execution time of the instructions and the cache access time, which can alter due to a cache hit or a cache miss.

The WCET analysis computes a sound and accurate bound for r in the end, across all feasible paths of the program. Note that this variable is always initialized to 0 and the only operation allowed upon it is a constant increment. Given a symbolic state $s \equiv \langle \ell, m, [\![s]\!] \rangle$ and a transition $tr : \ell \xrightarrow{op} \ell'$, the amount of increment at s by executing tr will be evaluated by the execution time of the LLVM instructions and the access time for seq. r is not used in any other way.

3.3.2 The Machine State

Since in this chapter we are limiting the machine state to the data and instruction cache, we define the symbolic state with the machine state assigned to each symbolic state to be a tuple of the form $\langle c_i, c_d \rangle$, where c_i and c_d are depicting instruction and data caches. As a result a symbolic state would be depicted by $\langle \ell, m, \sigma, \Pi \rangle$ where $m \equiv \langle c_i, c_d \rangle$.

In the domain of WCET analysis where time is the resource of interest and in the presence of caches, the updMachineState function would be equivalent to the standard Update function from the abstract cache semantics for *must* analysis from [TFW00]. Although the must analysis is originally only used to model the abstract instruction cache, since our framework performs loop unrolling, the must analysis is still expressive enough to model the abstract data cache.

Given a program point ℓ, an operation $op \in Ops$, and a symbolic store σ, the function $acc(\ell, op, \sigma)$ denotes the sequence of memory block accesses by executing op at the symbolic state $s \equiv \langle \ell, m, \sigma, \cdot \rangle$. In this chapter for short, we denote $acc(\ell, op, \sigma)$ by seq. While the program point ℓ identifies the instruction cache access, the sequence of data accesses are obtained by considering both op and σ together. We denote the updating machine state by $m' \equiv updMachineState(op, \sigma, m)$.

3.3.3 Witness and Dominating Condition

Intuitively, witness Γ is a path that depicts the WCET of a subtree. More specifically, it is depicted by $\Gamma \stackrel{\text{def}}{=} \langle t, \Upsilon, \pi \rangle$ where t is the (static) execution time of the instructions along the path assuming all the memory accesses are cache hits, Υ is the sequence of all memory accesses along the path, and π is the path constraints along the witness.

In case Υ contains consecutive accesses to the same memory block, all-but-first accesses in that sub-sequence can be classified as Always Hit and, importantly they will not affect the resulting cache state. As an optimization, we consider them redundant and remove them from Υ. In set-associative caches the chance that a memory access gets omitted in this way is more, because each memory access is checked against its consecutive accesses in the cache set. This helps reducing the size of Υ.

The timing of a witness is obtained dynamically from t and replaying the sequence Υ under an incoming cache state c. The feasibility of a witness w.r.t. to an incoming context is determined by checking if $[\![\pi]\!] \wedge [\![s]\!]$ is satisfiable. Throughout this thesis, we abbreviate $[\![\pi]\!] \wedge [\![s]\!]$ by $[\![\Gamma]\!]$.

We next discuss dominating condition δ, another customizable component of our analysis. Intuitively, this is a description of what cache configuration is needed in order that the witness *remains optimal* in a similar subtree. That is, in an analysis of the latter subtree, the witness remains the longest path. More specifically, the constraints in the dominating condition are either of the form $\mathsf{age}(m_i) < k$ or $\mathsf{age}(m_i) \geq k$, where age is a function returning the relative age of m_i in the cache and k is a non-negative integer. As an example, $\mathsf{age}(m_i) < A$ means the memory block m_i is in the cache; in contrast, $\mathsf{age}(m_i) \geq A$ indicates the memory block is not in the cache. We next present the customized Summarize-a-Trans, Combine-Witness and Merge-Witness functions.

```
function Summarize-a-Trans($s, tr$)
    Let $s$ be $\langle \ell, m, \sigma, . \rangle$ and Let $tr$ be $\ell \xrightarrow{op} \ell'$
⟨37⟩ $t$ := Execution-Time($op$); $\Upsilon := acc(\ell, op, \sigma)$
⟨38⟩ Iterate through $\Upsilon$ and remove repeating accesses
⟨39⟩ $i := 0; \mathcal{M} := \emptyset; w := t$
⟨40⟩ foreach $mem \in$ Reverse($acc(\ell, op, \sigma)$) do
⟨41⟩     Add $\langle mem, i \rangle$ into $\mathcal{M}; i := i + 1$
⟨42⟩     $w := w$ + Acc-Time($mem, m$)
    endfor
⟨43⟩ $\Delta_m := \langle \mathcal{M}, i \rangle$
⟨44⟩ return $[\ell, true, \langle t, \Upsilon, [\![op]\!]_\sigma \rangle, w, true, op_\Delta, \Delta_m]$
end function
```

Figure 3.1: Function to summarize a transition

Summarize-a-Trans Function

Summarize-a-Trans in Figure 3.1 computes a summarization for a single transition tr at state s. Because no infeasible path has been discovered, the interpolant Ψ is just $true$. There is a single path, thus the dominating condition is $true$ (the machine state is unconstrained). Moreover, the machine state summary for a single path is simply generated from the reverse order of the sequence of all memory accesses, namely $acc(\ell, op, \sigma)$, and i, increasing from 0, which indicates the relative age of the memory accesses in the instruction/data cache states in the end of the path (lines 39-41 and 43). Note that for brevity, here we demonstrate the steps to generate the machine state summary for a fully-associative cache. These steps can trivially be extended for set-associative caches. The abstract transformer Δ_p (for program variables) is the operation op itself, but translated to the language of input-output relation. As an example, x := x + 1 is translated to $x_{out} = x_{in} + 1$. We use op_Δ to denote such translated *op*.

```
function Combine-Witness($\Gamma_1, \Gamma_2, \delta_1, \delta_2$)
     Let $\Gamma_1$ be $\langle t_1, \Upsilon_1, \pi_1 \rangle$ and Let $\Gamma_2$ be $\langle t_2, \Upsilon_2, \pi_2 \rangle$
⟨45⟩ $t = t_1 + t_2$
⟨46⟩ if (Last($\Upsilon_1$) $\equiv$ first($\Upsilon_2$)) then $\Upsilon_2$ := Remove-First($\Upsilon_2$)
⟨47⟩ $\Upsilon = \Upsilon_1 \cdot \Upsilon_2$
⟨48⟩ $\pi := \pi_1 \wedge \pi_2$
⟨49⟩ $\delta'_2 := true$
⟨50⟩ foreach $\{\text{age}(mem_i) < k\} \in \delta_2$ do
⟨51⟩    $AlwaysTrueFlag = false$
⟨52⟩    foreach $mem_j \in \Upsilon_1$ do
⟨53⟩       if $mem_i \equiv mem_j$ then $AlwaysTrueFlag = true$
⟨54⟩       else if Conflict($mem_i, mem_j$) then $k := k - 1$
        endfor
⟨55⟩    if $AlwaysTrueFlag \equiv true$ then $skip$;
⟨56⟩    else if $k > 0$ then $\delta'_2 := \delta'_2 + \{\text{age}(mem_i) < k\}$
⟨57⟩    else if $k \leq 0$ then $\delta'_2 := \delta'_2 + false$
     endfor
⟨58⟩ foreach $\{\text{age}(mem_i) \geq k\} \in \delta_2$ do
⟨59⟩    $AlwaysFalseFlag = false$
⟨60⟩    foreach $m_j \in \Upsilon_1$ do
⟨61⟩       if $k \geq 0 \wedge mem_i \equiv m_j$ then $AlwaysFalseFlag = true$
⟨62⟩       else if Conflict($mem_i, m_j$) then $k := k - 1$
        endfor
⟨63⟩    if $AlwaysFalseFlag \equiv true$ then $\delta'_2 := \delta'_2 + false$
⟨64⟩    else if $k \geq 0$ then $\delta'_2 := \delta'_2 + \{\text{age}(mem_i) \geq k\}$
⟨65⟩    else if $k < 0$ then $skip$;
     endfor
⟨66⟩ $\delta = \delta_1 \wedge \delta'_2$
⟨67⟩ return $\{\langle t, \Upsilon, \pi \rangle, \delta\}$
end function
```

Figure 3.2: Combining (Vertically) Witnesses and Dominating Conditions

Combine-Witness and Merge-Witness Functions

In Figure 3.2, Combine-Witness produces a witness and a dominating condition, by compounding the witnesses and the dominating conditions of the two subtrees, where one suffixes the other. This can be understood as a *sequential* composition.

The static timing of the witness t is initialized as the sum of t_1 and t_2 (line 45). Let mem be the last access in Υ_1. If mem is also the first access in Υ_2, it would always be a cache hit and is removed from Υ_2 (line 46). The combined Υ is then the concatenation of Υ_1 and Υ_2 (line 47). Next, the witness path constraint π is computed as the conjunction of π_1 and π_2 (line 48).

The combined dominating condition δ is computed as the conjunction of δ_1 and a condition δ'_2, in line 66. Intuitively, δ'_2 describes a cache state m, such that if we perform all the accesses in Υ_1 on the cache states inside m, we will produce a cache state m' which satisfies δ_2.

The computation of δ'_2 is a precondition computation, but in the nature of caches. More specifically, δ'_2 is initialized to $true$ (line 49). Next, all the conditions in δ_2 are updated w.r.t. Υ_1. If a condition become always true, it is not added to δ_2 (lines 53 and 55 and line 65). Otherwise, k is decreased by the number of conflicting memory blocks in Υ_1 (line 54 and line 62) and the condition is added to δ'_2 (line 56 and 64). There is a special case, where a condition always resolves to *false*, thus, $false$ is added to δ'_2 (line 57). Lines 61 and 63 test for another similar case. As a result, δ'_2 would always resolve to false. This scenario rarely happened in our experiments.

In Figure 3.3, Merge-Witness produces a witness and a dominating condition, by compounding the witnesses and the dominating conditions of two sibling subtrees. We need to choose one witness as the dominating witness out of the two input witnesses. Moreover, the combined dominating condition must ensure the dominance of each witness (in its respective subtree) and the dominance of the chosen witness over the other witness.

The dominating condition δ is initialized as the conjunction of the two dominating conditions (line 68). We next compare the two WCET values; and we select the one with higher timing as the dominating witness. After line 70, the chosen witness and its corresponding WCET and dominating condition are captured by Γ_1, WCET$_1$ and δ_1.

Next, we test if δ is sufficient to ensure that Γ_1 dominates Γ_2. Given a condition δ, a witness *dominates* another witness if its minimum timing is more than the maximum timing of the other. The minimum timing is calculated by: (1) first determine some accesses in the Υ component are necessary misses as the consequence of the condition δ; (2) classifying the remaining accesses in Υ as cache hits. Whereas the maximum timing is calculated in the opposite manner: (1') first determine some accesses in the Υ component are necessary hits as the consequence of

```
function Merge-Witness($m, \Gamma_1, \Gamma_2, r_1, r_2, \delta_1, \delta_2$)
      Let $\Gamma_1$ be $\langle t_1, \Upsilon_1, \pi_1 \rangle$ and Let $\Gamma_2$ be $\langle t_2, \Upsilon_2, \pi_2 \rangle$
⟨68⟩ $\delta := \delta_1 \wedge \delta_2$
⟨69⟩ if $(r_1 < r_2)$
⟨70⟩    swap$(\Gamma_1, \Gamma_2)$, swap$(r_1, r_2)$, swap$(\delta_1, \delta_2)$
⟨71⟩ if $(t_1 + \text{Min-Time}(\Upsilon_1, \delta) \geq t_2 + \text{Max-Time}(\Upsilon_2, \delta))$
⟨72⟩    return $\{\Gamma_1, r_1, \delta\}$
⟨73⟩ foreach $mem_i \in \Upsilon_1$ do
⟨74⟩    if (Not-Constrained$(mem_i, \delta)$)
⟨75⟩        $\delta := \delta \wedge \{\text{age}(mem_i) \geq A\}$
⟨76⟩        if $(t_1 + \text{Min-Time}(\Upsilon_1, \delta) \geq t_2 + \text{Max-Time}(\Upsilon_2, \delta))$
⟨77⟩            return $\{\Gamma_1, r_1, \delta\}$
      endfor
⟨78⟩ foreach $mem_j \in \Upsilon_2$ do
⟨79⟩    if (Not-Constrained$(mem_j, \delta)$)
⟨80⟩        $\delta := \delta \wedge \{\text{age}(mem_j) < A\}$
⟨81⟩        if $(t_1 + \text{Min-Time}(\Upsilon_1, \delta) \geq t_2 + \text{Max-Time}(\Upsilon_2, \delta))$
⟨82⟩            return $\{\Gamma_1, r_1, \delta\}$
      endfor
end function
```

Figure 3.3: Merging Witnesses and Dominating Conditions

the condition δ; (2') classifying the remaining accesses in Υ as cache misses. This *dominance test* is shown in line 71.

If Γ_1 dominates Γ_2, then Γ_1 is returned as the dominating witness with δ as the dominating condition. If not, we need to further constrain the dominating condition δ.

First, for each access mem_i in Υ_1, if mem_i has not been constrained in δ, $\mathsf{age}(mem_i) \geq A$ is added to δ (lines 75). This cache constraint might increase the minimum timing of Γ_1 and lead to passing the dominance test. If the dominance test indeed succeeds, Γ_1, WCET_1 and δ are returned.

If we have not succeeded yet, we can do similarly for each mem_j in Υ_2. Note the difference that now we add the cache constraint of the form $\mathsf{age}(mem_j) < A$, with the hope to reduce the maximum timing of Γ_2 enough that the dominance test can be passed (line 81).

At the end of the first **for** loop, $\mathsf{Min}(\Upsilon_1, \delta)$ would be larger than (or equal to) the original timing of Γ_1 (w.r.t. cache context c) while at the end of the second **for** loop, $\mathsf{Max}(\Upsilon_2, \delta)$ would be less than (or equal to) the original timing of Γ_2 (w.r.t. cache context c). In other words, eventually, we will end up with a condition δ so that Γ_1 dominates Γ_2.

3.3.4 Machine State Summary

In the WCET analysis presented in this chapter, the machine state summary would only capture the relation between the abstract cache components. In short, we will call it as cache summary in this chapter. The cache summary presented in this section can be used as part of the machine summary for other resource analyses presented in the subsequent chapters. The cache summary is one of the contributions of the work presented in this chapter and to the best of our knowledge it is the first time that the concept of summarizing the cache for program analysis is presented[1].

Suppose the state is at program point ℓ_1 is s and a summarization of $subtree(s, \ell_2)$ is reused at another visit to ℓ_1 with an incoming cache state c_1. Then c_2, the cache state at ℓ_2, can be generated by applying the cache abstract transformer to c_1. That is, the transformer over-approximates the memory accesses along the feasible paths, which start from s and end at ℓ_2.

Let us first review on abstract set-associative cache for must analysis. An abstract set-associative cache c is consisted of N cache sets, where $N = C/(BS{*}A)$, C is the cache capacity and BS is block size. Specifically, c is $[cs_1, ..., cs_N]$ where each cs_j is a cache set. In turn, each cache set is a set of cache lines, i.e., $cs = [l_1, ..., l_A]$. We use $cs(l_i) = mem$ to indicate the presence of a memory block mem in a cache-set, where i describes the relative age of the memory block and not the physical position in the cache hardware. The cache abstract transformer Δ_c is partitioned to N independent abstract transformers of respective cache-sets $\Delta_c \equiv [\Delta_{s_0}, ...\Delta_{s_{N-1}}]$. In the process of applying a cache abstract transformer on a cache state, each abstract transformer of a cache-set is applied to the corresponding cache-set.

Each cache-set abstract transformer Δ_s is depicted by $\langle \mathcal{M}, n \rangle$, where $\mathcal{M}$ is a set of pairs $\langle mem, i \rangle$. Each pair indicates a memory block mem and its relative age in the cache i. Moreover, n depicts the number of cache lines that the memory blocks in $\mathcal{M}$ are loaded to. It is computed as the maximum i in the sequence $\mathcal{M}$

[1]We would like to also mention that cache summary in our work is different from the concept of non-preemptive fixed-priority scheduling where execution times depend on history from [A$^+$12c]. The concept of cache summary presented in this section foresees the sequence of memory accesses in a sub-tree based on a similar sub-tree, explored in the past, which can be quite different from the idea of using history for task scheduling.

plus 1. While computing the cache abstract transformer, only the memory blocks with an age less than the associativity A are stored. The rest of the memory blocks would naturally be pushed out of the cache and we do not need to maintain them in the abstract transformer. Thus, it is always true that $n <= A$ and the size of the cache abstract transformer is *linear* w.r.t. the cache capacity.

At the time of reuse, each $\Delta_s \equiv \langle \mathcal{M}, n \rangle$ in the cache abstract transformer is applied to its respective cache-set. First, the memory blocks in the cache-set are aged n times. Next, for each pair $\langle mem, i \rangle$ in $\mathcal{M}$, the memory block mem is loaded to its cache-set with relative age i. Considering that the pairs in $\mathcal{M}$ maintain the memory blocks accessed in $subtree(s, \ell_2)$, the cache abstract transformer simulates the updates and merges of the cache state along the paths in $subtree(s, \ell_2)$.

Example 6. *For example, consider the sequence of memory blocks $\langle m_1, m_2, m_3, m_2 \rangle$. Assume, the cache summary of this sequence of memory blocks for a fully associative cache would be $\langle 3, ([m_2, 0], [m_3, 1], [m_1, 2]) \rangle$. Now, consider a fully associative cache of size 4 which initially contains m_0:*

0	*1*	*2*	*3*
m_0			

By applying the memory block sequence $\langle m_1, m_2, m_3, m_2 \rangle$ to the cache the final cache state would be:

0	*1*	*2*	*3*
m_2	m_3	m_1	m_0

If we apply the cache summary $\langle 3, ([m_2, 0], [m_3, 1], [m_1, 2]) \rangle$ to the initial cache state, in the first step the items which are already loaded into the cache are aged by n (the static number stored in the cache summary which is 3):

0	*1*	*2*	*3*
			m_0

Next, each of the $[m_2, 0], [m_3, 1], [m_1, 2]$ *tuples in the cache summary are loaded into the cache with their relative ages:*

0	*1*	*2*	*3*
m_2	m_3	m_1	m_0

As it can be seen, the generated cache state would be the same to the cache state as if we had loaded the memory blocks one by one.

The size of the cache summary is of the order of $O(Capacity)$, where $Capacity$ is the cache capacity. It is so, since, in the process of generating the cache summary only the memory blocks which their relative age is less than or equal to the cache *associativity* are stored. The rest of the memory blocks would be naturally pushed out of the cache. As a result, while our analysis remains sound, the size of the cache summaries will remain constant w.r.t. to the cache capacity. Next, we will describe how to generate a cache summary.

Combine-Summary and Merge-Summary Functions

Consider Combine-Summary in Figure 3.4. The cache abstract transformer is first initialized to the cache abstract transformer of the prefix subtree (line 83). Next, since the memory accesses in the suffix subtree are more recent along the feasible paths, before adding them to the transformer (line 89), all the previous memory accesses in $reverse(\mathcal{M})$ are aged by 1 (line 86), while mem itself is not visited. If mem is visited, due to a more recent access ($\langle m, 0\rangle$) it is removed from $\mathcal{M}$ (lines 87). Finally, the new value for n is calculated (line 91). Note that storing the pairs

```
function Combine-Summary($\Delta_{m1}$,$\Delta_{m2}$)
     Let $\Delta_{m1}$ be $\langle \mathcal{M}_1, n_1 \rangle$ and Let $\Delta_{m2}$ be $\langle \mathcal{M}_2, n_2 \rangle$
⟨83⟩ $\mathcal{M} := \mathcal{M}_1; n := n_1$
⟨84⟩ foreach $\langle mem, k \rangle \in \mathcal{M}_2$ do
⟨85⟩     foreach $\langle mem', i \rangle \in reverse(\mathcal{M})$ do
⟨86⟩         if $mem' \not\equiv mem$ then update $\langle mem', i \rangle$ to $\langle mem', i+1 \rangle$
             else
⟨87⟩             Remove $\langle mem, k \rangle$ from $\mathcal{M}$
⟨88⟩             break
         endfor
⟨89⟩     Add $\langle mem, 0 \rangle$ to the beginning of $\mathcal{M}$
     endfor
⟨90⟩ foreach $\langle m, i \rangle \in \mathcal{M}$ do
⟨91⟩     if $i > n$ then $n := i + 1$
⟨92⟩     if $i \geq A$ then Remove $\langle mem, i \rangle$ from $\mathcal{M}$; $n := A$
     endfor
⟨93⟩ return $\langle \mathcal{M}, n \rangle$
end function

function Merge-Summary($\Delta_{m1}$,$\Delta_{m2}$)
     Let $\Delta_{m1}$ be $\langle \mathcal{M}_1, n_1 \rangle$ and Let $\Delta_{m2}$ be $\langle \mathcal{M}_2, n_2 \rangle$
⟨94⟩ $\mathcal{M} := \emptyset$
⟨95⟩ foreach $\langle mem, i \rangle \in \mathcal{M}_1 \wedge \langle mem, j \rangle \in \mathcal{M}_2$ do
⟨96⟩     $\mathcal{M} := \mathcal{M} + \langle mem, \max(i, j) \rangle$
     endfor
⟨97⟩ return $\langle \mathcal{M}, \max(n_1, n_2) \rangle$
end function
```

Figure 3.4: Combining (Vertically) and Merging (horizontally) Machine State Summaries

with relative age more than the associativity A would be redundant. Such pairs are removed from $\mathcal{M}$ in line 92 and the value of n is updated accordingly.

Merge-Summary in Figure 3.3 joins two cache state summaries. Since the cache states in the cache states are updated based on the semantics of abstract cache for must analysis, similarly the intersection of the memory accesses on the respective cache states are preserved with their maximum age. The memory access sequence $\mathcal{M}$ is initialized to $\emptyset$. Next, for each memory access m that is in the memory access sequences of the left and right subtrees $\mathcal{M}_1$ and $\mathcal{M}_2$, it is added to $\mathcal{M}$ with the maximum age from $\mathcal{M}_1$ and $\mathcal{M}_2$ (line 96). Finally, $\mathcal{M}$ is returned with the maximum number of cache lines that memory accesses in $\mathcal{M}$ will be loaded to $\mathsf{max}(n_1, n_2)$ (line 97).

Extension to other Cache Policies

As stated in the beginning of this section, our framework performs instruction and data cache analysis with LRU cache policy. In order to extend the cache analysis to support other cache policies, a sound cache summarization should be defined. Our inspection shows that if the memory blocks in a cache set can be ordered (by a ranking), a sound cache summarization can be defined for them. As a result, our framework can be extended to other cache policies such as MRU or FIFO where memory blocks are ranked based on the order they entered the cache. However, our framework seems not to be extendable to PLRU cache policy. Finally, we like to note that our framework can support cache hierarchies while it remains scalable. In order to define cache hierarchies, each cache hierarchy is added as an abstract cache to the machine state and the update function should be defined such that all cache states are updated with respect to the hierarchy at each transition. As a result our framework does not need to change substantially when changing to other cache policies.

3.4 Example Analysis

Consider the CFG and the symbolic execution tree in Figure 3.5. Here we assume a direct-mapped cache with 3 cache sets, initially empty, and a *cache miss penalty* of 10 cycles. Consider accesses to memory blocks m_1, m_2, m_3, and m_4, where only m_1 and m_3 conflict with each other in the first cache set. Note that in Figure5.8(b),

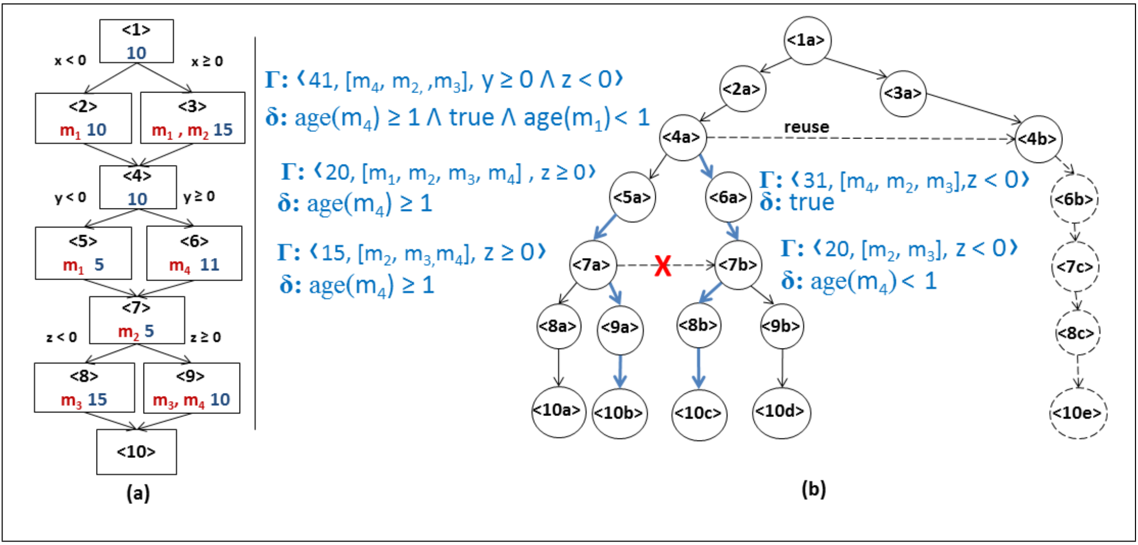


Figure 3.5: (a) a CFG (with memory accesses and static instruction timing shown in each block); and (b) Our Analysis Tree

we have not (fully) drawn the subtree below node $\langle 4b\rangle$.

Suppose the subtree $\langle 7a\rangle$ has been analyzed, and its summarization is $[\langle 7\rangle, \overline{\Psi}, \Gamma, w, \delta, \Delta_p, \Delta_c]$. We now explain the components of this summarization. The interpolant Ψ is easily determined as *true* because all (two) paths of this subtree are feasible. Next, because the incoming cache state contains only m_1, the timing of the sub-path $\langle 7a\rangle$, $\langle 8a\rangle$, $\langle 10a\rangle$ is $40 = (\mathbf{10} + 5 + \mathbf{10} + 15)$, with both accesses as cache misses. Similarly, the timing of the other sub-path $\langle 7a\rangle$, $\langle 9a\rangle$, $\langle 10b\rangle$ is $45 = (\mathbf{10} + 5 + \mathbf{10} + \mathbf{10} + 10)$. So, the sub-path $\langle 7a\rangle$, $\langle 9a\rangle$, $\langle 10b\rangle$ is longer than the other and it is chosen as the worst-case path[2] of subtree $\langle 7a\rangle$. Consequently, the witness Γ is computed as $\langle 15, [m_2, m_3, m_4], z \geq 0\rangle$, where 15 is the static timing of the witness path, $[m_2, m_3, m_4]$ are the memory accesses along the path, and $z \geq 0$ is the (partial) path constraints of the path. Moreover, the WCET of the subtree (w) is 45. Next, we capture a dominating condition (δ) as $\mathsf{age}(m_4) \geq 1$ (The cache associativity of a direct-mapped cache is 1). This condition is sufficient to ensure

[2]When it is clear, we often use "path" to mean "sub-path".

that the chosen path dominates (i.e., is longer than) any other path in the subtree.

The abstract transformer Δ_p is the trivial one where the output is the same as the input. This is because in this example we abstract away all the instructions executed by the basic blocks. The memory blocks m_2 and m_3 are accessed along the sub-path $\langle 7a\rangle$, $\langle 8a\rangle$, $\langle 10a\rangle$. The abstract transformer of the first cache set is $\Delta_{s0} \equiv \langle[\langle m_3, 0\rangle], 1\rangle$, where $\langle m_3, 0\rangle$ indicates the relative age of m_3 in the first cache set of the cache state at $\langle 10\rangle$ and 1 shows the number of cache lines that the memory blocks are loaded to. Similarly, the relative age of m_2 in the second cache set is 0 and the transformer of the second cache set is $\Delta_{s1} \equiv \langle[\langle m_2, 0\rangle], 1\rangle$. The abstract transformer of the third cache set Δ_{s2} is empty $\langle[\,], 0\rangle$. In a similar manner, the set abstract transformers for the sub-path $\langle 7a\rangle$, $\langle 9a\rangle$, $\langle 10b\rangle$ are $\Delta_{s0} \equiv \langle[\langle m_3, 0\rangle], 1\rangle$, $\Delta_{s1} \equiv \langle[\langle m_2, 0\rangle], 1\rangle$ and $\Delta_{s2} \equiv \langle[\langle m_4, 0\rangle], 1\rangle$. The respective set abstract transformers are joined at $\langle 7a\rangle$. The joined abstract transformer would maintain the common memory block accesses from both abstract transformers and the maximum of the number of cache lines where memory blocks are loaded to. The memory accesses m_2 and m_3 are the common accesses on both sub-paths, so $\Delta_c \equiv [\Delta_{s0}, \Delta_{s1}, \Delta_{s2}] \equiv [\langle[\langle m_3, 0\rangle], 1\rangle, \langle[\langle m_2, 0\rangle], 1\rangle, \langle[\,], 1\rangle]$.

In short, after analyzing $\langle 7a\rangle$, we also have computed a summarization $[7, \mathit{true}, \langle 15, [m_2, m_3, m_4], z \geq 0\rangle, 45, \mathsf{age}(m_4) \geq 1, \mathit{Id}(\mathit{Vars}), \Delta_c]$. For brevity, in what follows, we do not detail on abstract transformers Δ_p and Δ_c.

Next we propagate the analysis of $\langle 7a\rangle$ to its parent $\langle 5a\rangle$ whose summarization is now updated so that the witness is stored in the form $\langle 20, [m_1, m_2, m_3, m_4], z \geq 0\rangle$, where 20 is computed as the sum of: (1) the static timing of block $\langle 5\rangle$, which is 5; (2) the static timing of the witness for $\langle 7a\rangle$, which is 15. The dominating condition is $\mathsf{age}(m_4) \geq 1$, as before.

We fast forward to node $\langle 7b \rangle$, and consider now if the above analysis of $\langle 7a \rangle$ can be *reused*. That is, even though we have depicted the subtree $\langle 7b \rangle$ in full, could we in fact have simply declared that the witness in the subtree below $\langle 7b \rangle$ would remain the same as the witness in subtree below $\langle 7a \rangle$? (Recall that the witness in the subtree below $\langle 7a \rangle$ spans along the program points $\langle 7 \rangle$, $\langle 9 \rangle$, $\langle 10 \rangle$.) Unfortunately, the answer is negative, and the reason is that the dominating condition, $\mathsf{age}(m_4) \geq 1$, is not met because m_4 is in the cache at $\langle 7b \rangle$. This non-reuse is depicted by a red cross. We thus have to analyze $\langle 7b \rangle$ fully. We get a different longest sub-path this time, $\langle 7b \rangle$, $\langle 8b \rangle$, $\langle 10c \rangle$, with the witness $\langle 20, [m_2, m_3], z < 0 \rangle$. The dominating condition is also different: $\delta : \mathsf{age}(m_4) < 1$.

Finally, this analysis of $\langle 7b \rangle$ is propagated for its parent $\langle 6a \rangle$. The dominating condition is $\mathsf{age}(m_4) < 1$ which always holds due to the access of m_4 at $\langle 6 \rangle$. Thus the dominating condition for $\langle 6a \rangle$ is simply *true*.

Having now analyzed both $\langle 5a \rangle$ and $\langle 6a \rangle$, we can now compute an analysis for their common parent $\langle 4a \rangle$. Here the observed longest sub-path is $\langle 4a \rangle$, $\langle 6a \rangle$, $\langle 7b \rangle$, $\langle 8b \rangle$, $\langle 10c \rangle$, and the witness is stored as $\langle 41, [m_4, m_2, m_3], y \geq 0 \wedge z < 0 \rangle$. The dominating condition is conjoined from: (a) the dominating condition of its left child $\langle 5a \rangle$; (b) the dominating condition of its right child $\langle 6a \rangle$; and (c) the reason for the dominance of the above observed longest path over the other path. In particular, δ is $\mathsf{age}(m_4) \geq 1 \wedge \mathit{true} \wedge \mathsf{age}(m_1) < 1$.

Now we can exemplify reuse on the subtree $\langle 4b \rangle$. We first check if the context of $\langle 4b \rangle$ implies the interpolant computed for $\langle 4a \rangle$. Because all paths from $\langle 4a \rangle$ are feasible, the interpolant is *true*, thus, it trivially holds. We then check if the dominating condition holds. Examining the cache context of $\langle 4b \rangle$, indeed m_1 is in the cache and m_4 is not in the cache. Furthermore, the witness is still feasible w.r.t. the incoming context ($x \geq 0$). So we can reuse the *witness* of $\langle 4a \rangle$, yielding the timing of 61. We remark here that the timing of the sub-path $\langle 4b \rangle$, $\langle 6b \rangle$, $\langle 7c \rangle$, $\langle 8c \rangle$, $\langle 10e \rangle$ is less than the timing of $\langle 4a \rangle$,$\langle 6a \rangle$, $\langle 7b \rangle$, $\langle 8b \rangle$, $\langle 10c \rangle$ because now m_2 is present in the cache at $\langle 4b \rangle$.

Finally, we easily arrive at the WCET of the entire tree, thus, the entire example program, to be 106 cycles (= 10 + **10** + **10** + 15 + 61, since the accesses to m_1 and m_2 at $\langle 3a \rangle$ are cache miss).

Let us reconsider the same example using a *pure* abstract interpretation (AI) framework such as [TFW00]. A pure AI method would typically perform merging at the three join points: $\langle 4 \rangle$, $\langle 7 \rangle$, $\langle 10 \rangle$. Importantly, it discovers that at $\langle 4 \rangle$, m_1 *must* be in the cache. Thus, the access to m_1 at $\langle 5 \rangle$ is hit. However, at $\langle 7 \rangle$, AI has to conservatively declare that m_4 is not in the cache. As a result the access to m_4 at $\langle 9 \rangle$ will be cache miss. Consequently, the final worst case timings for the basic blocks that have some memory accesses are: ($\langle 2 \rangle$,**20**), ($\langle 3 \rangle$,**35**), ($\langle 5 \rangle$,**5**), ($\langle 6 \rangle$,**21**), ($\langle 7 \rangle$,**15**), ($\langle 8 \rangle$,**25**), ($\langle 9 \rangle$,**30**).

If we aggregate using a path-insensitive high-level analysis (such as [CJ11]), the WCET estimate is 121 (= 10 + max(20, 35) + 10 + max(5, 21) + 15 + max(25,30)). We cannot improve the estimate for this example, since the timing of the witness w.r.t. new context remains the same. However, as our experiments in the next section shows, in general, the timing generated by our framework is more accurate.

3.5 Experimental Evaluation

The data and instruction cache settings in our experiments is borrowed from [wtc14] for ARM9 target processor. Our instruction and data caches are separate. A cache state c contains two separate abstract caches $\langle c_i, c_d \rangle$, where c_i is a 4KB abstract instruction cache and c_d is a 4KB abstract data cache. The cache configurations are write-through, with no-write-allocate, 4-way set associative L1 cache with LRU replacement policy. The cache miss and cache hit latencies are respectively 10 and 0 cycles.

Because we fully unroll loops in our analysis, it is sufficient to employ a must analysis for precisely tracking the data cache, as opposed to a persistence analysis. We follow the treatment as in [FW98] for loading memory ranges into the cache for persistence analysis[3] when a data access cannot be resolved to a single memory address, meaning that the blocks in the memory address range are not loaded into the cache, but the blocks already in the cache are relocated as if all the blocks in the memory address range were loaded into the cache.

3.5.1 Results

We used an Intel Core i5 @ 3.2 Ghz processor having 4Gb RAM for our experiments and built our system upon CLP($\mathcal{R}$) [JMSY92] and Z3 as the constraint solver, thus providing an accurate test for feasibility. The analysis was performed on LLVM IR which, while being expressive enough, a program's transition system can be easily constructed. The LLVM instructions are simulated for a RISC architecture. We use Clang 3.2 [Cla14] to generate the IR.

[3]Huynh et. al. in [H$^+$11] have fixed a safety issue with the treatment of loading memory ranges into the cache from [FW98]. However, this safety issue occurs in the semantics of abstract cache for persistence analysis and does not affect the semantics of abstract cache for must analysis, which is used by our method.

Table 3.1 presents the results of the comparison of our method with the two state-of-the-art algorithms explained in Section 3.2.5:

• **`AI+SAT`$\oplus$`ILP`** implements the algorithm in [BCR13]. It comprises the state-of-the-art micro-architectural modeling (AI+SAT) combined with an ILP formulation for WCET aggregation.

• **`AI+SAT`$\oplus$`Unroll_s`** implements a *hypothetical* algorithm, to benefit from combining the `AI+SAT` low-level analysis and the high-level analysis in [CJ11]. This combined algorithm generates *static timing* for each basic block before aggregating results via a path analysis phase.

• **`Unroll_d`** is the algorithm presented in this chapter. This further improves on the already quite accurate hypothetical algorithm above because we now accommodate *dynamic timing*. As explained in the earlier sections, this entails more cost. Our results below show that this cost is bearable.

In Table 3.1, the columns **T(s)** and **State** denote the running time and number of states (in symbolic execution) respectively. The symbol ∞ denotes out-of-memory. The WCET precision improvement is computed as $\frac{B-U}{B} \times 100\%$, where U is the WCET obtained using our analysis algorithm, and B is the WCET obtained using the baseline approach. In order to highlight the importance of reuse, we tabulate separate results for the cases where it is employed or not. The last two columns, separated by a vertical double line, summarize the improvement of `Unroll_d` over the other two analyses.

We have divided our benchmark programs, which are quite standard in evaluating WCET analysis algorithms, into three groups, separated by horizontal double lines:

Table 3.1: Comparing our Algorithm (`Unroll_d`) to the State-of-the-art

Benchmark	**LLVM**	`AI⊕ILP`		`AI⊕Unroll_s`		`Unroll_d`						`Unroll_d` vs	
						w. reuse			w.o. reuse			`AI⊕`	`AI⊕Un`
	LOC	**T(s)**	**WCET**	**T(s)**	**WCET**	**T(s)**	**State**	**WCET**	**T(s)**	**State**	**WCET**	`ILP`	`roll_s`
tcas	736	0.84	1427	9.07	1212	21.36	2389	1112	-	∞	-	22.07%	8.25%
nsichneu	12879	161.58	85845	504.88	66808	709.03	3776	48388	-	∞	-	43.63%	27.57%
statemate	3345	13.89	12382	248.41	9101	358.94	4152	7644	-	∞	-	38.27%	16.01%
ndes	1755	11.45	304369	37.95	174266	38.92	1065	148368	-	∞	-	51.25%	14.86%
fly-by-wire	2459	1.32	12171	10.97	9761	11.16	279	8751	-	∞	-	28.10%	10.35%
adpcm	2876	4.82	39088	106.53	33676	118.92	1617	31574	-	∞	-	19.22%	6.24%
compress	1334	9.18	478191	179.43	31665	204.82	1622	28670	911.38	10984	28180	94.00%	9.46%
ud	536	2.08	33515	1.87	15792	1.77	638	12132	1.96	797	12092	63.80%	23.18%
janne_ complex	119	0.13	1718	0.11	1219	0.16	98	1119	0.26	137	1119	34.87%	8.20%
fft1	1346	6.43	403179	45.62	280188	48.13	966	268378	100.52	1820	268378	33.43%	4.22%
bsort100	128	0.14	752681	9.4	752630	19.34	1440	637580	-	∞	-	15.29%	15.29%
edn	1226	1.47	437158	534.28	437158	676.11	2369	321028	-	∞	-	26.56%	26.56%
cnt	269	0.17	21935	0.29	21935	0.44	230	19355	1.56	1426	19355	11.76%	11.76%
matmult	286	1.75	874348	5.38	874348	6.5	906	621458	-	∞	-	28.92%	28.92%
jfdctint	693	0.08	20332	1.02	20332	1.43	254	17572	0.9	328	17572	13.57%	13.57%
fdct	831	0.08	17442	0.05	17442	0.13	58	14572	0.04	70	14572	16.45%	16.45%

Benchmarks with lots of Infeasible Paths: The first group contains `statemate` and `nsichneu` from Mälardalen benchmarks [MÖ6] and `tcas`, a real life implementation of a safety critical embedded system. `tcas` is a loop-free program with *many* infeasible paths, which is used to illustrate the performance of our method in analyzing loop-free programs. On the other hand, `nsichneu` and `statemate` are programs which contain loops of big-sized bodies, also with many infeasible paths. These benchmarks are often used to evaluate the scalability of WCET analysis algorithms [W+08].

Standard Timing Analysis Benchmarks with Infeasible Paths: This group contains standard programs from [MÖ6], and `fly-by-wire` from [N+06].

Benchmarks with Simple Loops: This group contains a set of academic programs from [MÖ6]. Though the loops in these programs are simple for high-level analysis, they contain memory accesses that a fixed-point computation might resolve to a range of memory addresses, leading to imprecise low-level WCET analysis.

3.5.2 Discussion on Precision

The generated WCET by `Unroll_d` for the first group of benchmarks, compared to `AI+SAT⊕ILP`, on average is improved by 34%; compared to `AI+SAT⊕Unroll_s`, the number is 17%. Focusing on `nsichneu` and `statemate`, it can be seen that part of the improvement of `Unroll_d` over `AI+SAT⊕ILP` comes from the detection of infeasible paths (i.e., the common improvement between `Unroll_d` and `AI+SAT⊕Unroll_s` over `AI+SAT⊕ILP`). The improvement of `Unroll_d` over `AI+SAT⊕Unroll_s`, on the other hand, is due to infeasible path detection directly reflected in the tracking of micro-architectural states. This avoids lossy merging of cache states at the join points in the CFG.

For a loop-free program like `tcas`, the improvement of `Unroll_d` over the other two analyses is clearly not advantaged by tighter loop bounds in unrolling, nor disadvantaged by fixed-point computation in `AI+SAT`$\oplus$`ILP`. Next, consider the fact that the (high-level) infeasible paths detected by `Unroll_d` and `AI+SAT`$\oplus$ `Unroll_s` are the same. Even so, `Unroll_d` is more accurate by 8%. Once again, this improvement comes from our integration of low-level analysis with high-level analysis, making infeasible path detection reflected in the precise tracking of micro-architectural states.

For benchmarks in the second group, `Unroll_d` produces significantly more accurate WCET than `AI+SAT`$\oplus$`ILP`, on average 42%, peaking at 94%. In `compress`, `ud` and `ndes`, many infeasible paths have to do with loops, and being able to detect them improves the WCET estimates dramatically. `AI+SAT`$\oplus$`Unroll_s` performs relatively well on this group of benchmarks. However, for `ud`, `ndes` and `fly-by-wire`, the accuracy improvement of `Unroll_d` over `AI+SAT`$\oplus$`Unroll_s` is still noticeable. Further investigation reveals that two of these benchmarks contain memory accesses which are resolved to address ranges in the `AI` component – ultimately is still a fixed-point computation – leading to imprecise analysis results from the combined algorithm.

The effect of such memory accesses on analysis precision can be seen more clearly by examining the third benchmark group. `Unroll_d` is still better than the other two algorithms by 18% on average. These benchmarks do not contain many infeasible paths nor complicated loops and that is the reason why `AI+SAT`$\oplus$ `Unroll_s` does not produce better estimates than `AI+SAT`$\oplus$`ILP`. However, these benchmarks contain memory accesses which are resolved to address ranges in a fixed-point computation, leading to the imprecision of AI+SAT$\oplus$ILP. In contrast, Unroll_d performs loop unrolling, thus it can precisely resolve the addresses of the accesses, leading to superior precision.

In summary, in terms of precision, Unroll_d outperforms the other two algorithms in all benchmarks. The WCET estimations from Unroll_d have improved 33% on average compared to AI+SAT$\oplus$ILP and 14% on average compared to AI+SAT$\oplus$Unroll_s. These improvements clearly uphold our proposal that performing WCET analysis in one integrated phase in the presence of *dynamic timing* will enhance the precision over modular approaches. However, the scalability of our method is not yet discussed.

3.5.3 Discussion on Scalability

As expected, reuse is important for scalability. For most of the benchmarks (8 out of 12) the analysis cannot finish without reuse. Between the benchmarks in the first group which contain many infeasible paths (tcas, nsichneu and statemate), none of the benchmarks can be analyzed without reuse. The two largest benchmarks, nsichneu and statemate, are used as an indicator of the scalability of the WCET tools. The WCET analysis for nsichneu and statemate, uses at most 53% and 40% of the 4GB available. It is worth noting that, for nsichneu, the overhead of the analysis time and memory usage compared to AI+SAT$\oplus$Unroll_s is 31% and 40%, respectively, while the precision is improved by 27%.

In conclusion, our analysis framework relies a lot on reuse for scalability. From these experiments we can infer that only small size programs where the number of paths is limited can be analyzed without reuse. In the next section, we will elaborate on the point that similar to the symbolic simulation algorithm in [CJ11], our integrated analysis remains super-linear with regard to both time and space complexity.

3.5.4 Analysis of Benchmarks with Complicated Loops

We have performed a set of experiments on benchmarks containing nested or complicated loops from [M&ouml;06]. These benchmarks are analyzed not only to show the ability of our analyzer to generate tight WCET for the benchmarks, but also to maintain the super-linearity of the symbolic execution WCET analyzer in the integrated WCET analysis.

Table 3.2, presents the results of the comparison of our symbolic execution analysis, `Unroll_d`, with `AI+SAT`$\oplus$`ILP` and `AI+SAT`$\oplus$`Unroll_s` for these benchmarks. In this table, the WCET estimated by `Unroll_d` compared to `AI+SAT`$\oplus$`ILP` on average is improved by 50% and peaks at 97% for expint and insertsort. From the benchmarks in this group, bubblesort, expint, fft1, fir, insertsort, janne_complex and ud contain complicated loops which the number of the iterations of the loops is related to the context reaching the loop. In order to have a fair comparison between the three analyses, we provided the ILP formulation with the tightest possible loop bounds. Our aim was to eliminate the effect of the loop bounds for complicated loops on the estimated WCET. We like to elaborate on the results of the analyses for fir and bubblesort. These two benchmarks, while containing complicated loops, possess the least improvement. It is noteworthy, that the generated LLVM IR for insertsort, showing the best improvement in the generated WCET, is slightly different from the CFG of the C code. This has resulted in a much more costly path with less flow to receive maximum flow in the ILP formulation. The `AI+SAT`$\oplus$ `Unroll_s` analysis is able to tackle this issue in the high level analysis, but the issue still remains in the low-level analysis due to the fixed point computation. However, `Unroll_d` is able to generate a more precise WCET estimation.

Table 3.2: The results of our analysis, `Unroll_d`, compared to `AI+SAT`⊕`Unroll_s` and `AI+SAT`⊕`ILP` to Illustrate the Superlinearity of Loop Unrolling

Benchmark	**Size**	`AI+SAT⊕ILP`		`AI+SAT⊕Unroll_s`		`Unroll_d`			
		Time(s)	**WCET**	**Time(s)**	**WCET**	**Time(s)**	**WCET**	**vs AI+SAT ⊕Unroll_s**	**vs AI+SAT ⊕ILP**
fir		0.09	17977	0.25	14247	0.34	13957	2.04%	22.36%
ud	5	2.08	33515	1.87	15792	1.77	12132	23.18%	63.80%
janne_complex		0.13	1718	0.11	1219	0.16	1119	8.20%	34.87%
expint		0.14	453451	19.38	16831	28.16	16791	0.24%	96.30%
fft1	8	6.43	177059	5.53	74524	3.36	71514	4.04%	48.57%
fft1	16	6.43	266821	12.07	143342	10.74	137232	4.26%	33.43%
fft1	32	6.43	403179	45.62	280188	48.13	268378	4.22%	63.80%
bubblesort	25	0.14	46706	0.43	46655	0.82	37715	19.16%	19.25%
bubblesort	50	0.14	187656	1.72	187605	3.27	156285	16.69%	16.72%
bubblesort	100	0.14	752681	9.4	752630	19.34	637580	15.29%	15.29%
ns	5	4.94	100416	3.04	100416	0.57	82836	17.51%	17.51%
ns	10	4.94	1506020	3.34	1506020	1.12	1266420	15.91%	15.91%
ns	20	4.94	2338610	4.31	2338610	2.71	20674700	11.59%	11.59%
insertsort	25	0.12	68969	0.41	10124	0.64	6994	30.92%	89.86%
insertsort	50	0.12	294994	1.94	21124	3.48	14524	31.24%	95.08%
insertsort	100	0.12	1219540	11.95	43124	26.54	29584	31.40%	97.57%

The estimated WCET in all of these benchmarks show a significant improvement. Part of the improvement comes from the ability to find the exact number of loop iterations for complicated loops. This improvement is shared between our integrated analysis, `Unroll_d` , and `AI+SAT`$\oplus$`Unroll_s`. However, comparing `Unroll_d` and `AI+SAT`$\oplus$`Unroll_s`, the estimated WCET on average is improved by 12% and peaks at 31% for insertsort. The extra improvement comes from the integrity of the WCET analysis.

Finally, we like to note one other contribution of our method, which is performing loop unrolling in super-linear time. Our analysis shares this contribution with `AI+SAT`$\oplus$`Unroll_s` . It is reached through depth-wise loop compression, i.e., reusing across the loop iterations and generating and using compound summarizations where many loop iterations can be summarized at once. This summarization can be used later to avoid exploring large portions of loops. By comparing the size parameter of the benchmarks in table 3.2 (second column) with the analysis time it can be clearly seen that the super-linear complexity of the brute-force loop unrolling remains in our integrated analysis. For example, in ns the complexity is $O(n^4)$ while the analysis is performed in a super-linear complexity.

3.5.5 Comparison of Analysis Results with Simulations

In this section, we address the hypothesis that our analysis tool generates realistic WCET bounds. In order to test this hypothesis, We have simulated the benchmarks from Table 3.1 on the cycle accurate GEM5 simulator [B$^+$11] and compared the results.

GEM5 is an emerging simulator which is able to evaluate different architectural designs. Our simulations were performed on TimingSimpleCPU, which supports timing accesses memory, 4kB L1 instruction cache and 4kB L1 data cache, ARM instruction set and simple memory format, which was the closest to the hardware settings of our experiments. The static binaries passed to GEM5 were cross compiled with arm-linux-gnueabihf-gcc.

Figure 3.6 presents the comparison of the generated WCET and the results of the simulation of the benchmarks. As it can be seen the simulations and the generated WCET in most cases follow the same trend, which verifies our hypothesis. However, for nsichneu, ndes and edn there is a difference in the simulation result and the generated WCET. Our further investigation demonstrates that this difference can come from the different software characteristics of these benchmarks.

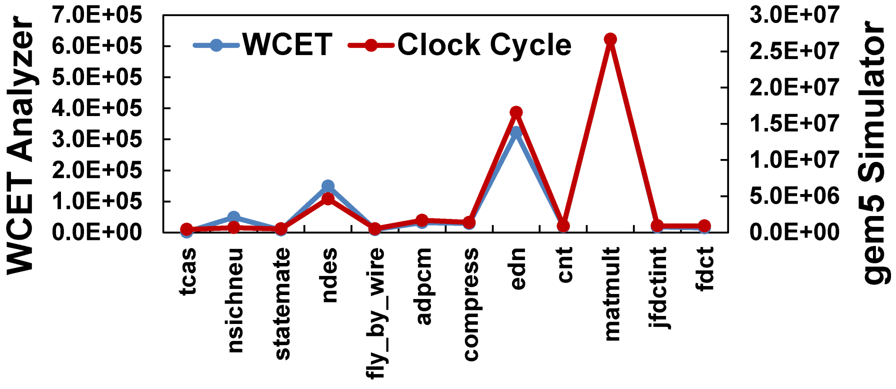


Figure 3.6: Comparison of the generated WCET with GEM5 simulations for benchmarks in Table 3.1

The research in [Als14] examines the relation between executions of a program and its software characteristics, such as having single paths, containing loops or nested loops, array manipulation, bit operation, etc. It uses regression algorithms to find a relation between GEM5 simulations for Mälardalen benchmarks [Mö6] and their software characteristics. In the research, the benchmarks are classified into different groups based on the program characteristics presented in [Mö6] and for each classification group a relationship between the simulated clock cycles and the software characteristics are derived. From our simulated benchmarks, ndes individually falls into one classification group. This can be the reason for the slight difference that we see between the simulation result and the generated WCET. However, nsichneu falls into a classification group with some other benchmarks. It has been stated that the WCET for nsichneu is unknown [KKZ13] and any simulation in general can be far from the WCET. This can be the reason of the slight difference seen between the simulation results and the generated WCET for nsichneu.

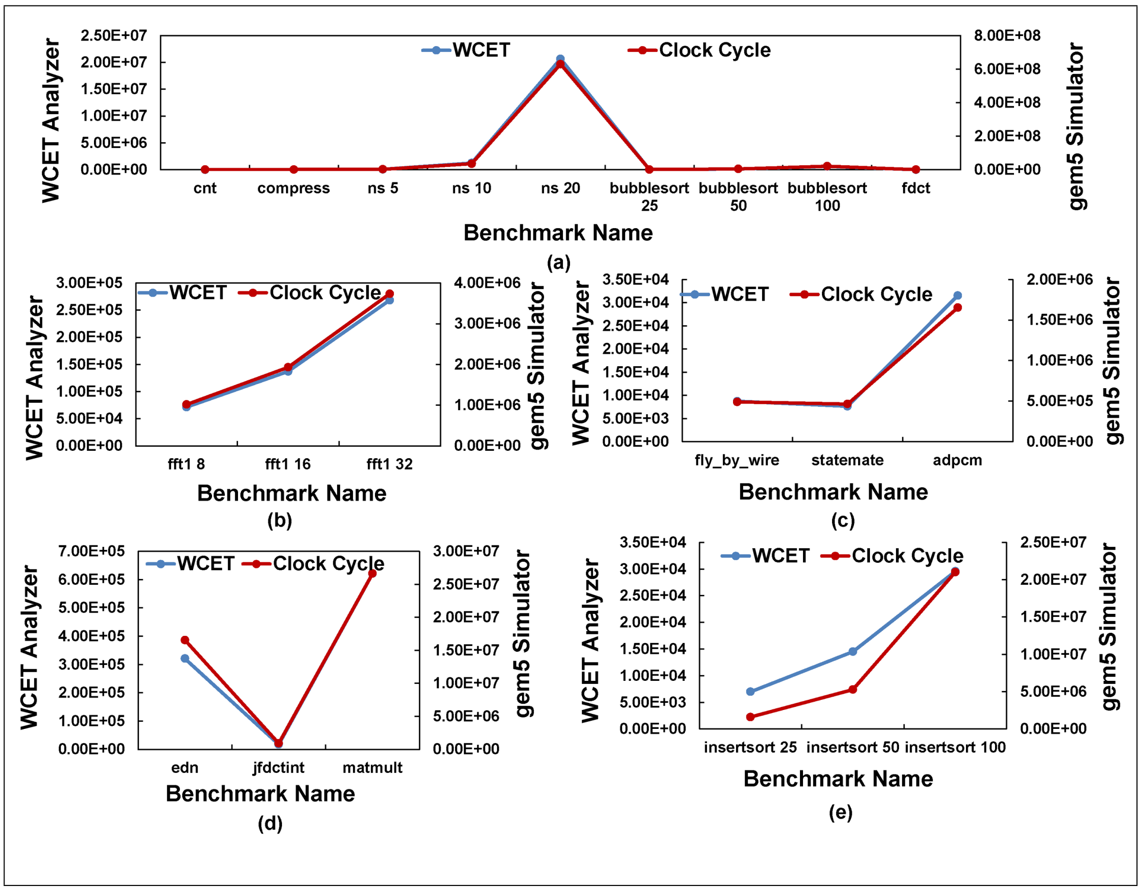


Figure 3.7: Comparison of the generated WCET with GEM5 simulations for Different Benchmarks Groups

We have followed the same classifications presented in [Als14] and compared the generated WCET and the simulation results. The results of this comparison for benchmark groups is presented in Figure 3.7. From the simulated benchmarks, tcas and ndes individually fall into one classification group and cannot be compared to other benchmarks. The LNA classification group of the benchmarks, which contain nested loops and array/matrices contains cnt, compress and fdct. We have compared these benchmarks with other benchmarks in Figure 3.7 (a). In Figure 3.7 (b) and (e), fft1 and insertsort benchmarks are compared with each other for different sizes.

On the other hand, the benchmarks in group L containing non-nested loops are nsichneu, statemate and adpcm. Moreover, fly-by-wire from Papabench [N^+06] has similar program characteristics and is added to this group. We have compared statemate, adpcm and fly-by-wire with each other in Figure 3.7 (c). We have excluded nsichneu due to the reason explained above. Moreover, edn, jfdctint and matmult are compared in Figure 3.7 (d). In general, these comparisons show that in the majority of cases the generated WCET and the simulation results follow a similar trend. The slight differences in the trends may be due to the point that the WCET analysis is performed on LLVM IR, while the simulations are performed on binary.

3.6 Summary

In this chapter, we have customized our framework for the WCET analysis of programs with consideration of a cache micro-architecture. At its core, the framework is a symbolic execution, which preserves the program's operational semantics in detail, down to the cache. The only abstraction performed is that the analysis of one loop iteration is summarized; importantly, the analysis proceeds precisely across loop iterations. The key challenge, scalability, was obtained by using a custom notion of reuse. In realistic benchmarks, it was shown that the extreme attempt at precision in fact pays off because there was a significant increase in precision, and this was obtained in a reasonable time.

Chapter 4

Symbolic Memory High-watermark Analysis with Symbolic Bounds

4.1 Introduction

Traditionally, in safety-critical embedded systems, it was recommended not to use dynamic memory allocation because of two main reasons: (a) the allocation instructions might take longer than expected, resulting in the failure of temporal constraints in hard real-time systems; and (b) the memory fragmentation issue. As a result, stack was the only memory that grows dynamically during execution. Worst-case stack usage was estimated by methods such as the one proposed in [KF14]; the estimate is compared with the available memory to ensure stack overflow errors does not occur.

In the past few years, there have been advances in both hardware and software of embedded systems. The drop in the hardware cost, the development of customized operating systems for embedded systems, and finally the advent of constant time memory allocation algorithms with a reasonable handling of memory fragmentation [MRC03, M+08] are among these advances. Besides these, as the embedded systems become more complex, the need to use third-party code – which might require dynamic memory allocation – becomes more inevitable. As a result, dynamic memory allocation has now been used more frequently in embedded software [A+15].

The use of dynamic memory allocation in embedded systems, raised more concerns on ensuring the reliability of embedded systems used for safety-critical tasks. Thus, methods have been developed to avoid heap and stack overflows in safety-critical systems [RRW05]. The worst-case memory consumption of a program can be estimated to ensure the reliability of such safety-critical systems against heap and stack overflow errors. Besides that, the estimate of the worst-case memory consumption would be useful in the design process of embedded systems to reduce hardware cost [T+13]. Furthermore, this estimate can be presented to the programmer or a customer who are interested in knowing the memory footprint of an embedded system.

Memory is a *non-cumulative resource*, meaning that unlike time and energy where the maximum execution time or consumed energy of a path in a program is at the end of the path, the maximum memory used in a path can be at any place inside a path (e.g., right in the middle of the path). Thus, many approaches developed for worst-case analysis of cumulative resources, such as WCET analysis, becomes inapplicable. More specifically, these methods often abstract away the orders between the acquires/releases, which is crucial for precise analysis of non-cumulative resource.

There has been a large body of work for *automatically* deriving *symbolic upper bounds* of memory consumption. Such analyses can provide a bound even when the program loops or recursions are not statically bounded. A bound generated by these methods is *parametric* in two types of program inputs: (1) the inputs that determine the maximum depths of the loops and recursions; and (2) the other program inputs. It is worth to note that the generated bound is often a *non-linear* formula over the first type of inputs.

Resource analysis of imperative programs with non-regular loop patterns and *signed* integers, to model both memory allocation and deallocation, has long been an open problem. By "non-regular", we mean that the loop does not behave *uniformly* across different iterations. We now mention some notable related work.

COSTA [A$^+$07, A$^+$12a], formulating the problem using the framework of cost relations, can infer parametric upper-bounds on the memory consumption of Java programs with region-based garbage collection. It was then extended [FMH14] to generate more refined cost relations. On the other hand, [CHS15] performs amortized resource analysis for C programs. However, these methods are quite limited in coping with non-linear formulas, in the sense that either the bounds generated are too imprecise or they have to require manual user interaction. They are further challenged by the programs of which the termination can only be decided if path-sensitivity is carefully taken into account.

In this chapter, we present a memory high-watermark analysis, based upon the analysis framework presented in Chapter 2. Our analysis *precisely* computes and summarizes the memory consumption of each iteration. The key result from the loop unrolling is that non-regular loop patterns can be analyzed efficiently, often in a linear number of steps.

The main contribution of the analysis framework presented in this chapter is that *the bound generated by our analysis is symbolic.* This is mainly because programs can allocate and/or deallocate a non-fixed amount of memory (e.g., via some input variable that is not statically determined). Our bound, however, is not parametric w.r.t. program inputs dictating the maximum depths of the program loops. As a result, we do not need to deal with the challenging problem of inferring closed-form expressions for the loops. This enables our method to have a higher level of automation, while producing more accurate bounds.

In detail, given a program, our analysis starts by constructing the symbolic execution tree, from which an estimate of the memory high-watermark can be easily extracted. Being highly path-sensitive, our analysis disregards infeasible combinations of allocations/deallocations from consideration, thus producing accurate bounds. In the previous chapter, we introduced the concept *reuse with interpolation and dominance* to achieve scalable symbolic execution for integrated WCET analysis. In this chapter, though we use the same method for scalability, we still need to address two major challenges:

• First, it is the issue of *non-cumulative* resource. This requires the interplay between the *net* usage and the *high-watermark* usage of memory. As will be shown later, to accommodate this, we need to store more than one witness and dominating condition.

• Second, it is the issue of dealing with symbolic cost of an instruction, as opposed to concrete cost in many related work, as well as in Chapter 3. The need to compare between symbolic expressions leads us to the usage of the standard max function. Importantly, how it is used in tandem with the capturing of "dominance conditions" is one key contribution of this chapter. We elaborate more in Section 4.3.

4.1.1 Overview Examples

Consider the C program from Figure 4.1. The program points are shown in brackets, e.g., ⟨1⟩. The allocations (deallocations) in the program are annotated with increment (decrement) statements (in red color). The resource variable r captures the resource of interest: memory. Note that in this chapter both increment and decrement operations are allowed on it.

```
⟨1⟩     int main(int j){
⟨2⟩         int n; r = 0;
⟨3⟩         if(j > 0){
⟨4⟩             n = 10;
⟨5⟩             char *matrix = malloc(n); r = r + n;
⟨6⟩             /* normal computation */
⟨7⟩             free(matrix); r = r - n;
            }else{
⟨8⟩             n = 5
⟨9⟩             char *matrix = malloc(n + 10); r = r + (n + 10);
⟨10⟩            /* normal computation */
⟨11⟩            free(matrix); r = r - (n + 10);
            }
⟨12⟩        return 0;
        }
```

Figure 4.1: An Annotated C program

As elaborated in Chapter 2, our analysis computes a sound and precise bound over r (possibly symbolic here) in the end, across all feasible paths of the program. Note that due to the non-cumulative nature of the resource of interest, the largest value of the resource variable r can be at any place along a path (and not in general at the end of a path).

For the program in Figure 4.1, the highest value of r in the `then` branch is 10 (at ⟨5⟩) and in the `else` branch is 15 (at ⟨10⟩). These two values are compared at the parent node, namely ⟨3⟩. Because the highest value of r in the `else` branch is larger, we say the `else` branch *dominates* the `then` branch, under the current context. Then the path ⟨3⟩, ⟨8⟩, . . . , ⟨11⟩ is returned as the dominating path in the program. As a result, 15 – the highest value of r in the dominating path – is returned as a sound estimate for the worst-case memory consumption of the program.

Before stepping into a full analysis example, let us revisit the scalability issue. As stated in the previous section, one contribution of this chapter is to adapt the concept of "reuse with interpolation and dominance" to the setting where the increment/decrement amount of the resource usage (memory) can be a *symbolic* value.

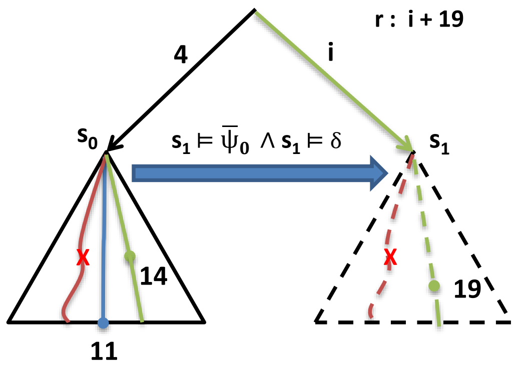


Figure 4.2: Reuse with Interpolation and Dominance

Figure 4.2 depicts a symbolic execution tree, where each triangle represents a subtree. The symbolic states s_0 and s_1 act as the program contexts for the left and right subtrees, respectively; and they are different visits to the *same program point*. Our analysis starts with exploring the left subtree. After traversing the left subtree, we obtain a summarization, comprised of four main components:

1) An *interpolant* Ψ_0, which is a generalization of s_0 that captures a condition

preserving the infeasible paths of the subtree. Infeasible paths are marked with a red cross.

2) A witness path denoted by Γ_h, which gives rise to the worst-case memory consumption of the subtree. This path is indicated in green color.

3) A second witness path, denoted by Γ_n, which captures the highest *net* memory usage at the end of the subtree. This path is indicated in blue color. The use of two witness paths is critical for safely combining summarizations (presented in Section 4.3).

4) A dominating condition δ, a formula which sufficiently guarantees that the dominating path *remains optimal*, i.e., the worst-case path in the subtree, when encountering a new context.

Considering we are now at s_1. Suppose that all the paths that were infeasible in s_0 stay infeasible, i.e., $s_1 \models \Psi_0$, and the dominating condition applies, i.e., $\mathsf{DOM}(\delta, s_1)$. This allows us to "reuse" the previous summarization. We then need to *replay* the witness path Γ_h under the context s_1. This, importantly, can lead to new *value* of the path (now 19), which is different from the original value (14). This is because the valuations of some symbolic expressions (or variables) are different, under the new context s_1, as opposed to the old context s_0 and may also hit the spike in another point along the path (as shown in Figure 4.2).
Backtracking to the root, and assuming that $i > 0$, thus $i + 19 > 4 + 14$. Therefore the right path in green is chosen as the overall dominating path. We then can conclude the analysis on the whole tree with the *symbolic* value $i + 19$.

Next, consider the C program in Figure 4.3 where we will discuss the concepts under the presence of a loop. Figure 4.4 depicts a symbolic execution tree of the

```
⟨0⟩  void main(int c){
⟨1⟩      assume(c ≥ 0); int N = 3;
         r = 0;
⟨2⟩      int** matrix = malloc(5 * sizeof(int*));
         r = r + 5 * 8;
⟨3⟩      char * b = (char*) malloc(c);
         r = r + c;
⟨4⟩      for (int i = 0; i ≤ N; i++){
⟨5⟩          if ((i % 2) == 0){
⟨6⟩              matrix[i]=malloc(i*sizeof(int));
                 r = r + (i * 4);
             }
         }
⟨7⟩      free (b); r = r - c;
     }
⟨8⟩
```

Figure 4.3: A Complicated Allocation Pattern

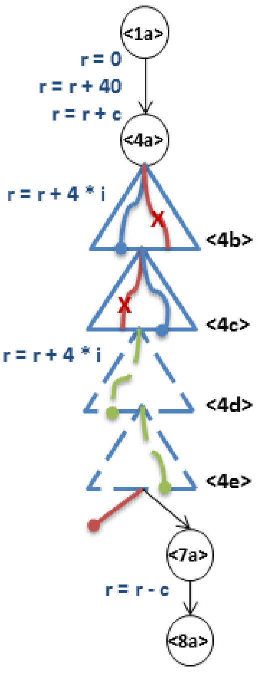


Figure 4.4: Analysis of the Complicated Allocation Pattern

corresponding program, where each triangle represents a loop iteration. For each node, in additional to a program point, we also use a letter to distinguish multiple visits to the same program point. An infeasible node is identified with a red bullet. For instance, at $\langle 4e \rangle$, the red bullet indicates that it is not possible to re-enter the loop body. For readability, the program points $\langle 2 \rangle$ and $\langle 3 \rangle$ are not shown in the tree.

We note that loops are exhaustively unrolled and contexts (of feasible paths) are merged in the end of each loop iteration.

Starting the analysis, the value of r is successively increased by 40 and then by the value of c (input argument) from program points $\langle 1 \rangle$ to $\langle 4 \rangle$. In the first loop iteration, the `then` branch is feasible and the value of r is increased by the value of $i * 4$, which is 0. Note that the `else` branch (colored in red in the symbolic execution tree) is *infeasible* because $i = 0 \wedge i \ \% \ 2 \ \neq \ 0$ is equivalent to $false$.

At $\langle 4b \rangle$, the analysis moves to the second iteration where the `then` branch is *infeasible*, because $(i = 1 \wedge i \ \% \ 2 \ == \ 0)$ is equivalent to $false$. There are no allocation/deallocations in the `else` branch, thus the value of r is unchanged. Similarly, in the third and fourth iterations the value of r is increased by 8 $(2 * 4)$ and 0, respectively. Finally, only at $\langle 4e \rangle$, exiting the loop is possible, while re-entering is not. We continue with the node $\langle 7a \rangle$, where r is then decreased by c. Because c is non-negative, the maximum value of r is reached at $\langle 7a \rangle$ which is $48 + c$.

Now let us go a bit deeper into the technicality. After the traversal of the first iteration, the maximum increase/decrease in the value of r in the iteration, which is $+(i * 4)$, was stored in a summarization of the loop iteration as a witness Γ_h. (The second witness path Γ_n coincides with Γ_h in this example.) Since there is only one feasible path in the loop iteration, the dominating condition is *true* and the interpolant stored in the summarization is $i \ \% \ 2 \ \neq \ 0$ which is enough to capture the reason of infeasibility.

In a following loop iteration where the interpolant and the dominating condition hold, for example at $\langle 4c \rangle$, the summarization of the first iteration is then reused. The analysis of the new iteration can be deduced to be $+(8)$, without the need of exploring all other paths in the loop iteration.

We also note that this summarization cannot be applied to the second iteration at $\langle 4b \rangle$. This is because the interpolant test fails. Fast forwarding, we finally mention that in Figure 4.4, the respective triangles of the third and the fourth iterations are shown in dotted lines to indicate that they were not explored in full. Instead, we reuse the summarizations of other iterations.

4.2 Related Work

We will briefly review three groups of related work.

4.2.1 Instrumentation Tools

Several different dynamic analysis tools have been developed which perform different forms of memory analysis; they can be categorized under *instrumentation* tools. Such tools often start by profiling an input program before analyzing the collected data; depending on the granularity of the data and the analysis overhead, we can broadly classify into “lightweight” and “heavyweight”.

Firstly, Valgrind [NS07], is a tool that has been widely used for memory debugging. One of its components, Massif, is a heap profiler that can measure the heap usage in a current execution of the program. Secondly, DynamoRio [B$^+$03], has a memory debugging tool which can be used to detect heap-overflow errors. One state-of-the-art tool from IBM, Pin [L$^+$05], tracks the amount of system resources used by a program. Finally, WMTrace [P+11], is most relevant tool to our analysis. It tracks memory allocation events in a multi-threaded program and in a post processing phase, it measures the worst-case heap usage of the program. All these methods are based on dynamic analysis, and not able to calculate the worst-case memory consumption.

4.2.2 Worst-case Stack Usage Analysis

Worst-case stack analysis is important for detecting stack overflows in safety-critical embedded systems. One state-of-the-art tool is AbsInt's StackAnalyzer [KF14]. It is a variant of value analysis performed on memory cells and CPU registers where the highest value on stack pointer is reported as the worst-case stack usage. This approach employs interval analysis and for precision, it is context-sensitive. Contexts are differentiated by a call string, which is bounded to some N (for scalability). The value analysis keeps updating the intervals till a fixpoint is reached. The highest value on the stack pointer shows the worst-case stack usage.

Recently, [C+14] employs a variant of Hoare logic to establish bounds on stack usage of C programs. However, this method cannot be extended for dynamic heap allocation. In its current formulation, it requires the size of the stack frame to be static.

In contrast, our method can be used for analyzing both worst-case heap and stack consumption. Importantly, our approach is path-sensitive, thus it produces more precise results.

4.2.3 Worst-case Heap Usage Analysis

Recently, object oriented languages have been proposed to be used in real-time critical systems [B+13, Sch15]. In such languages, besides WCET analysis,

analysis of worst-case heap consumption is also crucial for ensuring safety of the deployed systems [PHS10].

One attempt is [PHS10], targeting Java. It employs IPET-based framework, originated from WCET analysis. The method does not take into account memory deallocations and thus the bounds it produces would be imprecise. Another similar work is [A$^+$13], which only measures the allocations and assumes scope-based memory model: all the allocations are deallocated with the entire scope.

Our most closely related work is the static analysis presented in [HIE12], where an algorithm, extended from [CJ11, CJ13], has been outlined to perform non-cumulative resource analysis. This algorithm, however, assumes that the amount of resource consumed by each basic block is a constant. In this chapter, we make no such assumption. In fact, to accommodate that, we need to introduce a new form of reuse, as detailed in Section 4.3.

Last but not least, inferring *parametric* bounds, for memory consumption of imperative and object-oriented programs, has been an important research topic [A$^+$07, A$^+$12a, B$^+$08, FMH14, HAH11, CHS15, H$^+$16]. We have carefully discussed their representatives in Section 4.1.

4.3 General Framework

In this section, we will present the customized resource analysis framework for MHW analysis. In Table 2.1, we indicated the list of the terms and functions needed to be defined for customizing the resource analysis framework. In the rest of this section, we will define these terms and functions for MHW analysis.

4.3.1 Basic Operations and Memory Cost Model

Recall from Chapter 2 that we model a program by a transition system. The set of basic operations are either assignments, "assume" operations or memory allocations/deallocations. The set of all program variables is denoted by *Vars* including a special variable r tracking the amount of memory consumption. An assignment $x := e$ corresponds to assign the evaluation of the expression e to the variable x. The expression $\mathit{assume}(\mathit{cond})$ means: if the conditional expression *cond* evaluates to true, execution continues; otherwise it halts. Moreover, $\mathit{alc}(+, e)$ or $\mathit{alc}(-, e)$ corresponds to a memory allocation or deallocation, respectively, of size e. These operations are compiled from the `malloc` and `free` statements in the input C programs. We shall use $\ell \xrightarrow{op} \ell'$ to denote a transition relation from $\ell \in \mathcal{L}$ to $\ell' \in \mathcal{L}$ executing the operation $op \in \mathit{Ops}$. Similarly, the transition step would be defined as follow.

Definition 8 (Transition Step). *Given $\langle \mathcal{L}, l_0, \longrightarrow \rangle$, a transition system, and a symbolic state $s \equiv \langle \ell, \sigma, \Pi \rangle \in \mathit{SymStates}$, the symbolic execution of transition $tr : \ell \xrightarrow{op} \ell'$ returns another symbolic state s' defined as:*

$$s' \overset{\text{def}}{=} \begin{cases} \langle \ell', \sigma, \Pi \wedge \mathit{cond} \rangle & \textit{if } op \equiv \textit{assume(cond)} \\ \langle \ell', \sigma[x \mapsto [\![e]\!]_\sigma], \Pi \rangle & \textit{if } op \equiv x := e \\ \langle \ell', \sigma[r \mapsto r + [\![e]\!]_\sigma], \Pi \rangle & \textit{if } op \equiv \textit{alc(+,e)} \\ \langle \ell', \sigma[r \mapsto r - [\![e]\!]_\sigma], \Pi \rangle & \textit{if } op \equiv \textit{alc(-,e)} \end{cases}$$

□

4.3.2 The Machine State

The machine state for MHW analysis is empty, since estimating the memory consumption of a program does not rely on the internal state of any of the micro-architectural features. As a result, we define the symbolic state with the machine state assigned to each symbolic state to be empty ($\langle\rangle$).

In the domain of MHW analysis, since the machine state is an empty tuple, the updMachineState is equivalent to the identity function (Id), where it returns the exact input state as output: $m' := updMachineState(seq, m) \equiv Id(m)$.

4.3.3 Witness and Dominating Condition

Next, we discuss the concept of *witness*. As explained in Section 4.1.1, a high-watermark witness and a net-usage witness are stored in the summarization. A high-watermark witness is a sub-path from the root of the subtree $subtree(s, \ell_2)$ to a program point ℓ inside the subtree where the resource variable r reaches its peak value. It is depicted by $\Gamma_h \equiv \langle \Upsilon_h, \pi_h \rangle$ where Υ_h is the sequence of $alc(\pm, e)$ depicting all memory allocation or deallocations and π_h is the path constraints along the witness.

The witness of highest net-usage is a sub-path from the root of the subtree $subtree(s, \ell_2)$ to the program point ℓ_2 where the resource variable r has the highest value at ℓ_2 and it is depicted by $\Gamma_n \equiv \langle \Upsilon_n, \pi_n \rangle$.

Note that the two witnesses can be different. Also, to reduce the size of the witness, consecutive allocations of concrete amount are merged into one. Similarly for consecutive deallocations.

Because non-constant allocations may be evaluated to different values, with different contexts. For such evaluation, we rely on the *estimating upper-bound* and the *estimating lower-bound* functions.

Definition 9 (Estimating Upper-bound (EUB)). *Given a symbolic state $s \equiv \langle \ell, [\![s]\!] \rangle$ and a non-constant allocation of the amount captured by an expression e, the function* EUB$(e, [\![s]\!])$ *returns the smallest expression ub over symbolic input parameters and concrete values such that $[\![s]\!] \models e \leq ub$. In case no such upper-bound can be generated, e is returned.* □

Definition 10 (Estimating Lower-bound (ELB)). *Given a symbolic state $s \equiv \langle \ell, [\![s]\!] \rangle$ and a non-constant deallocation of the amount captured by an expression e, the function* ELB$(e, [\![s]\!])$ *returns the largest expression lb over symbolic input parameters and concrete values such that such that $[\![s]\!] \models lb \leq e$.* □

In summary, EUB and ELB are to over-estimate and under-estimate non-constant allocations and deallocations, respectively. Note that, in the worst case, ELB can always return 0 as a trivial lower bound. Generating sound, but not precise bounds using EUB or ELB would affect the overall precision of the analysis.

Example 7 (Witness Path). *Assume the following sequence of allocating/deallocating statements along a symbolic path, which is selected as a high-watermark witness:*

$$x = \texttt{malloc}(10), y = \texttt{malloc}(5), \texttt{free}(y), \texttt{free}(x), z = \texttt{malloc}(c)$$

This sequence would be stored as $\Upsilon_h \equiv ([+, 15], [-, 15], [+, c])$.

When the summarization is reused, the high-watermark memory usage is computed by replaying the sequence Υ_h under the new incoming context s, making use of the EUB and ELB functions. For example, given that $[\![s]\!] \equiv c < 5$, Υ_h in the above example will be approximated by $\Upsilon_1 = [+, 15], [-, 15], [+, 4]$, which gives us a high-watermark of 15. In a different context where $[\![s]\!] \equiv c < 20$, Υ_h will be approximated by $\Upsilon_1 = [+, 15], [-, 15], [+, 19]$, which gives us a high-watermark of 19.

Finally, similar to Chapter 3, the feasibility of a witness $\Gamma \equiv \langle \Upsilon, \pi \rangle$ w.r.t. to an incoming context s is determined by checking if $[\![\pi]\!] \wedge [\![s]\!]$ is satisfiable. In what follows, we abbreviate $[\![\pi]\!] \wedge [\![s]\!]$ by $[\![\Gamma]\!]$.

We say that two nodes in a symbolic execution tree are *similar* if they refer to the same program point. Thus, two subtrees are similar if they share the same entry and exit program points.

We next discuss dominating condition, another component of our analysis of a subtree. Each dominating condition is generated with respect to a witness. Intuitively, this answers the question "in what context of a similar subtree does the witness *remain optimal*?"

More specifically, the constraints in the dominating condition are typically of the form $x \leq y$ where x, y are either program variables or some concrete values — note that at least one must be a variable. The dominating condition is computed by abstracting the context that gives rise to dominance in the first place. We next present the customized Summarize-a-Trans function.

Summarize-a-Trans Function

Summarize-a-Trans, presented in Figure 4.5 computes a summarization for a single transition tr at state s. This can be seen as a basic step in our algorithm. We first elaborate on the computation of the witnesses and the high-watermark usage mhw. First, Υ is initialized to the sequence of allocations/deallocations (line 98), i.e., $[+, e]$ and/or $[-, e]$ in op. Next, consecutive concrete allocation/deallocations are merged by iterating through Υ once (line 99 and 100). Moreover, for each

```
function Summarize-a-Trans($s, tr$)
      Let $s$ be $\langle \ell, \sigma, \Pi \rangle$ and Let $tr$ be $\ell \xrightarrow{op} \ell'$
⟨98⟩ $\Upsilon$ := Sequence of (de-)allocations in $op$
⟨99⟩ Iterate on $\Upsilon$ and merge consecutive concrete allocations
⟨100⟩ Iterate on $\Upsilon$ and merge consecutive concrete deallocations
⟨101⟩ $i := 0; netusg := mhw := 0$
⟨102⟩ foreach $[\texttt{sign}, m] \in \Upsilon$ do
⟨103⟩     if sign $is +$ then $netusg := netusg + m$
⟨104⟩     else $netusg := netusg - m$
⟨105⟩     if $netusg > mhw$ then $mhw := netusg$
      endfor
⟨106⟩ $\Gamma_h := \langle \Upsilon, [\![op]\!]_\sigma \rangle$;  $\Gamma_n := \langle \Upsilon, [\![op]\!]_\sigma \rangle$
⟨107⟩ return $[\ell, true, \Gamma_h, \Gamma_n, mhw, true, true, op_\Delta]$
```

Figure 4.5: Summarize-a-Trans Function

(de-)allocation the *netusg* is updated in lines 103 and 104. If the value of *netusg* is greater than the high-watermark value, *mhw* is updated and ℓ' is stored as the peak location (line 105).

Next, the path constraint for each witness is computed by projecting *op* onto the set of program variables w.r.t. the symbolic store σ, denoted as $[\![op]\!]_\sigma$ (line 106). We now elaborate on the rest of the components stored in the summarization. Because no infeasible path has been discovered, the interpolant Ψ is just *true*. There is a single path, thus the dominating conditions are *true*. The abstract transformer Δ_p is the operation *op* itself, but translated to the language of input-output relation. As an example, $\mathsf{y} := \mathsf{y} + 1$ is translated to $y_{out} = y_{in} + 1$. We use op_Δ to denote such translated *op*.

Compounding Two Summarizations

We presented the high view steps to compounding two summarizations in Chapter 2. However, since that the summarizations in this chapter contain both high-watermark and net-usage witness, we will present and updated version of the

```
function Compose($s, S_1, S_2$)
      Let $S_1$ be $[\ell_1, \Psi_1, \Gamma_{h1}, \Gamma_{n1}, mhw_1, \delta_{h1}, \delta_{n1}, \Delta_{p1}]$
      Let $S_2$ be $[\ell_2, \Psi_2, \Gamma_{h2}, \Gamma_{n2}, mhw_2, \delta_{h2}, \delta_{n2}, \Delta_{p2}]$
      Let $\Gamma_{n1}$ be $\langle \Upsilon_{n1}, \pi_{n1} \rangle$
⟨108⟩ $\Delta_p := \Delta_{p1} \wedge \Delta_{p2}$
⟨109⟩ $\Psi := \Psi_1 \wedge$ Pre-Cond($\Delta_{p1}, \Psi_2$)
⟨110⟩if ($mhw_1 >$ net-usg($s, \Upsilon_{n1}$) $+ mhw_2$)
⟨111⟩     $mhw := mhw_1$
⟨112⟩     $\Gamma_h := \Gamma_{h1}$; $\delta_h := \delta_{h1}$
     else
⟨113⟩     $mhw :=$ net-usg($s, \Upsilon_{n1}$) $+ mhw_2$
⟨114⟩     $\{\Gamma_h, \delta_h\} :=$ Combine-Witnesses($\Delta_{p1}, \Gamma_{n1}, \Gamma_{h2}, \delta_{n1}, \delta_{h2}$)
⟨115⟩     $\{\Gamma_n, \delta_n\} :=$ Combine-Witnesses($\Delta_{p1}, \Gamma_{n1}, \Gamma_{n2}, \delta_{n1}, \delta_{n2}$)
⟨116⟩ return $[\ell_1, \overline{\Psi}, \Gamma_h, \Gamma_n, mhw, \delta_h, \delta_n, \Delta_p]$
end function

function JoinHorizontal($s, S_1, S_2$)
      Let $S_1$ be $[\ell, \Psi_1, \Gamma_{h1}, \Gamma_{n1}, mhw_1, \delta_{h1}, \delta_{n1}, \Delta_{p1}]$
      Let $S_2$ be $[\ell, \Psi_2, \Gamma_{h2}, \Gamma_{n2}, mhw_2, \delta_{h2}, \delta_{n2}, \Delta_{p2}]$
⟨117⟩ $mhw :=$ max($mhw_1, mhw_2$)
⟨118⟩ $\Psi := \Psi_1 \wedge \Psi_2$
⟨119⟩ $\Delta_p := \Delta_{p1} \vee \Delta_{p2}$
⟨120⟩ $\{\Gamma_h, \delta_h\} :=$ Merge-Witness-h($\Gamma_{h1}, \Gamma_{h2}, \delta_{h1}, \delta_{h2}$)
⟨121⟩ $\{\Gamma_n, \delta_n\} :=$ Merge-Witness-n($s, \Gamma_{n1}, \Gamma_{n2}, \delta_{n1}, \delta_{n2}$)
⟨122⟩  return $[\ell, \Psi, \Gamma_h, \Gamma_n, mhw, \delta_h, \delta_n, \Delta_p]$
end function
```

Figure 4.6: Helper Functions

Compose and JoinHorizontalfunction in Figure 4.6.

Compounding Vertically Two Summarizations: Consider that $subtree(s_2, \ell_3)$ suffixing $subtree(s_1, \ell_2)$, where $s_2 \equiv \langle \ell_2, [\![s_2]\!] \rangle$ and $s_1 \equiv \langle \ell_1, [\![s_1]\!] \rangle$. In other words, a path π_1 from ℓ_1 to ℓ_2 followed by a path π_2 from ℓ_2 to ℓ_3 corresponds a path π in $subtree(s_1, \ell_3)$. The Compose function (in Figure 4.6) returns a summarization for $subtree(s_1, \ell_3)$ by compounding the two existing summarizations, respectively for $subtree(s_1, \ell_2)$ and $subtree(s_2, \ell_3)$.

The abstract transformer Δ_p is computed as the conjunction of the input

```
function Combine-Witnesses($\Delta_p, \Gamma_1, \Gamma_2, \delta_1, \delta_2$)
        Let $\Gamma_1$ be $\langle \Upsilon_1, \pi_1 \rangle$ and Let $\Gamma_2$ be $\langle \Upsilon_2, \pi_2 \rangle$
⟨123⟩ $\pi := \pi_1 \;\wedge\; \pi_2$
⟨124⟩ $\delta_2' := \mathit{true}$
⟨125⟩ foreach $\mathit{cond} \in \delta_2$ do
⟨126⟩     $\delta_2' := \delta_2' \;\wedge\;$ Pre-Cond($\Delta_p, \mathit{cond}$)
        endfor
⟨127⟩ $\delta := \delta_1 \;\wedge\; \delta_2'$
⟨128⟩ $\Upsilon_2' := [\,]$
⟨129⟩ foreach $[\mathtt{sign}, m] \in \Upsilon_2$ do
⟨130⟩     $\Upsilon_2' :=$ Add $[\mathtt{sign},$ Pre-Cond$(\Delta_p, m)]$ into $\Upsilon_2'$
        endfor
⟨131⟩ $\Upsilon := \Upsilon_1 \;\bullet\; \Upsilon_2'$      // concatenation
⟨132⟩ return $\{\langle \Upsilon, \pi \rangle, \delta\}$
```

Figure 4.7: Combining Witness Formula and Dominating Conditions

abstract transformers (line 108), with proper variable renaming. Note that in our implementation, abstract transformers are computed using polyhedral domain. We employ Δ_p to generate *one* continuation context, before proceeding the analysis with subsequent program fragments. Next, the desired interpolant must capture the infeasibility of S_1, as well as the infeasibility of S_2 given that we treat $\mathit{subtree}(s_1, \ell_2)$ as an abstract transition, of which the operation is Δ_p. We rely on the function Pre-Cond, which in line 109 under-approximates the weakest-precondition of the post-condition Ψ_2 w.r.t. to the transition relation Δ_p.

Next we update the high-watermark witness. Here the net-usage witness becomes important. In the combined subtree, the high-watermark is chosen by comparing (1) the high-watermark of the prefix tree and (2) the (worst) net-usage of the prefix subtree plus the high-watermark of the suffix subtree (line 110). In case (1) is greater, the witness and the dominating condition of the prefix subtree is returned (lines 111 and 112). Otherwise, the net-usage witness and its dominating condition of the prefix subtree are combined with the high-watermark witness and the corresponding dominating condition of the suffix subtree (lines 113 and 114). This is achieved by calling the function Combine-Witnesses. Finally, we again invoke Combine-Witnesses to combine the net-usage witnesses and their respective dominating conditions (line 115).

In Figure 4.7, Combine-Witnesses produces a witness and a dominating condition, by compounding the witnesses and dominating conditions of the two subtrees, where one suffixes the other. This can be understood as a *sequential* composition.

First, the path constraint π is simply the conjunction of π_1 and π_2 (line 123). Next, the combined dominating condition δ is computed as the conjunction of δ_1 and a condition δ_2', in line 127, where intuitively, δ_2' describes a set of conditions, such that δ_2' is a precondition of δ_2 w.r.t. to the transition relation Δ_p. Similarly, the allocation/deallocations in Υ_2 are updated w.r.t. to the transition relation Δ_p and stored in Υ_2'.

Compounding Horizontally Two Summarizations: Given two summarizations of two subtrees rooted at two nodes which are siblings, we want to propagate

```
function Merge-Witness-h($\Gamma_1, \Gamma_2, mhw_1, mhw_2, \delta_1, \delta_2$)
⟨133⟩ $\delta := \delta_1 \wedge \delta_2$
⟨134⟩ if ($true \models mhw_1 \geq mhw_2$) then return $\{\Gamma_1, \delta\}$
⟨135⟩ if ($true \models mhw_1 \leq mhw_2$) then return $\{\Gamma_2, \delta\}$
⟨136⟩ return $\{\langle \mathsf{max}(\Upsilon_1, \Upsilon_2), \pi_1 \wedge \pi_2\rangle, \delta\}$

function Merge-Witness-n($s, \Gamma_1, \Gamma_2, \delta_1, \delta_2$)
      Let $\Gamma_1$ be $\langle \Upsilon_1, \pi_1\rangle$ and Let $\Gamma_2$ be $\langle \gamma_2, \pi_2\rangle$
⟨137⟩ $\delta := \delta_1 \wedge \delta_2$
⟨138⟩ if ($true \models \mathsf{net\text{-}usg}(s, \Upsilon_1) \geq \mathsf{net\text{-}usg}(s, \Upsilon_2)$)
⟨139⟩     return $\{\Gamma_1, \delta\}$
⟨140⟩ if ($true \models \mathsf{net\text{-}usg}(s, \Upsilon_1) \leq \mathsf{net\text{-}usg}(s, \Upsilon_2)$)
⟨141⟩     return $\{\Gamma_2, \delta\}$
⟨142⟩ return $\{\langle \mathsf{max}(\Upsilon_1, \Upsilon_2), \pi_1 \wedge \pi_2\rangle, \delta\}$
```

Figure 4.8: Merging Witness Formulas and Dominating Conditions

the information back and compute the summarization for the (common) parent node. While propagation can be achieved by Compose, we need JoinHorizontal (presented in Figure 4.6) to "merge" the contributions of the two children to the parent node. Note that unlike Compose, we need to select the path with the larger memory high-watermark usage between the two witnesses of the input summarizations. Thus, the high-water mark usage would be the maximum of the mhw_1 and mhw_2 (line 117). Moreover, all the infeasible paths in both sub-structures must be maintained, thus the desired interpolant is the conjunction of the two input interpolants (line 118). On the other hand, the abstract transformer Δ_p is computed straightforwardly as the disjunction of the input abstract transformers (line 119). Finally, Merge-Witnesses-h is invoked to merge the high-watermark witnesses and the respective dominating conditions (line 120) and similarly Merge-Witnesses-n is invoked to merge the net-usage witnesses and the respective dominating conditions (line 121).

In Figure 4.8, Merge-Witnesses-h produces a high-watermark witness and a dominating condition, by compounding the witnesses and dominating conditions of two sibling subtrees. We need to choose one witness from the two input witnesses. The combined dominating condition must ensure the dominance of each witness (in its respective subtree) and the dominance of the chosen witness, which produces a higher value of r, over the other.

The dominating condition δ is initialized as the conjunction of the two dominating conditions (line 133). Next we compare the two high-watermarks mhw_1 and mhw_2; if mhw_1 is greater or equal to mhw_2, Γ_1 dominates Γ_2, and Γ_1 is returned as the dominating witness with δ as the dominating condition (line 134). If not, we check if mhw_2 is greater or equal to mhw_1 and then we return Γ_2 as the dominating witness with δ as the dominating condition (line 135). If both tests fail, this could happen when we deal with symbolic expressions, we then employ the max function, delaying the dominance test to a higher level in the symbolic execution tree (line 136) with the hope that another witness might dominate this path. Similarly, the Merge-Witnesses-n function produces a net-usage witness and a dominating condition. Here we make use of the function net-usg, which extracts the net usage given a context s and a sequence of allocations/deallocations (Υ_1 or Υ_2). This function can be easily implemented, we omit the detail due to space reason.

4.3.4 Machine State Summary

In MHW analysis, since the machine state is empty, the machine state summary is equal to the identity function (Id), which returns the same machine state: $\Delta_m \equiv Id(m)$.

4.4 Example Analysis

Figure 4.9(a) presents the CFG of an example program. Its symbolic execution tree is depicted in Figure 4.9(b). Both the CFG and the tree are annotated with the updates on the resource variable r (in red color). Assume that b is a symbolic input parameter for this example program. The variable of interest r is initialized to 0 between nodes $\langle$1a$\rangle$ and $\langle$2a$\rangle$. The value of r can be seen beside node $\langle$2a$\rangle$ (in green color). Note that in Figure 4.9(b), we do not (fully) show the subtree below node $\langle$5b$\rangle$ and that we do not discuss the abstract transformers in detail.

In the left-most path, the value of r is updated to 20 at $\langle$5a$\rangle$, 0 at $\langle$8a$\rangle$ and c at $\langle$11a$\rangle$, which is a symbolic value. Note that c is a random number in the range of $[0, b-1]$ and cannot be determined statically. In the second path, reaching $\langle$10a$\rangle$, the path constraints contains both $j = 0$ and $j < 0$ (only relevant constraints are

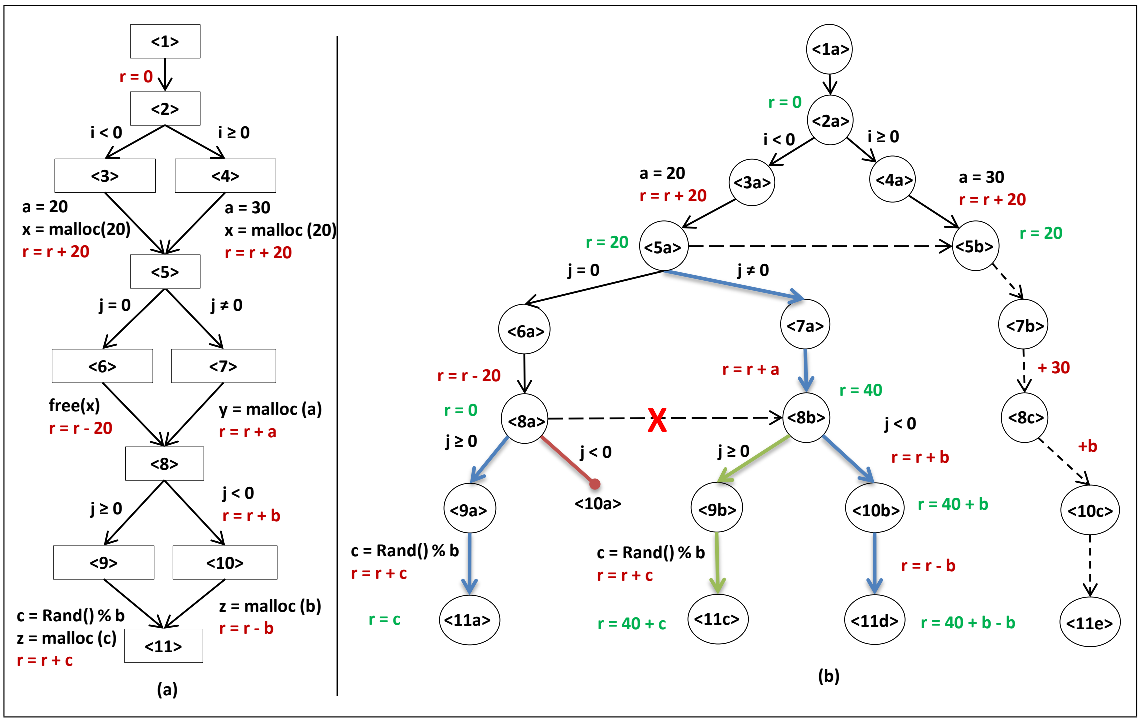


Figure 4.9: (a) The CFG of an annotated program (b) The analysis tree of the program

shown for brevity), thus an infeasible path is detected.

After finishing the subtree beneath ⟨8a⟩, the following summarization is computed and stored:

$$[\langle 8\rangle, \overline{\Psi}, \Gamma_h, \Gamma_n, mhw, \delta_h, \delta_n, \Delta_p],$$

where the stored interpolant is $\overline{\Psi} \equiv j \geq 0$, which is a succinct reason for the infeasibility of the right sub-path; the stored dominating conditions δ_h and δ_n are *true*, given that there is only one feasible path. Similarly, the only feasible path (in blue color) is stored both as the high-watermark witness $\Gamma_h \equiv \langle([+, c]), j \geq 0\rangle$ and as the net-usage witness $\Gamma_n \equiv \langle([+, c]), j \geq 0\rangle$, where $j \geq 0$ is the witness path constraint and $([+, c])$ is the sequence of the allocation/deallocations along the witness path. Finally, mhw, the worst-case heap memory consumption of the subtree, is computed to $b-1$, by evaluating the memory consumption of the high-watermark witness. This, in turn, is achieved by invoking the function EUB with c as the first argument and the current context as the second argument.

Continuing the analysis, consider the pair of nodes $\langle 8a \rangle$ and $\langle 8b \rangle$. At node $\langle 8b \rangle$, the value of r is 40, due to the memory allocation from $\langle 7a \rangle$ to $\langle 8b \rangle$. We will check the reuse conditions here. We *first* check whether the stored *interpolant* ($\overline{\Psi}$) is implied. This does not hold. The key reason is: some infeasible path in $\langle 8a \rangle$ is in fact feasible in $\langle 8b \rangle$. As a result, reuse does not happen and the node $\langle 8b \rangle$ is expanded.

After the analysis of the subtree beneath $\langle 8b \rangle$, a summarization is generated from the analysis of node $\langle 8b \rangle$. The summarization would be $[\langle 8 \rangle, \overline{\Psi}', \Gamma'_h, \Gamma'_n, mhw',$ $\delta'_h, \delta'_n, \Delta'_p]$. The stored interpolant $\overline{\Psi}'$ is simply *true* because both paths in the subtree are feasible.

Comparing the peak value of r along these two feasible sub-paths, it can be seen that the value of r is larger in the right sub-path. So, the right sub-path is chosen as the high-watermark witness and Γ'_h is stored as $\langle ([+, b], [-, b]), j < 0 \rangle$. Consequently, mhw' is set to b.

Now, we need to capture a dominating condition such that when it holds, it is guaranteed that the chosen witness path dominates all the other path(s) in the subtree. For any symbolic state s, it is the case that $[\![s]\!] \models \text{EUB}(c, [\![s]\!]) <$ $\text{EUB}(b, [\![s]\!])$ (since C is a random value in the range $[0, b-1]$). Thus the stored dominating condition δ'_h is simply *true*.

On the other hand, the value of r at $\langle 11c \rangle$ is $40 + c$, which is higher than the value of r at $\langle 11d \rangle$, which is 40. So the net-usage witness is the sub-path $\langle 8b \rangle$, $\langle 9b \rangle$, $\langle 11c \rangle$ (with green color) $\Gamma'_n \equiv \langle ([+, c]), j \geq 0 \rangle$. Moreover, the net-usage dominating condition δ'_n would also be simply *true*. For brevity, we would not discuss net-usage witnesses and their dominating conditions in the rest of this example.

Continuing the analysis, consider the pair of nodes $\langle 5a \rangle$ and $\langle 5b \rangle$. We will show that reuse can in fact take place here. Please take note, without proof, that: (1) the high-watermark witness of the subtree rooted at $\langle 5a \rangle$ is the rightmost feasible path $\langle 5a \rangle$, $\langle 7a \rangle$, $\langle 8b \rangle$, $\langle 10b \rangle$, $\langle 11d \rangle$ (in blue color), stored as $\langle ([+, a], [+, b]), j < 0 \rangle$; (2) The interpolant of interest is *true*; and the dominating condition for the high-watermark witness is also *true*.

Now we can exemplify the reuse at $\langle 5b \rangle$. We first check if the context of $\langle 5b \rangle$, called $[\![s_{5b}]\!]$ implies the interpolant computed after finishing $\langle 5a \rangle$. In this case, the interpolant is *true*, thus the implication trivially holds. We then check whether the dominating condition, which is *true*, holds. This is also trivially satisfied, thus we can reuse the *high-watermark witness* of $\langle 5a \rangle$, yielding an overall worst-case memory consumption of $20 + \text{EUB}(a, [\![s_{5b}]\!]) + \text{EUB}(b, [\![s_{5b}]\!])$, which is simplified to be $50 + b$.

We remark here that, the worst-case consumption of the sub-path $\langle 5 \rangle$ $\langle 7 \rangle$ $\langle 8 \rangle$ $\langle 10 \rangle$ $\langle 11 \rangle$ for contexts $\langle 5a \rangle$ and $\langle 5b \rangle$ are indeed different. It is because, fundamentally, the worst-case consumption of a symbolic path is dependent on its context. In this particular example, the valuation of a in the two contexts is different.

Finally, we easily arrive at the worst-case heap memory consumption of the entire tree, thus the entire example program, to be $50 + b$ which is a symbolic bound considering that b has been an input argument for this example program.

4.5 Experimental Evaluation

Table 4.1 compares our framework with the state-of-the-art tools and methods. We have elaborated these methods and tools in Sections 4.1 and 4.2. The four first tools perform dynamic analysis and hence are unable to perform worst-case heap analysis (WCHA) and worst-case stack analysis (WCSA). The second four tools either only perform WCHA or WCSA, while our framework performs both WCHA and WCSA.

The main tools and methods that can be compared to our framework is first [HIE12] which generates non-symbolic bounds while our method is able to generate symbolic bounds. A second group of tools that can be compared to our analysis are COSTA [A$^+$07, A$^+$12a] and the methods presented in [CHS15] and [FMH14]. These methods might generate non-linear bounds and have limitations in handling with non-linear bounds. Our framework generates non-linear bounds only if a dynamic memory allocation/deallocation has a non-linear size. In other words, since our framework fully unrolls loops, our method does not require manual user interaction to solve non-linear formulas. Moreover, non-linear formulas, in general, might be more imprecise compared to the (probably linear) bound generated by our framework.

As a result, we can state that our method would be at least as precise as other methods. However, this precision might affect the scalability of our framework.

We evaluate our proposed algorithm using a number of benchmarks collected from the literature. The suite includes: (1) memory allocator tests such as `shbench`, `larson` and `cache-scratch` from Hoard benchmarks [BMBW00]; (2) embedded programs from MiBench [G$^+$01] and from Mälardalen WCET benchmarks [Mö6]);

Table 4.1: Comparison of Memory Analysis Tools and Methods

Tool	Analysis	WCHA	WCSA	Remark
Massif-Valgrind	Dynamic			Peak Analysis
DynamoRio				Detect heap-overflow errors
Pin				Tracks used system resources
WMTrace				Worst-case heap usage
StackAnalyzer	Static		*	Worst-case stack usage (variant of value analysis)
[C$^+$14]			*	Worst-case stack usage (Static frames)
[PHS10]		*		Does not consider deallocations
[A$^+$13]		*		Scope-based deallocations
[HIE12]		*	*	Generates non-symbolic bounds
COSTA & [FMH14]		*	*	Limited in coping with non-linear formulas, either: · Generate imprecise bounds · Require manual user interaction
[CHS15]				
Our Framework (Unroll_d)		*	*	1. Generating symbolic bounds 2. Generate bounds only parametric to input variables 3. Not requiring manual user interaction

and (3) heap manipulating benchmarks from [LLV15].

Table 4.2 presents the results of our experiments. The third column indicates the values to which input parameters are concretized. *Time*, *States* and *Reuses* columns present the running time, the number of visited states, and the number of reuses in each benchmark instance. In other words, they illustrate the cost of our algorithm.

Among the benchmarks, `nsichneu`, `statemate` and `ndes` contain many (possibly infeasible) paths. They are to stress the scalability of our algorithm. Benchmarks `cache-thrash` and `cache-scratch` are used to test active and passive false sharing. To test for memory fragmentation, `shbench` is often used. It randomly allocates and deallocates random memory chunks of memory. As illustrated in

Table 4.2: Result of Analysis of Experimental Benchmarks on Our Algorithm Unroll_d

Benchmark	**LOC**	**Input Parameters (Concretized)**	**Time**	**States**	**Reuses**	**Bound**
larson	614	threads = 1, num_chunks = 100	55.55	613	198	240050000
ndes	219	N.A.	21.21	643	201	11214
puzzle	197	N.A.	164.43	1094	354	204
fasta	121	N.A.	0.23	91	17	755
chomp	401	N.A.	1.59	153	36	6800
cache-thrash	120	threads = 1, iterations = 100, objSize = 1, repetitions = 10	7.96	344	108	1
cache-thrash	120	threads = 2, iterations = 100, objSize = 1, repetitions = 10	8.00	329	103	1
cache-thrash	120	threads = 1, iterations = 200, objSize = 1, repetitions = 10	58.96	644	208	1
cache-thrash	120	threads = 2, iterations = 200, objSize = 1, repetitions = 10	57.60	629	203	1
cache-scratch	126	threads = 1, iterations = 100, objSize = 1, repetitions = 10	9.32	350	108	9
cache-scratch	126	threads = 2, iterations = 100, objSize = 1, repetitions = 10	9.26	338	104	18
cache-scratch	126	threads = 1, iterations = 200, objSize = 1, repetitions = 10	68.86	650	208	9
cache-scratch	126	threads = 2, iterations = 200, objSize = 1, repetitions = 10	69.40	638	204	18
statemate	1090	N.A.	233.63	3553	1296	64
nsicheneu	3144	N.A.	791.75	3639	1376	112
shbench	121	threads = 1, Nalloc = 100	554.39	4461	1408	43600
himenobmtxpa	272	N.A.	175.45	1472	406	57344
dry	491	N.A.	0.3	142	39	112
fft1 (main)	234	MAXWAVES = 8	5.23	88	23	16 * MAXSIZE + 64
nsieve-bits	33	N.A.	4.74	552	192	sz / 8 + 4
ffbench	287	Asize = 10	138.56	1550	508	262160

Section 4.4, our analysis can generate bounds even for programs where memory allocations/deallocations are highly randomized. Finally, `larson` is a famous benchmark which simulates a server. Similar to `shbench` it has a random behavior in memory allocation/deallocation. The analyzed benchmarks are categorized into four groups, separated by a double line in Table 4.2. We discuss each individual group as below.

The first group of benchmarks contain complicated patterns of allocations and/or deallocations (e.g., inside loops and conditional branches). In these benchmarks, path-sensitivity plays a crucial role in generating a precise worst-case estimate. Although the method presented in [HIE12] also benefits from path-sensitivity, one key distinction of this work is the employment of symbolic witnesses, which make our analysis applicable to programs with unbounded allocations/deallocations.

Moreover, let us elaborate on `puzzle` to highlight the impact of addressing the issue of non-cumulative resource *directly*, as opposed to applying some known approaches for analyzing cumulative resource. In `puzzle`, the outer loop iterates 5 times, and in each iteration it acquires memory, performs some operations in some inner loops and releases the memory at the end of each outer loop iteration. Analyzing memory as a cumulative resource would return 1020 as an estimate of the worst-case consumption, which is 5 times larger than the bound produced by our method.

The second group of benchmarks contains `cache-thrash` and `cache-scratch` which are analyzed for different input parameters. Both `cache-thrash` and `cache-scratch` are multi-threaded benchmarks. We have analyzed their local computation for the cases when number of threads (*nthreads*) are either 1 or 2 (exact bounds). In both benchmarks, the number of the iterations of an inner loop is determined by *repetitions*/*nthreads*. As a result, the execution with *nthreads* = 2 is indeed shorter than the execution with *nthreads* = 1. Finally, note that the generated bound for `cache-scratch` is dependent on the number of threads.

The third group of benchmarks contains `nsichneu` and `statemate`. These two benchmarks contain very large loops which iterate twice. These benchmarks are often used by WCET research community to test the scalability aspect of an algorithm, especially when it is based upon symbolic execution. Our algorithm is able to fully analyze these benchmarks, demonstrating the potential of our new concept of "reuse" to scale in the presence of symbolic bounds.

Finally, the last group of benchmarks contain loops with large number of iterations, where the ability to reuse compounded summarizations to avoid state explosion is crucial. For example, `himenobmtxpa` contains 14 loops with the nested level of 3 and `shbench` contains a complicated loop pattern with the nested level of 3. Among these benchmarks, `shbench` can be executed in both single-threaded and multi-threaded forms which we have analyzed it in the single threaded form. We highlight that the bounds generated for `fft1` and `nsieve-bits` are symbolic bounds.

4.6 Summary

In this chapter, we presented memory high-watermark analysis with symbolic bounds. Memory high-watermark analysis is important since it is a non-cumulative resource analysis where the resource of interest can be consumed or released.

Moreover, we extended the concept of *reuse with interpolation and domination* in the presence of *symbolic bounds*. Memory high-watermark analysis does not get affected from any of the micro-architecture features. As a result, the machine state was empty. However, as illustrated in the examples, our resource analysis framework can still return more precise bounds based on the point that the contribution of each basic block in the memory consumption is dynamic. Finally, we presented the results of our experiments on memory-intensive benchmarks to verify both the precision and scalability of our framework.

Chapter 5

Integrated Worst-case Energy Consumption Analysis with Cache and Pipeline

5.1 Introduction

In this chapter, we will present a worst-case energy consumption analysis. Energy consumption is one of the important non-functional features in embedded systems. A lower energy consuming embedded system can function for a longer time in an environment with limited access to energy. The software being executed on the embedded system has a profound effect on the consumed energy. The energy consumption of an application can vary based on the design of the software, its algorithms, the programming language used and the compiler and its optimizations. Measuring energy consumption of software in general needs specific knowledge about the underlying hardware and also instrumentation, which is not an easy task for the programmers or software engineers.

On the other hand, knowing an estimation of the energy consumption of a program beforehand is very useful. It can help understanding the effect of the code on the energy consumption of the final system without instrumentation or even having access the system.

One approach in the literature has been measuring an average energy usage of a program. This measured value can be reported as an indicator of the average energy usage of software on a specific hardware to the end-user [GGP$^+$14]. On the other hand, a second approach [PS01, B$^+$05, V$^+$14] states in cases where energy is a scarce resource, *energy consumption analysis* is an important factor for the safety of embedded systems. The lack of access to infinite energy source can be either because: 1- The energy source of these embedded systems is not always available and cannot guarantee an infinite operation in energy harvesting systems (e.g., solar powered sensor nodes); or 2- Such embedded systems do not have access to rechargeable energy resources after being deployed (e.g., underwater sensor nodes). These researches suggest that an energy consumption analysis should be performed to guarantee that an embedded system would have enough energy to perform the set of assigned tasks.

In general, the issues with measurement-based methods in timing analysis remains valid in the energy consumption analysis too. Measurement-based methods cannot give a safe upper-bound on the energy consumption of a program.

In order to evaluate if a system can operate in an environment with limited access to energy, an upper-bound on the *worst-case energy consumption* (WCEC) of a program should be generated. The WCEC of a program can be a reliable guarantee to evaluate the safety of the operation of an embedded system in an environment with limited access to energy.

Similar to worst-case execution time, the WCEC of a program is not known in general and WCEC analysis is performed to generate a safe and precise upper-bound on the energy usage of the program.

Although, there has been extensive research on the worst-case execution time analysis of the programs, unfortunately, the current worst-case execution time analysis tools cannot be used to determine the worst-case energy behavior of programs. Due to the complicated micro-architectural features in modern processors, in general, the path resulting in the worst-case execution time is not the same as the path resulting in the WCEC of a program [J+06]. Indeed, it is not true that if a path has a lower execution time, its energy consumption would be less too [H+14b].

5.1.1 Integrated WCEC Analysis

Static analysis methods can generate safe upper bounds for the energy consumption of software provided that a *worst case cost model* for the energy consumption of the underlying hardware is present. Similar to timing analysis accurate energy consumption of a program involves low-level micro-architectural analysis. The WCEC analysis is more difficult compared to the worst-case execution time analysis, since, the complexity of the micro-architectural features can have more effect on the energy consumption of a basic block compared to its execution time. In other words, energy consumption is more data dependent, meaning that a small change in the value of a variable might result in a significant difference in the energy consumption of a certain path. As a result, the methods utilizing fixed point computation in WCEC analysis might lead in a greater overestimation.

This point brings us to the main proposal of this chapter, where we propose that in the domain of energy consumption (where the nature of a consumed resource is very much data dependent), *integrated resource analysis* is more plausible compared to other methods. Since it will remain sound and yet it is able to generate precise bounds on the worst-case energy consumption behavior of a program. In the following sections, we will present our resource analysis customized for WCEC analysis. Our WCEC analysis:

1. Is able to remain precise (to a great extent) concerning the effect of data-dependency on the consumed energy across different paths[1].

2. Extends the cache analysis presented in Chapter 3 with first in-order and then out-of-order superscalar pipeline analysis as an important micro-architectural feature affecting the energy consumption in programs.

5.1.2 Pipeline Analysis

Many researches have been performed on integrating pipeline analysis into worst-case analyses (details are available in survey [W+08]). In general, most of these analyses, for scalability reasons, perform pipeline analysis and the analysis of other micro-architecture elements such as caches separately. However, for accuracy reasons, all micro-architectural elements should be analyzed in an integrated analysis[2].

Most of the energy harvesting embedded systems are built using in-order processors (due to their lower energy consumption compared to out-of-order processors). In in-order pipelines, all of the pipeline stages execute instructions based on the instruction order, while this is not the case in out-of-order pipelines.

[1]Our analysis does not consider battery tailing effect.

[2]In the presence of complicated patterns such as out-of-order execution, the state of one micro-architectural feature can affect the internal state of other micro-architectural features [H+03]. So, specially for out-of-order pipelines, integrated low-level analysis should be performed to ensure the soundness of the analysis.

In this chapter, we first focus on presenting an integrated analysis where high-level path analysis and low-level analysis of cache and in-order pipeline are performed at the same time. Our aim is to show that such an analysis would remain scalable in the presence of cache and in-order pipeline. We should note that we follow the abstract cache semantics from [SF99]. By performing pipeline analysis during the symbolic execution tree traversal, we also model the following features:

- **Superscalarity**
- **Worst-case Energy Consumption Analysis**
- **The Multiplicity of Resources:** Our processor model can contain two ALUs, and arithmetic instructions (except multiply and divide) can be issued to any of them.
- **Resource Capacity:** Some resources such as buffers and queues have limited capacity and thus introduce dependencies between instructions in an earlier pipeline stage and other instructions at a later stage.
- **Instruction and Data Cache:** We have modeled instruction and data cache and their interactions with the pipeline.

We model a processor with in-order superscalar pipeline, instruction and data cache. Our model can be parametrized w.r.t to the cache configuration, the number of entries in the instruction window, the latency of the functional units, etc. The pipeline has a standard 5-stage pipeline consisting of Instruction Fetch (IF), Instruction Decode & Dispatch (ID), Instruction Execute (EX), Write Back (WB) and Commit (CM).

Figure 5.1 presents the in-order superscalar processor which we will be using in the analysis framework in this chapter. The pipeline consist of five stages:

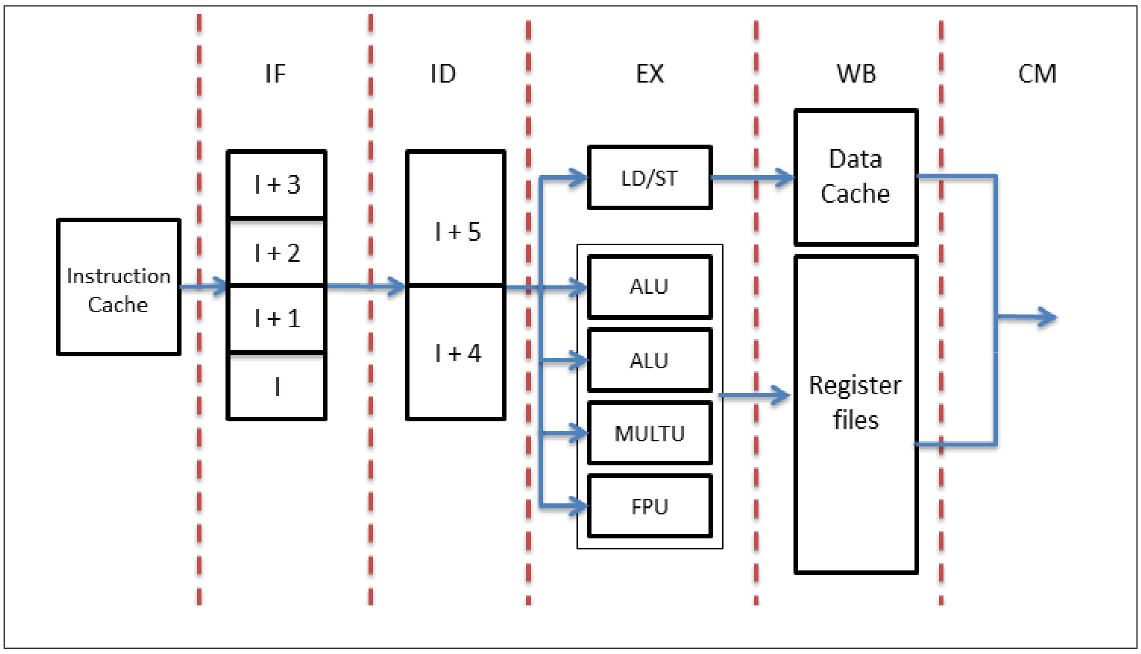


Figure 5.1: In-order processor model

1. **Instruction Fetch (IF)**: In this stage, instructions are fetched from the memory in the program order into the instruction fetch buffer. We assume a 4-entry buffer in this chapter which can load two instructions into the buffer per clock cycle.

2. **Instruction Decode & Dispatch (ID)**: In this stage, instructions are decoded and dispatched in program order for execution. Two instructions at most can be decoded and dispatched per clock cycle.

3. **Instruction Execute (EX)**: At this stage the earliest instruction is issued to its corresponding functional unit for execution. A new instruction can be assigned to an execution unit when the previous instruction in the unit has

finished execution. The components in the processor model are two ALU units, one MULTU unit for integer multiplications, one FPU unit for floating point multiplications and one unit for load/store instructions.

4. **Write Back (WB)**: At this stage the execution of an instruction has finished and the results are written to the registers or passed to the data cache.

5. **Commit (CM)**: In this stage, the instructions which have finished writing their output to the registers and are removed from the pipeline. Two instructions can be committed per cycle.

In the end of this chapter, we will present the steps needed to extend our analysis to out-of-order execution in Section 5.6.

5.1.3 Analysis of Loop-free Programs

We first focus on a sound analysis of loop-free programs. Our analysis performs a depth-first traversal of the symbolic execution tree of a program. At each node in the tree, a unique instruction and data cache, and a pipeline state are assigned to the nodes (as seen in Figure 5.3). Our analysis explores the symbolic execution tree and reports the resource cost of the longest path.

Unlike [The04, H^{+}03] where a pipeline state could have many successor states, our analysis usually can generate one precise successor pipeline state. The only time where more than one successor pipeline state may be generated is when a data cache access points to a memory range access. However, since we do not perform merge on machine states in our analysis[3], more than one successor state is relatively rare and in our experiments only in two of the benchmarks a pipeline

[3]We only merge the program context in the end of the loops. So, in loop-free programs we do not perform merge at any point and in programs with loops the machine states are not merged.

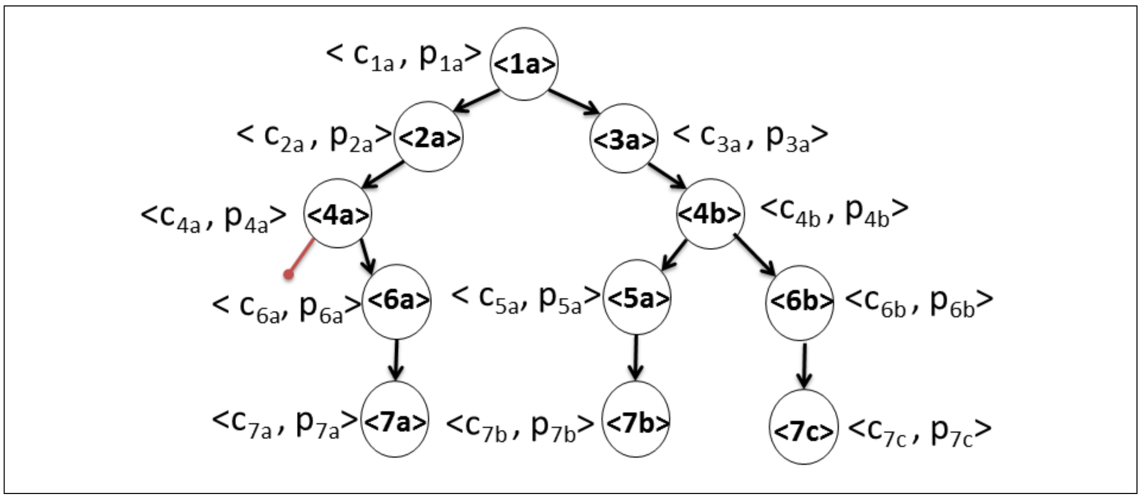


Figure 5.2: Full Symbolic Execution Tree

state had more than one successor state.

In Chapter 3 we employed *reuse with interpolation and dominance* to perform exhaustive symbolic execution in an environment where the contribution of each basic block to the worst-case resource consumption was *dynamic* w.r.t. the context. However, it only covered cache states. In this chapter, we extend reuse to be performed in the presence of *cache and in-order pipeline*.

Figure 5.3 informally depicts a symbolic execution tree, where each triangle presents a subtree. The contexts for the left and right subtrees (s_0 and s_1) contain the machine state reaching the subtree. The machine state for the left and right subtrees are depicted by $m_0 \equiv \langle c_0, p_0 \rangle$ at ℓ_0 and $m_1 \equiv \langle c_1, p_1 \rangle$ at ℓ_1 containing the cache and pipeline states.

In our setting, we keep the interpolant and witness test from Chapter 3 and we enhance the domination test to be able to express the dominance of the witness path w.r.t. the incoming context. We now explain the dominating condition.

If at s_1 the pipeline and cache states p_1 and c_1 were exactly the same as p_0 and c_0, we could safely reuse the summarization of s_0. However, in general, such cases

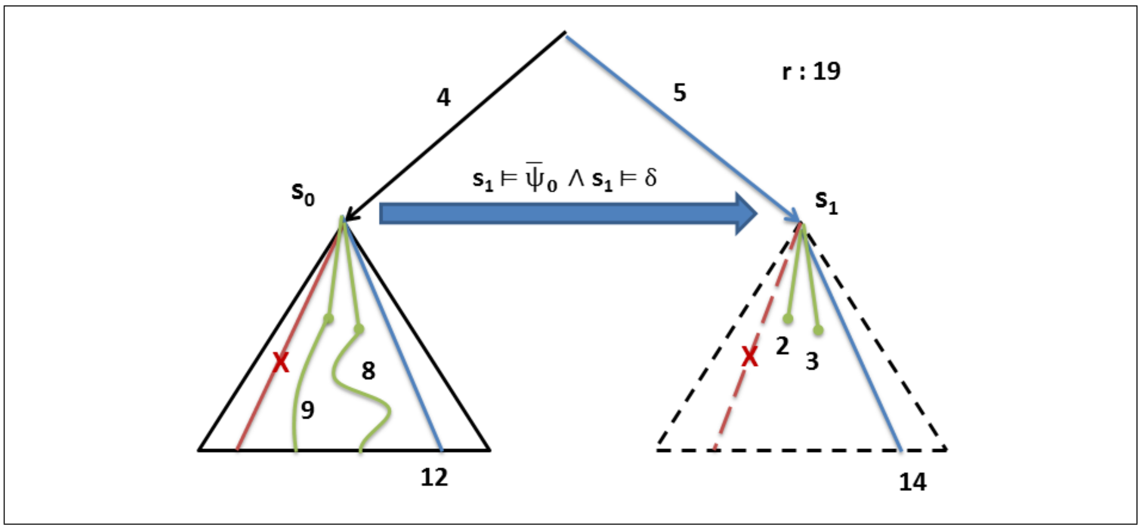


Figure 5.3: Reuse Step

do not happen often. The instructions in the pipeline state might be different, and also the memory blocks in the cache can be different.

We define *successor nodes* inside a subtree as nodes where the pipeline state at these nodes does not contain any instruction from before the subtree. Successor nodes are chosen in all feasible paths except the witness (highlighted with green color in Figure 5.3). We can check if the pipeline state at successor nodes of s_1 (successor states of p_1) remain similar enough to the successor states of p_0. If so, the witness path will remain the dominating path in the new context. For the successor states of p_1 to remain similar enough to the successor states of p_0, we will compare the energy cost of the witness under the new cache and pipeline state against *the maximum possible energy cost of other paths from successor nodes* plus *the energy cost of the path from the root to the successor nodes.* If the energy cost of the witness is more, we conclude the cache and pipeline state are similar enough to maintain the dominance of the witness. As illustrated in Figure 4.9, the energy cost of the witness is 14, while the sum of (1) the highest energy cost of the other paths from the root of the subtree to the successor nodes and (2) the maximum energy cost of the rest the path is $(3+8 \equiv 11)$. Note that the maximum possible energy cost of the paths (from successor nodes to the end of the subtree) is computed based on the worst-case cache state at the successor node.

5.1.4 Extension to Programs with Loops

We adopt the loop unrolling framework presented in Chapter 2. However, unlike [R+06], where in analyzing loops, the contexts are merged at the merge points (the end of loop iterations in our framework), here to remain sound the machine states are not merged and all the contexts in the end of a loop iteration are passed to the next iteration.

5.2 Related Work

5.2.1 Worst-case Energy Consumption Analysis

Cycle-accurate power simulators such as Wattch [BTM00] and Simplepower [Y+00] perform power simulation for one path and in general are not able to generate a safe estimation of the WCEC for a program. Wattch produces estimations of energy consumption by modeling micro-architectural systems such as cache and branch prediction. It can model software on different architectures with the precision of up to 10% of the commercial low-level hardware modeling tools. Furthermore, JoulTrack [SC01] is an Instruction-level energy estimation method which does not consider the effect of cache miss or branch prediction.

The work presented in [GGP+14] develops a static analyzer that works on LLVM IR. Their analysis generates cost relations which represent the cost of running the program in terms of its input. These cost relations are then solved to generate the energy consumption.

Other promising works such as [W$^+$15, J$^+$06] perform WCEC analysis. The work presented in [J$^+$06], to the best of our knowledge, is the first research presenting WCEC analysis. On the other hand, [W$^+$15] generates a comprehensive approach to WCEC analysis. It presents a method that enables WCEC computations based on relative energy models and measurements. It performs WCEC estimates by combining different techniques for three different hardware platforms. However, these works are only applicable to in-order pipelines.

The majority of the literature on WCEC analysis (like us), assume that a worst-case cost model is either present or can be generated. Recall from Chapter 1 that modular approaches can generate a worst-case energy consumption for each basic block in the low-level analysis phase. Although, this has been a valid assumption among the resource analysis community, the researchers from the hardware community doubt this point. Such researchers believe that due to the high level of data dependency in the energy consumption domain, a worst-case energy consumption value generated for a basic block in the low-level phase is either not safe or very imprecise [PKME15]. As an example, there have been examples where the energy cost of a certain path returned as the WCEC path has had a different energy cost (differing up to 20 % based on the input data) [KE15]. This issue seems to be a valid point and the current approaches can not perform full data-dependent WCEC analysis. This issue is an open problem and can be a topic for future research. In industry is solved, by adding a safety margin to the generated bound by WCEC analyses.

5.2.2 Out-of-order Pipeline Analysis

In literature, many works focus on in-order pipelines. However, the number of works addressing both in-order and out-of-order pipeline analysis is limited. We will review some of the more important methods.

Lundqvist et al. in [LS98, LS99] proposed an integrated approach using symbolic execution for computing the WCET bounds where all possible contexts where enumerated. Although, their method is able to generate safe bounds for complicated execution patterns in pipelines, it does not scale to a reasonable level.

Three important algorithms have been proposed to address out-of-order execution in pipelines. The first work presented in [The04, H$^+$03] modeled the out-of-order processor PowerPC 755. It generates reduced concrete pipeline model for performing pipeline analysis. The reduced concrete pipeline state are maintained for each basic block and are updated along the program points. However, a pipeline state may have up to several different successor states where certain memory accesses cannot be resolved to cache hit/cache miss. The analysis is sound, however, it can take a long time (12 hrs on average for an avionics program) as reported in [S$^+$07] and consume a huge amount of memory.

The work presented by Li et al [LRM06], first identifies the dependency among pipeline stages and then uses an iterative technique to estimate the earliest/latest start and completion time of each basic block in the presence of out-of-order execution by an interval. The process continues till it reaches fix-point. Although, their method considers overlapping between the execution of basic blocks in pipelines, the paper does not present any proof that their method generates safe bounds. Furthermore, their model does not take into account data cache analysis.

Rochange and Sainrat [RS09b] report in their experiments that the bounds returned by [LRM06] is quite imprecise. Their method is an extension over [LRM06] which attempts to decrease this imprecision by incorporating a set of parameters to represent the resource usage of pipeline resources in each basic block. They present a proof in their paper. However, their model only takes into consideration out-of-order execution in pipelines and assumes perfect cache.

Our method performs integrated WCEC analysis and considers (in-order and out-of-order) pipeline, instruction and data cache states at the same time.

5.3 General Framework

In Chapter 2, we presented our resource analysis framework. In this section, we will customize the analysis framework for WCEC analysis with emphasis on instruction and data cache, and in-order superscalar pipelines.

5.3.1 Basic Operations and Energy Cost Model

Given a program point ℓ, an operation $op \in Ops$, and a symbolic store σ, we denote the sequence of instructions in op by $instr$ and the function $acc(\ell, op, \sigma)$ denotes the sequence of memory block accesses by executing op at the symbolic state $s \equiv \langle \ell, m, \sigma, \cdot \rangle$. In this chapter for short, we denote $acc(\ell, op, \sigma)$ by $memseq$ and the sequence of instructions by $instr$.

In WCEC analysis the variable of interest r models the consumed energy. Note that this variable is always initialized to 0 and the only operation allowed upon it is a constant increment.

The WCEC analysis computes a sound and accurate bound for r in the end, across all feasible paths of the program. Given a symbolic state $s \equiv \langle \ell, m, [\![s]\!] \rangle$ and a transition $tr : \ell \xrightarrow{op} \ell'$, the amount of increment at s by executing tr will be evaluated by the energy consumption in different components in the superscalar in-order processor and other energy consuming components. r is not used in any other way.

We have presented the energy cost model that we will be using through this chapter in Appendix B. In our cost model, we have utilized parts of the energy cost model presented in [J^+06]. However, we have updated the cost model to fit our analysis framework. We also have indicated the parts where our analysis can compute the energy cost more accurately.

5.3.2 The Machine State

In this chapter, the machine state consists of the data and instruction cache and the pipeline. As a result, we define the machine state attached to each symbolic state to be a tuple of the form $\langle c_i, c_d, p\rangle$, where c_i and c_d are depicting instruction and data caches and p is the pipeline state. As a result, a symbolic state would be depicted by $\langle \ell, m, \sigma, \Pi\rangle$ where $m \equiv \langle c_i, c_d, p\rangle$.

A concrete pipeline state can be modeled by a set of $\langle ins, s, t\rangle$ tuples where ins depicts an instruction, s depicts a pipeline stage and t depicts the time remaining to finish executing instruction ins in stage s. A pipeline state then would be modeled as $[\langle ins_i, s_j, t_1\rangle \ldots \langle ins_k, s_l, t_2\rangle]$ which indicates that ins_i is at state s_j, taking t_1 clock cycles to finish its execution in stage s_j and so on.

An update function $\mathcal{U}$ can be defined which cycle-wise updates a pipeline state based on a previous pipeline state p, the set of instructions in a basic block $inst_B B$ and the dependency graph of a basic block dep: $p = \mathcal{U}(p', inst, dep)$

By cycle-wise update we mean that the pipeline state p is updated from cycle to cycle till reaching p'.

The sequence of instructions, $instr$, is used to update the pipeline state and the instruction cache access. The updMachineState function updates both the cache and pipeline states. The updMachineState function which was used to update the cache state in Chapter 3 is used to update both the instruction and data cache states. For the pipeline state, the pipeline is updated following the standard process in in-order superscalar processors.

5.3.3 Witness and Dominating Condition

In WCEC analysis, the witness (Γ) is the sequence of the program points demonstrating the most energy consuming path in an explored subtree. In WCEC analysis and due to the existence of in-order pipeline, the contribution of the $Energy_{reg}$, $Energy_{wk}$, $Energy_{FPU}$ and $Energy_{MULTU}$ is only affected by the sequence of the instructions on the path. As a result, the contributions of these energy consuming elements *remain static* regardless of *the machine state* reaching a node. Moreover, although $Energy_{ALU1}$ and $Energy_{ALU2}$ would alter based on the machine state, $Energy_{ALU1} + Energy_{ALU2}$ would remain constant.

In the presence of in-order pipeline, the rest of the energy consuming elements in a path cannot be considered static. As a result, witness is stored as the *sequence of the instructions* on the witness path which will be used to dynamically measure the energy consumption of these elements based on the incoming machine state plus the energy cost of static items. The energy of these dynamic elements are depicted by the energy in instruction cache $Energy_{ic}$ and data cache $Energy_{dc}$, the clock energy $clock_{path}$, selection logic energy $selection_{path}$, the leakage energy $leakage_{path}$ and sum of the switch-off energy in different components $Switchoff$. As a result, the energy consumption of a witness path is obtained dynamically and can alter based on the incoming machine state.

In WCEC analysis, the dominating condition (δ) should guarantee the dominance in the presence of both cache states and pipeline state. As briefly explained in Section 5.1.1, the dominating condition would reason about the energy cost of the witness based on the machine state at reuse point compared to the maximum energy cost from the successor nodes to the end of the subtree plus the energy from the root to the successor nodes.

In general, the energy cost is affected from *the cache states* and *the data dependency* and *contentions* in the in-order pipeline. Due to the in-order nature of the pipeline, the contentions will always result to the earlier instructions to be executed. So, they would be the same regardless of the context.

In order to consider data dependencies, the successor nodes are chosen after a window of instructions, such that the instructions from before the root of the subtree have been committed. As a result, the data dependencies after these successor nodes remains the same and the maximum energy cost of the paths after successor nodes can be computed using empty cache.

The window of instructions from the root to the successor nodes would be the set of epilogue instructions in all the paths emanating from the node at reuse point. In general, since our framework performs analysis on the symbolic execution tree and w.r.t. to the in-order pipeline model, the set of epilogue instructions of such paths would be at most $8 - 1 = 7$ instructions (Figure 5.4).

Finally, the energy cost from the root to successor nodes is computed based on the new machine state which considers the different access time in the cache access time and data dependencies in the new pipeline state.

As a result, the dominating condition is a set of instructions from the root to successor nodes and a respective maximum energy from the successor nodes to the end of the sub-path. At reuse time, the energy cost of these instruction sets is added to the respective maximum energy stored with them and compared to the energy cost of the witness in the new context. In case the energy cost of the witness is still more, the summarization can be reused and the energy cost of the witness is returned (as illustrated in Figure 5.4). In our experiments, the number of successor

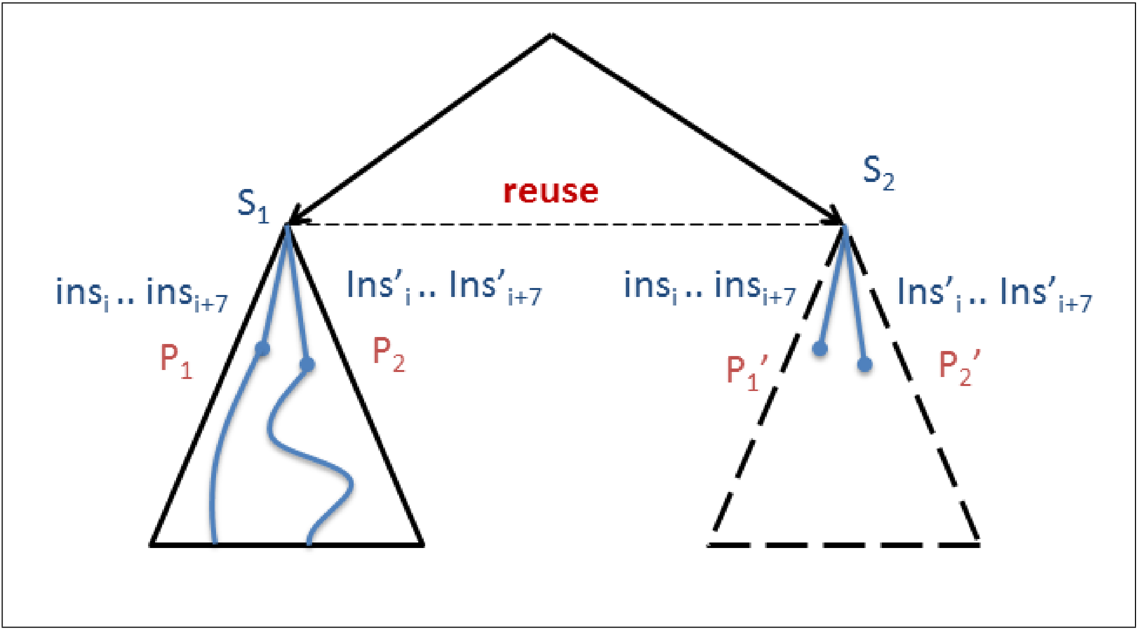


Figure 5.4: Window of Epilogue Instructions from Root to Successor Nodes

nodes are bounded by $O(1)$. We next present the customized Summarize-a-Trans, Combine-Witness and Merge-Witness functions.

```
function Summarize-a-Trans($s, tr$)
      Let $s$ be $\langle \ell, m, \sigma, . \rangle$ and Let $tr$ be $\ell \xrightarrow{op} \ell'$
      Let $m \equiv \{c_i, c_d, p\}$
⟨143⟩ $\Gamma$ := Sequence of instructions in $op$
⟨144⟩ $Energy_p$ := Energy($p$);
⟨145⟩ Execute Instructions in $p$
⟨146⟩ $wcec$ := Energy($p$) $-$ $Energy_p$;
⟨147⟩ return $[\ell, true, \Gamma, wcec, true, op_\Delta]$
```

Figure 5.5: Summarize-a-Trans Function

Summarize-a-Trans Function

Summarize-a-Trans computes a summarization for a single transition tr at state s. This can be seen as a basic step in our algorithm. Because no infeasible path has been discovered, the interpolant Ψ is just *true*. There is a single path, thus the witness is simply the sequence of the instructions in the transition and the dominating condition is also simply *true*. Moreover, the *wcec* is measured by the difference in the energy cost of the pipeline state before and after executing the instructions (lines 144 - 146). The abstract transformer Δ_p is the operation *op* itself, but translated to the language of input-output relation. As an example, y := y + 1 is translated to $y_{out} = y_{in} + 1$. We use op_Δ to denote such translated *op*.

```
function Merge-Witness($s, \Gamma_1, \Gamma_2, \delta_1, \delta_2$)
    Let $s$ be $\langle \ell, m, \sigma, . \rangle$ and Let $tr$ be $\ell \xrightarrow{op} \ell'$
    Let $m \equiv \{c_i, c_d, p\}$
⟨148⟩ if (max($\Gamma_1, p$) $\leq$ max($\Gamma_2, p$))
⟨149⟩     $swap(\Gamma_1, \Gamma_2), swap(\delta_1, \delta_2)$
⟨150⟩ $instr_{succ}$ := Generate-Succ($\Gamma_2$);
⟨151⟩ $max_{succ}$ := Max-from-Succ($\Gamma_2$);
⟨152⟩ $\delta$ := Union($\delta_1, \delta_2$)
⟨153⟩ $\delta := \delta \wedge \langle instr_{succ}, max_{succ} \rangle$
⟨154⟩ return $\{\Gamma_1, \delta\}$
end function
```

Figure 5.6: Merge-Witness Function

The Combine-Witness and Merge-Witness Functions

The Merge-Witness function in Figure 5.6, presents the step to merge horizontally two witnesses and their respective dominating conditions. Due to the complicated nature of WCEC analysis with pipeline, in order to choose the dominating path, the pipeline state is also needed. In the first step, based on the energy consumption of each of the witnesses the longer witness is chosen (line 148 to 149). We now elaborate on the computation of the dominating condition. The dominating condition is consisted of the instructions to the successor node and the max energy cost from the root of the successor node to the end of the path. The union of the dominating conditions are computed (line 152) and next, a successor node is generated on the second witness and the instructions from the root to the successor node (generated in line 150) and the maximum energy cost from the successor node to the end of the subtree (generated in line 151) is added to the dominating condition (line 153).

```
function Combine-Witness($s, tr, \Gamma', \delta'$)
      Let $s$ be $\langle \ell, m, \sigma, . \rangle$
      Let $tr$ contain $\{instr\}$
      Let $m \equiv \{c_i, c_d, p\}$
⟨155⟩ $\Gamma = \Gamma'$
⟨156⟩ foreach $ins_i \in instr$ do
⟨157⟩     $\Gamma = \Gamma + ins_i$
      endfor
⟨158⟩ $\delta := [\,]$
⟨159⟩ foreach $\langle instr_{succ}, max_{succ} \rangle \in \delta'$ do
⟨160⟩     $\delta :=$ Union($\delta$ , update-succ($instr_{succ}, max_{succ}, \Gamma$))
      endfor
⟨161⟩ return $\{\Gamma, \delta\}$
end function
```

Figure 5.7: Combine-Witness function

The Combine-Witness function in Figure 5.7 combines two witnesses and dominating conditions. Note that tr can be an abstract transition, by that we mean that it is an abstraction of several transitions between program point ℓ_1 to ℓ_2. An abstract transition is the summarization between ℓ_1 and ℓ_2. As a result, in case tr is an abstract transition, $instr$ would be generated from the witness (Γ) in the summarization.

First, for each of the instructions in the respective transition ($instr$), it is added to the beginning of the witness list Γ (line 157). Next, the dominating condition is updated with regard to the new generated witness Γ in line 160. In other words, for each $\langle instr_{succ}, max_{succ} \rangle$ in δ' the update-succ function updates $\langle instr_{succ}, max_{succ} \rangle$. Note that the update-succ function can return more than one $\langle instr_{succ}, max_{succ} \rangle$

sets. The output of the update-succ function is next generated by union with δ.

5.3.4 Generating Machine State Summaries

In this chapter, the machine state summary should be redefined. In Chapter 3, we introduced cache summary which captured the relation between the abstract cache components. The cache summary was needed when a summarization was reused across a loop iteration. In the presence of in-order execution pipelines the pipeline and cache states cannot be merged.

As a result, the witness (the most energy consuming path) is used to generate the pipeline and cache states at the end of the reused subtree. The combination of the cache summaries and witness will act as the machine state summary. Customizing the Combine-Summary and Merge-Summary functions would be quite similar to the functions presented in Chapter 3.

5.4 An Example Analysis

In the example presented in this section, we will illustrate the logic of the reuse step which ensures the dominance of the witness over another path in a subtree with respect ta a new machine state. Moreover, this example clarifies the point that the reuse step is sound.

Consider the CFG and the symbolic execution tree in Figure 5.8. Each rectangle in the CFG represents a basic block where the program point (e.g., $\langle 1a \rangle$) and the instructions in the basic block can be seen. In front of each instruction, the respective functional unit (executing it) is depicted in red. For simplicity, in this example we assume perfect data cache. The pipeline is a scalar 5 stage in-order pipeline, initially empty, where all stages take only 1 clock cycle to complete. The functional units in the EX stage of the pipeline are MEM and ALU. The time taken in all the pipeline stages is equally 1 clock cycle (CC). Note that in Figure 5.8(b), we have not (fully) drawn the subtree below node $\langle 4b \rangle$.

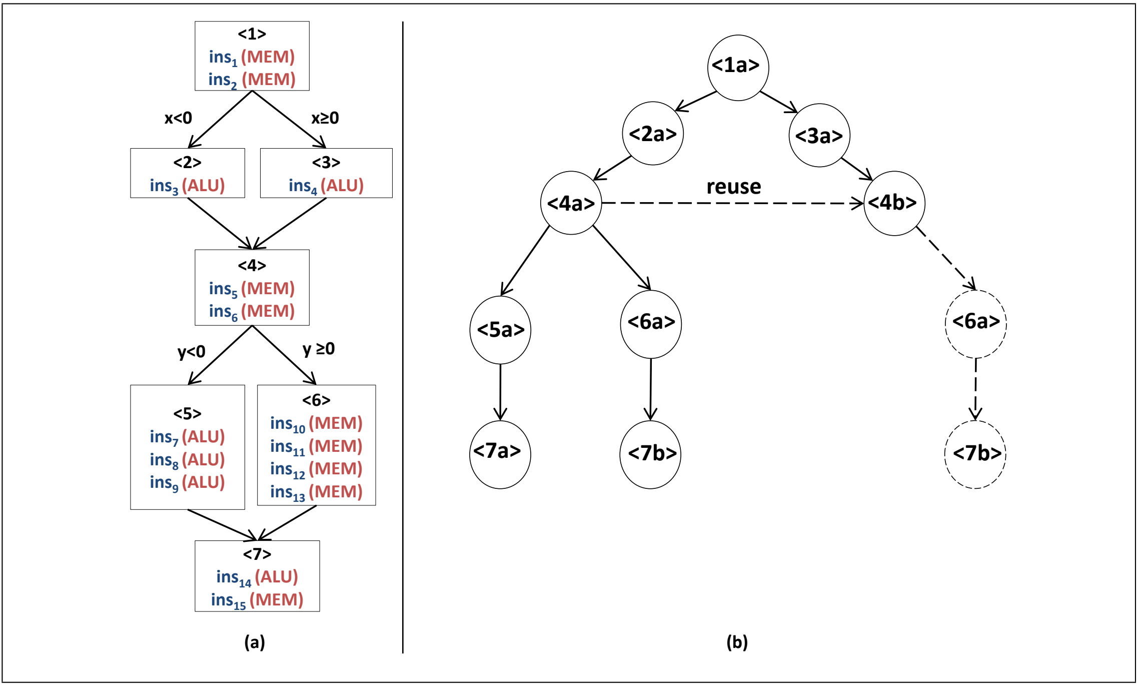


Figure 5.8: (a) a CFG (with instructions and their respective execution unit shown in each block); and (b) Our Analysis Tree

The analysis starts with empty pipeline state at $\langle 1a \rangle$. The pipeline state at $\langle 2a \rangle$ and $\langle 4a \rangle$ is $[\langle ins_1, ID \rangle, \langle ins_2, IF \rangle]$ (CC 2) and $[\langle ins_1, MEM \rangle, \langle ins_2, ID \rangle, \langle ins_3, IF \rangle]$ (CC 3). The analysis continues updating the pipeline state clock-wise till all the instructions leave the pipeline state. The energy cost of the sub-path from $\langle 4a \rangle$ would be 123 mJoules. The pipeline state at each clock cycle can be seen in Table 5.1.

Note that the pipeline state at program point $\langle 7a \rangle$ does not contain any instruction from before the root of the subtree ($\langle 4a \rangle$). As a result, $\langle 7a \rangle$ is chosen as the successor node. The instructions between the root $\langle 4a \rangle$ and the successor node $\langle 7a \rangle$ are $ins_5, ins_6, ins_7, ins_8, ins_9$. Moreover, the maximum energy cost from the

Table 5.1: Pipeline States Along the First Path

PP	**Pipeline State**	**Clk**
$\langle 4a \rangle$	$[\langle ins_1, MEM \rangle, \langle ins_2, ID \rangle, \langle ins_3, IF \rangle]$	3
	$[\langle ins_1, WB \rangle, \langle ins_2, MEM \rangle, \langle ins_3, ID \rangle, \langle ins_5, IF \rangle]$	4
$\langle 5a \rangle$	$[\langle ins_1, CM \rangle, \langle ins_2, WB \rangle, \langle ins_3, ALU \rangle, \langle ins_5, ID \rangle, \langle ins_6, IF \rangle]$	5
	$[\langle ins_2, CM \rangle, \langle ins_3, WB \rangle, \langle ins_5, MEM \rangle, \langle ins_6, ID \rangle, \langle ins_7, IF \rangle]$	6
	$[\langle ins_3, CM \rangle, \langle ins_5, WB \rangle, \langle ins_6, MEM \rangle, \langle ins_7, ID \rangle, \langle ins_8, IF \rangle]$	7
$\langle 7a \rangle$	$[\langle ins_5, CM \rangle, \langle ins_6, WB \rangle, \langle ins_7, ALU \rangle, \langle ins_8, ID \rangle, \langle ins_9, IF \rangle]$	8
	$[\langle ins_6, CM \rangle, \langle ins_7, WB \rangle, \langle ins_8, ALU \rangle, \langle ins_9, ID \rangle, \langle ins_{14}, IF \rangle]$	9
	$[\langle ins_7, CM \rangle, \langle ins_8, WB \rangle, \langle ins_9, ALU \rangle, \langle ins_{14}, ID \rangle, \langle ins_{15}, IF \rangle]$	10
	$[\langle ins_8, CM \rangle, \langle ins_9, WB \rangle, \langle ins_{14}, ALU \rangle, \langle ins_{15}, ID \rangle]$	11
	$[\langle ins_9, CM \rangle, \langle ins_{14}, WB \rangle, \langle ins_{15}, MEM \rangle]$	12
	$[\langle ins_{14}, CM \rangle, \langle ins_{15}, WB \rangle]$	13
	$[\langle ins_{15}, CM \rangle]$	14

successor node $\langle 7a \rangle$ till the end of the path is stored with the instruction window: $[\langle 7 \rangle, \langle ins_5, ins_6, ins_7, ins_8, ins_9 \rangle, 108]$.

Moving to the right sub-path, the energy cost of the second path from $\langle 4a \rangle$ would be 125 mJoules and the pipeline state at the program points can be seen in Table 5.2.

Table 5.2: Pipeline States Along the Second Path

PP	**Pipeline State**	**Clk**
$\langle 4a \rangle$	$[\langle ins_1, MEM \rangle, \langle ins_2, ID \rangle, \langle ins_3, IF \rangle]$	3
$\langle 6a \rangle$	$[\langle ins_1, CM \rangle, \langle ins_2, WB \rangle, \langle ins_3, ALU \rangle,$	5
$\langle 7b \rangle$	$[\langle ins_6, CM \rangle, \langle ins_{10}, WB \rangle, \langle ins_{11}, MEM \rangle, \langle ins_{12}, ID \rangle, \langle ins_{13}, IF \rangle]$	8

After traversing the second path, similar to the first path an instruction window and maximum energy for the rest of the path is generated and stored: $[\langle 7 \rangle, \langle ins_5, ins_6, ins_{10}, ins_{11}, ins_{12}, ins_{13} \rangle, 52]$.

Continuing to $\langle 4a \rangle$ the two paths are compared and the path on the right is chosen as the witness. Next, the $[SuccNode, Witness, MaxEnergy]$ of the other path is stored as the dominating condition.

Fast forwarding the analysis to the right subtree at $\langle 4b \rangle$, this dominating condition is tested on the pipeline state $[\langle ins_1, MEM \rangle, \langle ins_2, ID \rangle, \langle ins_4, IF \rangle]$ (CC 3). Assume that there is some dependency between the nodes $\langle ins_4, CM \rangle$ and $\langle ins_5, MEM \rangle$.

The energy cost of the witness with regard to the current pipeline state is computed which is 128 mJoules. Next, the energy cost of the instructions in the instruction window in the new pipeline state is computed which is $71 + 3 \equiv 74$ mJoules this time. The energy cost is added to the maximum energy stored with the instruction window and compared to the energy cost of the witness: $74 + 52 < 128$. Since the energy cost of the witness is larger we can reuse and return the energy cost of the witness as the WCEC of the subtree.

The analysis is continued and we can come to the WCEC of the whole program which is 150.

5.5 Experimental Evaluation

We used an Intel Core i5 @ 3.2Ghz processor having 4Gb RAM for our experiments and built our system upon CLP($\mathcal{R}$) [JMSY92] and Z3 as the constraint solver, thus providing an accurate test for feasibility. The analysis was performed on LLVM IR which, while being expressive enough, maintains the general CFG of the program. Our framework receives a C program as input and produces a transition system from the LLVM IR of the program. The LLVM instructions are simulated for a RISC architecture. We use Clang 3.2 [Cla14] to generate the IR. The data and instruction cache settings in our experiments were similar to Chapter 3 and the pipeline settings have been presented in Section 5.1.2.

Table 5.3 presents the results of our analysis on a set of benchmarks. The benchmarks used in our experiments are from embedded programs (`fft1` from MiBench [G+01] and the rest from [Mö06]). The second column presents the size parameter of the benchmarks. We compare the result of our analysis framework unroll_d with the cycle accurate power simulator Wattch [BTM00]. The third column reports the result of simulation on Wattch. The *state* and *reuse* columns in Table 5.3 present the number of visited states and reused states in the analysis. The time column reports the analysis time and the WCEC column reports the worst-case energy consumption for the benchmarks.

5.5.1 Discussion on Precision

We have compared our analysis results with the Wattch power simulator. As it can be seen in Table 5.3 and Figure 5.9, in all cases our analysis result overestimates the Wattch simulations. Among the benchmarks, we were able to simulate the worst-case path on **edn**, **matmult**, **ns**, **fdct**, **bubblesort** and **jfdctint**. On these benchmarks, the WCEC estimation by our analysis is compared to the actual WCEC of the program. The results show in average our analysis results have 73% overestimation. On the other benchmarks, finding the worst-case path is not trivial. So, we compared our analysis result with the simulation of the benchmarks without changes. As a result, in some cases the simulated path might be far from the WCEC path.

Table 5.3: Results of Analysis of Benchmarks with In-order Pipeline

Benchmark	**Size**	**Wattch Simulation (nJ)**	**Unroll_d**			
			Time	**State**	**Reuse**	**WCEC (nJ)**
edn		1964967.475	606.42	2047	571	3909066.00
compress		170967.348	268.64	1666	398	15205680.00
ndes		634466.991	27.11	714	246	1848224.00
adpcm		Not Runnable	262.51	1100	255	265783.00
fft1	**8**	181638.497	2.87	313	51	546861.00
matmult	**20**	4379671.380	2.16	462	114	7290374.00
ns	**5**	358722.764	0.11	78	18	645745.80
ns	**10**	6165369.361	0.24	142	40	9531186.00
ns	**20**	107642166.000	0.6	266	78	147183400.00
fir		364811.870	3.78	343	129	685058.00
ud	**5**	301514.463	1.41	514	53	302174.40
expint		118812.395	25.26	867	247	147140.00
cnt	**10**	Not Runnable	0.25	142	38	133072.80
fdct	**8**	125910.057	0.14	56	14	205605.60
jfdctint		302495.221	1.41	251	77	354978.80
bubblesort	**25**	311460.787	0.62	194	40	314025.60
bubblesort	**50**	732892.352	2.41	387	84	1279203.80
bubblesort	**100**	1518204.211	13.02	773	172	5163788.00
two_shapes	**50**	165419.981	2.02	375	47	179106.20
two_shapes	**100**	428837.735	10.73	750	97	695741.20
two_shapes	**200**	1455017.221	89.09	1500	197	2742264.00

5.5.2 Discussion on Scalability

The three groups of the benchmarks used in our experiments are separated by a double line:

Benchmarks with Long Witnesses: The first group contains benchmarks with mainly long nested loops. The length of the witness becomes quite long in these benchmarks and in order for our analysis to scale, we had to do some adjustments for scalability in the analysis of these benchmarks. We overestimate the later part of the witness to force its size to remain within our threshold. These group of benchmarks illustrate the performance of our method in analyzing benchmarks

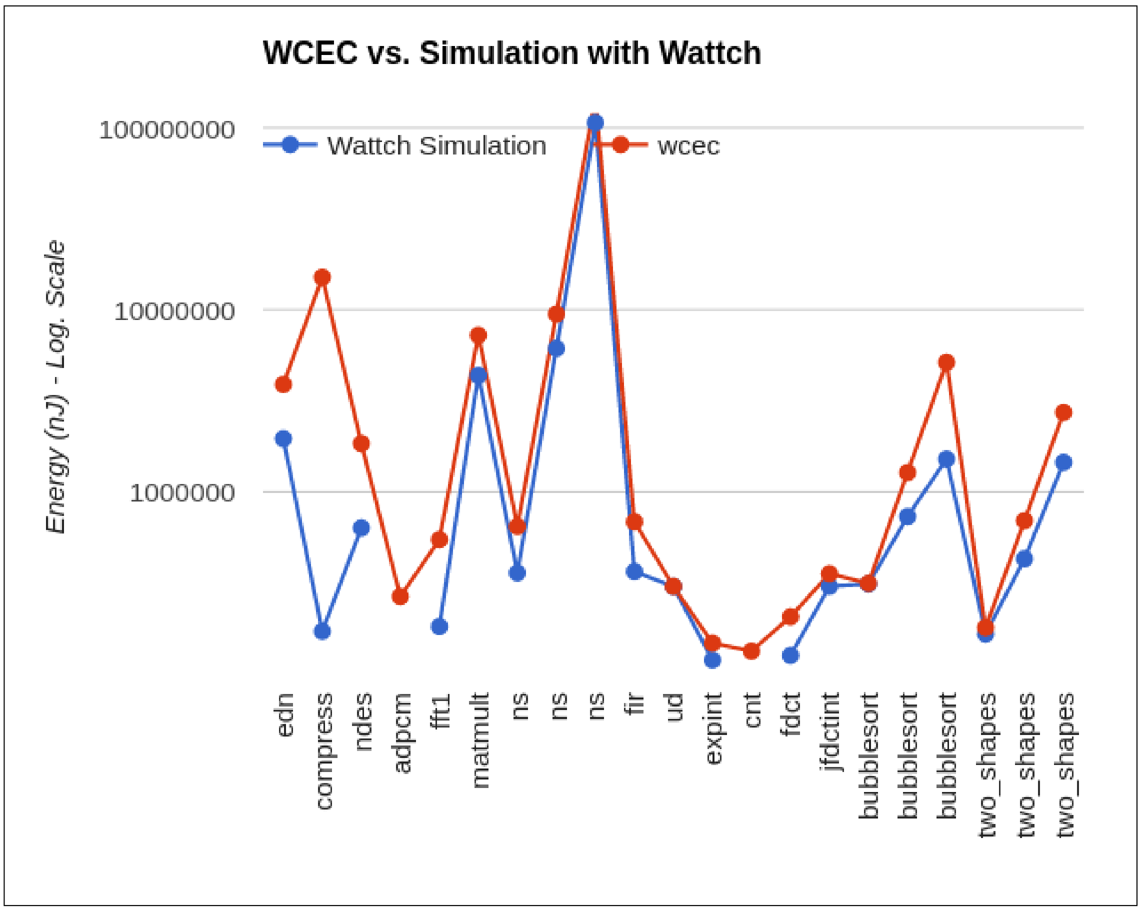


Figure 5.9: Comparison of Estimated WCEC with Wattch Simulation

with very long paths. Due to the large number of infeasible paths in these benchmarks, the analysis of these benchmarks can also determine the precision of our proposed analysis.

Benchmarks with Long Witnesses and Complicated Loops: This group of benchmarks contains standard programs from [MÖ6] which although contained long witnesses, the length of the witness did not exceed the threshold. The reason is that in these benchmarks, the number of the iterations of the complicated loop patterns are precisely measured by our analyses loop unrolling technology. As a result, the length of the witness remains below the threshold and the analysis can generate precise results for these benchmarks.

Academic Benchmarks: Although, the loops in these benchmark programs are considered to be simple, they contain memory accesses which might be resolved to a range of memory addresses, leading to the imprecision in the low-level analysis approaches that employ a fixed point computation.

In conclusion, with the adjustment to handle long witness paths our framework is able to scale on reasonable size benchmarks. This adjustment can affect the precision of our results for benchmarks analyzed with the adjustment, but still the overestimation falls in an acceptable range.

One other limitation of our framework is that the processor model does not consider all micro-architecture features. In comparison with the Wattch simulator, we tried to compare our analysis with a similar processor setting. We had to add bounded overestimation to handle the missing micro-architecture features (such as branch prediction). Although, our estimations would still be sound, but a more accurate processor model can increase the precision of our analysis. In other words, our analysis is not bounded to a specific pipeline model, but with a more accurate processor model a more precise analysis can be provided.

5.6 Extension to Out-of-order Pipelines

In Section 5.1.2, we stated that most of embedded systems with limited access to energy operate on in-order pipelines. However, a precise and scalable out-of-order pipeline analysis has been long an open problem. We explored three state-of-the-art out-of-order pipeline analysis methods in Section 5.2.2 and we elaborated on the existing issues in out-of-order pipeline analysis.

Now, we present a further discussion on extending the analysis framework presented in this chapter to out-of-order pipelines. We update the dominating condition to preserve soundness in the presence of timing anomaly introduced by out-of-order pipelines.

The extension in this section is presented to demonstrate that our WCEC analysis framework can also be performed on out-of-order superscalar pipelines, instruction and data cache at the same time. We like to note that, in order to remain sound we need to be conservative in the reuse step. This might affect the scalability of our analysis on out-of-order pipelines.

Figure 5.10 presents the out-of-order superscalar processor which we will be using in the analysis framework in this section. The pipeline has a standard 5-stage pipeline consisting of Instruction Fetch (IF), Instruction Decode & Dispatch (ID), Instruction Execute (EX), Write-Back (WB) and Commit (CM). The instruction fetch, decode, and commit stages are performed in program order, and the execute and write-back stages, instructions can be processed out-of-order:

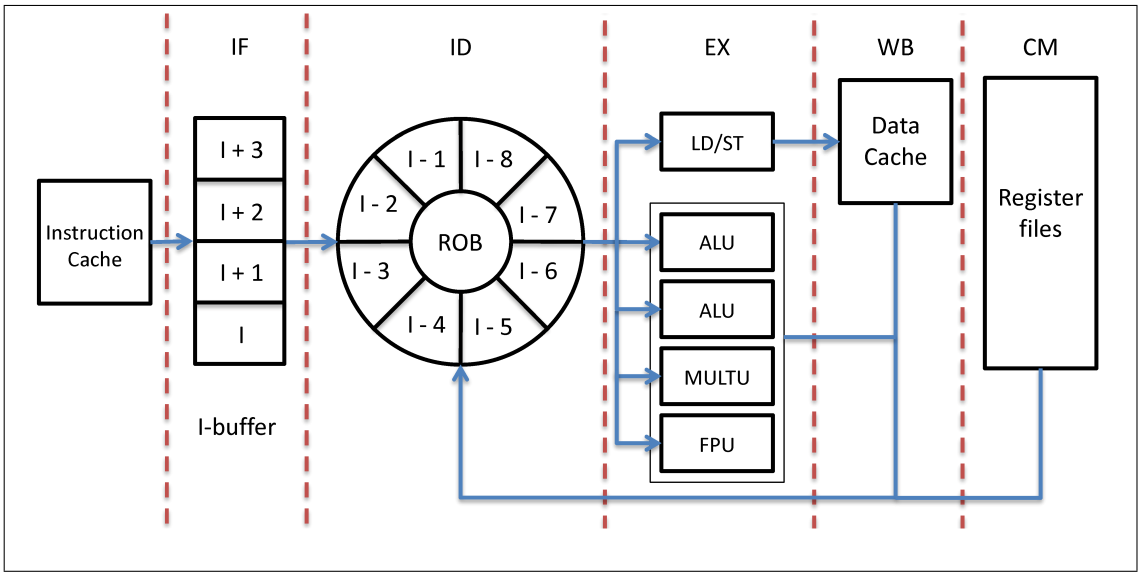


Figure 5.10: Out-of-order processor model

1. **Instruction Fetch (IF)**: In this stage instructions are fetched from the memory in the program order into the instruction fetch buffer I-buffer. We assume a 4-entry I-buffer in this section which can load two instructions into the I-buffer per clock cycle.

2. **Instruction Decode & Dispatch (ID)**: In this stage instructions are decoded in the I-buffer and dispatched in program order into the ROB buffer. We assume an 8-entry buffer in this section. Instructions are stored in this buffer from the time they are dispatched to the time they are committed. Two instructions at most can be decoded and dispatched per clock cycle.

3. **Instruction Execute (EX)**: At this stage, the earliest instruction in the ROB, which its arguments are ready is issued to its corresponding functional unit for execution. This stage executes the instructions in an out-of-order behavior, and a new instruction can be assigned to an execution unit when the previous instruction in the unit has finished execution. The components in the processor model are two ALU units, one MULTU unit for integer multiplications, one FPU unit for floating point multiplications and one unit for load/store instructions.

4. **Write-Back (WB)**: At this stage the execution of an instruction has finished and the results are passed to the following waiting instructions in the ROB buffer. In case all other operands of an instruction in the ROB are ready, the instruction could be executed in the next cycle. For the sake of simplicity, we will assume that an instruction will write-back immediately after finishing execution. The WB stage is also an out-of-order stage.

5. **Commit (CM)**: In this stage the instructions which have finished the WB

stage write their output to the registers and free the ROB register in the program order. Two instructions can commit per cycle.

5.6.1 Timing Anomaly

Out-of-order execution in current modern processors can result in *timing anomaly*. Timing anomaly is referred to the case where a local worst-case/best-case might not result in global worst-case/best-case. Consider an instruction I with the possible execution times of E_1 and E_2, which lead to different worst-case execution time estimations T_1 and T_2. E_1 and E_2 can be different due to some reasons such as cache hit/cache miss, etc. A timing anomaly happens when $T_1 > T_2$, but $E_1 < E_2$. Generally, in the worst-case execution time analysis, we need to keep track of the state of the hardware components and their possible effects on the overall worst-case execution time.

Three types of timing anomaly have been identified by Reineke et al. [R+06]. Scheduling timing anomaly, which has been thoroughly studied, happens when a faster execution time of a basic block leads to the global worst case. The second form of timing anomaly, speculation timing anomaly, where an initial cache hit changes the order of executed instruction and an increase in the overall execution time. Finally, the third type is cache timing anomaly which is caused by non-LRU cache replacement policies.

In the presence of out-of-order pipeline, the effect of timing anomaly should be considered in the worst-case execution time analysis. This issue would still be valid for WCEC analysis and it becomes worse since energy is more data dependent compared to timing.

5.6.2 Dominating Condition and Machine State Summary

In the presence of out-of-order pipeline, the dominating condition, and the machine state summary should be updated to ensure that the safety of our analysis is maintained. Due to the existence of out-of-order pipeline, the local worst-case will not generally result in the global worst case and the reuse test presented in Section 5.3.3 is no longer sound. The domination test should ensure all the components affecting the energy cost of the paths in a subtree remains the same at reuse time. As a result, the dominating condition here would be a record of the ready time and start time of the instructions at successor nodes. Ready time is the time that an instruction is ready to enter a pipeline stage and start time is the time that it actually enters that particular stage. Due to pipeline stalls, these two times might not be the same. For example, $Start_{IF1} \leq Ready_{IF1}$ indicates start time of instruction 1 at stage IF in the pipeline is always less than its ready time. In other words the difference between the ready time and the start time indicates a pipeline stall in that stage. Since the pipeline model is non-preemptive, by storing these times and checking it at the successor nodes at reuse point, we can make sure the components affecting the energy cost would be the same at reuse point. The drawback of this reuse test is that the number of conditions, in this case can grow exponentially.

In order to address this issue, we adopt the concept of dependency graph from [LRM04]. As a result, besides the dominating condition from Section 5.3.3 a subset of the dependency graph is stored and tested at reuse point. We will illustrate the dominating condition test based on the dependency graph, in the example presented in Section 5.6.4.

Dependency Graph

Dependency graphs can model the dependencies and also contentions that may occur in pipelines [B+06]. The dependency graph models the following dependency and contentions [LRM06]:

1. Dependencies among pipeline stages of the same instruction.
2. Dependencies due to finite-sized buffers and queues
3. Dependencies due to out-of-order execution stages
4. Data dependencies among instructions
5. Contention relations modeling structural hazards in the pipeline

We should note that the dominating condition though sound, is conservative in analysis of large programs and our analysis of out-of-order pipeline might not scale on large benchmarks.

5.6.3 Analysis of Loops

In order for the analysis to remain sound, our analysis avoids merging at any point inside the symbolic execution tree. As a result, there will be no need to store machine state summaries. Note that by avoiding merge at any point inside the loop our analysis will remain sound and will be quite precise. However, the size of the symbolic execution tree can grow exponentially in terms of the loop size. This can affect the scalability of our method.

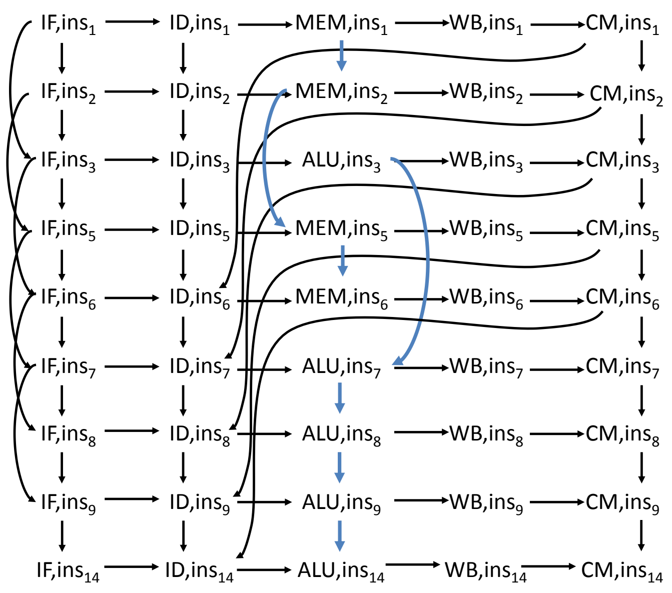
5.6.4 Example Analysis

Consider the CFG and the symbolic execution tree of the example presented in Section 5.4. This time we analyze the example program in Figure 5.8(a) on an out-of-order pipeline. The pipeline is a scalar 5 stage pipeline, initially empty, where all stages take only 1 clock cycle to complete. The IF, ID and CM stages are in-order and the stages EX and WB are out of order. The functional units in the EX stage of the pipeline are MEM and ALU. The time taken in all the pipeline stages is equally 1 clock cycle (CC). For brevity, in this example we only explain the extra dominating condition for out-of-order pipeline.

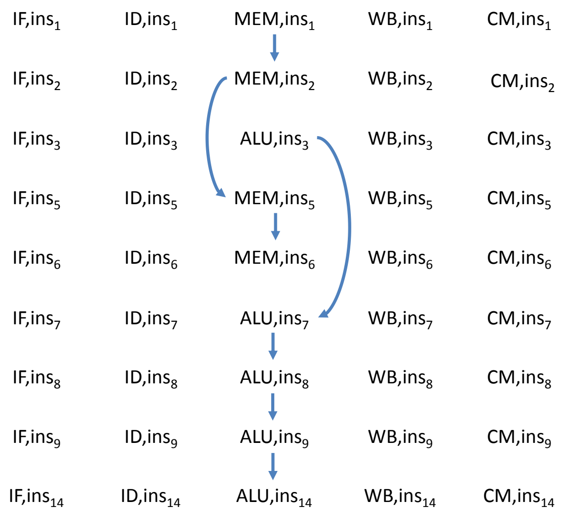
Similar to before, the analysis starts with empty pipeline state at $\langle 1a \rangle$. The pipeline state at $\langle 2a \rangle$ and $\langle 4a \rangle$ is $[\langle ins_1, ID \rangle, \langle ins_2, IF \rangle]$ (CC 2) and $[\langle ins_1, MEM \rangle, \langle ins_2, ID \rangle, \langle ins_3, IF \rangle]$ (CC 3). The analysis continues updating the pipeline state clock-wise till all the instructions leave the pipeline state. The energy cost of this path would be 197 mJoules. The pipeline state at each clock cycle can be seen in the table below:

PP	**Pipeline State**	**Clk**
$\langle 4a \rangle$	$[\langle ins_1, MEM \rangle, \langle ins_2, ID \rangle, \langle ins_3, IF \rangle]$	3
	$[\langle ins_1, WB \rangle, \langle ins_2, MEM \rangle, \langle ins_3, ID \rangle, \langle ins_5, IF \rangle]$	4
$\langle 5a \rangle$	$[\langle ins_1, CM \rangle, \langle ins_2, WB \rangle, \langle ins_3, ALU \rangle, \langle ins_5, ID \rangle, \langle ins_6, IF \rangle]$	5
	$[\langle ins_2, CM \rangle, \langle ins_3, WB \rangle, \langle ins_5, MEM \rangle, \langle ins_6, ID \rangle, \langle ins_7, IF \rangle]$	6
	$[\langle ins_3, CM \rangle, \langle ins_5, WB \rangle, \langle ins_6, MEM \rangle, \langle ins_7, ID \rangle, \langle ins_8, IF \rangle]$	7
$\langle 7a \rangle$	$[\langle ins_5, CM \rangle, \langle ins_6, WB \rangle, \langle ins_7, ALU \rangle, \langle ins_8, ID \rangle, \langle ins_9, IF \rangle]$	8
	$[\langle ins_6, CM \rangle, \langle ins_7, WB \rangle, \langle ins_8, ALU \rangle, \langle ins_9, ID \rangle, \langle ins_{14}, IF \rangle]$	9
	$[\langle ins_7, CM \rangle, \langle ins_8, WB \rangle, \langle ins_9, ALU \rangle, \langle ins_{14}, ID \rangle, \langle ins_{15}, IF \rangle]$	10
	$[\langle ins_8, CM \rangle, \langle ins_9, WB \rangle, \langle ins_{14}, ALU \rangle, \langle ins_{15}, ID \rangle]$	11
	$[\langle ins_9, CM \rangle, \langle ins_{14}, WB \rangle, \langle ins_{15}, MEM \rangle]$	12
	$[\langle ins_{14}, CM \rangle, \langle ins_{15}, WB \rangle]$	13
	$[\langle ins_{15}, CM \rangle]$	14

After traversing the first path, the dependence graph at $\langle 5a \rangle$ is as follow:


IF,ins1 ID,ins1 MEM,ins1 WB,ins1 CM,ins1
IF,ins2 ID,ins2 MEM,ins2 WB,ins2 CM,ins2
IF,ins3 ID,ins3 ALU,ins3 WB,ins3 CM,ins3
IF,ins5 ID,ins5 MEM,ins5 WB,ins5 CM,ins5
IF,ins6 ID,ins6 MEM,ins6 WB,ins6 CM,ins6
IF,ins7 ID,ins7 ALU,ins7 WB,ins7 CM,ins7
IF,ins8 ID,ins8 ALU,ins8 WB,ins8 CM,ins8
IF,ins9 ID,ins9 ALU,ins9 WB,ins9 CM,ins9
IF,ins14 ID,ins14 ALU,ins14 WB,ins14 CM,ins14

Note that the prefix instructions $ins_1, ins_2, ins_3, ins_5$ and ins_6 are added to the dependence graph since these instructions exist in the pipeline state at $\langle 5a \rangle$. Moreover, the instructions ins_7, ins_8, ins_9 and ins_{14} are added to the dependence graph, since these instructions exist in pipeline state when the last prefix instruction ins_6 is committed (at CC9). All the black edges in this dependence graph would always hold in future visits to $\langle 5 \rangle$. However, the blue edges might not hold. So, we generate a reduced dependence graph from blue edges:


IF,ins1 ID,ins1 MEM,ins1 WB,ins1 CM,ins1
IF,ins2 ID,ins2 MEM,ins2 WB,ins2 CM,ins2
IF,ins3 ID,ins3 ALU,ins3 WB,ins3 CM,ins3
IF,ins5 ID,ins5 MEM,ins5 WB,ins5 CM,ins5
IF,ins6 ID,ins6 MEM,ins6 WB,ins6 CM,ins6
IF,ins7 ID,ins7 ALU,ins7 WB,ins7 CM,ins7
IF,ins8 ID,ins8 ALU,ins8 WB,ins8 CM,ins8
IF,ins9 ID,ins9 ALU,ins9 WB,ins9 CM,ins9
IF,ins14 ID,ins14 ALU,ins14 WB,ins14 CM,ins14

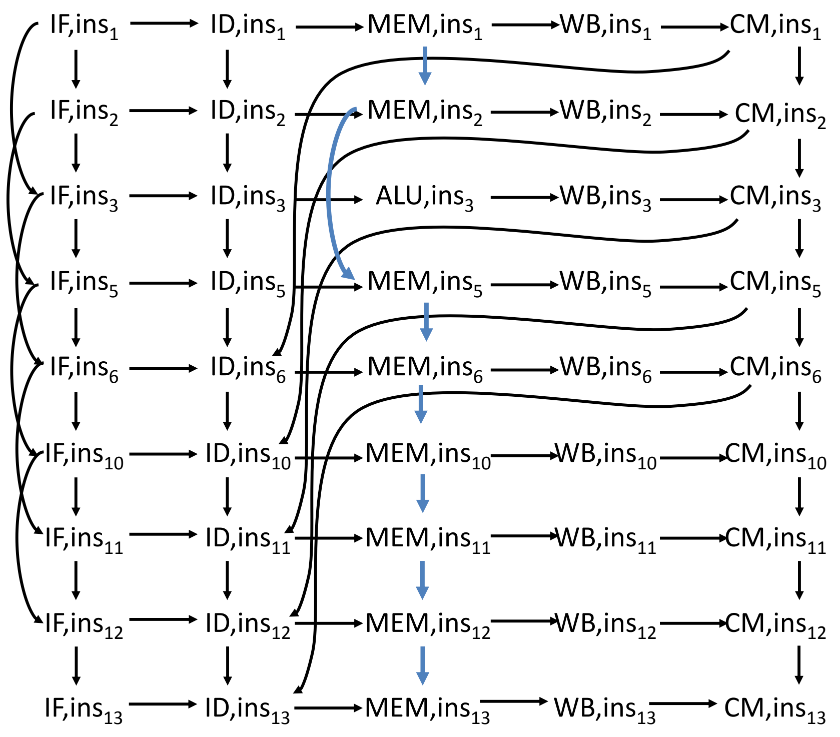
Next, we will generalize it and store it as the dominating condition:

$$\begin{tikzcd}
\text{MEM,Pre} \arrow[r] & \text{MEM,ins}_5 \arrow[r] & \text{MEM,ins}_6 \\
\text{ALU,Pre} \arrow[r] & \text{ALU,ins}_7 \arrow[r] & \text{ALU,ins}_8 \arrow[r] & \text{ALU,ins}_9 \arrow[r] & \text{ALU,ins}_{14}
\end{tikzcd}$$

Moving to the right sub-path, the execution time of the second path would be 15 CCs and the pipeline state at the program points can be seen in the table below:

PP	Pipeline State	Clk
$\langle 4a\rangle$	$[\langle ins_1,MEM\rangle, \langle ins_2,ID\rangle, \langle ins_3,IF\rangle]$	3
$\langle 6a\rangle$	$[\langle ins_1,CM\rangle, \langle ins_2,WB\rangle, \langle ins_3,ALU\rangle,$	5
$\langle 7b\rangle$	$[\langle ins_6,CM\rangle, \langle ins_{10},WB\rangle, \langle ins_{11},MEM\rangle, \langle ins_{12},ID\rangle, \langle ins_{13},IF\rangle]$	8

After traversing the second path, the dependence graph at $\langle 6a\rangle$ is as follow.



IF,ins1 ID,ins1 MEM,ins1 WB,ins1 CM,ins1
IF,ins2 ID,ins2 MEM,ins2 WB,ins2 CM,ins2
IF,ins3 ID,ins3 ALU,ins3 WB,ins3 CM,ins3
IF,ins5 ID,ins5 MEM,ins5 WB,ins5 CM,ins5
IF,ins6 ID,ins6 MEM,ins6 WB,ins6 CM,ins6
IF,ins10 ID,ins10 MEM,ins10 WB,ins10 CM,ins10
IF,ins11 ID,ins11 MEM,ins11 WB,ins11 CM,ins11
IF,ins12 ID,ins12 MEM,ins12 WB,ins12 CM,ins12
IF,ins13 ID,ins13 MEM,ins13 WB,ins13 CM,ins13

The reduced dependence graph is as follow:

$$\begin{tikzcd}
\text{IF,ins}_1 & \text{ID,ins}_1 & \text{MEM,ins}_1 \arrow[d] & \text{WB,ins}_1 & \text{CM,ins}_1 \\
\text{IF,ins}_2 & \text{ID,ins}_2 & \text{MEM,ins}_2 \arrow[dd, bend right] & \text{WB,ins}_2 & \text{CM,ins}_2 \\
\text{IF,ins}_3 & \text{ID,ins}_3 & \text{ALU,ins}_3 & \text{WB,ins}_3 & \text{CM,ins}_3 \\
\text{IF,ins}_5 & \text{ID,ins}_5 & \text{MEM,ins}_5 \arrow[d] & \text{WB,ins}_5 & \text{CM,ins}_5 \\
\text{IF,ins}_6 & \text{ID,ins}_6 & \text{MEM,ins}_6 \arrow[d] & \text{WB,ins}_6 & \text{CM,ins}_6 \\
\text{IF,ins}_{10} & \text{ID,ins}_{10} & \text{MEM,ins}_{10} \arrow[d] & \text{WB,ins}_{10} & \text{CM,ins}_{10} \\
\text{IF,ins}_{11} & \text{ID,ins}_{11} & \text{MEM,ins}_{11} \arrow[d] & \text{WB,ins}_{11} & \text{CM,ins}_{11} \\
\text{IF,ins}_{12} & \text{ID,ins}_{12} & \text{MEM,ins}_{12} \arrow[d] & \text{WB,ins}_{12} & \text{CM,ins}_{12} \\
\text{IF,ins}_{13} & \text{ID,ins}_{13} & \text{MEM,ins}_{13} & \text{WB,ins}_{13} & \text{CM,ins}_{13}
\end{tikzcd}$$

Moreover, The generalized dependence graph would be as follow:

$$\begin{tikzcd}
\text{MEM,Pre} \arrow[r] & \text{MEM,ins}_{10} \arrow[r] & \text{MEM,ins}_{11} \arrow[r] & \text{MEM,ins}_{12} \arrow[r] & \text{MEM,ins}_{13}
\end{tikzcd}$$

Continuing to $\langle 4a \rangle$ the generalized dependence graphs from $\langle 5a \rangle$ and $\langle 6a \rangle$ will be merged with generalized dependence graphs at $\langle 4a \rangle$ and the result would be:

$$\begin{tikzcd}
\text{MEM,Pre} \arrow[r] & \text{MEM,ins}_5 \arrow[r] & \text{MEM,ins}_6 \\
\text{ALU,Pre} \arrow[r] & \text{ALU,ins}_7 \arrow[r] & \text{ALU,ins}_8 \arrow[r] & \text{ALU,ins}_9 \arrow[r] & \text{ALU,ins}_{14}
\end{tikzcd}$$

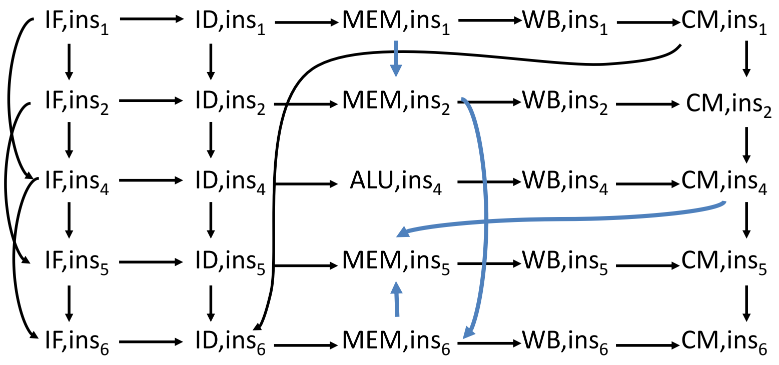
$$\begin{tikzcd}
\text{MEM,Pre} \arrow[r] & \text{MEM,ins}_5 \arrow[r] & \text{MEM,ins}_6 \arrow[r] & \text{MEM,ins}_{10} \arrow[r] & \text{MEM,ins}_{11} \arrow[r] & \text{MEM,ins}_{12} \arrow[r] & \text{MEM,ins}_{13}
\end{tikzcd}$$

Furthermore, the common part can be removed since it affects both paths similarly:

$$\begin{tikzcd}
\text{ALU,Pre} \arrow[r] & \text{ALU,ins}_7 \arrow[r] & \text{ALU,ins}_8 \arrow[r] & \text{ALU,ins}_9 \arrow[r] & \text{ALU,ins}_{14}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{MEM,Pre} \arrow[r] & \text{MEM,ins}_{10} \arrow[r] & \text{MEM,ins}_{11} \arrow[r] & \text{MEM,ins}_{12} \arrow[r] & \text{MEM,ins}_{13}
\end{tikzcd}$$

Moving to the right subtree at $\langle 4b\rangle$, this dominating condition is tested on the pipeline state[$\langle ins_1$,*MEM*$\rangle$, $\langle ins_2$,*ID*$\rangle$, $\langle ins_4$,*IF*$\rangle$] (CC 3). Assume that there is some dependency between the nodes CM, ins_4 MEM, ins_5. As a result, this time ins_5 is delayed and enters MEM after ins_6:



The pipeline state is updated cycle-wise till the instructions in the generalized dependence graphs at $\langle 4b\rangle$ reach EX state and it is tested if the edges in the generalized dependence graphs still hold, which is true. So the summarization for $\langle 4\rangle$ is reused and the pipeline state on the right sub-path ($\langle 4b\rangle$, $\langle 6c\rangle$, $\langle 7c\rangle$) is replayed and the total energy cost for the path is computed which is 221 mJoules:

PP	Pipeline State	Clk
$\langle 4b\rangle$	[$\langle ins_1$,*MEM*$\rangle$, $\langle ins_2$,*ID*$\rangle$, $\langle ins_4$,*IF*$\rangle$]	3
$\langle 6c\rangle$	[$\langle ins_1$,*CM*$\rangle$, $\langle ins_2$,*WB*$\rangle$, $\langle ins_4$,*ALU*$\rangle$, $\langle ins_5$,*ID*$\rangle$, $\langle ins_6$,*IF*$\rangle$]	5
$\langle 7c\rangle$	[$\langle ins_5$,*WB*$\rangle$, $\langle ins_6$,*ST*$\rangle$, $\langle ins_{10}$,*WB*$\rangle$, $\langle ins_{11}$,*MEM*$\rangle$, $\langle ins_{12}$,*IF*$\rangle$, $\langle ins_{13}$,*IF*$\rangle$,]	8

Continuing the analysis to the root the WCEC of the whole tree is returned as 247 mJoules.

5.7 Summary

In this chapter we presented the worst-case energy consumption analysis. We highlighted that in the energy domain, which is data dependent to a high scale, modular approaches can generate more imprecise bounds. We proposed that the best application for integrated analysis is when the resource of interest is data-dependent (such as energy). Our framework is able to precisely track the machine state across the paths in the symbolic execution tree. This will give us the ideal precision to measure the worst-case energy cost of a program. However, the scalability of our method relies on the possibility of reuse.

Comparing to the state-of-the-art, our method beside inheriting the precision from infeasible path detection and more precise cache analysis (from Chapter 3), can generate more precise energy cost in the pipeline analysis (wake-up logic, selection and clock energies, explained in Appendix B). Our intention in this chapter has been on ensuring that our framework can remain scalable while performing superscalar pipeline analysis and WCEC analysis. We were able to extend the fundamental notion of reuse in the presence of in-order execution while remaining sound and scalable and extend reuse in the presence of out-of-order execution while remaining sound.

Our pipeline analysis can be applied to other architectures as long as a sound transition function for the pipeline can be defined to represent the changes in the internal state of the pipeline. We consider timing anomaly in the out-of-order pipeline analysis. However, we do not consider timing anomaly in in-order pipeline analysis and also we do not consider multi-threaded pipelines.

Chapter 6

Conclusion and Future Work

In this thesis, we have presented a framework which performs resource analysis with the help of a fully path-sensitive integrated symbolic execution framework. Our framework, presented in Chapter 2, performs resource analysis in one integrated phase where the low-level analysis and high-level analysis are performed at the same time. The main contribution of our method is that our algorithm precisely tracks the underlying micro-architectural features represented by the machine state along the symbolic execution. This, then, allows our framework to precisely estimate the total consumed resource along different paths.

Our framework addresses the scalability issue with the notion of reuse. Reuse has been used for scalability of path-based methods [CJ11, CJ13]. However, in this thesis, first, we extended the notion of reuse in the presence of *dynamic resource consumption model*, where a realistic assumption is that the contribution of each basic block to the resource consumption of a program is not a constant value. Secondly, we extended the notion of reuse in the presence of *symbolic resource consumption model*, where the contribution of each basic block to the resource consumption of a program is a symbolic value.

In Chapter 3, we customized our resource analysis framework for WCET analysis in the presence of instruction and data caches. The WCET analysis with data cache analysis adds a new dimension (memory range being accessed by data cache) to the problem. The experiments presented shows that our analysis can scale on realistic benchmarks such as nsichneu.

Our framework performs instruction and data cache with LRU cache policy. In order to extend the cache analysis to support other cache policies, a sound cache summarization should be defined. Our inspection shows that if the memory blocks in a cache set can be ordered (by a ranking), a sound cache summarization can be defined for them. As a result, our framework can be extended to some of the cache policies such as FIFO. Finally, we like to note that our framework can support cache hierarchies while it remains scalable. The result of our research in Chapter 3 is published in [CJMa].

Next, we extended our resource analysis framework to estimate the MHW of a program in Chapter 4. Note that although none of the micro-architectural features have an effect on the memory consumption in MHW analysis, the resource of interest, memory, is a non-cumulative resource. In other words, since the memory can be acquired and released the MHW of a program is no longer at the end of its paths and rather, it can be at any node along a path. We extended our framework to be able to perform symbolic resource analysis on non-cumulative resources in this chapter.

As elaborated in Chapter 4, compared to the state-of-the-art methods, our method can perform both stack and heap analysis and also generate bounds which are less complex (parametric to input arguments only). The result of our research in Chapter 4 is published in [CJMb].

Finally, in Chapter 5, we extended our resource analysis framework to perform WCEC analysis in the presence of in-order and out-of-order superscalar processors. The machine state would track the pipeline and cache states precisely along the paths during the analysis and we were able to extend the fundamental notion of reuse in the presence of in-order execution while remaining sound and scalable and extend reuse in the presence of out-of-order execution while remaining sound.

Comparing to the state-of-the-art, our method beside inheriting the precision from infeasible path detection and more precise cache analysis (from Chapter 3), can generate more precise energy cost in the pipeline analysis (wake-up logic, selection and clock energies). Our intention in this chapter has been on ensuring that our framework can remain scalable while performing superscalar pipeline analysis and WCEC analysis.

Our pipeline analysis can be applied to other architectures as long as a sound transition function for the pipeline can be defined to represent the changes in the internal state of the pipeline. We consider timing anomaly in the out-of-order pipeline analysis. However, we do not consider timing anomaly in in-order pipeline analysis and also we do not consider multi-threaded pipelines.

We will end this chapter by proposing some future directions to the research in this thesis for interested readers.

Extension to Binary Analysis

As discussed in Chapter 2, our resource analysis framework performs resource analysis on LLVM IR. However, analyzing binary code can be performed for the analysis of highly safety-critical systems. The analysis of binary code is cumbersome and costly. Moreover, certain issues such as unbounded jumps which the jump location can only be identified at run-time, makes performing symbolic execution on binary code quite difficult.

The research presented in [H$^+$13] and implemented in the respective tool T-Crest can bridge this gap. In this method, by the help of a control-flow relation graph, each LLVM basic block is related to one or more flow paths in the binary code and the timing of each LLVM basic block can then be measured precisely. By embedding this method with our analysis framework, we would be able to perform resource analysis on machine code. Moreover, note that since our framework is path-sensitive we might be able to track more precisely the context reaching an LLVM basic block and as a result prune certain flow paths in the respective control-flow relation graph.

Pipeline Analysis with Branch Prediction

In the WCEC analysis presented in Chapter 5, we assumed perfect branch prediction where all the predictions of the branch instruction are correct. One important extension of the research presented in Chapter 5 is to extend the resource analysis framework with computing the effect from the branch prediction. Note that the branch prediction affects both in-order and out-of-order execution. The soundness of our framework relies on the soundness of the reuse step, and the integration of a realistic branch prediction should be performed in a way which it would not affect the soundness of the reuse step. In this way, we would be able to estimate the WCEC of a program with more precision.

Summarizing Functions

Our proposed framework can be extended by adding the means to generate summarization of functions and reusing it in later on calls to the functions. It should be noted that a function might be called several times during the execution of a program and generating summarizations for the functions will extend the scalability of our framework. The main idea will be to analyze functions as loops with one iteration. By the help of this method, we can use the concepts proposed for loops (witness, dominating condition and machine state summary) in the summarization of functions.

Bibliography

[A$^+$07] Elvira Albert et al. Cost analysis of java bytecode. In *Programming Languages and Systems*, pages 157–172. Springer, 2007.

[A$^+$12a] Elvira Albert et al. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[A$^+$12b] Saswat Anand et al. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[A$^+$12c] Björn Andersson et al. Non-preemptive scheduling with history-dependent execution time. In *ECRTS*, pages 363–372, 2012.

[A$^+$13] Jeppe L Andersen et al. Worst-case memory consumption analysis for scj. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 2–10. ACM, 2013.

[A$^+$15] David Atienza Alonso et al. Dynamic memory management optimization for multimedia applications. In *Dynamic Memory Management for Embedded Systems*, pages 167–192. Springer, 2015.

[aiT] aiT Worst-Case Execution Time Analyzers. URL `http://www.absint.com-/ait/index.htm`.

[Als14] Mohammad Alshamlan. A regression approach to execution time estimation for programs running on multicore systems. 2014.

[B$^+$03] Derek Bruening et al. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003.

[B$^+$05] Enrico Bini et al. Speed modulation in energy-aware real-time systems. In *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 3–10. IEEE, 2005.

[B^+06] Jonathan Barre et al. Modeling instruction-level parallelism for wcet evaluation. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '06, pages 61–67, Washington, DC, USA, 2006. IEEE Computer Society.

[B^+08] Víctor Braberman et al. Parametric prediction of heap memory requirements. In *Proceedings of the 7th international symposium on Memory management*, pages 141–150. ACM, 2008.

[B^+11] Nathan Binkert et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[B^+13] Thomas Bøgholm et al. Towards harnessing theories through tool support for hard real-time java programming. *Innovations in Systems and Software Engineering*, 9(1):17–28, 2013.

[B^+14] Abhijeet Banerjee et al. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 588–598. ACM, 2014.

[BCR13] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. Precise micro-architectural modeling for wcet analysis via ai+sat. In *19th RTAS*. IEEE, 2013.

[BMBW00] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[Bou] Bound-T time and stack analyser. URL `http://www.bound-t.com`.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. *Wattch: a framework for architectural-level power analysis and optimizations*, volume 28. ACM, 2000.

[$\check{C}^+$] Pavol Černý et al. Segment abstraction for worst-case execution time analysis. In *ESOP 2015.*

[C^+76] Lori Clarke et al. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, (3):215–222, 1976.

[C^+14] Quentin Carbonneaux et al. End-to-end verification of stack-space bounds for c programs. In *ACM SIGPLAN Notices*, volume 49. ACM, 2014.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis. In *POPL*, pages 238–252. ACM, 1977.

[CH] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL, Pages 84–96, 1978.*

[chr14] Chronos WCETanalysis tool. www.comp.nus.edu.sg/r̃pembed/chronos, January 2014.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *PLDI*, pages 467–478. ACM, 2015.

[CJ11] Duc-Hiep Chu and Joxan Jaffar. Symbolic simulation on complicated loops for wcet path analysis. In *Embedded Software (EMSOFT)*, pages 319–328. IEEE, 2011.

[CJ13] Duc-Hiep Chu and Joxan Jaffar. Path-sensitive resource analysis compliant with assertions. In *Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2013.

[CJMa] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. Precise cache timing analysis via symbolic execution. In *RTAS 2016*.

[CJMb] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. Symbolic execution for memory consumption analysis. In *LCTES 2016*.

[Cla14] clang: a c language family front-end for llvm. http://www.clang.llvm.org, 2014. Accessed: 2015-02-01.

[CP00] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, 2000.

[CR11] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *RTSS*, pages 193–203. IEEE, 2011.

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *European Conference on Parallel Processing*, pages 1298–1307. Springer, 1997.

[Flo14] Florida WCETanalysis tool. http://moss.csc.ncsu.edu/mueller/, January 2014.

[FMH14] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In *Programming Languages and Systems*, pages 275–295. Springer, 2014.

[FV98] Klaus Friedewald and AG Volkswagen. Design methods for adjusting the side airbag sensor and the car body. In *Proceedings of 16th International Technical Conference on the Enhanced Safety of Vehicles, Windsor, Ontario, Canada, 31 May-4 June 1998. VOLUME 3*, 1998.

[FW98] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES*, pages 16–30. Springer, 1998.

[G$^+$01] Matthew R Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[G$^+$05] Jan Gustafsson et al. Towards a flow analysis for embedded system c programs. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 287–297. IEEE, 2005.

[GEL06] Jan Gustaffson, Andreas Ermedahl, and Björn Lisper. Algorithms for infeasible path calculation. In *OASIcs-OpenAccess Series in Informatics*, volume 4, 2006.

[GGP$^+$14] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, and Kerstin Eder. Static energy consumption analysis of llvm ir programs. *arXiv preprint arXiv:1405.4565*, 2014.

[GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. *SIGPLAN Not.*, 45(6), June 2010.

[H$^+$03] Reinhold Heckmann et al. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[H$^+$11] Bach Khoa Huynh et al. Scope-aware data cache analysis for wcet estimation. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212. IEEE, 2011.

[H$^+$13] Benedikt Huber et al. Combined wcet analysis of bitcode and machine code using control-flow relation graphs. In *ACM SIGPLAN Notices*, volume 48, pages 163–172. ACM, 2013.

[H$^+$14a] Julien Henry et al. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *RTNS*, pages 43–52. ACM, 2014.

[H$^+$14b] Timo Hönig et al. Proactive energy-aware programming with peek. In *Proceedings of the 2014 International Conference on Timely Results in Operating Systems*, pages 6–6. USENIX Association, 2014.

[H$^+$16] Rémy Haemmerlé et al. A transformational approach to parametric accumulated-cost static profiling. In *International Symposium on Functional and Logic Programming*, pages 163–180. Springer, 2016.

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *POPL'11*, pages 357–370, 2011.

[HIE12] CHU DUC HIEP. *Interpolation Methods for Symbolic Execution.* PhD thesis, NATIONAL UNIVERSITY OF SINGAPORE, 2012.

[J$^+$05] Hans Jacobson et al. Stretching the limits of clock-gating efficiency in server-class processors. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 238–242. IEEE, 2005.

[J$^+$06] Ramkumar Jayaseelan et al. Estimating the worst-case energy consumption of embedded software. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 81–90. IEEE, 2006.

[JMNS12] Joxan Jaffar, Vijayaraghavan Murali, Jorge A Navas, and Andrew E Santosa. Tracer: A symbolic execution tool for verification. In *Computer Aided Verification*, pages 758–766. Springer, 2012.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J Stuckey, and Roland HC Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS 14(3)*, 14(3):339–395, 1992.

[JSV08] Joxan Jaffar, Andrew E Santosa, and Razvan Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*, 2008.

[JSV09] Joxan Jaffar, Andrew E Santosa, and Răzvan Voicu. An interpolation method for clp traversal. In *CP*, 2009.

[K$^+$02] Raimund Kirner et al. Fully automatic worst-case execution time analysis for matlab/simulink models. In *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 31–40. IEEE, 2002.

[KE13] Steve Kerrison and Kerstin Eder. Energy modelling and optimisation of software for a hardware multi-threaded embedded microprocessor. *University of Bristol, Bristol, Tech. Rep*, 2013.

[KE15] Steve Kerrison and Kerstin Eder. Energy modeling of software for a hardware multithreaded embedded microprocessor. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(3):56, 2015.

[KF14] Daniel Kästner and Christian Ferdinand. Proving the absence of stack overflows. In *Computer Safety, Reliability, and Security*. Springer, 2014.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KKZ13] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Wcet squeezing: on-demand feasibility refinement for proven precise wcet-bounds. In *RTNS*, pages 161–170. ACM, 2013.

[Kop11] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer, 2011.

[Kru14] Michael Kruse. Perfrewrite–program complexity analysis via source code instrumentation. *arXiv preprint arXiv:1409.2089*, 2014.

[L$^+$] Hanbing Li et al. Tracing flow information for tighter wcet estimation: Application to vectorization. In *RTCSA 2015*.

[L$^+$05] Chi-Keung Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[L$^+$10] Mingsong Lv et al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 339–349. IEEE, 2010.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[LLV15] Llvm test suite guide. URL http://llvm.org/releases/2.2/docs/Testing Guide.html, 2015.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Not.*, 30(11):88–98, 1995.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *TODAES 4(3)*, 4(3), 1999.

[LRM04] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for software timing analysis. In *RTSS*, pages 92–103. IEEE, 2004.

[LRM06] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 34(3):195–227, 2006.

[LS98] Thomas Lundqvist and Per Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *(LCTES),1998*, pages 1–15. Springer, 1998.

[LS99] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *RTS*, 1999.

[M06] Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2006.

[M$^+$08] Miguel Masmano et al. A constant-time dynamic storage allocator for real-time systems. *Real-Time Systems*, 40(2):149–179, 2008.

[M$^+$13] Aravind Machiry et al. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[MHH02] Frank Mehnert, Michael Hohmuth, and Hermann Hartig. Cost and benefit of separate address spaces in real-time operating systems. In *23rd RTSS*, pages 124–133. IEEE, 2002.

[MRC03] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. Dynamic storage allocation for real-time embedded systems. *Proc. of Real-Time System Simposium WIP*, 2003.

[N$^+$06] Fadia Nemer et al. Papabench: a free real-time benchmark. In *WCET*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[NS] Kartik Nagar and YN Srikant. Path sensitive cache analysis using cache miss paths. In *VMCAI 2015*.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[P$^+$11] OFJ Perks et al. Wmtrace: a lightweight memory allocation tracker and analysis framework. 2011.

[PB00] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *J. RTS*, 2000.

[PHS10] Wolfgang Puffitsch, Benedikt Huber, and Martin Schoeberl. Worst-case analysis of heap allocations. In *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 464–478. Springer, 2010.

[PKME15] James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Data dependent energy modelling: A worst case perspective. *arXiv preprint arXiv:1505.03374*, 2015.

[PS01] Padmanabhan Pillai and Kang G Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM SIGOPS Operating Systems Review*, number 5, pages 89–102. ACM, 2001.

[PSSG10] Preeti Ranjan Panda, BVN Silpa, Aviral Shrivastava, and Krishnaiah Gummidipudi. *Power-efficient system design*. Springer Science & Business Media, 2010.

[Q$^+$00] Gang Qu et al. Function-level power estimation methodology for microprocessors. In *Proceedings of the 37th Annual Design Automation Conference*, pages 810–813. ACM, 2000.

[R$^+$06] Jan Reineke et al. A definition and classification of timing anomalies. *WCET*, 4, 2006.

[RRW05] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):751–778, 2005.

[RS09a] Jan Reineke and Rathijit Sen. Sound and efficient wcet analysis in the presence of timing anomalies. In *WCET*, page 101, 2009.

[RS09b] Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. In *Transactions on High-Performance Embedded Architectures and Compilers II*, pages 222–241. Springer, 2009.

[S^+07] Jean Souyris et al. Computing the worst case execution time of an avionics program by abstract interpretation. In *OASIcs-OpenAccess Series in Informatics*, volume 1. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[S^+12] Carsten Sinz et al. Llbmc: A bounded model checker for llvm's intermediate representation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 542–544. Springer, 2012.

[SC01] Amit Sinha and Anantha P Chandrakasan. Jouletrack: a web based tool for software energy profiling. In *Proceedings of the 38th annual Design Automation Conference*, pages 220–225. ACM, 2001.

[Sch94] Werner Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, 1994.

[Sch15] Martin Schoeberl. Scala for real-time systems? In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 13. ACM, 2015.

[SF99] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *ACM SIGPLAN Notices*, volume 34, pages 35–44. ACM, 1999.

[T^+94] Vivek Tiwari et al. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[T^+13] Philip W Trinder et al. Resource analyses for parallel and distributed coordination. *Concurrency and Computation: Practice and Experience*, 25(3):309–348, 2013.

[TFW00] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by seperate cache and path analyses. *RTS 18(2/3)*, 18(2/3):157–179, May 2000.

[The04] Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models*. PhD thesis, Universitätsbibliothek, 2004.

[V^+14] Marcus Volp et al. Has energy surpassed timeliness? scheduling energy-constrained mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 275–284. IEEE, 2014.

[Val] Valgrind instrumentation framework. http://valgrind.org/.

[W^+97] Randall T White et al. Timing analysis for data caches and set-associative caches. In *RTAS*, pages 192–202. IEEE, 1997.

[W^+08] Reinhard Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[W^+15] Peter Wägemann et al. Worst-case energy consumption analysis for energy-constrained embedded systems. *ECRTS*, 4, 2015.

[Wil06] Reinhard Wilhelm. Determining bounds on execution times. In Richard Zurawski, editor, *Handbook on Embedded Systems*, chapter 14. CRC Press, 2006.

[wtc14] Wcet tool competition 2014. URL www.mrtc.mdh.se/projects/WTC/, 2014.

[Y^+00] Wu Ye et al. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings of the 37th Annual Design Automation Conference*, pages 340–345. ACM, 2000.

[ZGSV11] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis*, pages 280–297. Springer, 2011.

[ZK15] Michael Zolda and Raimund Kirner. Calculating wcet estimates from timed traces. *RTS*, pages 1–50, 2015.

Appendix A

Compiling LLVM IR to Transition System

LLVM is a modular compiler, which compiles high-level languages to an intermediate representation named LLVM IR. LLVM IR is in static single assignment (SSA) form representation.

LLVM IR instructions are arranged in basic blocks (labeled with unique names). The basic blocks are a sequence of instructions, $inst_1 \ldots inst_n$, such that all instructions up to $inst_{n-1}$ are not branch or return instructions. The last instruction $inst_n$ is always *br* or *ret*. Moreover, the *Phi* instruction always appears as the first instruction in the basic block.

Our symbolic execution framework receives a C program as an input and produces a respective transition system in CLP($\mathcal{R}$) [JMSY92]. The transition system is passed to the resource analysis framework. We use the open-source C/C++ front-end Clang 3.2 [Cla14] to compile a C program and generate the LLVM IR. Next, a LLVM pass, which we have developed, reads the LLVM IR and generates the transition system. Finally, our resource analysis framework analyses the transition system.

During the compilation, for each basic block BB, and its set of input and output variables, $Vars_{input}, Vars_{output}$, the updates on each of the variables in $Vars_{output}$ compared to $Vars_{input}$ is captured through a set of constraints and is stored in the generated transition system.

Note that besides capturing the relations between the input and output variables for a BB, the framework needs more low-level information for the resource analysis to be able to precisely generate and update the machine state. For example in WCET analysis, the sequence of memory block accesses should also be captured.

A.1 The LLVM Pass

The LLVM pass, that we have developed, transforms the LLVM IR to a respective transition system in CLP($\mathcal{R}$). The LLVM pass performs the transformation in nine phases:

Phase 1:

In the first phase, three groups of variables are collected:

1. Global variables
2. Local variables needed for infeasibility test
3. Local variables accessed by data cache are collected to create the data map

The global variables are collected and added to both the list of global variables list and also to the data map. Since LLVM introduces many intermediate variables, not all of the local variables are collected. Only the local variables which their value is important for the feasibility test or data range calculation are collected. All the variables which appear in the arguments of the *branch* and *store* instructions are tested and the ones which are needed for the infeasibility test or data range calculation are added to the list of local variables. The data range calculation is performed by *load* instructions. In the next step, all the dynamic range data accessed in *load* instructions are then tested and the variables which their value has an effect on the data access ranges are collected (similar to taint analysis, but from bottom to top). This process is repeated till it reaches a fixed point (where all the important variables and all the variables writing to them, are collected).

Phase 2:

In phase 2, the function names and their input arguments are collected.

Phase 3:

In phase 3, each basic block is mapped to a program point and the CFG structure is collected and stored too.

Phase 4:

This phase is the core of the compiler where the information of the transition system is collected from the LLVM IR and is stored in the related data structures. For brevity, we will only explain phase 4 and phase 5 for WCET analysis. In this phase, the transitions between the basic blocks are generated and the constraints needed for infeasibility test and data range calculation are collected in this phase.

For each transition, firstly, the list of global and local variables is generated. Note that the transitions keep the list of incoming variables and outgoing variables. The updated variables in the transition are kept in the outgoing list.

Secondly, a function collects all the data accesses in the basic blocks and stores them in the transitions. Data accesses are in *store* and *load* instructions. In the *store* instruction all the data accesses are assumed as cache misses (based on the point that a write-through cache configuration is used in the experiments).

The *load* instructions data accesses might be a cache hit or a cache miss. On the other hand, the access can be a constant access (accessing to a specific memory location) or a dynamic access (the accesses address is resolved at run-time). For the constant accesses the constant address is recorded and for the dynamic accesses a constraint is generated which calculates the accessed address or a range of data addresses at run-time.

Finally, the constraints updating the variables in the transitions are collected. We like to highlight the following instructions and explain more on the information collected from them:

- *Load* instruction: In addition to collecting the data range accesses information, in case the load instruction loads from a pointer location, then the constraint is generated and added to the transition.
- *Store* instruction: Beside collecting the information from data range accesses, in case the store instruction updates a variable in the collected variables list, recursively a constraint is generated where it captures the update made to the variable. For example, for the following piece of LLVM IR, the generated constraint would be $A4 = A3 - A1 + A2$.

  ```
  %Add = add %A1, %A2
  %Sub = sub %A3, %Add
  Store %Sub, %A4
  ```

- *GetElementPtr* instruction: Since, this instruction stores pointer accesses and array manipulation addresses, its information is used for calculation of the dynamic data cache accesses.
- *Br* instruction: This instruction is usually the final instruction in the basic blocks and represents conditional or unconditional jump. In the case of unconditional jump, only one transition would be generated. However, for conditional jumps two or three transitions are generated and for each transition the constraint which indicates the condition of the jump is added to the transition.

- *Cmp* instruction: In the *icmp* and *fcmp* instructions, a comparison type and its arguments is collected which is used later in the *Br* instruction.
- *Ret* instruction: The *ret* instruction is the last instruction in each function and for each *ret* instruction in a function the respective transition is updated with the information of the returned value of the function. For the main function these transitions are marked as the transitions to the leaf nodes.

Phase 5:

In phase 5, the instructions in a basic block and their addresses are added to the transitions. This information is used to capture the instruction cache accesses.

Phase 6:

In phase 6 the information regarding loops (such as loop-head, exit-edge, back-edge and nested level of the loops) are collected. The information is used to perform loop simplification. Loop simplification is performed in phase 7.

Phase 7:

In this phase, first a dummy node is added at the end of the loops and all the back-edges in a loop are first redirected to this dummy node and then an edge is added from this dummy node to the loop-head. As a result the loops have only one back-edge and the dummy nodes are selected as the abstraction point (merge point) in our analysis of loops.

Phase 8:

In phase 8 loops are virtually unrolled once (similar to the virtual step in AI framework) is performed. The virtual unrolling is performed to ensure the comparison to AI framework is a fair comparison.

Phase 9:

Finally, in the phase 9 the transitions and the generated CFG are printed out into a CLP($\mathcal{R}$) file.

Appendix B

Energy Consumption Model in Embedded Systems

In this Appendix, we will present the energy consumption model for the in-order and the out-of-order pipeline model presented in Chapter 5.

The energy consumed by an embedded system is governed by the physical properties of the hardware components in use, the power management capabilities of that hardware, and the activity driven by the software running on it [KE13]. Jayaseelan et. al. in [J+06] has argued that although power and energy are two terms used reciprocally, energy is the important metric regarding battery life.

The energy consumption of a task running on a processor is defined as $E = P \times t$, where P is the average power and t is the execution time. In other words, energy is the integral of power over a period of time, $E = \int_0^T P(t)dt$. Energy is measured in Joules whereas power is measured in Watts (Joules/Sec).

Power consumption consists of two main components: dynamic power and leakage power $P = P_{dynamic} + P_{leakage}$. Dynamic power is caused due to the switching activity in the underlying hardware and it is data dependent and related to the executed program.

Leakage power captures the power lost from the leakage current irrespective of switching activity.

Furthermore, in clock cycled processors part of the energy is consumed in the clock. This consumed energy is referred to as the clock energy. Moreover, we refer to the power consumed in the idle cycles as switch-off power.

B.1 Energy Consumption Model

The energy consumption of a path can be calculated through the following formula:

$$energy_{path} = dynamic_{path} + leakage_{path} + switchoff_{path} + clock_{path} \tag{B.1}$$

Where the total energy consumption during the execution time of the path, $energy_{path}$, is the summation of $dynamic_{path}$ (the energy consumed due to the switching activity), and $leakage_{path}$, $switchoff_{path}$ and $clock_{path}$, (the energy consumed due to the leakage power, switch-off power, and clock power).

On the other hand, the energy consumption of a program can be categorized into two separate time-dependent and time-independent components which can be estimated separately:

1. **Instruction-specific energy:** The energy consumed by an instruction, which can come from the energy consumed in the ALU units or the fetching of parameters which can resolve to a cache hit/cache miss. This energy is totally time-independent.

2. **Pipeline-specific energy:** The energy consumed in different micro-architectural features such as pipeline. This energy cannot be related to a certain instruction and is relevant to the execution time.

B.1.1 Instruction-specific Energy

The instruction-specific energy of a path is the dynamic power consumed due to the switching activity of the instructions in that path:

$$dynamic_{path} = \sum_{ins \in path} dynamic_{ins} \tag{B.2}$$

Where $dynamic_{ins}$ is the dynamic power consumed by an instruction ins.

The energy consumed by an instruction as it travels through the pipeline would be as follows:

- **Fetch and decode:** The energy consumed due to the fetch, decode and instruction cache access are consumed in this stage. We will follow the instruction cache model presented in Chapter 3 to measure the energy consumed in this phase in the presence of the instruction cache. P_{ic} depicts the power consumed to access the instruction cache per cycle.

- **Register access:** The energy consumed for the register file access due to reads and/or writes can alter for different classes of instructions. Throughout this thesis we assumed realistic clock gating [J+06]. As a result, the energy consumption in the register file for an instruction is proportional to the number of register operands. $P_{RegisterAccess}$ depicts the register-access power consumed to access one operand by an instruction in a clock cycle.

- **Branch prediction:** Currently we assume perfect branch prediction. Examining and extending our framework to model branch prediction can be a goal for future research. For now, this energy consumption is fixed as a constant value.

- **Wake-up logic:** The wake-up logic informs the dependent instructions that their arguments are ready while writing the result onto the result bus. The energy consumed in the wake-up logic is proportional to the number of output operands for an instruction. P_{WakeUp} is the power consumed in one clock cycle for making the dependent instructions ready for one output operand. Wake-up logic energy is only consumed in out-of-order pipelines.

- **Selection logic:** After wake-up logic notifies an instruction that all its arguments are ready, the instruction is chosen by the selection logic for execution. It selects an instruction from a pool of ready instructions. Note that, due to the existence of instructions with higher priority, an instruction might be accessed by the selection logic a few times till it gets executed. $P_{Selection}$ depicts the power consumed by the selection logic for each clock cycle that the selection logic chooses an instruction for execution from a pool of ready instructions. Unlike other methods which conservatively consider the selection logic operates at every clock cycle, in our approach, besides the end of the loop iterations, we can accurately identify the clock cycles that the selection logic is accessed. Similar to Wake-up logic, the energy for selection logic is only consumed in out-of-order pipelines.

- **Functional units:** The energy consumed in the execution stage for an instruction depends on the functional unit it uses and its latency. Since our framework in able to precisely track the context reaching each basic block, unlike other methods, for variable latency instructions, we usually can track precisely the consumed energy in the functional unit. Otherwise, similar to other methods such as [J$^+$06], we will assume the maximum energy consumption.

  The power consumed by each component C when it is active is depicted by P_C. The power consumed by these components in the active mode is measured by P_{ALU}, P_{MULTU} and P_{FPU}. The energy which the execution of each instruction utilizes is then measured from the delay of the instruction times the power consumed in the component. Moreover, the access to the data cache and hence the energy consumption for load/store units is measured here. P_{dc} depicts the power consumed to access the data cache per cycle.

B.1.2 Pipeline-specific Energy

Pipeline-specific energy is consisted of three components: switch-off energy, clock-energy and leakage energy. All three energy components are influenced by the execution time of the paths:

- **Switch-off Energy:** The switch-off energy refers to the power consumed in an idle unit when it is disabled through clock gating. The switch-off energy corresponding to a basic block can now be defined as:

$$switchoff_{path} = \sum_{C \in Components} switchoff(C) \tag{B.3}$$

Where $Components$ is the set of all hardware components. Since we track the pipeline state and the accesses to different components such as caches along each path, we will measure the switch-off energy of each component for each clock cycle. $Switchoff_C$ is the power consumed per clock cycle by each component C in the idle state. Following the realistic clock gating from [J$^+$06] we consider the switch-off power of a component $Switchoff_C$, being 10% of its peak power. As a result, the switch-off power for each component is as follow:

$$switchoff(C) = (ExecutionTime_{path} - access_{path}(C)) * P_C * 10\% \tag{B.4}$$

Where $ExecutionTime_{path}$ is the execution time of the basic blocks along a path with regard to the context which reaches the basic blocks, $access_{path}(C)$ is the total number of accesses to a component C by the instructions in a path and P_C is the active power consumed by the component C. The switch-off power of the components are represented by $Switchoff_{ALU}$, $Switchoff_{MULTU}$ and $Switchoff_{FPU}$.

- **Clock Network Energy:** In modern high-frequency microprocessors, nearly 70% of the active (switching) power is consumed by the clock circuit [J+05]. Clock gating helps in reducing the dynamic power by pruning part of the components being idle in the clock tree [PSSG10]. The power consumed by the clock network while having clock gating is depicted by P_{Clock}. P_{Clock} is proportionate to the number of the active components in each clock cycle. It can at most reach the peak power used by the clock power $\textit{clock-power}_{cycle}$.

  Since, our framework measures the energy consumed in a path based on a concrete pipeline state, it is able to determine which components are active and which components are idle in each clock cycle and as a result measure the P_{Clock} with a more precision. P_{Clock} at each clock-cycle is then used to measure the energy consumed in the clock network ($clock_{Path}$). However, this measurements needs the list of the components which are clock gated in a processor and also the amount of clock power which is utilized. In general, these information might not be available or the producer may not disclose such information with a processor. In such cases we can follow the estimation technique used in [BTM00, J+06] to measure the energy consumed in the clock network:

$$clock_{Path} = nonGatedClock_{Path} \times \left(\frac{circuit_{Path}}{nonGatedCircuit_{Path}}\right) \tag{B.5}$$

  where $nonGatedClock_{Path}$ is the clock energy without gating and can be defined as:

$$nonGatedClock_{Path} = clockPower_{cycle} \times \mathrm{WCET}_{Path} \tag{B.6}$$

where *clock-power*$_{cycle}$ is the peak power consumed per cycle in the clock network and WCET$_{Path}$ is the worst-case execution time of basic blocks in the path. Moreover, $circuit_{Path}$ is the power consumed in all the components except clock network in the presence of clock gating:

$$circuit_{Path} = dynamic_{Path} + switchoff_{Path} + leakage_{Path} \tag{B.7}$$

nonGatedCircuit$_{Path}$, on the other hand, is the power consumed in all the components except clock network in the absence of clock gating:

$$nonGatedCircuit_{Path} = circuitPower_{cycle} \times \mathrm{WCET}_{Path} \tag{B.8}$$

$circuit - power_{cycle}$ is a constant defining peak dynamic plus leakage power per cycle excluding the clock network.

- **Leakage Energy:** The energy lost due to the leakage current regardless of the circuit activity per clock cycle is the leakage energy. For measuring the leakage energy we follow the measurement presented in [J^{+}06]:

$$leakage_{path} = P_{leakage} \times ExecutionTime_{path} \tag{B.9}$$

  Where $P_{leakage}$ is the power lost per processor cycle from the leakage current regardless of the circuit activity. This quantity is a constant given a processor architecture. $ExecutionTime_{path}$ is the execution time of the basic blocks along a path with regard to the context which reaches the basic blocks.

Revisiting the formula to measure the instruction-specific energy, the dynamic energy for a path can be measured by the following formula:

$$\begin{aligned} dynamic_{path} &= Energy_{ic} + Energy_{dc} + Energy_{reg} + Energy_{wk} + \\ & Energy_{ALU1} + Energy_{ALU2} + Energy_{FPU} + Energy_{MULTU} \end{aligned}$$

Where $Energy_{ic}$ is the energy consumed to access the instruction cache, $Energy_{dc}$ is the energy consumed to access the data cache, $Energy_{reg}$ is the energy consumed to access the register file, $Energy_{wk}$ is the energy consumed by the wake-up logic and $Energy_{ALU1}$, $Energy_{ALU2}$, $Energy_{FPU}$, $Energy_{MULTU}$ are the energies consumed by $ALU1$, $ALU2$, FPU and $MULTU$ along the path.

Furthermore, we depict the energy consumed by the clock with $clock_{path}$, selection logic with $selection_{path}$, the leakage energy with $leakage_{path}$ and sum of the switch-off energy in different components by $Switchoff$.

Example 8 (Computing Energy Cost of a Sub-path on an Out-of-order Pipeline)**.**

Consider the control flow graph (CFG) of a program fragment in Figure B.1(a). We abstract a basic block, shown as a rectangle, using (1) the beginning program point of the block; (2) and the sequences of instructions in the basic block. Two outgoing edges signify the branching structure, while the branch conditions are labeled beside the edges. The branch condition is shown beside each conditional branch.

The processor model has an 8-entry ROB, 4-entry I-buffer queue and the following functional units: two single-cycle integer ALU, an integer multiplier with $1 \sim 4$ cycle latency and a floating-point multiplier with $1 \sim 12$ cycle latency. Moreover, all the constant power consumptions of different devices are set to 1.

For simplicity, in this example, we consider *perfect data cache* (a single-cycle load/store unit) and *perfect branch prediction.* As a result, all accesses to the data cache would be cache hit and also all the predictions for the next branch would be correct prediction.

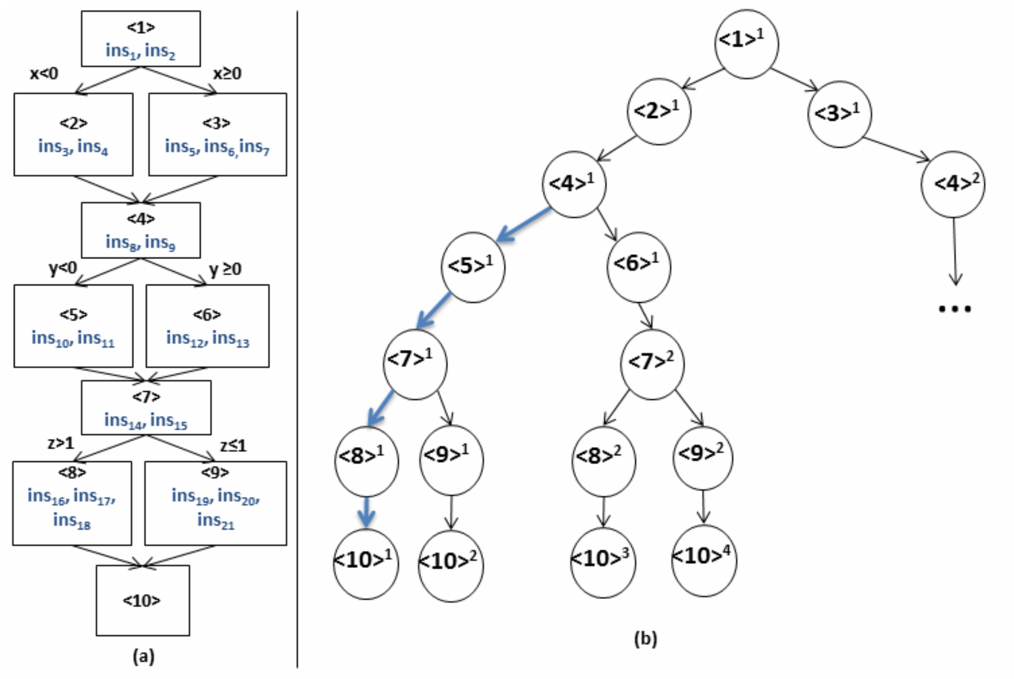


Figure B.1: (a) program control flow graph with accessed memory blocks and instruction running time shown inside each block, (b) the analysis tree of the program

However, the effect of the instruction cache is considered and an instruction cache hit would take one clock cycle, while an instruction cache miss would cost 10 clock cycles.

In this example, we consider the out-of-order superscalar RISC processor presented in Figure 5.10 and for the sake of illustration the size of the instructions is set to 2 bytes. As a result, instructions ins_1 to ins_{16} fall in the first memory block m_1 and the rest of the instructions (ins_{17} to ins_{21}) fall into m_2.

Next, in Figure B.1(b), we depict a *symbolic execution tree* of the program. The nodes, shown as circles, represent the program points, with superscripts to distinguish the multiple occurrences. Each path denotes a *symbolic execution* of the program. Each node is associated with a symbolic state $s \equiv \langle \ell, c, [\![s]\!] \rangle$, which preserves the context reaching the node. While the context is not explicitly shown in the figure (since the basic blocks are shown only abstractly), we shall make use of some obvious properties of the context of some nodes. Note that we have not (fully) drawn the subtree below node $\langle 4 \rangle^{(2)}$ in Figure B.1(b). In order to focus on the role of the dominating condition for pipeline in reuse, in this example all paths are feasible paths. In order to investigate more on the role of infeasible paths we refer the interested reader to the example presented in Section 2.2.2.

Table B.1: Instruction Details

Instruction	Latency of Instructions	Execution Unit	No. of Input Arguments	No. of Output Arguments
ins_1	1	LD/ST	1	1
ins_2	1	LD/ST	1	1
ins_3	4	MULTU	2	1
ins_4	1	LD/ST	1	1
ins_5	1	ALU	2	1
ins_6	4	MULTU	2	1
ins_7	1	LD/ST	1	1
ins_8	3	MULTU	2	1
ins_9	1	LD/ST	1	1
ins_{10}	12	FPU	2	1
ins_{11}	1	LD/ST	1	1
ins_{12}	1	ALU	2	1
ins_{13}	1	LD/ST	1	1
ins_{14}	1	ALU	2	1
ins_{15}	1	LD/ST	1	1
ins_{16}	4	MULTU	2	1
ins_{17}	1	LD/ST	1	1
ins_{18}	1	LD/ST	1	1
ins_{19}	1	ALU	2	1
ins_{20}	1	LD/ST	1	1
ins_{21}	1	LD/ST	1	1

Since the execution time, execution unit, number of input arguments and number of output arguments affects the energy cost of an instruction, Table B.1 presents such information for each of the instructions. For example, ins_8 is of the integer multiplication

Table B.2: Data dependencies between the instructions

Data Dependency
$ins_8 \rightarrow ins_9$
$ins_9 \rightarrow ins_{12}$
$ins_9 \rightarrow ins_{14}$
$ins_{12} \rightarrow ins_{16}$
$ins_{14} \rightarrow ins_{19}$

instructions with 3 cycle latency, 2 input arguments and 1 output argument. Furthermore, Table B.2, presents the list of data dependencies between the instructions. For example, ins_{16} can only be executed after the output of ins_{12} is written back.

We will compute the energy cost of the four paths in the left subtree. We start with the left most path in the symbolic execution tree. Note that in the beginning at node $\langle 1 \rangle^1$ the variable of interest r which tracks the energy consumption is set to 0. The execution of the leftmost path with can be seen in Figure B.2. The execution time of this path takes 32 clock cycles. The energy consumption for this path is calculated accordingly and can be seen in Table B.3.

Table B.3: The energy consumption for the leftmost path (Path 1)

Component	Consumed Energy
$Energy_{ic}$	$25 \times P_{ic}$
$Energy_{dc}$	$8 \times P_{dc}$
$Energy_{reg}$	$18 \times P_{Register-access}$
$Energy_{wk}$	$5 \times P_{Wake-up}$
$Energy_{ALU1}$	$1 \times P_{ALU}$
$Energy_{ALU2}$	$0 \times P_{ALU}$
$Energy_{FPU}$	$12 \times P_{FPU}$
$Energy_{MULTU}$	$11 \times P_{MULTU}$
$Leakage$	$32 \times P_{leakage}$
$Clock$	$\frac{75}{7} \times \text{clock-power}_{cycle}$
$Selection$	$10 \times P_{selection}$
$Switchoff$	$3.1 \times Switchoff_{ALU} + 3.2 \times Switchoff_{ALU} + 2 \times Switchoff_{FPU} + 2.1 \times Switchoff_{MULTU}$

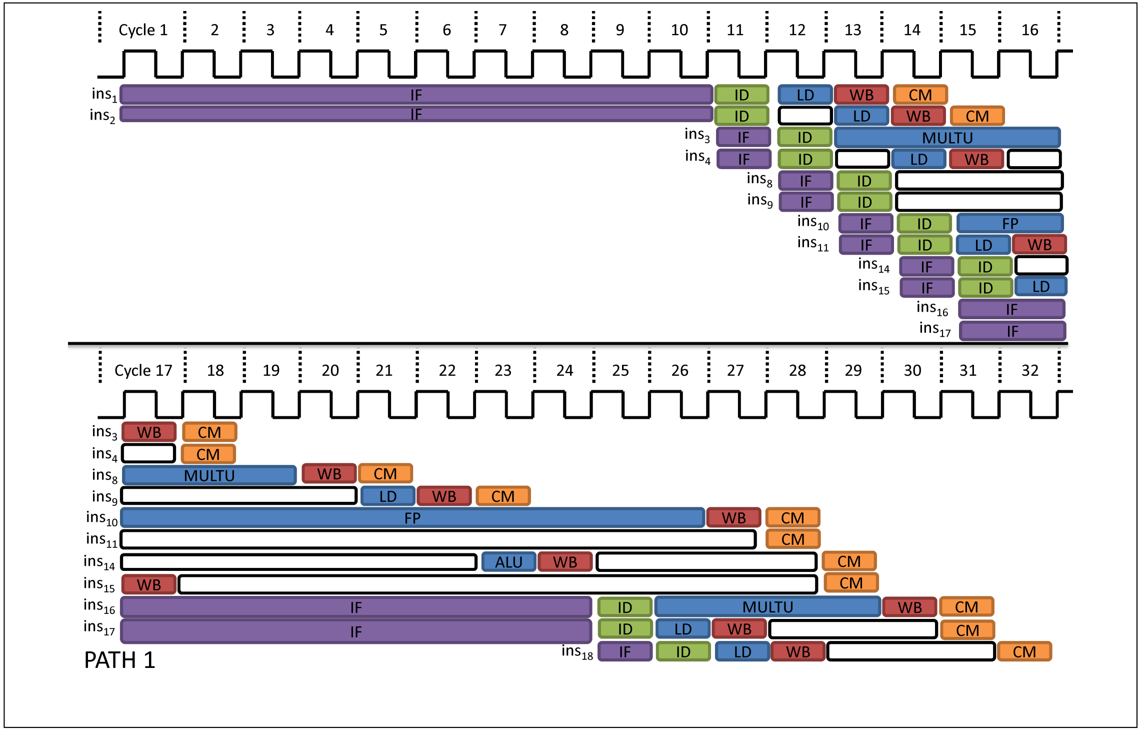


Figure B.2: The energy analysis of the leftmost path (Path 1)

The numbers in Table B.3 are generated from the execution times in Figure B.2 and the cost model presented in this appendix. For example, $Energy_{ALU1}$ is estimated from the formula $1 \times \; Power_{ALU}$ since ALU_1 is active in clock cycles 23 in Figure B.2. Furthermore, the *Leakage* energy is calculated from $32 \times \; P_{leakage}$, since path 1 takes 32 clock cycles to be executed. On the other hand, the *Selection* energy is calculated from $10 \times \; Power_{selection}$, since the selection logic is accessed in clock cycles $12, 13, 14, 15, 16, 17, 21, 23, 26$ and 27. Finally, we would like to mention the *Clock* energy. At most in a clock cycle, 7 gated components are consuming the clock power (instruction cache, data cache, ALU 1, ALU 2, MULTU, FPU and Register File). Counting the number of clock cycles that these components are in active mode results in 75. While this number can at most reach 7×32, where 32 is the execution time of the path. As a result, knowing that $\text{clock-power}_{cycle}$ is the maximum power consumed in the clock network, the formula $\frac{75}{7} \times \text{clock-power}_{cycle}$ estimates the *clock* energy in the path. Fixing all power consumptions to 1, the total energy consumption of the leftmost path would be 143.11 Micro-watt.

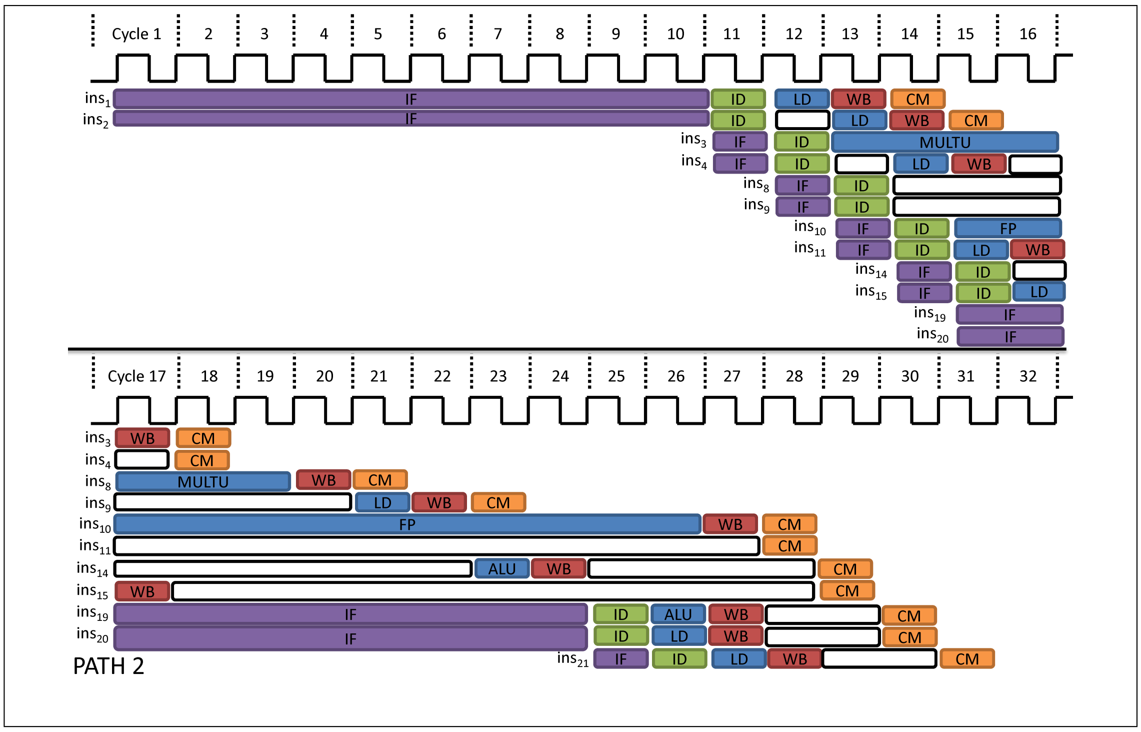


Figure B.3: The energy analysis of path 2

Table B.4: The energy consumption for path 2

Component	Consumed Energy
$Energy_{ic}$	$25 \times Power_{ic}$
$Energy_{dc}$	$8 \times Power_{dc}$
$Energy_{reg}$	$18 \times P_{Register-access}$
$Energy_{wk}$	$5 \times P_{Wake-up}$
$Energy_{ALU1}$	$2 \times Power_{ALU}$
$Energy_{ALU2}$	$0 \times Power_{ALU}$
$Energy_{FPU}$	$12 \times Power_{FPU}$
$Energy_{MULTU}$	$7 \times Power_{MULTU}$
$Leakage$	$31 \times P_{leakage}$
$Clock$	$\frac{72}{7} \times$ clock-power$_{cycle}$
$Selection$	$10 \times Power_{selection}$
$Switchoff$	$2.9 \times Switchoff_{ALU} + 3.1 \times Switchoff_{ALU} + 1.9 \times Switchoff_{FPU} + 2.4 \times Switchoff_{MULTU}$

Moving to the second path, the execution time of the leftmost path with respect to the pipeline state can be seen in Figure B.3. The execution time of this path takes 31 clock cycles. The energy consumption for this path is calculated accordingly and can be seen in Table B.4. The total energy consumption of this path would be 138.88 Micro-watt.

As it can be seen, the energy consumption of the second path is less than the first path. So the energy consumption of the sub-path $\langle\langle 7\rangle^1, \langle 8\rangle^1, \langle 10\rangle^1\rangle$ is more than the sub-path $\langle\langle 7\rangle^1, \langle 9\rangle^1, \langle 10\rangle^2\rangle$ (since the paths share the first part $\langle\langle 1\rangle^1, \langle 2\rangle^1, \langle 4\rangle^1, \langle 5\rangle^1, \langle 7\rangle^1\rangle$). As a result, in an analysis, the witness of the subtree beneath $\langle 7\rangle^1$ is $\langle\langle 7\rangle^1, \langle 8\rangle^1, \langle 10\rangle^1\rangle$.

Moving to the third path in the symbolic execution tree. The execution time of this path with respect to the pipeline state can be seen in Figure B.4. The execution time of this path takes 32 clock cycles. The energy consumption for this path is calculated accordingly and can be seen in Table B.5. The total energy consumption of the leftmost path would be 132.35 Micro-watt.

Table B.5: The energy consumption for path 3

Component	Consumed Energy
$Energy_{ic}$	$25 \times Power_{ic}$
$Energy_{dc}$	$8 \times Power_{dc}$
$Energy_{reg}$	$18 \times P_{Register-access}$
$Energy_{wk}$	$5 \times P_{Wake-up}$
$Energy_{ALU1}$	$2 \times Power_{ALU}$
$Energy_{ALU2}$	$0 \times Power_{ALU}$
$Energy_{FPU}$	$0 \times Power_{FPU}$
$Energy_{MULTU}$	$11 \times Power_{MULTU}$
$Leakage$	$32 \times P_{leakage}$
$Clock$	$\frac{69}{7} \times \text{clock-power}_{cycle}$
$Selection$	$10 \times Power_{selection}$
$Switchoff$	$3 \times Switchoff_{ALU} + 3.2 \times Switchoff_{ALU} + 3.2 \times Switchoff_{FPU} + 2.1 \times Switchoff_{MULTU}$

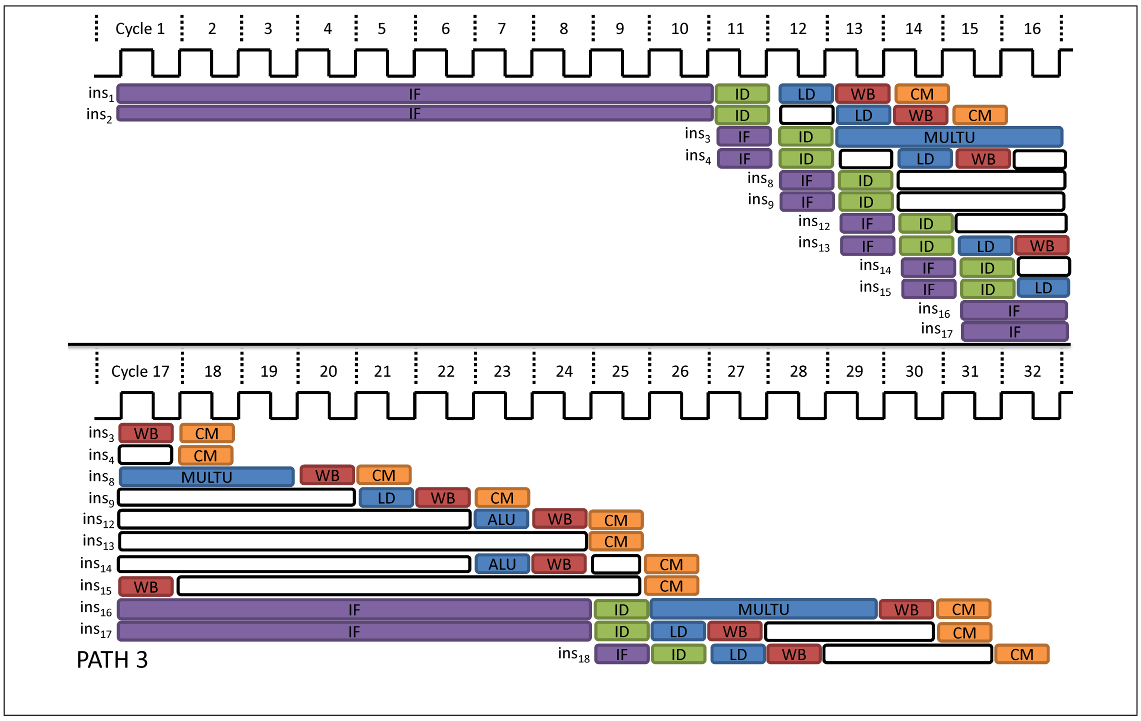


Figure B.4: The energy analysis of path 3

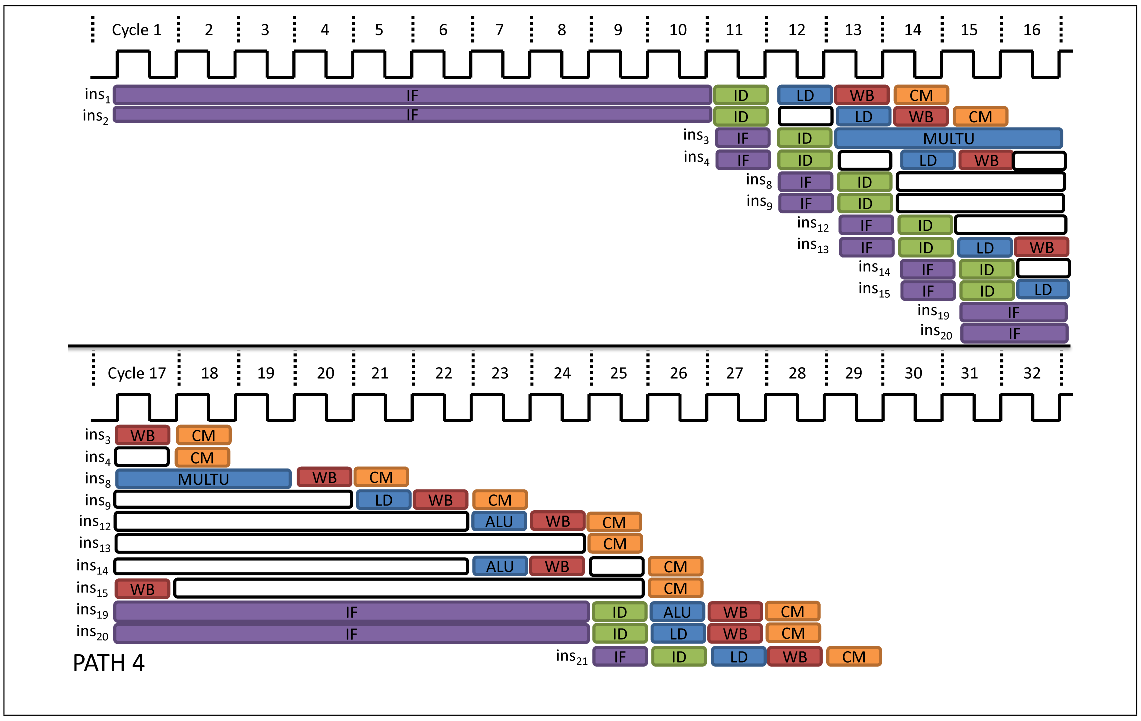


Figure B.5: The energy analysis of path 4

Table B.6: The energy consumption for path 4

Component	Consumed Energy
$Energy_{ic}$	$25 \times Power_{ic}$
$Energy_{dc}$	$8 \times Power_{dc}$
$Energy_{reg}$	$18 \times P_{Register-access}$
$Energy_{wk}$	$5 \times P_{Wake-up}$
$Energy_{ALU1}$	$3 \times Power_{ALU}$
$Energy_{ALU2}$	$0 \times Power_{ALU}$
$Energy_{FPU}$	$0 \times Power_{FPU}$
$Energy_{MULTU}$	$7 \times Power_{MULTU}$
$Leakage$	$29 \times P_{leakage}$
$Clock$	$\frac{60}{7} \times \text{clock-power}_{cycle}$
$Selection$	$10 \times Power_{selection}$
$Switchoff$	$2.6 \times Switchoff_{ALU} + 2.9 \times Switchoff_{ALU} + 2.9 \times Switchoff_{FPU} + 2.2 \times Switchoff_{MULTU}$

Continuing the depth first traversal in the symbolic execution tree to $\langle\langle 7\rangle^2, \langle 9\rangle^2, \langle 10\rangle^4\rangle$. The execution time of the fourth path with respect to the pipeline state can be seen in Figure B.5. The execution time of this path takes 29 clock cycles. The energy consumption for this path is calculated accordingly and can be seen in Table B.6. The total energy consumption of the leftmost path would be 124.17 Micro-watt. Moving back to node $\langle 4\rangle^1$, we can note that the sub-path $\langle\langle 4\rangle^1, \langle 5\rangle^1, \langle 7\rangle^1, \langle 8\rangle^1, \langle 10\rangle^1\rangle$ (in blue color) is the most energy consuming sub-path in the tree and in an analysis would be chosen as the witness path.