MICROARCHITECTURE MODELING FOR TIMING ANALYSIS OF EMBEDDED SOFTWARE

LI XIANFENG

(B.Eng, Beijing Institute of Technology)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2005

ACKNOWLEDGEMENTS

I am deeply grateful to my supervisors, Dr. Abhik Roychoudhury and Dr. Tulika Mitra. I sincerely thank them for introducing me such an exciting research topic and for their constant guidance on my research. I consider myself very fortunate to be their first Ph.D. student and because of this I had the privilege to receive their guidance almost exclusively in my junior graduate years (Some times I feel guilty for taking them so much time).

I have also benefited from Professors P.S. Thiagarajian, Samarjit Chakraborty and Wong Weng Fai. They have given me many insightful comments and advices. Their lectures and seminars not only have been another source of knowledge and inspirations for me, but also have been excellent examples for how to communicate scientific thoughts.

The weekly seminars of our embedded systems research group have been a unique forum for us to exchange ideas. I have learnt a lot by either presenting my own work or by listening to the talks given by our group members or visiting professors. I will certainly miss it after I leave our group.

I would like to thank the National University of Singapore for funding me with research scholarship and for providing such an excellent environment and services. My thanks also go to the administrative and support staff in the School of Computing, NUS. Their support is more than what I have expected.

I thank my friends Dr. Zhu Yongxin, Chen Peng, Luo Ming, Shen Qinghua and Daniel Högberg, with whom I play tennis and badminton. Doing sports has made my life here more fun and less stressful. I would also miss my other friends and lab mates Liang Yun, Pan Yu, Kathy Nguyen Dang, Wang Tao, Andrew Santosa,

Marciuca Gheorghita, Mihail Asavoae, Sufatrio Rio, Xie Lei and Wang Zhanqing. Our discussions, gatherings and other social activities made my stay at NUS enjoyable.

I have special thanks to my parents, my brother and sister for their love and encouragement. To make me concentrate on my study, they were even trying to conceal from me a serious illness of my mother when she was suffering it a couple of years ago.

Most of all, this thesis would not have been possible without the enormous support of Cailing, my wife. She has sacrificed a great deal ever since I decided to pursue my Ph.D. study. As an indebted husband, I hope this thesis could be a gift to her, and I take this chance to make a promise that I will never leave her struggling alone in the future.

The work presented in this thesis was partially supported by National University of Singapore research projects R252-000-088-112 and R252-000-171-112. They are gratefully acknowledged.

TABLE OF CONTENTS

SUMMARY

Worst Case Execution Times (WCET) of tasks are an essential input to the schedulability analysis of hard real-time systems. Obtaining the WCET of a program by exhaustive simulation over all sets of data input is often unaffordable. As an alternative, static WCET analysis predicts the worst case without actually running the program. One important yet difficult problem for static WCET analysis is to model the hardware features which have a great impact on the execution time of the program. In this thesis, we study the features that are commonly found in high performance processors but have not been effectively modeled for WCET analysis.

First, we model out-of-order pipelines. This in general is difficult even for a basic block (a sequence of instructions with single-entry and single-exit points) if some of the instructions have variable latencies. This is because the WCET of a basic block on out-of-order pipelines cannot be obtained by assuming maximum latencies of the individual instructions; on the other hand, exhaustively enumerating pipeline schedules could be very inefficient. In this thesis, we propose an innovative technique which considers all possible pipeline schedules but avoids an enumeration on them.

Next, we present a technique for dynamic branch prediction modeling. Dynamic branch predictions are superior to static branch predictions in terms of accuracy, but are much harder to model. There are very few studies dealing with dynamic branch predictions and the known techniques are limited to some relatively simpler schemes. Our technique can effectively model a variety of dynamic prediction schemes including the popular two-level branch predictions used in current commercial processors. We also study the effect of speculative execution (via branch prediction) on instruction caching and capture it by augmenting an existing instruction cache analysis.

Finally, we integrate the analyses of different features into a single framework. The features being modeled include an out-of-order pipeline, a dynamic branch predictor, and an instruction cache. Modeling multiple features in combination has long been acknowledged as a difficult problem due to their interactions. However, the combined analysis in our work does not need significant changes to the modeling techniques for the individual features and the analysis complexity does not increase sharply, indicating a good extensibility for incorporating more microarchitectural features.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Real-time Embedded Systems

Today a large portion of computing devices are serving as components of other systems for the purpose of data processing, control or communication. These computing devices are called *embedded systems*. The application domains of embedded systems are diverse: ranging from mission-critical systems, such as aviation systems, power plant monitoring systems, vehicle engine control systems, etc, to consumer electronics, such as mobile phones, mp3 players, etc.

Many of the embedded systems are required to interact with the environment in a timely fashion and they are called *real-time systems*. The correctness of such systems depends not only on the computed results, but also on the time at which the results are produced. Real-time systems can be further divided into two classes: *hard real-time systems* and *soft real-time systems*. Hard real-time systems do not allow any violation of their timing requirements. They are typically mission-critical systems such as vehicle control systems, avionics, automated manufacturing and sophisticated medical devices. With such systems, any failure to meet their deadlines may cause disastrous loss. In contrast, soft real-time systems can tolerate occasional misses of deadlines. For example, in voice communication systems or multimedia streaming applications, the loss or delay of a few frames may be tolerable. In this thesis, we are concerned with hard real-time systems.

1.2 Worst Case Execution Time Analysis

Typically, a hard real-time system is a collection of *tasks* running on a set of hardware resources. Each task repeats periodically or sporadically and can be characterized by a *release time*, a *deadline*, and a *computation time*. The *schedulability analysis* is concerned with whether it is possible to find a schedule for the tasks such that they all complete executions within their deadlines each time they are released (ready to execute).

Clearly, to perform schedulability analysis, the computation time for each task needs to be known a priori. Furthermore, to guarantee that the deadline is met in any circumstance, the Worst Case Execution Time (WCET) should be used as input instead of average case execution time. In reality, it may not be possible to know an exact WCET of a task and a conservative estimate is used. Tight WCET estimates are of primary importance for schedulability analysis as they reduce the waste of hardware resources. In this thesis, we study efficient methods for WCET estimations.

The Worst Case Execution Time to be studied in this thesis is defined as the maximum possible execution time of a task running on a hardware platform without being interrupted. There are several points for this definition to be noted. First, a simplified assumption is made that the task is executed uninterruptedly, while in a hard real-time system the task may be interrupted, e.g., by a higher priority task. The impact of interruptions on the execution of a task is another topic and it is beyond our research scope in this thesis. Second, the WCET is hardware-specific as the execution time of a task depends on the underlying hardware platform. Last, the execution time of a task varies with different data input and the WCET should cover all possible sets of data input.

In general, there are two approaches to determine the WCET of a task, or equivalently, the WCET of a program (as we are now shifting from a multi-tasking context

of schedulability analysis to a single task context of WCET determination, we will use the term program instead of task). The first approach is to obtain the WCET by simulating or by actually running the program on the target hardware over all sets of possible data input. However simulation or execution can only examine one set of data input each time. On the other hand, most non-trivial programs have a tremendous number of sets of possible data input, rendering an exhaustive simulation over all of them unaffordable. Another approach is to estimate the WCET by static analysis, which studies the program, derives its timing properties, and makes an estimation on the WCET without actually running the program. Static WCET analysis is expected to have the following properties:

- Conservative. The analysis should not underestimate the actual WCET, otherwise the system which is reported by the analysis as "safe" may actually fail. For example, the task is assigned a computation time which is above the reported WCET but lower than what is required for the actual worst case, resulting in its deadline being missed in some circumstances.

- Tight. The analysis should be reasonably close to the actual WCET, otherwise the task will be assigned an unnecessarily long computation time, i.e., a computation time no less than the estimated WCET. With the increase of computational requirement for each task, the promise of schedulability on the target hardware platform decreases and more powerful and expensive hardware platform may be needed.

- Efficient. The static analysis should be efficient in both time and space consumption.

Note the first property is compulsory and the other two are desirable.

Since the execution time of a program is affected by two factors: (a) the data input to the program, and (b) the hardware platform on which the program is running, their

effects need to be studied for WCET determination. The first factor mainly affects the execution path of a program and the second factor affects instruction timing, i.e., how long an instruction executes. Correspondingly, static WCET analysis can be divided into three sub-problems.

The first sub-problem is called **program path analysis**. It works on either the source program or the compiled code and derives program flow information such as what are the feasible paths and infeasible paths that an execution can go through. Later on, during the search of the worst case execution path, the identified infeasible paths will be excluded from consideration. Therefore the more infeasible paths are discovered, the more efficient and accurate the computation of the WCET.

The second sub-problem is called **microarchitecture modeling**. It is concerned with instruction timing. Traditionally, the execution time of an instruction is either a constant or easy to predict on processors with simple architectures. Modern processors, however, employ aggressive microarchitectural features such as pipelining, caching and branch prediction to improve the performance of the applications running on them. These features, which are designed to speed up the average-case execution, pose difficulties for instruction timing prediction. Firstly, the execution time of an instruction is no longer a constant, e.g., a cache miss may result in a much longer execution time that a cache hit does. Furthermore, the variation of instruction timing can be highly dynamic, e.g., without detailed execution history information, it may be unclear whether a cache access is a hit or a miss. Microarchitecture modeling studies the impact of the microarchitectural features on the executions of instructions. It provides instruction timing information which later on will be used to evaluate the costs of the execution paths during the search for the worst case execution path.

The third sub-problem is called **WCET calculation**. With the program path information and instruction timing information, the costs of the program paths are evaluated and the maximum one will be taken as the estimated WCET. In contrast

to the simulation approach, where program paths are evaluated individually, static WCET analysis performs this task more efficiently by simultaneously considering a set of paths which share some common properties. The correctness of the WCET calculation (the estimated WCET is not an underestimation to the actual WCET) relies on the earlier two sub-problems. First, no feasible paths are excluded by the program path analysis, otherwise the estimated WCET would be an underestimation in case the worst case execution path is among the excluded ones. Second, instruction timing estimated by microarchitecture modeling should be conservative, such that the cost of each program path will not be underestimated. On the other hand, the tightness of the estimated WCET depends on the first two sub-problems as well: the more infeasible paths are discovered, the less infeasible paths (which may have longer execution times than the feasible paths) are to be considered; and the more accurate the instruction timing, the tighter the estimation of the paths. There has been a few WCET calculation methods, which are different in the way that program paths are evaluated and the way instruction timing information is used. We will discuss them in the related work.

1.3 Contributions

In this thesis, we study microarchitecture modeling for WCET analysis. Our goal is to develop a framework for microarchitecture modeling which accurately estimates the timing effects of the three most popular microarchitectural features: instruction caching, branch prediction and pipelining (in-order/out-of-order). The framework should have an extensible structure, such that the modeling of more features can be conveniently incorporated. The contributions of this thesis can be summarized as follows.

- We propose a technique for out-of-order pipeline modeling. In out-of-order

pipelines, an instruction can execute if its operands are ready and the corresponding resource is available, irrespective of whether earlier instructions have started execution or not. Since out-of-order execution improves processor's performance significantly by replacing pipeline stalls with useful computations, it has become popular in high performance processors. The main challenge to out-of-order pipeline modeling is that out-of-order pipelines exhibit a phenomenon called *timing anomaly* [50], where counterintuitive events may arise. For example, a cache miss may result in shorter overall execution time of the program than a cache hit does, which means assuming a cache miss somewhere the actual cache access result is not available may be *not conservative*. Unfortunately, existing techniques largely rely on these conservative assumptions to make accuracy-performance trade-offs by only considering conservative cases. In the presence of timing anomalies, such trade-offs are no longer safe. As a result, all cases need to be examined. However, examining the possible cases individually could be very inefficient. In this thesis, we address the timing anomaly problem by proposing a novel technique which avoids enumerating the individual cases. Our technique is a fixed-point analysis over time intervals, where multiple cases of an event at a point are represented as an interval. This way, these cases can be studied in one go, and at the same time the analysis result obtained is still safe as long as the interval covers all cases.

- We develope a framework for the modeling of a variety of dynamic branch prediction schemes. The presence of branch instructions introduces control dependencies among different parts of the program. Control dependencies cause pipeline stalls called *control hazards* [30]. Current generation processors perform control flow speculation through branch prediction, which predicts the outcome of a branch instruction long before the actual outcome is available. If the prediction is correct, then execution proceeds without any interruption.

Otherwise (known as misprediction), the speculatively executed instructions are undone, incurring a branch misprediction penalty. If branch prediction is not modeled, all the branches in the program have to be assumed mispredicted to avoid underestimation. However, a majority of the branches can be correctly predicted in reality, which means the estimated WCET will be very pessimistic if branch prediction is not modeled. In this thesis, we propose a generic and parameterizable framework by using Integer Linear Programming (ILP). Since it is integrated with our ILP-based WCET calculation method, it can make good use of program path information for a tight estimate. Our framework can model the popular branch prediction schemes, including both global and local ones [51, 73].

- We propose a framework for combined analyses of the three features: out-of-order pipelining, branch prediction and instruction caching. The major issue with the combined analyses of multiple features is the sharp increase of the analysis complexity due to their interactions. By decomposing the timing effects of the various features into local timing effects (which affect nearby instructions) and global timing effects (which affect remote instructions), our combined analyses are divided into two levels: local analyses and global analyeses. By doing so, we can keep the analysis at a reasonable complexity, yet we can still receive good accuracy. The combined analyses of the three features also suggest that our framework has a good extensibility in the sense that incorporating the modeling of more microarchitectural features into the existing framework can be achieved with reasonable effort.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. The next chapter presents an overview of the approach taken in this thesis. Chapter 3 surveys the literature of WCET analysis.

Chapter 4 presents the out-of-order pipeline analysis. Branch prediction analysis is discussed in Chapter 5, where its integration with an ILP-based instruction cache analysis is also discussed. The combined analysis the three features is presented in Chapter 6. Finally, Chapter 7 gives a summary on what have been achieved in this thesis and points out possible future work.

CHAPTER II

OVERVIEW

In this chapter, we provide an overview of the approach taken in this thesis. First, we give some background information on the three microarchitectural features: out-of-order pipelining, branch prediction, and instruction caching. Then we introduce a concrete processor model used in this thesis. Next we present our overall approach for WCET analysis. Finally, we introduce the experimental setup used throughout this thesis.

2.1 Background on Microarchitecture

Microarchitecture is the term used to describe the resources and methods used to achieve architecture specification of processors. Modern processors employ aggressive microarchitectural features such as pipelining, caching and branch prediction to improve the performance of the applications running on them. The purpose of this section is to give some background information on the three popular microarchitectural features studied in this thesis.

2.1.1 Pipelining

The execution of an instruction naturally involves several tasks performed sequentially, or in other words, the execution proceeds through several stages. Therefore, instead of starting the execution of an instruction after the completion of an earlier one, we can overlap the executions of multiple instructions, where each one is in a particular execution stage at a time. This implementation technique is called pipelining. Ideally, if the execution which takes T time units to execute is divided into N pipeline stages with equal latencies, there can be an instruction completing

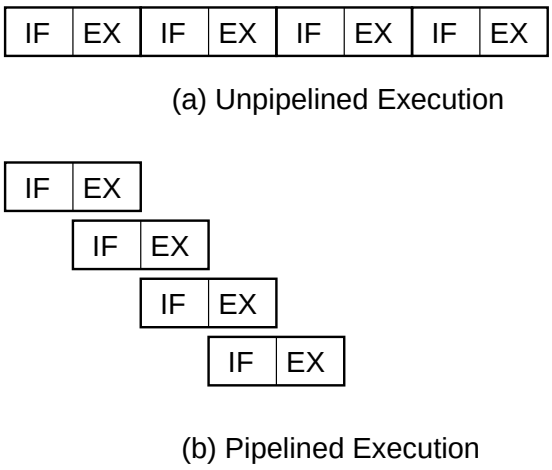


Figure 2.1: The Speedup of Pipelined Execution

execution each T/N time units, achieving a speedup of factor N. The speedup of pipelined execution is illustrated in Figure 2.1. With a two-stage pipeline, the execution takes roughly half the execution time of the unpipelined execution for four instructions. Modern processors have much deeper pipelines and the improvement is more substantial.

However, the ideal speedup of pipelined execution is often not reached because there are some events preventing the instructions from proceeding through the pipeline smoothly. These events are called *hazards* in the literature [30]. There are three classes of hazards.

- *Structural hazards.* Some of the resources needed by an instruction are currently unavailable, e.g., occupied by another instruction.
- *Data hazards.* Some of the data operands on which an instruction depends are currently unavailable, e.g., an operand to be provided by an earlier instruction is still under computation.
- *Control hazards.* The next instruction to be executed is currently unknown, e.g., due to branches or other control flow transfer instructions.

Because of these hazards, the execution time of an instruction or a sequence of instructions is not straightforwardly predictable, resulting in difficulties for timing analysis. This problem becomes more serious with aggressive pipelining mechanisms such as out-of-order execution. On an out-of-order pipeline, instructions can proceed through some of the pipeline stages out of their program order. This rise of complexity makes the hazards harder to predict. For example, in an out-of-order pipeline, a structural hazard happening to an instruction might be caused by either an earlier instruction or a later instruction, while in an in-order pipeline, it can only be caused by an earlier instruction.

2.1.2 Branch Prediction

The motivation for branch prediction is to address control hazards. When a conditional branch is executed, it computes the address of the subsequent instruction to be executed. There can be two possible outcomes: *taken* or *not taken*. If the branch outcome is taken, the subsequent execution will be redirected to a target address indicated by the branch instruction, otherwise it is not taken and the execution continues sequentially. However, the branch outcome is often available somewhere late in the pipeline, which means the processor does not know what to do between the interval from the start of the branch instruction to its production of the outcome.

If we do nothing with control hazards and let the processor idly wait for the branch outcome (the waiting time is called a branch penalty), we will have a significant performance loss. Hennessy and Patterson [30] have shown that for a program with a 30% branch frequency and a branch penalty of three clock cycles, their processor with branch stalls achieves only about *half* the ideal speedup with pipelining.

In light of this, various techniques have been proposed to reduce branch stalls. One effort is to reduce the branch penalty by computing the branch outcome and the target address as early as possible. However, constrained by the inherent nature

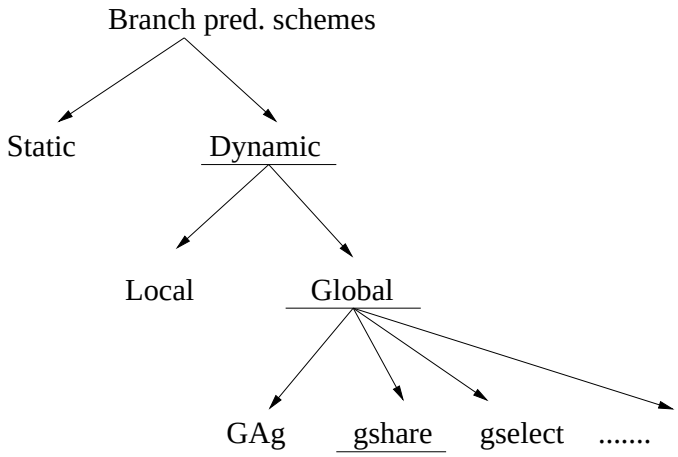


Figure 2.2: Categorization of Branch Prediction Schemes

of the pipelined execution, the computation of the branch outcome often cannot be done immediately after or very close to the start of the branch's execution, thus the branch stall cannot be completely overcome. In fact, on current processors with deep pipelines, the branch penalty can be over ten clock cycles.

Another method is to predict the branch outcome before it is available, such that the processor can continue execution along the predicted direction instead of idly waiting for the actual outcome. In case the prediction is correct, the branch penalty is completely avoided, otherwise it is a misprediction and some recovery actions must be taken to undo the effects of the wrong path instructions. The interval from the time the wrong path instructions entering the pipeline to the time the execution resuming on the correct path is called a *misprediction penalty*. It is the delay compared to the scenario of a correct prediction and is usually equal to or slightly higher than the branch penalty.

A variety of branch prediction schemes have been proposed and they can be broadly categorized as *static* and *dynamic* (see Figure 2.2; the most popular category in each level is underlined). In a static scheme, a branch is predicted the same direction every time it is encountered. Either the compiler can attach a prediction bit to every branch through analysis, or the hardware can perform the prediction

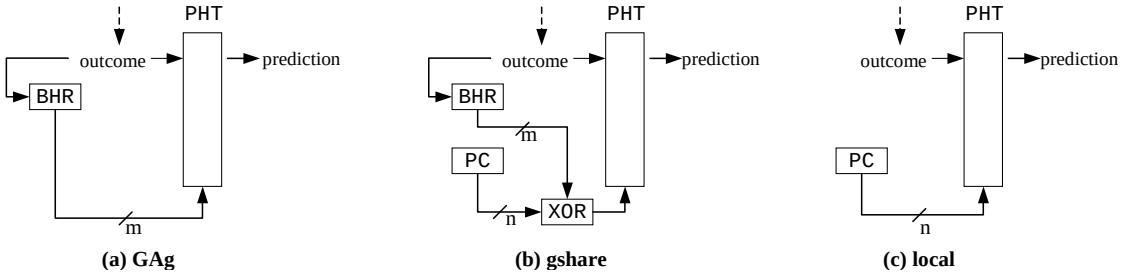


Figure 2.3: Illustration of Branch Prediction Schemes. The branch prediction table is shown as PHT, denoting Pattern History Table.

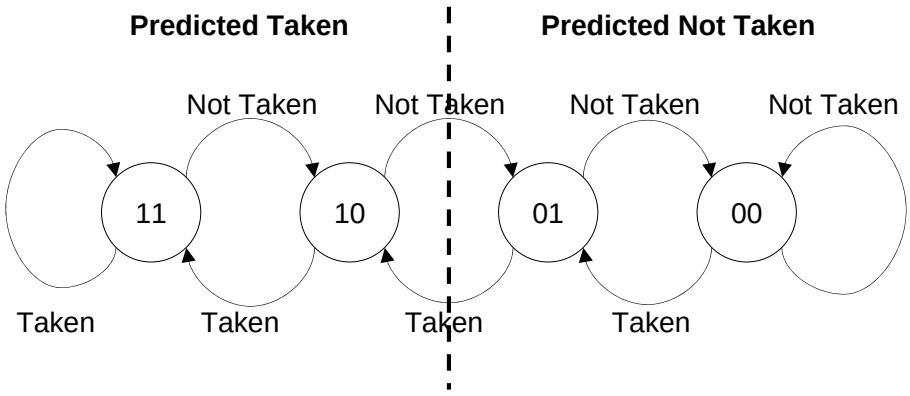


Figure 2.4: Two-bit Saturating Counter Predictor

using simple heuristics, such as backward branches are predicted taken and forward branches are predicted non-taken. Static schemes are simple to realize and easy to model. However, they do not make very accurate predictions.

Dynamic schemes predict the outcome of a branch according to the execution history. The first dynamic technique proposed is called *local branch prediction* (illustrated in Figure 2.3(c)), where the prediction of a branch is based on its last few outcomes. It is called "local" because the prediction of a branch is *only* dependent on its *own* history. This scheme uses a 2^n-entry *branch prediction table* to store past branch outcomes, and this table is indexed by the n lower order bits of the branch address. Obviously, two or more branches with the same lower order address bits

will map to the same table entry and they will affect each other's predictions (constructively or destructively). This is known as the *aliasing effect*. In the simplest case, each prediction table entry is one-bit and stores the last outcome of the branch mapped to that entry.

In this thesis, for simplicity of disposition, we discuss our modeling only for the one-bit scheme. When a branch is encountered, the corresponding table entry is looked up and used for prediction; and the entry will be updated after the outcome is resolved. In practice, two-bit saturating counters are often used for prediction, as show in Figure 2.4. Furthermore, the two-bit counter can be extended to n-bit scheme straightforwardly. We are aware that subsequent to our work, Bate and Reutemann [4] have developed techniques to extend the state-of-the art for modeling an n-bit saturating counter (in each row of the prediction table).

Local prediction schemes cannot exploit the fact that a branch outcome may be dependent on the outcomes of other recent branches. The *global branch prediction* schemes can take advantage of this situation [73]. Global schemes use a single shift register called *branch history register (BHR)* to record the outcomes of the n most recent branches. As in local schemes, there is a branch prediction table in which predictions are stored. The various global schemes differ from each other (and from local schemes) in the way the prediction table is looked up when a branch is encountered. Among the global schemes, three are quite popular and have been widely implemented [51]. In the *GAg* scheme (refer to Figure 2.3(a)), the BHR is simply used as an index to look up the prediction table. In the popular *gshare* scheme (refer to Figure 2.3(b)), the BHR is XOR-ed with the last n bits of the branch address (the PC register in Figure 2.3(b)) for prediction table look-up. Usually, *gshare* results in a more uniform distribution of table indices compared to *GAg*. Finally, in the *gselect (GAp)* scheme (not illustrated in Figure 2.3 but can be derived from the gshare scheme), the BHR is concatenated with the last few bits of the branch address to look up the table.

Note that even with accurate branch prediction, the processor needs the target address of a taken branch instruction. Current processors employ a small branch target buffer to cache this information. We have not modeled this buffer in our analysis technique; its effect can be easily modeled via techniques similar to instruction cache analysis [43]. Furthermore, the effect of the branch target buffer on a program's WCET is small compared to the total branch misprediction penalty. This is because the target address is available at the beginning of the pipeline whereas the branch outcome is available near the end of the pipeline.

2.1.3 Instruction Caching

Caching in our context is a mechanism used to bridge the gap between a faster processor and a relatively slower memory. A cache is a small, fast memory close to the processor that accommodates the most recently accessed code or data in the memory. If the data item needed by the processor is found in the cache, it is called a *cache hit*, otherwise the processor has to get it from the main memory and it is called a *cache miss*. The cost of a cache miss is called *cache miss penalty*. The caching mechanism is effective thanks to the *principle of locality*, which says that programs tend to reuse data and instructions they have used recently. It has been observed that a program may spend 90% of its execution time on only 10% of the code. Thus, by storing the recently accessed data in the cache, we will have a high chance of visiting them again from the cache in the future.

Program instructions and data can be cached either in a single storage, called von Neumann architecture, or in physically separate storages, called Harvard architecture. For embedded systems, Harvard architecture is more widely used. This makes it possible to study instruction caching and data caching separately. In this thesis we only study instruction caching.

Now we look at the organization of a cache with a simplified view. A cache is

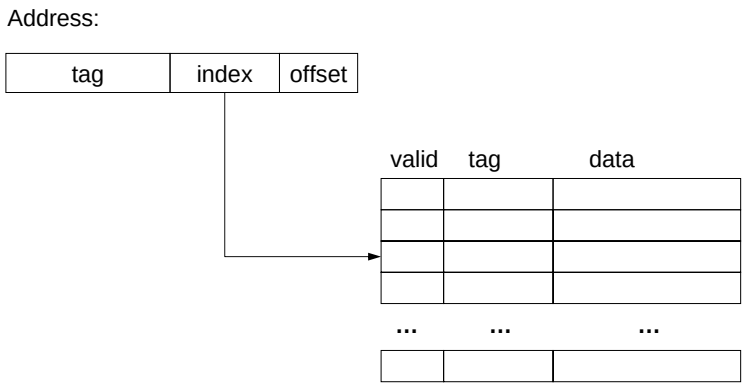


Figure 2.5: The Organization of a Direct Mapped Cache

organized in fixed-size blocks, each of which accommodates consecutive data items located in the memory (called memory blocks). Depending on where a memory block can be placed in the cache, there are three organization categories.

- If a memory block has only one place to go in the cache, the cache is called *direct mapped*.
- If a memory block can be placed anywhere in the cache, the cache is called *fully associative*.
- If a memory block can be placed in a restricted set of places in the cache, the cache is called *set associative*.

Direct mapped cache and fully associative cache can be viewed as two special cases of set associative caches. In this thesis, for simplicity of disposition, we will take direct mapped cache as an example, but our work can be extended to set associative caches. Figure 2.5 gives a simplified view of the organization of a direct mapped cache. A direct mapped cache is divided into multiple cache lines. Each cache line has three portions: a *data* portion which contains the memory block; a *tag* portion which is used to differentiate multiple possible memory blocks mapped to the same cache line; and a *valid* bit to indicate whether the cache line contains any valid data. When the

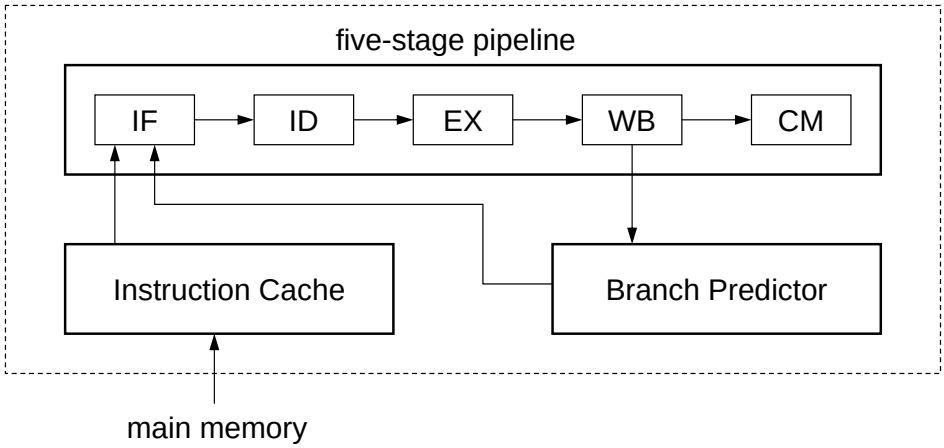


Figure 2.6: The Block Diagram of the Processor

processor accesses a data item, it dispatches the address of the data item to the cache. The address is divided into three fields as shown in Figure 2.5: The *index* field is used to determine which cache line to access; the *tag* field is used to decide whether the cache line contains the desired data (true if the tag field matches the tag portion of the corresponding cache line); and the block *offset* field is used to select the desired data item from the corresponding cache line. In case the memory block is not in the cache, access is directed to the main memory, and the memory block fetched from the main memory will displace the current one from the corresponding cache line.

2.2 A Processor Model

In this section we present the processor model used in this thesis. It is a simplified version of the SimpleScalar `sim-outorder` processor model [6], which is in turn based on [67]. The processor consists of three components: an out-of-order pipeline, a branch predictor and an instruction cache. The block diagram of the processor and the interactions among the three components are shown in Figure 2.6.

The pipeline consists of five stages. The interaction between the pipeline and the instruction cache takes place at the instruction fetch stage (IF on the diagram),

where the pipeline dispatches an instruction address to the instruction cache and the instruction is sent to the pipeline upon a hit, otherwise the instruction will be fetched from the main memory and the instruction cache is updated accordingly. The interaction between the pipeline and the branch predictor takes place at two stages. In the IF stage, the pipeline consults the branch predictor for the subsequent instruction to be executed. In the WB stage, where computed results are available, the branch predictor is updated with the branch outcome if the instruction is a conditional branch. The interaction between the branch predictor and the instruction cache is indirect (via the pipeline). The content of the instruction cache can be changed by the branch prediction in the following way: If the branch prediction is incorrect, the pipeline will execute instructions on the wrong path, which might bring some instructions into the instruction cache and displace some existing instructions. The instruction cache does not change the state of the branch predictor because the state of the branch predictor is only updated by the branch outcomes of the program, which is independent of the behaviors of both the pipeline and the instruction cache. Next, we give the organization of the pipeline and explain in more details how an instruction is executed by this processor.

The pipeline is shown in Figure 2.7. It consists of the following components: an instruction buffer (I-buffer), which accommodates instructions that have been fetched from the instruction cache or main memory, but yet to be decoded and executed; a circular reorder buffer (ROB), which accommodates instructions that have been decoded, but have not completed execution; several functional units which carry out the operations specified by the instructions; register files which hold computed results, including an integer register file and a floating-point register file.

An instruction proceeds through the five-stage pipeline as follows.

1. **Instruction Fetch (IF)**. In this stage, the instruction specified by the the program counter is fetched from the instruction cache or memory into the I-buffer.

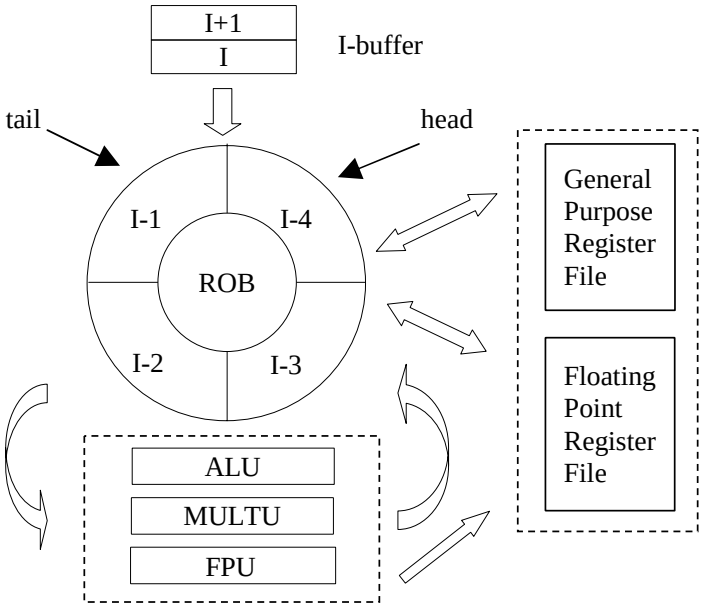


Figure 2.7: The Organization of the Pipeline

There are several rules dictating the behavior of the IF stage. Instructions enter and leave the I-buffer in program order. If the I-buffer is full, the processor stops fetching more instructions until the earliest instruction leaves the I-buffer.

2. **Instruction Decode & Dispatch (ID)**. In this stage, the earliest instruction in the I-buffer is removed from the I-buffer, decoded, and dispatched into the ROB. The instruction is stored there until it commits (see CM stage). The instruction decode cannot proceed if the ROB is full or the I-buffer is empty.

3. **Instruction Execute (EX)**. In this stage, an instruction in the ROB is issued to its corresponding functional unit for execution when all its operands are ready and the functional unit is available. If more than one instruction corresponding to a function unit are ready for execution, the earliest instruction has the highest priority. We assume that the functional units are not pipelined, that is, an instruction can be issued to a functional unit F only after the previous instruction occupying F has completed execution. We also assume that the

number of instructions issued in a clock cycle is only bounded by the number of functional units. The EX stage exhibits true *out-of-order* behavior as an instruction can start execution irrespective of whether earlier instructions have started execution or not.

4. **Write Back (WB)**. In this stage, instructions that have finished execution forward their results to awaiting instructions, if any, in the ROB. If all the operands of an awaiting instruction become ready, the instruction will be among the candidates scheduled for execution in the next cycle. We assume that there is no contention in the WB stage, that is, any instruction who has finished execution can always write its results back at this stage. Clearly, the WB stage exhibits *out-of-order* behavior as well.

5. **Commit (CM)**. This is the last stage where the earliest instruction which has completed the WB stage writes its output to the register files and frees its ROB entry. Note that the instructions commit *in program order*. Therefore, even if an instruction has completed its WB stage, it still has to wait for the earlier instructions to commit. We assume at most one instruction can commit each cycle.

In summary, in this processor model, EX and WB are the pipeline stages where instructions can proceed out-of-order, but resource contentions (contention for functional units) only happen in the EX stage.

2.3 Our Framework

In this section, we provide an overview of our approach for WCET analysis and microarchitecture modeling. As mentioned in Section 1.2, there are three sub-problems for WCET analysis: program path analysis, microarchitecture modeling, and WCET

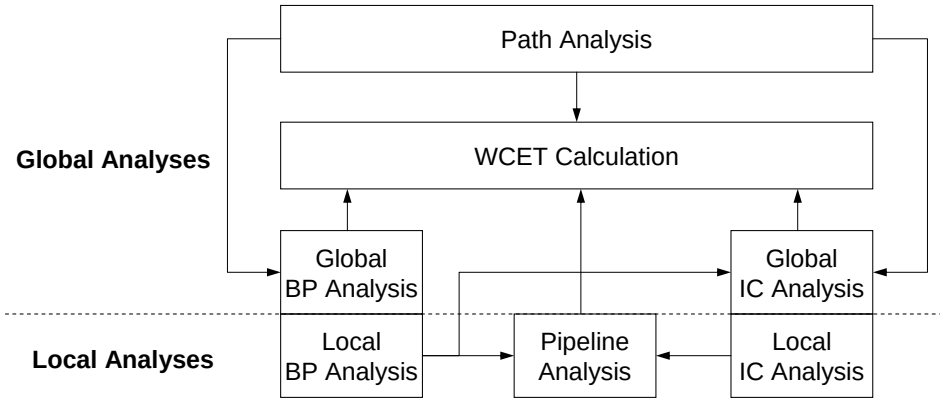


Figure 2.8: The WCET Analysis Framework

calculation. Our approach to performing these sub-problems and handling their interactions is illustrated in Figure 2.8. We divide the analyses into two levels: local analyses and global analyses, depending on whether global program flow information is needed or not in the respective analysis.

2.3.1 Program Path Analysis and WCET Calculation

The purpose of program path analysis is to identify feasible paths which later on will be used by WCET calculation. There has been extensive research work in this direction. Since our focus in this thesis is microarchitecture modeling, we do not propose new techniques for program path analysis, and the existing program path analysis techniques may be adopted here. The rest part is mainly for WCET calculation.

WCET calculation evaluates the costs of the program paths and takes the maximum one as the WCET. In contrast to simulation, where each program path is evaluated separately (the major drawback of the simulation approach), WCET analysis evaluates multiple program paths simultaneously. The key problem is how the program paths are grouped for evaluation. There has been an approach proposed by Li and Malik [40] which uses Integer Linear Programming (ILP) to represent the program paths. We adopt their approach for WCET calculation in our work. The idea

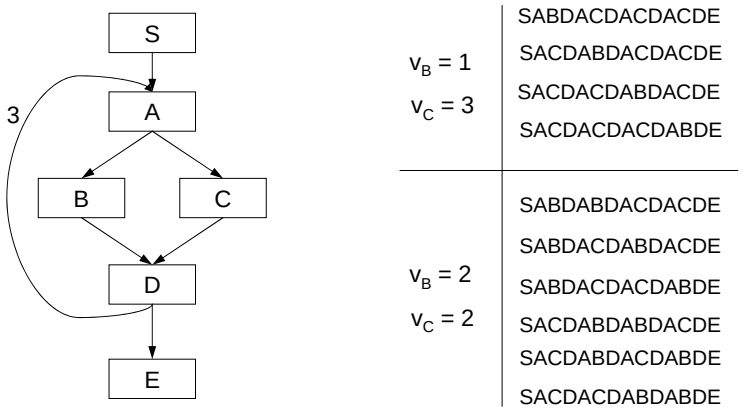


Figure 2.9: A Control Flow Graph Example

is as follows.

We work on the compiled code of the program. We first construct the Control Flow Graph (CFG) [1] for the program. The vertices of the graph are *basic blocks*, each of which is a sequence of instructions where flow of control can only enter from the beginning of the basic block and leave from the end. The basic blocks are connected by directed edges. There is an edge from block B_1 to block B_2 if and only if B_2 can follow the execution of B_1 in some execution sequence. The diagram on the left hand side of Figure 2.9 gives s simple CFG example.

Suppose the costs (execution times) of the basic blocks are known, then the execution time of a path can be calculated by first collecting the execution counts of the basic blocks on the path, then summing up the terms of the execution counts weighted by their costs. More formally, given a path P, its execution time T_P can be represented by the following equation.

$$T_P = \sum_{i=1}^{N} cost_i * v_i$$

where $cost_i$ and v_i are the cost and the execution count of block B_i respectively. If P does not contain block B_i, v_i is set to zero.

As mentioned earlier, static analysis evaluates a set of paths (or a segment of a

set of paths) at a time. The ILP approach achieves this by exploiting the fact that if two paths P_1 and P_2 have the same execution counts for each of their corresponding basic blocks, that is to say, they only differ in the execution order of the basic blocks, then their execution time will be the same (under the assumption that the costs of each basic block in the two paths are identical). From another point of view, the ILP assigns feasible execution counts to the basic blocks and give them an evaluation. This assignment actually represents a collection of paths with the same execution time, hence they need to be evaluated only once by the ILP solver. The right hand side in Figure 2.9 gives a concrete example. Suppose the loop from A to D iterates four times. Since there is an "if-then-else" branch inside the loop, each iteration the control flow may go through either B or C, thus there can be 16 paths of the program in total. By assigning one to the execution count of B ($v_B = 1$) and three to the execution count of C ($v_C = 3$), there can be four paths satisfying this situation and having the same execution time. These paths are listed in the upper half on the right hand side. Similarly, with $v_B = 2$ and $v_C = 2$, there are six paths that can be evaluated together (listed in the lower half on the right hand side).

Above we have discussed in an intuitively way on how program paths are grouped by ILP for evaluation. Actually, an ILP solver can do an even better job by exploiting relationships between different groups of paths (sets of execution counts). For details, please refer to [65, 69]. Formally, the WCET of the program with N basic blocks can be formulated as the maximization of the following problem.

$$Time = \sum_{i=1}^{N} cost_i * v_i \tag{2.1}$$

We call Equation 2.1 the *objective function*. The ILP solver maximizes $Time$ by trying to assign different execution counts to v_i. Obviously, there must be some constraints on the execution counts that can be assigned. A ready set of constraints

come from the control flow information. They are given as follows.

$$v_i = \sum_j e_{j \to i} = \sum_j e_{i \to j} \tag{2.2}$$

where $e_{i \to j}$ is the count of control flow transfer from block B_i to block B_j. Equation 2.2 captures the fact that the execution count of a basic block is equal to the sum of incoming control flow as well as the sum of outgoing control flow. Furthermore, for the *start* and *end* blocks, which execute exactly once, we have

$$v_{start} = v_{end} = 1 = \sum_i e_{start \to i} = \sum_i e_{i \to end} \tag{2.3}$$

The flow constraints by themselves are not enough. For instance, a program typically has loops whose iterations must be bounded, but above constraints by no means give such bounds. The loop bounds can either be derived by the program path analysis or be provided manually. For example, if we found that the loop in Figure 2.9 can iterate no more than four times, we add a bound $v_A \leq 4$ to the existing constraints. Besides the compulsory loop bounds, some more flow facts discovered by the program path analysis can be transformed to constraints to further bound the possible execution count assignment. For example, suppose $cost_B$ is larger than $cost_C$ in Figure 2.9, if the program path analysis finds out that B can only execute a limited number of times (less than the loop iterations) and this fact is transformed into an extra constraint, then the ILP solver will not be able to assign v_B a loop iteration count which leads to an unnecessarily overestimated WCET.

WCET calculation works on the scope of the global program, thus it belongs to the global analyses in our framework in Figure 2.8.

It worth noting that when microarchitecture is modeled, the cost of a basic block varies under different execution scenarios. In that case, we will identify the timing events that affect the cost and refine the execution of a basic block into a few secenarios, each of which may have a distinct cost and its occurrences will be bounded by microarchitecture modeling. The objective function will be changed accordingly.

2.3.2 Microarchitecture Modeling

Some of the timing effects of the microachitecture are mainly exercised in a local scope, and their analyses need no much program flow information. Pipelining is a typical example, where adjacent instructions affect each other, but remote instructions such as those who have completed execution do not affect instructions currently in the pipeline. As a result, pipeline analysis is performed at the level of basic blocks with very limited program flow information taken into account (e.g, a short sequence of instructions preceding or succeeding the analyzed basic block).

For instruction caching and branch prediction, it is well known that they exhibit global timing effects in the sense that an earlier cache access or branch instruction can update the state of the instruction cache or the branch predictor, which will affect future cache accesses or branch predictions. How long the effect is exercised is highly dynamic. For example, a cache access to an instruction I may displace another instruction I' from the cache; when I' will be visited again depends on the program path taken from I to I'. We call the analyses for the global effects global analyses ("Global IC Analysis" and "Global BP Analysis" in Figure 2.8). To receive reasonably accurate results, global program flow information needs to be taken into account for global instruction cache analysis and global branch prediction analysis.

On the other hand, instruction caching and branch prediction have local effects – mainly on the pipeline. For example, a cache miss results in a longer latency of the corresponding pipeline IF stage, and a branch misprediction results in the flush of the pipeline. We call the analyses for the local effects local analyses ("Local IC Analysis" and "Local BP Analysis" [1] in Figure 2.8).

Local analyses. Since pipeline is the place where instructions are executed and the execution time is accounted, the pipeline analysis is taken as the core of local level

[1] Note local branch prediction analysis is not the analysis for **local branch prediction schemes**.

analyses, while the local analyses of the other two features, instruction caching and branch prediction, are incorporated into the pipeline analysis with their effects on the corresponding pipeline stages being captured (indicated by the arrows from "Local IC Analysis" and "Local BP Analysis" to "Pipeline Analysis" in Figure 2.8).

Global analyses. The global instruction cache analysis and the global branch prediction analysis are concerned with the occurrences of the timing effects, e.g., cache misses and branch mispredictions. Li et al. [41, 43] have proposed an ILP-based instruction cache analysis which can be conveniently integrated with their ILP-based WCET calculation. In our global branch prediction analysis, to better exploit the program flow information, we also use ILP to model the global behavior of branch prediction (The technical details appears in Chapter 5). Recall in Section 2.2, we have mentioned that the state of the instruction cache can be affected by the behavior of the branch prediction. Now we revisit this issue with a perspective of global/local effects. Clearly, a misprediction, which may affect the cache state, has no impact on how a cache miss or hit affects the pipeline; rather, by changing the cache state, it affects whether a future cache access is a hit or a miss. Therefore, an arrow is drawn from local branch prediction analysis to global instruction cache analysis. In Chapter 5, we will augment the instruction cache analysis by Li et al. to capture the branch prediction effect.

Now we show the changes to WCET calculation with microarchitecture modeling enabled. Since the execution time of a basic block varies with timing events (cache misses, branch mispreditions) that may happen in its execution, and on the other hand, the occurrences of the timing events are bounded by global analyses. The objective function in Equation 2.1 will be changed to the following form.

$$Time = \sum_{i=1}^{N} \sum_{sc \in SC_i} cost_i^{sc} * v_i^{sc} \tag{2.4}$$

where sc is an execution scenario of block B_i, e.g., it may carry relevant cache state

and branch prediction information. The possible execution scenarios of B_i are captured by the set SC_i. For different sc and sc' of the same B_i, $cost_i^{sc}$ and $cost_i^{sc'}$ are expected to be different. The occurrences of each sc is bounded by global analyses, such that for an sc which results in a higher $cost_i^{sc}$ than other scenarios, the corresponding v_i^{sc} will not be assigned an impossibly high count. Note the scenario mentioned here is generic – we will see concrete scenarios in the respective microarchitecture modeling chapters.

In summary, we decompose microarchitecture modeling into two levels: local level and global level. The local level analyses are concerned with the timing of the analysis units (e.g., basic blocks) by modeling local timing effects, and pipeline analysis is the core at this level. The global level analyses are concerned with the occurrences of timing events, and it works on the scale of the whole program. By decomposing microarchitecture modeling into two levels, the analyses can be performed with reasonably complexity and the microarchitecture modeling can be conveniently extended when more features are to be modeled.

2.4 Experimental Setup

We will conduct experiments to evaluate our out-of-order pipeline analysis, branch prediction analysis and the combined analysis of the three features. The experiments share some commonalities such as the benchmarks used, the methodology, and the experimental environment.

Benchmarks Table 2.1 lists the benchmark programs used for experiments. These programs have been used by other researchers for WCET analysis. Among them, `des`, `dhry`, `fdct`, `fft`, `isort` and `whet` were used by Li et al. [43], and the others are from the real-time research group at Seoul National University [63]. For some of the programs, e.g, `dhry`, `fdct`, `matsum`, `matmul` and `fft`, their branch conditions are not

Program	Description
des	Data Encryption Standard
dhry	Dhrystone benchmark
fdct	Fast Discrete Cosine Transform
fft	1024-point Fast Fourier Transformation
fir	FIR filter with Gaussian function
isort	Insertion sort of 100-element array
ludcmp	LU decomposition algorithm
matmul	Multiplication of two 10x10 matrices
matsum	Summation of two 100x100 matrices
minver	Inversion of a floating point matrix
qurt	Root computation of quadratic equations
whet	Whetstone benchmark

Table 2.1: The Benchmark Programs

dependent on input data, hence they have single execution paths. The program `fir` has a single path as well, but it has a relatively complex control flow, which means that it will be a tough job for program path analysis to derive flow information as tight as what a simulation can report. The rest programs have multiple execution paths. To be free of dependence on library calls, the data input is hard-coded into the program.

Methodology To evaluate the accuracy of our analysis, the estimated result should be compared against some reference one. Ideally, it should be the *actual* worst case. However, as explained earlier, it is often impossible to know the actual worst case. As an alternative, we use an approximate to the actual worst case by doing an inexhaustive simulation over some sets of data input which are likely to produce the worst case. We call the result obtained this way the *observed* worst case. Correspondingly, the result produced by our analysis is called the *estimated* worst case. The relationships of the three values are: observed WCET $\leq$ actual WCET $\leq$ estimated WCET. Finding a set of data input for a good observed worst case is not easy, especially when timing effects introduced by microarchitectural features come into play. What we do is to inspect the important parts of a program (with the timing effects in mind), e.g.,

the inner loops, to get an idea on how the executions of these important parts are affected by the data input, then we try to feed the program with a set of data input which is likely to maximize their execution.

For both the simulation and estimation, we use SimpleScalar [6], a popular architectural simulation toolset, for a variety of tasks. Our experiments start with the source program. The first step is to compile the source program into object code using the GCC compiler provided by SimpleScalar. This GCC version yields code of an instruction set architecture (ISA) which is a superset of MIPS ISA [60].

Then, we simulate the object code by using one of the SimpleScalar simulators with the selected data input. Whichever is used for simulation depends on which microarchiture features are being modeled. In addition, to match our processor configuration, the simulators are tailored and the parameters are set correspondingly.

Next, we conduct the analysis with a prototype analyzer written by us. It reads the object code, constructs control flow graphs (CFG), performs local and global microarchitecture modeling, and formulates an ILP problem by producing an objective function, a set of flow constraints as well as constraints from microarchitecture modeling. In addition, flow information collected by program path analysis or by user observation is transformed into an extra set of flow constraints, e.g., loop bounds, which are called functional constraints.

Finally, the ILP problem is submitted to an ILP solver and the objective function is maximized by the solver. The produced result will be the estimated worst case. In our experiments, we use CPLEX [15], a commercial ILP solver for this task.

Environment We run all the experiments on a 1.3 GHz Pentium IV machine with 1-GB main memory. The operating system is Linux-2.4.18. The parameters of the processor components will be reported together with the experimental results in the respective chapters.

CHAPTER III

RELATED WORK

The importance of Worst Case Execution Time (WCET) analysis has been recognized by the real-time community and substantial progress has been made over the past fifteen years. The earlier work include Kligerman and Stoyenko's Real-time Euclid [35], Shaw and Park's timing schema [66, 58], and Puschner and Koza's study [61] on the calculation of WCET and its decidability issue. In the early time WCET analysis was targeted towards simple hardware, on which the timing of an instruction is constant, thus no need for microachitecture modeling; and if no much optimization is done by the compiler, working on the source program would be sufficient. However, with the advent of modern processors which employ aggressive performance enhancement features, it is not a feasible option anymore for not doing microarchitecture modeling and WCET calculation is usually carried out on the compiled code.

In the rest of this section, we review the literature on three topics: program path analysis, microarchitecture modeling, and WCET calculation. They correspond to the three sub-problems for WCET analysis introduced in Chapter 1. Since WCET calculation is directly connected to the aim of the analysis (the WCET of the program) and the other two sub-problems are performed to enable and improve WCET calculation, our review will first cover WCET calculation methods, then the rest two topics.

3.1 WCET Calculation

There are primarily three WCET calculation methods: timing schema, path-based calculation, and Implicit Path Enumeration Technique (IPET).

```
1:  sumneg = sumpos = sumall = 0;
2:  for (i=0; i<1000; i++) {
3:      if (a[i] < 0)
4:          sumneg += a[i];
5:      sumall += a[i];
6:      if (a[i] > 0)
7:          sumpos += a[i];
8:  }
```

Figure 3.1: An Example of Infeasible Paths (by Healy and Whalley)

Timing Schema. Shaw and Park [66, 58, 57] proposed a tree-based approach called *timing schema*. It determines the execution times of program constructs with a bottom-up traversal of the syntax tree. Once the times of lower level constructs have been obtained, the time of the higher level construct containing them can be estimated. The advantage of this approach is that it is very efficient. However, the local estimation in timing schema cannot account for infeasible paths which are defined by constraints across multiple constructs. Consider the example in Figure 3.1 (which is from [27]). Clearly, the statements on Line 4 and 7 are mutually exclusive and any paths across the two statements in the same iteration are infeasible ones. Timing schema estimates the costs of the two `if` statements separately, with the executions of both the two statements on Line 4 and 7 being assumed true. Thus the estimated worst case for this example will arise from an infeasible path.

Timing schema has been adopted and extended by some other researchers [10, 11, 31, 44, 45, 46]. Lim et al. [44, 45] and Hur et al. [31] have used it for WCET analysis on RISC processors. In their work, they used new data structures and replaced some of the operations in the original timing schema with operations which work on these data structures. Their revised timing schema can better account for timing effects of the pipelines and the caches. Colin and Puaut [10, 11] have recognized the importance

of loop nestings for tight WCET estimates. In their work, a construct is estimated under the context of its different loop nestings. Because of this, the result for a construct i is a set of touples $< wcet_i, ln\_level_j >$ instead of a single $wcet_i$, where ln_level_j is a loop level in which the construct can be located. They have developed a static analysis tool named Heptane [1].

Path-based Calculation. To better exploit the correlations of different program parts. Some researchers work on program paths for WCET Calculation [3, 24, 27, 28, 49, 68]. Arnold et al. [3] and Healy et al. [24, 27, 28] search the longest *loop paths*[2] in each loop-nesting level. Infeasible loop paths found by program path analysis are disregarded (e.g., paths go through both Line 4 and 7 in the program in Figure 3.1). Furthermore, the longest loop path may only execute a limited number/range of iterations. In that case, the search continues on finding the next longest path as well as the iterations in which it can execute. This process terminates when all iterations are exhausted. Then, the cost of the loop can be calculated by summing up the terms of the longest paths weighted by their costs. This path-based calculation traverses the program hierarchically, such that when the cost of an outer loop is under calculation, the costs of its inner loops are available for use.

Stappert et al. [68] developed another path-based WCET calculation method. They construct a *scope graph* – a hierarchical representation of the program. The longest paths are searched for the scopes. To simplify the work, each scope may be expanded into some *virtual scopes*, where the iterations are covered by the same set of *flow facts* (flow information derived from programm path analysis). They then search the longest path in each virtual scope. If the longest path is an infeasible one, it is discarded and the search continues. Unlike the first path-based calculation,

[1] http://www.irisa.fr/aces/work/heptane-demo/heptane.html

[2] A loop path is a control-flow connected sequence of blocks in a loop which starts with the loop header and terminates at a block with a transition either to the loop head or out of the loop.

each virtual scope has a unique longest path applicable to all iterations of the virtual scope. Thus the cost of a virtual scope is simply the cost of the longest path times the scope iterations. The WCET of the program can be calculated via a bottom-up traversal of the scope graph.

Lundqvist and Stenström [49] used cycle-level symbolic simulation technique for WCET calculation. Symbolic simulation needs to handle two problems: unknown data values in data-manipulating instructions and unknown conditions in conditional branches. In the later case, both paths of the branch need to be simulated. Since the number of feasible paths across the entire program can be substantial, to reduce the paths maintained for simulation, they apply path merging, which is typically carried out at the beginning of each loop iteration. The path merging should guarantee that the execution following the merged path will not lead to a time lower than what an execution following any of the pre-merged paths can do. Symbolic simulation can exclude some infeasible paths. For example, if the branch condition evaluated is known, the false path will not be simulated.

IPET. Li and Malik [40] proposed a technique which considers all paths *implicitly* by using integer linear programming (ILP). Suppose the cost of each basic block B_i, denoted as $cost_i$, is known, and let its execution count be denoted as v_i, then the execution time of a complete program with N basic blocks can be expressed as $\sum_{i=1}^{N} cost_i * v_i$. Then the rest of the task is to maximize the value of this function over all valid combinations of the execution counts. The value of the execution count v_i can take is bounded by the control flow of the program as well as some extra flow information derived from program path analysis or observed by user. The path enumeration is implicit in the sense that each combination of execution counts actually captures a set of program paths which have the same execution counts for the corresponding basic blocks, but the orders in which the basic blocks are executed are

different. An example illustrating this is given in Figure 2.9 in the overview chapter. This approach (IPET) differs from the path-based approaches in the following aspects. First, the paths (implicitly enumerated) in IPET are entire program paths whereas the paths in most of the path-based approaches are segments of program paths, e.g., paths within loops. Second, IPET considers a set of paths having the same combination of execution counts whereas path-based approaches considers a single path during the longest path search. Last, a path in IPET does not contain temporal information (the order in which basic block are executed) whereas a path in path-based approaches specifies a deterministic execution order for the basic blocks on the path. Note that both approaches can have some optimizations to speed up the search for the longest path. For example, in a path-based approach, Dijkstra's algorithm for longest-path search can be used to more efficiently find the longest path in a loop or a scope [13]; in IPET, the ILP solver can employ very aggressive algorithms to explore the relationships between different combinations of execution counts (refer to [65, 69] for more details).

Because of its simplicity and efficiency for path enumeration, the availability of powerful ILP solvers and a potential for a closer integration with microarchitecture modeling (will be explained later), the IPET approach has been adopted by some other researchers including us for WCET calculation [8, 37, 54, 70, 71, 38, 39].

3.2 Microarchitecture Modeling

Microarchitectural features, especially pipelining and caching, have caught a lot of attention for accurate WCET analysis. We review the various microarchitecture modeling techniques in this section.

Extended timing schema. Researchers at Seoul National University [31, 44, 45, 46] proposed a technique for modeling RISC processors. They extended timing schema

to account for pipeline and cache effects. In their work, the time-bound for a program construct in the original timing schema is replaced by a data structure called worst case timing abstraction (WCTA). It contains a set of elements, each of which corresponds to a possible worst case path in the program construct. An element in a WCTA consists of a time-bound for its respective path and a reservation table, which captures the use of pipeline stages and instruction interactions. When two adjacent constructs are concatenated, path concatenation is realized by concatenating the reservation tables in the two constructs, where interactions between instructions across construct borders are modeled. After concatenation, a prune operation may discard some concatenated reservation tables which can not be the worst case.

To model instruction cache effects, they divide memory accesses in a path of a construct into three groups: first/last/other references to the cache lines. Cache hits/misses of the first references need to be resolved with execution information preceding the path and the last references are needed by paths succeeding it, thus they are remembered by augmenting the WCTA. When concatenating two paths across program constructs, the last references are used to resolve some of the hits/misses in the first references in the later path, and the first/last references of the concatenated path will be computed from the first/last references of the two concatenating paths.

Their treatment to the combination of the two analyses is simple: just superimpose cache miss penalties to the execution time obtained from the pipeline analysis, where instruction cache effects were not considered.

Flow analysis technique. The approach proposed by researchers at Florida State University [3, 24, 28] is based on flow analysis techniques found in optimizing compilers. The target architecture includes pipelines and instruction caches. They first perform instruction cache analysis by using a static cache simulator [55, 56]. The simulator analyzes the program control flow and categorizes instructions into four

classes: *always hit*, *always miss*, *first hit*, and *first miss*. The categorization information is associated with loop levels. For example, an instruction I categorized as *always miss* for an outer loop L_1 might be categorized as *first miss* for an inner loop L_2. This is to more accurately account for cache behaviors and the WCET.

Next, they perform pipeline analysis by using the cache category information. This work consists of two steps. First, they perform pipeline analysis for loop paths. To model the pipeline behavior, the key point is to model its structural hazards and data hazards. They use two data structures for this purpose. The structural/data hazards information stored in each path will be used by the path concatenation algorithm. Next, they perform loop analysis to predict the worst case execution time of a loop. To avoid the complexity of calculating all combinations of paths, they union the pipeline effects of the paths for a single iteration of a loop. The union operation should guarantee conservativeness for safe WCET of the loop.

Last, the timing analyzer predicts the WCET of the program by using the worst case execution times of the code segments containing loops, function calls etc. Like timing schema, this is done in a bottom-up manner.

Abstract Interpretation Researchers at Saarland University [22, 71] used *abstract interpretation* [14] for instruction cache analysis. The analysis consists of two steps. The first step is to collect *abstract cache states* at program points. Intuitively, in an abstract cache state, each cache line contains a set of memory blocks. They define two functions: an *abstract cache update* function, which specifies how an abstract cache state is updated by a cache access; and a *join* function, which combines two or more abstract cache states at program joins. By traversing the program flow, abstract cache states are updated and joined. In the second stage, the abstract cache states are used to categorize memory references into four categories: *always hit*, *always miss*, *persistent* and *not classified*. The category information will be used for subsequent

analysis where cache information is needed.

They have also used abstract interpretation for pipeline analysis [64]. They first introduce concrete pipeline semantics to model the pipeline behavior and capture pipeline hazards (structural and data). Instruction executions on the pipeline are described by updates of concrete pipeline states. A concrete pipeline state describes the occupancy of the pipeline stages by instructions, resource allocations and states of some other resources. Based on the concrete pipeline semantics, they build abstract pipeline semantics, in which an abstract pipeline state is a set of concrete pipeline states. Update on an abstract pipeline state is realized by updating each of the contained concrete pipeline states. In some cases, if the update involves some non-deterministic events (e.g., a load with unknown address), one concrete pipeline state is split into multiple successor states. If a successor state cannot be determined as impossible to be the worst case, it has to be kept in the new abstract state. They claim that in general the number of concrete states in an abstract state is small, therefore operations on abstract pipeline states are efficient.

In recent years, they have targeted their work to real-life modern processors. Langenbach et al. [36] modeled Motorola ColdFire-5307, and Heckmann et al. [29] modeled PowerPC-755, an out-of-order processor.

Integer linear programming. Li et al. [41, 42, 43] used integer linear programming (ILP) for instruction cache modeling and combined it with their ILP-based WCET calculation method. In their work, the cache behavior is modeled by a set of graphs called Cache Conflict Graphs (CCG) for a directly mapped instruction cache. The CCG models flow transfer information among memory blocks[3] mapping to the same cache line. Cache misses are captured as flow transfer between conflicting memory blocks. Variables and linear constraints are generated from the CCGs and are

[3] a sequence of instructions in a basic block which map to the same cache line

incorporated into the existing ILP problem. This way, the modeling of cache behavior is tightly coupled with the modeling of program flow. For set associative instruction caches, an extra set of graphs called Cache State Transition Graphs (CSTG) are introduced to model their more complicated behaviors. This ILP-based instruction cache modeling, due to its ability of using more detailed flow information, achieves good accuracies. On the other hand, its tight integration with WCET calculation results in an increase in analysis time, especially for set associative caches.

Symbolic simulation Lundqvist and Stenström [49] used cycle-level symbolic simulation technique for WCET calculation. Microarchitectural features such as caching and pipelining are modeled during the symbolic execution. The instruction cache state in the simulation is updated along an execution path and cache states from multiple paths are merged at a path join. Each cache line in the cache state contains either a block of program instructions or invalid content (for direct mapped cache). They have two merge strategies: pessimistic merge and optimistic merge. With the pessimistic merge, if the contents of the respective cache lines from two different paths are different, invalid content is assumed for the cache line in the merged cache state. Optimistic merge is based on the idea that if it is known in advance that one partial path does not belong to the worst case path, the cache state of this path is simply ignored by the path merge. In their work, they predict the worst case penalty and best case penalty that the cache state of each path can incur. For two partial paths P_1 and P_2, if the cost of P_1 plus its worst case penalty is less than the cost of P_2 plus the best case penalty of P_2, then P_1's cache state will be ignored in the merge. For pipeline modeling, they use pipeline reservation tables to maintain the pipeline state. A reservation table record when each resource (pipeline stages or register) is released. With the reservation tables, pipeline hazards (structural and data) can be captured. During the simulation, the reservation table can be updated for each instruction at

a time. For the path merge, the pipeline reservation tables are merged following the same strategy of cache states merge. The accuracy of this approach depends on how many infeasible paths can be identified during simulation and how many path merges can be applied with the optimistic merge.

Other techniques There are some other techniques on the modeling of pipelines and instruction caches. There are also some work on the modeling of other microarchitecture features such as branch prediction, data caching, prefetching etc.

Engblom [16] provides a comprehensive study of various pipelines for WCET analysis in his doctoral dissertation. His work for pipeline modeling is based on a concept called *timing effects*, which reflect the impact of an earlier instruction on subsequent instructions. Formally, given two consecutive instructions I_1 and I_2, let their isolated execution times be $T(I_1)$ and $T(I_2)$ respectively, and let their combined execution time be $T(I_1I_2)$, the timing effect is defined as $\delta_{I_1I_2} = T(I_1I_2) - (T(I_1) + T(I_2))$. Due to pipeline overlap, $\delta_{I_1I_2}$ is often negative and the timing effect is called negative timing effect. The concept of timing effect can be extended to a sequence of more than two instructions. A timing effect related to a long instruction sequence is called long timing effect. If long timing effects are absent or insignificant on a pipeline, then the execution time of an instruction sequence can be obtained by doing simulation on its short sub-sequences; otherwise, one either performs extensive simulations on both its short and long sub-sequences to get a tight estimate or trades accuracy for performance by ignoring the long time effects. Note a time effect can only be ignored if it is negative. Ignoring *positive* timing effects results in underestimation. Therefore, positive long timing effects pose a problem for this approach. Unfortunately, it has been observed in his dissertation that out-of-order pipelines exercise positive long timing effects.

Branch prediction started getting attention in recent years. Compared to instruction caching, dynamic branch prediction [51, 73] is more difficult to model as similar regular properties for instruction caching do not exist in dynamic branch prediction schemes. For instance, for some inner loops which can be completely accommodated by the cache, the accesses except for the first time to an instruction will always be hits as long as the execution is repeated within the loop. This spatial locality has been exploited by some techniques which differentiate instruction executions with respect to their execution contexts such as loop levels and function calls ([3, 24, 28] and the VIVU approach in [22, 71]). In contrast, spatial locality is not obvious or does not exist for dynamic branch prediction schemes. For example, a conditional branch which is repeatedly executed in an inner loop may disturb itself by changing its direction each time and making itself wrongly predicted. As a result, branch prediction modeling is expected to take more effort. The difficulties for branch prediction modeling have been discussed by Engblom [17].

To our knowledge, the first detailed branch prediction analysis for WCET was performed by Colin and Puaut [9]. They modeled the *Branch Target Buffer* (BTB), which can be found in a Intel Pentium processor. With the BTB scheme, a branch is either predicted according to its history in the BTB or is predicted as not taken if it is absent from the BTB. In their work, the evolution of the BTB state with program flow is studied and information is collected along with the evolution. Next, with the collected information, branch instructions are classified according to whether they are predicted by their history or by default. The classification is connected with correct predictions/mispredictions in the following way. Since their WCET calculation is based on timing schema, the worst case path taken in a construct is always the same path across different iterations, thereby a branch instruction always takes the same direction on the worst case execution path of the program. Thus, for a branch predicted by its history, the prediction is correct. For a branch predicted by default,

depending on its direction in the worst case path, it can be statically determined whether it is correctly predicted or mispredicted. Only for a branch whose source of prediction (by history or by default) is unknown, its prediction is assumed to be mispredicted for the sake of conservativeness. This way, the timing effects of branch predictions can be accounted for WCET analysis. It needs to be pointed out that above disposition takes a simplified view of their work. In fact, due to their extension to the original timing schema, the worst case path of a construct and the direction of a branch in it may not be globally unique, rather they are unique only in a specific loop level. But the rationale behind remains unchanged.

Another work on branch prediction analysis is by Bate and Reutemann [4]. They modeled *bimodal* branch predictors. Like Colin and Puaut [9], they tried to classify branch instructions. The difference is that their classification is based on semantic context of a branch, rather than using dynamic execution.

Comparison In this part, we compare the various modeling techniques (including ours).

As for instruction cache analysis, the flow analysis approach and the abstract interpretation approach perform it before WCET calculation; while in the extended timing schema, integer linear programming and symbolic simulation approaches, instruction cache analysis is integrated with WCET calculation. Integrated approaches have the potential of achieving more accurate results as more program path information can be used for cache analysis, but it may have a higher computation cost. For example, when the ILP approach is used for modeling set associative instruction caches, very long computation time has been observed. For separated approaches, the analysis results are general and conservative enough to be applicable to all possible program paths or to one of a few sets of program paths (when execution context information is imposed, e.g., loop levels and function calls). This can be viewed as

trading accuracy for performance. However, due to its locality, instruction caching can still be modeled with good accuracy by separated approaches if execution context information such as loop levels is used to distinguish the accesses of an instruction.

As for pipeline analysis, we compare our work with the various approaches. We model an out-of-order pipeline [39] where an instruction can be executed in variable latencies. For such a pipeline, considering only one latency for each instruction such as the longest one would be unsafe [50]. In contrast, most of the surveyed pipeline analysis approaches are only applicable to in-order pipelines. In addition, they assume that an instruction executes with a single latency or implicitly take the longest latency for estimation. Recently, the abstract interpretation approach has been applied to out-of-order pipelines [29]. However, as mentioned earlier, the issue is that the pipeline states are updated against each possible latency when a variable-latency instruction is encountered, leading to an accumulation of pipeline states along the estimation process. In case the sequence of instructions to be estimated is not very short and the pipeline is complex, this approach can result in state space explosion [72]. Our approach avoids enumerating the individual execution latencies of an instruction by using an interval to represent the latencies, and it employs an efficient fixed-point algorithm to iteratively tighten the intervals. Another advantage of our approach is its convenience for integrating with the analyses of other microarchitectural features. For example, it can either be integrated with an instruction cache analysis where cache accesses are classified as hits or misses before pipeline analysis is carried out, or be integrated with an ILP-based instruction cache analysis, where cache hits/misses are figured out during WCET calculation (in this thesis, we use the later approach). In contrast, most of the surveyed approaches have not demonstrated such a flexibility.

3.3 Program Path Analysis

Program path analysis studies a number of topics including automatic flow analysis for infeasible path detection and loop bounding, path annotation methods, source-code level to compiled-code level flow information translation, interaction with optimization compilers etc. Substantial research work has been done in this area.

Automatic flow analysis. Feasible/infeaisble path information is either manually provided or is explored automatically by flow analysis. The later approach has been investigated by many researchers.

Altenbernd [2] proposed a method to exclude false paths during the search for the worst case execution path. His work is a combination of path enumeration with pruning and symbolic execution. He used branch-and-bound algorithm to perform the actual path search in the control flow graph.

Ermedahl and Gustafsson [20] used symbolic execution to discover false paths and loop bounds. They work on abstract semantics of programs. The key concept is an environment σ_i^h, which captures the abstract values (split integer intervals) of variables at a program point i following a specific path h. Rules updating the environments at program points are generated based on program semantics. If a variable's abstract value in σ_i^h is $\perp$, which means empty value, then the path h to i is an infeasible one.

Lundqvist and Stenström [49] used cycle-level symbolic simulation for WCET calculation as well as infeasible path detection. In their work, the domain of variable values is extended with an extra value called *unknown*. When a conditional branch is reached and the value of the condition variable is not *unknown*, then the execution goes along one path and the execution along the other path is an infeasible one, which is simply not simulated. In case the condition value is an *unknown*, both paths need to be simulated.

Liu and Gomez [47, 48] proposed another technique using symbolic evaluation on partially known input structures. They work on the source-language level. In contrast to the earlier symbolic execution based techniques, they do not merge paths from loops. This reduces nondeterminism due to path merge but raises concerns on time and space complexity. They apply some program language transformations such as incremental computation and transformation of conditionals to make the analysis more efficient. In their experiments they observed that the analysis is still feasible for for inputs sizes in the thousands.

Above symbolic execution based methods need to iterate through the loops many times, which could be inefficient. Healy et al. [25, 26] implemented techniques to automatically determine the minimum/maximum number of iterations for loops. They do so by (1) identifying conditional branches within the loop that can affect the number of of loop iterations, (2) calculating the range of iterations these branches can be reached, and (3) calculating the minimum/maximum number of iterations with the information computed in (2). In another work [27], they developed techniques for automatic detection of branch constraints. They do so by analyzing the effect of a variable assignment on a branch and the correlation between the outcomes of different branches. The fall through or taken frequency of a branch in a loop may also be calculated by using value range analysis on loop induction variables. The branch constraints will be used in the subsequent analyses to exclude infeasible paths.

Ferdinand et al. [21] used abstract interpretation to detect infeasible paths. They call it *value analysis*, which computes for each processor register an interval of possible values. If at a conditional branch, the value interval for the branch condition indicates a deterministic direction, then the path along the other direction is an infeasible one.

Annotation methods. To make use of the feasible/infeasible path information, there should be methods to describe it.

Puschner and Koza [61] proposed a language called MARS-C. They use constructs like *scopes*, *markers*, and *loop sequences* to describe feasible/infeasible paths.

Park [57] developed a script language called IDL (*information description language*), which is subsequently translated into regular expressions. IDL can capture some frequent path relationships such as that a statement is executed a certain number of times, or that two statements are always executed together or they are mutually exclusive. The major problem is that manipulations on regular expressions, e.g., intersection of two regular expressions, are difficult.

Li and Malik [40] used linear constraints to specify the flow information, which they called functional constraints. Functional constraints can be used to give loop bounds, and relationship of execution counts among multiple basic blocks. They have shown that every IDL information clause in [57] can be transformed into functional constraints.

Colin and Puaut [9] proposed an annotating method for loops with variant number of iterations. They used couples of mathematical expressions instead of constants for inner loops whose iteration numbers are dependent on counter variables of outer loops. For example, $[maxiter, counter]$ is such a loop bound, where $maxiter$ is the maximum number of iterations and $counter$ is the loop counter value, both are mathematical expressions. These expressions are symbolically evaluated by Maple [7]. By using this annotating method, they have achieved significant accuracy improvement for programs having inner loops with variant number of iterations.

Engblom and Ermedahl [18] defined a language called *flow facts language* to describe complex flow information. They define flow facts for scopes, which are program segments under some execution context, e.g., a loop or a function call reached from a path. A flow fact consists of three parts: the name of a scope, a context specifier, which typically gives the iterations of the scope, and a constraint expression specifying the flow information. For example, a flow fact *foo: [1..10] :* $X_A \leq 2$ specifies that

a block A in the scope foo cannot execute more than twice in the first ten iterations of the scope. Depending on the WCET calculation method being used, not all flow facts can be accurately transformed to path information that can be used for that WCET calculation.

Translation and compiler support. Program path information is often provided on the source-program level, but WCET analysis is usually on the compiled-code level. Thus a translation of the annotations from source-program level to the compiled-code level is necessary. This is a non-trivial problem because optimizing compilers perform a lot of code transformations, which makes the mapping between source program constructs to instructions/basic blocks in the compiled code difficult.

Puschner [62] described a mapping function to translate path information on the source level to the assembly level. It assumes that the programs are compiled with moderate optimization. The mapping function traverses the parse tree of the source program. In each step down the tree it tries to find the corresponding assembly code by using information about the nesting of constructs, line numbers etc. in the assembly code. If the mapping fails on a construct, it outputs a warning.

Engblom et al. [19] proposed an approach called *co-transformation* for supporting the mapping of execution information from source program to compiled code. They defined a language called *Optimization Description Language* (ODL) to characterize what typical optimizations do. The co-transformation engine can be generated from the ODL source. To apply their work, the compiler needs to be modified slightly to tell the transformer what kind of optimizations have been done. As long as the optimization types performed by the compiler are described by ODL, the co-transformation can map the source code constructs to compiled code segments.

Kirner and Puschner [33, 34] developed another transformation method that is integrated into the compiler. The path information is transformed through all compiler

stages. Therefore substantial effort is needed to extend the existing compiler, but is paid by the ability to supporting strong code optimizations for WCET analysis.

Summary Above discussion covers several issues on program path analysis, which address the problem of providing program path information for WCET calculation from different aspects. More accurate program path information is essential for tight WCET estimates. On the other hand, automated path information derivation techniques and integration with compilers will facilitate WCET analysis and promote its application. In this thesis, we focus on microarchitecture modeling and do not explore new program path analysis methods. The existing techniques can be integrated with our work and this will be part of our future work.

CHAPTER IV

OUT-OF-ORDER PIPELINE ANALYSIS

Our aim in this chapter is to obtain a safe and tight WCET estimate for out-of-order pipelined execution without enumerating possible instruction schedules. Our technique is inspired by an iterative performance analysis technique for real-time distributed systems proposed by Yen and Wolf [74], which estimates the execution time of tasks with data dependencies and resource contentions. For estimating the WCET of a basic block, we exploit and augment their technique by treating individual instructions as tasks. Clearly, there are data dependencies between instructions in a program; resource contention is defined in terms of two instructions requiring the same functional unit. We then extend our solution for estimating the WCET of a basic block to arbitrary programs with complex control flows. This extension involves several steps. First, we apply the timing estimation technique to each basic block. Next, we bound the timing effects of instructions preceding or succeeding a basic block. Finally, Integer Linear Programming (ILP) technique is employed on the control flow graph to estimate the WCET of the entire program.

The rest of this chapter is organized as follows. In the next section we discuss the difficulties of out-of-order pipeline analysis and present an overview of our approach for addressing them. In Section 4.2 we present the analysis technique in two steps: in the first step, we develop the core algorithms for the execution of a basic block without considering its execution context; and in the next step we extend the algorithms to handle the issues related to the execution context of a basic block. In Section 4.3 we experimentally validate the analysis technique. The concluding remarks for out-of-order pipeline analysis appear in Section 4.4.

4.1 Background

4.1.1 Out-of-Order Execution

Modern processors such as the one presented in Section 2.2 employ out-of-order execution where the instructions can be scheduled for execution in an order different from the original program order. In such a processor, an instruction can execute if its operands are ready and the corresponding functional unit is available, irrespective of whether earlier instructions have started execution or not. Out-of-order execution improves processor's performance significantly as it replaces pipeline stalls (due to dependencies and/or resource contentions) with useful computations. However, the out-of-order execution exhibits a phenomenon called *timing anomaly* [1], which makes WCET analysis difficult.

4.1.2 Timing Anomaly

The problem of *timing anomaly* was originally discussed by Lundqvist and Stenström [50]. Let us consider an instruction I with two possible latencies l_{min} and l_{max} such that $l_{max} > l_{min}$. The variation of latency could due to different reasons: cache hit/miss for a load instruction, variable number of cycles taken by an arithmetic instruction like multiplication etc. Let us assume that the execution time of a sequence of instructions containing I is g_{max} (g_{min}) if I incurs a latency of l_{max} (l_{min}). The latencies of the other instructions in the sequence are fixed. A timing anomaly happens if either $(g_{max} - g_{min}) < 0$ or $(g_{max} - g_{min}) > (l_{max} - l_{min})$.

Figure 4.1 illustrates timing anomaly with an example. In the code fragment, instruction B depends on A, instruction C depends on B, and instruction E depends on D. Instructions A and E use the MULTU functional unit with latency of $1 \sim 4$ cycles and the other instructions use the single cycle ALU functional unit.

[1] It has been observed by Langenbach et al. [36] that timing anomaly can also happen to some in-order processors such as Motorola ColdFire 5307 where a unified cache for instruction/data is employed.

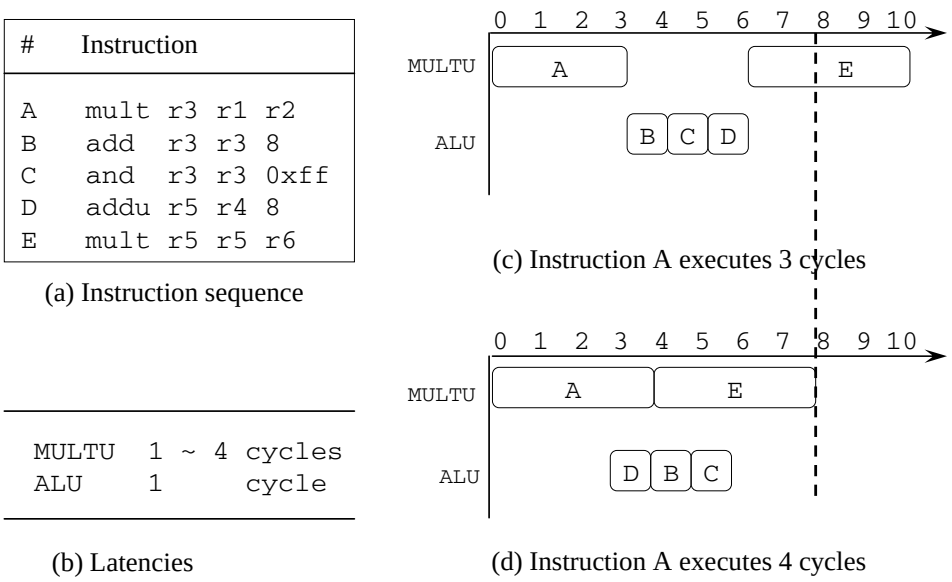


Figure 4.1: Timing Anomaly due to Variable-Latency Instructions

We illustrate two possible execution scenarios. In the first scenario illustrated in Figure 4.1(c), instruction A executes for three cycles – cycles $0-2$. Since A starts executing at cycle 0, it is ready for execution at cycle 0 or earlier. Therefore at the beginning of cycle 3, all of B, C, D are ready for execution; all of them are contending for the ALU. Thus, instructions B and C execute on cycles 3 and 4, respectively. Instruction D is ready for execution in cycle 3 itself, but it can only be scheduled for execution in cycle 5 after B and C (which appear earlier in program order). The overall execution time in this case is 10 cycles. In the second scenario as illustrated in Figure 4.1(d), A executes for four cycles. Now D is the only ready instruction in cycle 3 (B and C are still waiting for their operands); D executes in clock cycle 3 allowing E to start execution in clock cycle 4. The overall execution time in this case is only eight cycles. Thus, *a longer latency of A results in a shorter overall execution time.*

In the presence of timing anomaly, techniques which generally take the local worst case for WCET estimation no longer guarantee safe bounds. For example, it is not

safe to assume that the worst case cache behavior of a sequence of instructions results from a cache miss in all the instructions. For the same reason, it is not safe to assume the longest latency for variable-latency arithmetic instructions will lead to the overall WCET of a program. This prompts the need to consider all possible schedules of instructions. For a piece of code with N instructions and each of which has K possible latencies, a naive approach, which examines each possible schedule individually, will have to consider K^N schedules. We now explain the basic idea behind our approach which allows us to avoid such expensive enumeration.

4.1.3 Overview of the Pipeline Modeling

Given the control flow graph of a program, our WCET analysis method first derives a WCET estimate for each basic block. Then the basic block estimates are combined using Integer Linear Programming (ILP) to produce the program's WCET estimate (refer to Equation 2.1).

How do we find the WCET estimate for a basic block B_i? This is done by first considering the basic block's execution in isolation, that is, starting with an empty pipeline. We find the WCET estimate without enumerating instruction schedules as follows. We observe that the worst-case timing behavior of B_i occurs from maximum resource contention among instructions in B_i, that is, each instruction being delayed by maximum number of other instructions. We produce very coarse estimates for the time interval at which instructions in B_i can start/finish execution by initially assuming that any instruction in B_i can delay the others, except the contentions ruled out by data dependencies. The estimates allow us to rule out certain contentions – if the earliest time instruction I is ready for execution occurs after the latest time at which I' finishes, clearly I cannot delay I'. This allows us to further refine the estimates, thereby ruling out more contentions. The process continues until a fixed point is reached. The WCET of the basic block B_i (where B_i's execution starts with

an empty pipeline) is the maximum time between the fetch of B_i's first instruction and commit of B_i's last instruction.

Given the execution time estimate of B_i's execution starting with an empty pipeline, how do we find $cost_i$, block B_i's WCET estimate? We observe that the number of instructions before and after B_i which can affect the timing of B_i's execution is bounded by architectural parameters. Accordingly, we extend our timing estimation technique to operate on basic block with a prologue/epilogue (instructions before/after B_i which directly affect the timing of B_i). Time intervals for execution of instructions in prologue/epilogue are estimated conservatively by assuming maximum possible contentions. We also consider (a) the data dependencies between instructions in prologue and instructions in B_i, and (b) possible time overlap between instructions in B_i and instructions prior to B_i. In this way, we find the timing estimate of basic block B_i for all possible choices of prologue and epilogues. The maximum of these estimates is $cost_i$, the estimated WCET of B_i.

In the preceding, we have given an overview of our modeling technique which captures the timing effects of out-of-order pipelines. The technical details of this modeling will be presented in the following sections.

4.2 The Analysis

Our analysis technique is presented in two steps. First, we estimate the execution time of a basic block in isolation by assuming an empty pipeline at the beginning. Next, we extend the technique by taking into account the possible initial pipeline states and context instructions before/after the basic block.

4.2.1 Estimation for a Basic Block without Context

Our effort in this section is to develop an algorithm for estimating the WCET of a basic block executing on the out-of-order processor pipeline presented in Section

2.2. Instructions in a basic block are executed sequentially, that is, there is no non-determinism in terms of control flow transfer. The main advantage of our approach is that explicit enumeration of possible instruction schedules is avoided. Thus the estimation is both time and space efficient. The technical details are presented in the following order. First, we formulate the problem as an execution graph, which captures data dependencies and resource contentions — the two major factors dictating instruction executions. Next, based on the execution graph, we develop an algorithm which starts with very coarse yet safe estimates, and iteratively refines the estimates until a fixed point is reached.

Definition 4.1 (**Execution Graph).** *The execution graph for a basic block B under a pipeline model is defined as*

$$G_B = (V_B, DE_B)$$

where V_B represents all possible combination of instruction identifiers and pipeline stages for basic block B, and $DE_B \subseteq V_B \times V_B$ represents a dependency relation among nodes. For two nodes $u, v \in V_B$, we say that $(u, v) \in DE_B$ iff v can start execution only after u has completed execution; this is indicated by a solid directed edge from u to v in the execution graph. Clearly $(u, v) \in DE_B \Rightarrow (v, u) \notin DE_B$.

Apart from the dependency relation among nodes in an execution graph (denoted by solid edges), we also define a contention relation as follows. We do not make the contention relation part of the execution graph so as to clearly identify what we mean by "paths" in the exection graph; paths in the execution graph refer to chains of dependency edges. This will be required in our analysis.

Definition 4.2 (**Contention Relation).** *Let B be a basic block, and $G_B = (V_B, DE_B)$ be its execution graph. We define a contention relation $CE_B \subseteq V_B \times V_B$ such that for two nodes $u, v \in V_B$, we say that $(u, v) \in CE_B$ iff*

- *nodes u and v denote the EX stages of two different instructions I and J respectively, and*

- *instruction I and J can delay each other by contending for a functional unit.*

Our definition of contention relation is symmetric, that is, $(u, v) \in CE_B \Rightarrow (v, u) \in CE_B$. We will often show the contention between u and v as an undirected dashed edge in the execution graph.

We now explain the nodes, dependencies and contentions captured in an execution graph in details. This will also clarify how the dependency and the contention relations can be computed.

Let $Code_B = I_1 \ldots I_n$ represent the sequence of instructions in a basic block B. Then each node $v \in V_B$ is represented by a tuple: an instruction identifier and a pipeline stage denoted as $stage(instruction\_id)$. For example, the node $v = IF(I_i)$ represents the fetch stage of the instruction I_i. If basic block B contains n instructions, then $|V_B| = n \times P$ where P is the number of stages in the pipeline. Each node in the execution graph is associated with the latency of the corresponding pipeline stage. In our processor pipeline, all pipeline stages except EX have single cycle latency.

Our definition of dependency edges includes dependencies due to resource constraints and pipelined execution in addition to traditional data dependencies. We consider:

- Dependencies among pipeline stages of the same instruction. This is because an instruction must proceed from the first stage to the last last stage in order, for example, $ID(I_i)$ must follow $IF(I_i)$.

- Dependencies due to in-order execution in IF, ID, and CM pipeline stages. That is, different instructions should proceed through these pipeline stages in program order, for example, $IF(I_{i+1})$ can only start after $IF(I_i)$.

- Dependencies due to resource constraints as in full I-buffer or ROB. For example, assuming I-buffer has two entries, there will be no entry available for $IF(I_{i+2})$ before the completion of $ID(I_i)$ (which removes I_i from the I-buffer). Therefore, there should be a dependency edge $ID(I_i) \rightarrow IF(I_{i+2})$. Similarly, with a 4-entry ROB, there should be a dependency edge $CM(I_i) \rightarrow ID(I_{i+4})$ because $CM(I_i)$ frees up the entry occupied by I_i in the ROB. Note that we can draw these edges as both the I-buffer and the ROB are allocated and freed in program order.

- Data dependencies among instructions. If instruction I_i produces a result that is used by instruction I_j, then there should be a dependency edge $WB(I_i) \rightarrow EX(I_j)$.

The above summarizes the dependencies; we now describe the contention relation among nodes in the execution graph of a basic block B. We define contention relation CE_B among the EX stages of different instructions utilizing the same FU for execution. This is because contention can only happen in the EX stage with our pipeline model. For two instructions I_i, I_j in basic block B ($i \neq j$) we define $(EX(I_i), EX(I_j)) \in CE_B$ iff

1. instructions I_i and I_j utilize the same functional unit,

2. there is no path from $EX(I_i)$ to $EX(I_j)$ or from $EX(I_j)$ to $EX(I_i)$ in the execution graph G_B, and

3. $|i - j| < ROB\_size$

The second condition ensures that there is no dependency between the two nodes, i.e., they can indeed contend for a functional unit. The final condition simply excludes the possibility of two far-away nodes contending with each other. For example, if the ROB has four entries then clearly instructions I_i and I_{i+4} cannot coexist in the ROB. Note that the contention between two instructions obeys the following rules.

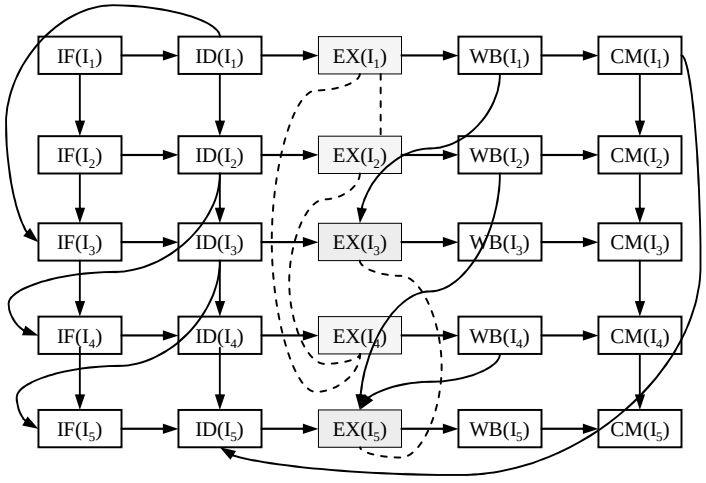
- If two instructions contend for a functional unit in the same clock cycle, the earlier instruction (according to program order) gets access to the functional unit, and
- Once an instruction gets access to a functional unit, it runs to completion without getting pre-empted.

Given two instructions I_i, I_j (where $i < j$, *i.e.* I_i appears earlier in program order) contending for a functional unit, suppose I_j becomes ready earlier than I_i. This is possible since I_i may be delayed due to data dependencies. Instruction I_j thus starts executing ahead of I_i. Meanwhile I_i may receive its operands and get ready. However, I_i now has to wait for the function unit to be free, that is, until I_j completes. This is how instructions later in the program order can delay the execution of an earlier instruction.

Figure 4.2 shows an example of execution graph. This graph is constructed from a basic block with five instructions as shown in Figure 4.2(a). In Figure 4.2(b), the edges $WB(I_1) \rightarrow EX(I_3)$, $WB(I_2) \rightarrow EX(I_5)$, and $WB(I_4) \rightarrow EX(I_5)$ reflect data dependencies. The other solid edges capture dependencies due to the structure of the pipeline and resource constraints. The dashed edges represent contention relations. The contention relation between $EX(I_1)$ and $EX(I_4)$ implies: (a) if instructions I_1 and I_4 are both ready to execute and the functional unit MULTU is free, then $EX(I_1)$ will be issued for execution as it is from an earlier instruction and thus has higher priority; and (b) if $EX(I_4)$ has already started execution before $EX(I_1)$ is ready, then $EX(I_4)$ will be allowed to complete and thereby delay $EX(I_1)$. Our execution graph is similar to the dynamic dependency graph among instructions of Fields et al. [23]. In their work, the dependency graph is obtained from a concrete simulation run, that is, a trace of dynamic instructions. Therefore, the actual resource contentions exercised in that particular run are known and the nodes are annotated with the execution latency as well as the wait time for a functional unit. They study how

```
I1:  mult  r6  r10  4
I2:  mult  r1  r10  r1
I3:  sub   r6  r6   r2
I4:  mult  r4  r8   r4
I5:  add   r1  r1   r4
```

(a) Code Example



(b) Execution Graph of the Code

Figure 4.2: A basic block and its execution graph. The solid edges represent dependencies and the dashed edges represent contention relations.

much each instruction can be delayed (the slack) without increasing the execution time of the run. Our execution graph is static and all possible resource contentions between instructions are represented for the purposes of static analysis.

Problem Definition Let B be a basic block consisting of a sequence of instructions $Code_B = I_1 \ldots I_n$ and let $G_B = (V_B, DE_B)$ be its execution graph. Estimating the WCET of B can be formulated as finding the maximum (latest) completion time of the node $CM(I_n)$ assuming that $IF(I_1)$ starts at time zero. Note that this problem is *not* equivalent to finding the longest path from $IF(I_1)$ to $CM(I_n)$ in the execution graph (taking the maximum latency of each pipeline stage). The execution time of a path in the execution graph is not a summation of the latencies of the individual nodes because of two reasons.

- The total time spent in making the transition from $ID(I_i)$ to $EX(I_i)$ is dependent on the contentions from other ready instructions.
- The initiation time of a node is computed as the *max* of the completion times of its immediate predecessors in the execution graph. This models the effect of dependencies, including data dependencies.

A Related Problem Given the problem formulated as an execution graph, we propose an iterative algorithm to estimate the WCET of a sequence of instructions. The basic structure of our algorithm is inspired by a performance analysis technique for real-time distributed systems [74] which analyzes a system consisting of several periodic tasks represented by task graphs. Each task consists of a partially ordered set of processes, and each process has lower and upper bounds on its computation time. The hardware architecture consists of a set of Processing Elements (PE) connected via communication edges. Processes are allocated to the PEs and priorities are assigned among the processes assigned to the same PE. A process P is scheduled to execute on

a processor E if (1) all of P's predecessors have completed execution, and (2) no higher priority process in running on E. P can possibly preempt a lower priority process to start execution; on the other hand, P may itself get preempted by higher priority processes during its execution. The algorithm estimates the worst case completion time of all the tasks.

The problem addressed by Yen and Wolf's algorithm is similar to our analysis problem in some key aspects. The similarities include the fact that the execution graph in our problem is similar to the task graph considered in [74]; both these graphs capture data dependencies between nodes. Furthermore there are resource contentions between the nodes and contending nodes are assigned priorities. However, there are some significant differences as well. First of all, [74] captures periodic tasks whereas the instructions in our execution graph are not periodic. More importantly, in [74] a higher priority process *hp* may delay a lower priority process *lp* by preemption; but *lp* cannot delay *hp*. However, in our problem, it is possible for a lower priority instruction (appearing later in program order) *li* to delay the execution of a higher priority instruction *hi*. As there is no preemption, if *li* is executing when *hi* becomes ready, then *li* is allowed to complete the execution and it delays the execution of *hi*. Such differences make the computation of the response time of a node v – the time when all of v's predecessors have completed execution to the time v completes execution – different in our problem.

Notations Before we discuss our WCET estimation method, we explain the notations used in our estimation algorithm. In the following, u, v denote nodes in the execution graph of the basic block B being analyzed.

- t_v^{ready}: Ready time of node v is defined as the time when all its predecessors have completed execution.
- t_v^{start}: Start time of node v is defined as the time when it starts execution.

Except for nodes corresponding to EX stages, $t_v^{start} = t_v^{ready}$. A node $EX(I_i)$ may not be able to start execution when it becomes ready if another instruction is using the corresponding functional unit, or some higher priority instructions (earlier than I_i in program order) are also ready. Therefore, $t_v^{start} \geq t_v^{ready}$.

- t_v^{finish}: Finish time of a node v is defined as the time when it completes execution. Pipeline stages other than EX need only one cycle to execute. Therefore, $t_v^{finish} = t_v^{start} + 1$. For EX stage, we add the minimum (maximum) latency of the functional unit to t_v^{start} when we compute its *earliest* (*latest*) finish time.

- $separated[u, v]$: If the executions of the two nodes u and v cannot overlap, then $separated[u, v]$ is assigned to *true*; otherwise, they might overlap and it is assigned to *false*.

- $instr\_id(v)$: The instruction id corresponding to a node v.

- $early\_contenders(v)$: Contending instructions that appear earlier in program order, i.e., the set of nodes u s.t. $(u, v) \in CE_B$ and $instr\_id(u) < instr\_id(v)$. Recall that that CE_B denotes the contention relation among the nodes in the execution graph of basic block B.

- $late\_contenders(v)$: Contending instructions that appear later in program order, i.e., the set of nodes u s.t. $(u, v) \in CE_B$ and $instr\_id(u) > instr\_id(v)$.

- min_lat_v, max_lat_v: Minimum and maximum execution latencies of node v.

Summary of our method As mentioned earlier, our problem is not equivalent to finding the longest path in the execution graph due to resource contentions and dependencies. We account for the timing effects of the dependencies by using a modified longest path algorithm that traverses the nodes in topologically sorted order. This topological traversal ensures that when a node is visited, the completion times of all

its predecessors are known. To model the effect of resource contentions, we conservatively estimate an upper bound on the delay due to contentions for a functional unit by other instructions. A single pass of the modified longest path algorithm computes loose bounds on the lifetime of each node. These bounds are used to identify nodes with disjoint lifetimes. These nodes are not allowed to contend in the next pass of the longest path search to get tighter bounds. These two steps repeat till either there is no change in the bounds or a pre-defined number of iterations have elapsed.

```
1  separated[., .] = false; step = 0;
2  foreach node v ∈ V do
3  |  earliest[t_v^start] := 0; earliest[t_v^finish] := min_lat_v;
   |  latest[t_v^start] := ∞; latest[t_v^finish] := ∞;
4  repeat
5  |  LatestTimes(G); EarliestTimes(G);
6  |  foreach u, v ∈ V do
7  |  |  if earliest[t_v^ready] ≥ latest[t_u^finish] then
   |  |  |  separated[u, v] = true;
8  |  |  if earliest[t_u^ready] ≥ latest[t_v^finish] then
   |  |  |  separated[u, v] = true;
9  |  step := step + 1;
   until separated[., .] are unchanged or step > limit;
10 WCET = latest[t_CM(I_n)^finish]; /* I_n is the last instruction of the basic block */
```

Algorithm 1: WCET Estimation for Execution Graph $G = (V, DE)$

Estimation Algorithm Algorithm 1 gives the outline for computing the WCET given an execution graph $G = (V, DE)$ corresponding to a basic block. The top level algorithm iteratively performs two operations: timing bounds computation and separations analysis. The first operation is done by *LatestTimes* and *EarliestTimes*, which compute the upper and lower timing bounds of the nodes. The second operation is done by re-assigning the values of $separated[u, v]$ for all nodes u, v. Basically, we find out pairs of nodes (u, v) whose lifetimes are guaranteed to not overlap; for these nodes we set $separated[u, v]$ to true. How do we find out pairs of nodes with

```
1  latest[t^{ready}_{IF(I_1)}] := 0;   /* I_1 is the first instruction of the basic block */
2  foreach node v ∈ V in topologically sorted order do
3      latest[t^{start}_v] := latest[t^{ready}_v];
4      S_late := late_contenders(v) ⋂ {u | ¬separated[u, v] ∧ earliest[t^{start}_u] <
       latest[t^{ready}_v]};
5      if S_late ≠ φ then
6          latest[t^{start}_v] :=
           min ( max_{u∈S_late} ( latest[t^{finish}_u] ), latest[t^{ready}_v] + max_lat_v − 1 );
7      S_early := early_contenders(v) ⋂ {u | ¬separated[u, v]} ;
8      if S_early ≠ φ then
9          tmp :=
           min ( max_{u∈S_early} ( latest[t^{finish}_u] ), latest[t^{start}_v] + |S_early| × max_lat_v );
10         latest[t^{start}_v] := max ( tmp, latest[t^{start}_v] );
11     latest[t^{finish}_v] := latest[t^{start}_v] + max_lat_v ;
12     foreach immediate successor w of v do
13         latest[t^{ready}_w] = max(latest[t^{ready}_w], latest[t^{finish}_v]);
```

Algorithm 2: LatestTimes($G = (V, DE)$)

```
1  earliest[t^{ready}_{IF(I_1)}] := 0;   /* I_1 is the first instruction of the basic block */
2  foreach node v ∈ V in topologically sorted order do
3      earliest[t^{start}_v] := earliest[t^{ready}_v];
4      S_late := late_contenders(v) ⋂ {u | ¬separated[u, v] ∧ latest[t^{start}_u] <
       earliest[t^{ready}_v] < earliest[t^{finish}_u]};
5      S_early := early_contenders(v) ⋂ {u | ¬separated[u, v] ∧ latest[t^{start}_u] ≤
       earliest[t^{ready}_v] < earliest[t^{finish}_u]};
6      S := S_late ⋃ S_early;
7      if S ≠ φ then
8          earliest[t^{start}_v] := max ( max_{u∈S} ( earliest[t^{finish}_u] ), earliest[t^{ready}_v] );
9      earliest[t^{finish}_v] := earliest[t^{start}_v] + min_lat_v ;
10     foreach immediate successor w of v do
11         earliest[t^{ready}_w] = max(earliest[t^{ready}_w], earliest[t^{finish}_v]);
```

Algorithm 3: EarliestTimes($G = (V, DE)$)

non-overlapping lifetimes? In our problem, given two nodes u and v in the execution graph, we simply set $separated[u, v]$ to true if $earliest[t_u^{ready}] \geq latest[t_v^{finish}]$ or $earliest[t_v^{ready}] \geq latest[t_u^{finish}]$.[2] Thus, the tighter the time intervals obtained, the more are the pairs of nodes that can be identified as separated. On the other hand, the more the number of separated pairs identified, the tighter are the timing intervals computed in subsequent iterations due to lesser number of competing nodes.

Algorithm 2 computes the latest ready, start, and finish times for each node of the execution graph. The latest start time of node v, denoted as $latest[t_v^{start}]$, is computed according to (a) its latest ready time $latest[t_v^{ready}]$ (which is obtained from the latest finish times of its predecessors), and (b) its contenders. We first consider the delay of v's start time by contenders later in program order. Note that the start time of node v can be delayed by *at most one* late contender. Obviously, a late contender $u \in late\_contenders(v)$ cannot delay v after v is ready (since v has higher priority). Therefore, late contenders who do not satisfy the condition $earliest[t_u^{start}] < latest[t_v^{ready}]$ are excluded. We also exclude the contenders who have been identified to be separated from v (*i.e.*, whose lifetimes cannot overlap with v). The delay from a late contender u is bounded by u's latest finish time $latest[t_u^{finish}]$. In addition, u cannot delay v by more than its maximum latency; thus, we have another bound $latest[t_v^{ready}] + max\_lat_v - 1$ where $max\_lat_u = max\_lat_v$ is the maximum latency of the contended functional unit. The minimum of the two bounds is taken.

Apart from the delay due to late contenders of node v, we also need to estimate the delay in v's start time due to its early contenders. Note that the early contenders appear before v in program order. So in the worst case, all of them, except those proved to be separated from v (*i.e.*, not overlapping with v's lifetime), can contend with v and delay its start time. This is captured on Lines 7–10 of Algorithm 2. First,

[2]There exist more sophisticated techniques for finding nodes with disjoint lifetimes in a graph *e.g.* see [52]. In our experiments we found that our simplified approach for identifying separated nodes substantially increases the efficiency of our WCET analysis.

it is obvious that the delay due to early contention cannot be beyond the time when all contenders have completed execution, so

$$t_v^{start} \leq \max_{u \in S_{early}} \left(latest[t_u^{finish}] \right)$$

On the other hand, the maximum delay is also bounded by $|S_{early}| \times max\_lat_x$ where each early contender executes for its maximum latency.

The latest finish time of v is obtained by simply adding the maximum latency of the functional unit to $latest[t_v^{start}]$ (Line 11). This is because an instruction cannot get preempted once it has started execution on a functional unit. The immediate successors of v get their latest ready times updated if v's latest finish time is higher than the current approximation of their latest ready times (Lines 12–13). In this way the *LatestTimes* algorithm estimates the latest ready/start/finish times of each node in the execution graph.

Similar to the algorithm *LatestTimes*, the *EarliestTimes* algorithm (see Algorithm 3) computes the earliest ready, start, and finish times of all nodes in the execution graph. The main difference is that we allow a node u to contend and thereby delay the earliest start time of a node v only if the contention can be *guaranteed*. A formal proof for the correctness of the algorithms is given in Appendix A.1.

4.2.2 Estimation for a Basic Block with Context

In the last section, our technique for estimating the WCET of a basic block B_i is based on the simplifying assumptions that execution of instructions outside B_i does not interact with B_i's execution and the initial pipeline state is empty. This is, however, an unrealistic assumption. In this section, we extend our technique to consider the instructions preceding and succeeding B_i.

The execution context of a basic block B_i is defined in terms of the instructions that directly affect the timing of B_i's execution. To model the execution time of a basic block B_i, we need to consider (1) contentions and data dependencies among

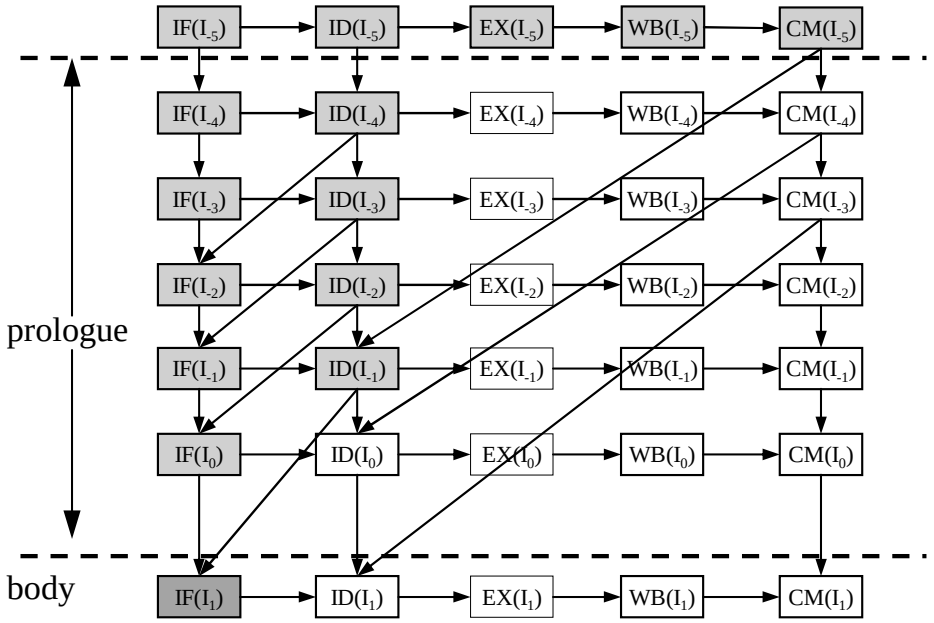


Figure 4.3: An Example Prologue

instructions prior to B_i and instructions in B_i, and (2) contentions between instructions in B_i and instructions after B_i[3]. The instructions before (after) a basic block B_i that *directly* affect the execution time of B_i constitute the contexts of B_i and are called the **prologue** (**epilogue**) of B_i. For example, assuming a 2-entry I-buffer and a 4-entry ROB, at most (4+2)-1 = 5 instructions can be in the pipeline when B_i enters the pipeline. Similarly, due to the 4-entry ROB, at most 4-1=3 instructions after B_i can contend with instructions in B_i. Of course, a basic block B_i may have multiple prologues and epilogues corresponding to the different paths along which B_i can be entered or exited. To capture the effects of contexts, our analysis constructs execution graphs corresponding to all possible combinations of prologues and epilogues. *Each execution graph consists of three parts: the prologue, the basic block itself (called the* **body***) and the epilogue.*

[3]Here, we only consider contentions but not dependencies because data dependencies between B_i and instructions after B_i cannot affect the execution time of B_i.

/* I_1 is the first instruction in the basic block $latest[t^{ready}_{IF(I_1)}] := 0$ */;

1 **foreach** *node* $v \in prologue$ **do**

2 $shaded[v] := false$;

3 **if** $paths(v, IF(I_1)) \neq \phi$ **then**

4 $shaded[v] := true$;

5 $latest[t^{finish}_v] := -\max_{\pi \in paths(v, IF(I_1))} \sum_{x \in nodes(\pi)} min\_lat_x$; /* Equation 4.2 */

6 $latest[t^{start}_v] := latest[t^{finish}_v] - min\_lat_v$; $latest[t^{ready}_u] := latest[t^{start}_u]$;

/* I_{-p} is the instruction just before the prologue */

7 $latest[t^{ready}_{CM(I_{-p})}] := -\max_{\pi \in paths(CM(I_{-p}), IF(I_1))} \sum_{x \in nodes(\pi)} min\_lat_x - 1$;

8 **foreach** *node* $v \in prologue$ *in topologically sorted order where* $shaded[v] = false$ **do**

9 $latest[t^{ready}_v] := \max_{\{u | u \to v\}} \left(latest[t^{finish}_u] \right)$;

10 $latest[t^{ready}_v] := \max \left(latest[t^{ready}_v], latest[t^{ready}_{CM(I_{-p})}] \right)$;

11 $latest[t^{start}_v] := latest[t^{ready}_v] + max\_lat_v - 1$; /* conservative late contention */

12 $S_{early} := early\_contenders(v)$;

13 **if** $S_{early} \neq \phi$ **then**

14 $tmp :=$ $\min \left(\max_{u \in S_{early}} \left(latest[t^{finish}_u] \right), \ latest[t^{start}_v] + |S_{early}| \times max\_lat_v \right)$;

15 $latest[t^{start}_v] := \max \left(tmp, latest[t^{start}_v] \right)$;

16 $latest[t^{finish}_v] := latest[t^{start}_v] + max\_lat_v$;

Algorithm 4: Estimation of latest times of prologue nodes

Time Intervals for Prologue Nodes Figure 4.3 shows a prologue with 5 instructions preceding the body. We need to estimate the time intervals of the start/ready/finish of prologue nodes in order to compute their effects on body nodes. As the execution context of the prologue itself is not clear, we conservatively estimate the time intervals as follows. We set the ready time of $IF(I_1)$ to 0 and then we derive the time intervals of the nodes in prologue with respect to the ready time of $IF(I_1)$. Algorithm 4 shows the computation of latest ready, start, and finish times of the nodes in the prologue. First, we observe that certain nodes in prologue (shaded in Figure 4.3) have at least one path to the node $IF(I_1)$ where I_1 is the first instruction in the *body*, that is, the basic block being analyzed. The latest finish time of a shaded prologue node is clearly bounded by $latest[t^{ready}_{IF(I_1)}] = 0$. Let u be a node in prologue with a path to $IF(I_1)$. Consider any path π connecting v and $IF(I_1)$, and let $nodes(\pi)$ be the nodes in π appearing between v and $IF(I_1)$. Clearly

$$latest[t^{finish}_v] \leq latest[t^{ready}_{IF(I_1)}] - \sum_{x \in nodes(\pi)} min\_lat_x \tag{4.1}$$

where min_lat_x is the minimum latency of node x. That is, the finish time of shaded prologue node v cannot be later than the right-hand-side expression in Inequality 4.1 even assuming an ideal execution where each node along the path from v to $IF(I_1)$ (a) becomes ready immediately at the completion of execution of its predecessor, (b) starts execution as soon as it becomes ready (i.e., there is no delay due to contention) and (c) executes as fast as possible by taking the minimum latency. Clearly, Inequality 4.1 holds for all paths between v and $IF(I_1)$. Therefore, for any shaded prologue node v (*i.e.* a node with a path to $IF(I_1)$) we can estimate the latest finish time of v as

$$latest[t^{finish}_v] \leq max_{\pi \in paths(v, IF(I_1))} \left(latest[t^{ready}_{IF(I_1)}] - \sum_{x \in nodes(\pi)} min\_lat_x \right) \tag{4.2}$$

where $paths\,(v, IF(I_1))$ is the set of paths between v and $IF(I_1)$ in the execution graph with prologue/epilogue. Since we compute the time intervals for prologue

nodes relative to ready time of $IF(I_1)$ we can set $latest[t^{ready}_{IF(I_1)}] = 0$ in Inequality 4.2; this is shown on Line 5 of Algorithm 4. In this way we compute the latest finish times of prologue nodes which have a path to $IF(I_1)$. Given the latest finish times, it is straightforward to estimate the latest start and ready times of these nodes (Line 6 of Algorithm 4).

For the rest of prologue nodes (unshaded nodes in Figure 4.3), the latest time calculation is similar to Algorithm 2 with some modifications (see Lines 8–16 of Algorithm 4). First, the processing of the nodes proceed in topologically sorted order. Thus, each of the unshaded nodes, when visited, has at least one predecessor node whose latest finish time has already been computed. Ready time of an unshaded node is estimated as the maximum of the finish times of its immediate predecessors (Line 9 of Algorithm 4). However, we still have not accounted for the immediate predecessors that belong to the pre-prologue part. This effect is conservatively estimated on Line 10 of Algorithm 4. We observe that all pre-prologue nodes should have completed execution by the time the commit stage of the last pre-prologue instruction ($CM(I_{-p})$ where p is the length of the prologue) is ready. Since $CM(I_{-p})$ has a path to $IF(I_1)$, its latest ready time can be computed easily (Line 7 of Algorithm 4). We bound the ready time of the unshaded prologue nodes by the ready time of $CM(I_{-p})$ to take care of the dependencies from the pre-prologue nodes. Latest start time of an unshaded prologue node is estimated conservatively from the latest ready time by taking into account the effect of contentions. First, we conservatively assume that late contention is always present. By definition, at most one late contender can delay an instruction. For early contenders, we do not need to look beyond the prologue as (1) all the pre-prologue nodes have completed execution by the ready time of the node $CM(I_{-p})$ and (2) the ready time of the prologue nodes have been bounded by the ready time of $CM(I_{-p})$ on Line 10. The maximum delay due to early contenders is estimated in a manner similar to Algorithm 2 (Lines 13–15 of Algorithm 4).

Earliest times of prologue nodes do not affect the WCET estimation significantly. Therefore, we conservatively assume earliest ready, start, and finish times of the prologue nodes as $-\infty$.

Time intervals for epilogue nodes Time intervals for epilogue nodes are initialized and iteratively tightened almost the same way as Algorithms 2 and 3 except for one difference: for the EX epilogue nodes which are from the last $ROB\_size - 1$ instructions, they may have late contenders beyond the epilogue, therefore we conservatively assume maximum late contentions for each of them when latest times are estimated.

Time intervals for body nodes Given the time intervals for prologue and epilogue nodes, the timing estimation of body nodes (i.e., the nodes in the basic block we are analyzing) still follows Algorithms 2 and 3. The only difference is that the dependencies and contention from the prologue nodes and late contentions from the epilogue nodes are taken into account in the estimation process.

Overlapped execution For a basic block B_i with instructions $I_1, \ldots, I_n$ the execution time estimate of B_i can be calculated as the time between the fetch of I_1 to the commit of I_n, that is, $t^{finish}_{CM(I_n)} - t^{ready}_{IF(I_1)}$. However, this definition does not produce tight timing estimates. This is because the execution of two or more successive basic blocks have some overlap due to the presence of the pipeline.

Definition 4.3. *The overlap δ between a basic block B_i and its preceding basic block B_j is the period during which instructions from both the basic blocks are in the pipeline, that is*

$$\delta = t^{finish}_{CM(I_0)} - t^{ready}_{IF(I_1)} \tag{4.3}$$

where I_0 is the last instruction of block B_j and I_1 is the first instruction of block B_i.

We want to avoid duplicating the overlap in time estimates of successive basic blocks. Therefore, we calculate the execution time estimate of a basic block with a given context as follows.

Definition 4.4. *For a basic block B_i with instructions $I_1, \ldots, I_n$ executed under a context (prologue and epilogue) ctx, its estimated execution time, denoted as $cost_i^{ctx}$, is the interval from the time when the instruction immediately preceding the basic block has finished commit to the time when its last instruction has finished commit, that is*

$$cost_i^{ctx} = t_{CM(I_n)}^{finish} - t_{CM(I_0)}^{finish} \tag{4.4}$$

where I_0 is the instruction immediately prior to B_i.

Note that the first basic block of the program does not have any preceding instructions. As a special case, we calculate its execution time as the time between the fetch of its first instruction and commit of its last instruction.

Now, we estimate $cost_i^{ctx}$ for basic block B_i with respect to the time at which the first instruction I_1 of B_i is fetched, *i.e.* $t_{IF(I_1)}^{ready} = 0$. Thus $cost_i^{ctx} = t_{CM(I_n)}^{finish} - \delta$. We can conservatively estimate $cost_i^{ctx}$ by finding the largest value of $t_{CM(I_n)}^{finish}$ and the smallest value of δ. The largest value of $t_{CM(I_n)}^{finish}$ is simply the quantity $latest[t_{CM(I_n)}^{finish}]$, calculated by our *LatestTimes* algorithm. The smallest value of the overlap δ is obtained from the following theorem.

Theorem 4.1.

$$\delta \geq \min_{\substack{u \\ u \to IF(I_1)}} \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \tag{4.5}$$

Proof. Let u be the node among $IF(I_1)$'s immediate predecessors with the longest (maximum) finish time. Then,

$$t_{IF(I_1)}^{ready} = t_u^{finish} \tag{4.6}$$

Clearly,

$$t^{ready}_{CM(I_0)} \geq t^{finish}_{u} + \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) \tag{4.7}$$

This is because $CM(I_0)$ can become ready only *after* its predecessors along the paths from u have executed. Therefore,

$$t^{finish}_{CM(I_0)} \geq t^{finish}_{u} + \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \tag{4.8}$$

From Equations (4.6) and (4.8), we get:

$$t^{finish}_{CM(I_0)} - t^{ready}_{IF(I_1)} \geq \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \tag{4.9}$$

By the definition of overlap, the above Equation can be re-written as

$$\begin{aligned} \delta &\geq \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \\ &\geq \min_{\substack{u \\ u \rightarrow IF(I_1)}} \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \end{aligned} \tag{4.10}$$

□

Above we have proved that the overlap is lower-bounded by the right hand side of Inequality 4.5, which will be used as the estimated minimum overlap. The complete proof for the correctness of the estimation for a basic block with context can be found in Appendix A.2.

Putting it all together Note that $cost^{ctx}_i$ is obtained *for a specific prologue and a specific epilogue of* B_i. Since a basic block in general has multiple choices of prologues and epilogues, they might result in different estimates. So, we estimate B_i's execution time under all possible combinations of prologues and epilogues, denoted as CTX_i, and $cost_i = \max_{ctx \in CTX_i} (cost^{ctx}_i)$, where $cost_i$ is the WCET of B_i used in the WCET

objective function given by Equation 2.1,

$$Time = \sum_{i=1}^{N} cost_i * v_i$$

This objective function is maximized over the constraints on v_i given by control flow equations, loop bounds and user-provided infeasible flow information. This is done by using an Integer Linear Programming solver like CPLEX.

4.3 Experimental Evaluation

In this section, we evaluate the accuracy of our estimation technique with twelve benchmarks listed in Table 2.1. Most of them contain variable-latency arithmetic instructions; few exceptions are `des`, `isort` and `matsum`, which do not contain any variable-latency arithmetic instructions.

The pipeline configuration for our experiments is as follows. It has a 4-entry I-buffer and an 8-entry ROB and it contains the following variable latency functional unit types: (a) an integer multiplication unit with $1 \sim 4$ cycle latency, (b) a floating point add unit with $1 \sim 2$ cycle latency, and (c) a floating point multiplication unit with $1 \sim 12$ cycle latency. In addition, the processor has an integer ALU unit and a load/store unit, each with one cycle latency. Note that we assume single-cycle latency for load/store unit because we have not modeled data cache. Since instruction caching and branch prediction has not been modeled so far, we simply assume every instruction fetch takes a single clock cycle and every branch instruction is correctly predicted, e.g., there is no pipeline stall caused by the two events.

The chart in Figure 4.3 compares in-order execution with out-of-order execution. For each benchmark, we give two bars. The first bar (lighter shade in Figure 4.3) denotes the following ratio

$$(Obs\_WCET_{in-order} - Obs\_WCET_{out-of-order}) \ / \ Obs\_WCET_{out-of-order}$$

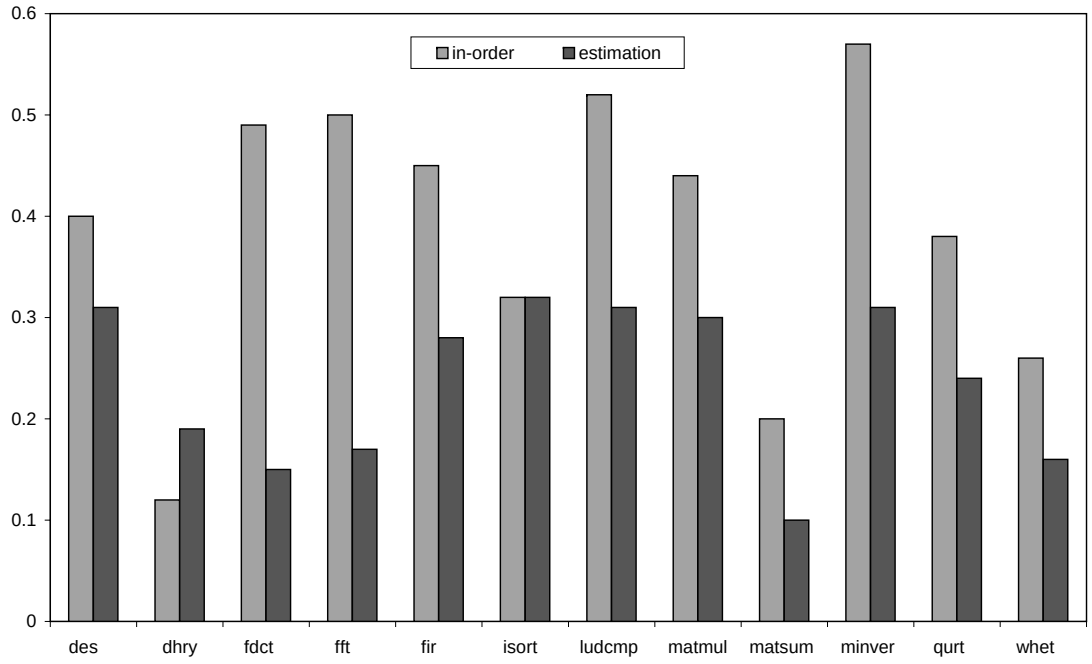


Figure 4.4: Increase of In-order Execution over Out-of-Order Execution and Overestimation for Out-of-Order Execution

where $Obs\_WCET_{in-order}$ ($Obs\_WCET_{out-of-order}$) is the observed WCET for in-order (out-of-order) pipelined execution. The second bar (darker shade in Figure 4.3) denotes the overestimation due to our analysis method for out-of-order execution, that is, the ratio

$$(Est\_WCET_{out-of-order} - Obs\_WCET_{out-of-order}) \ / \ Obs\_WCET_{out-of-order}$$

where $Est\_WCET_{out-of-order}$ is the estimated WCET produced by our analysis method. Why do we compare these two sets of bars for each benchmark? This is to investigate whether our out-of-order pipeline analysis can be replaced by a a simple, but pessimistic WCET analysis which considers in-order execution. Clearly, such an analysis would produce safe WCET estimates, we are trying to find out how tight such estimates will be. In Figure 4.3 we show that even the observed WCET for in-order execution $Obs\_WCET_{in-order}$ is appreciably larger than $Est\_WCET_{out-of-order}$, the estimated WCET produced by our method. Clearly, any analysis developed for in-order

Program	Obs. WCET	Est. WCET	Ratio	Analysis Time(sec.)	Solving Time(sec.)
des	52181	68218	1.31	0.76	0.01
dhry	121018	143633	1.19	2.35	0.01
fdct	9131	10503	1.15	0.12	0.01
fft	1087963	1268466	1.17	0.27	0.01
fir	43958	56104	1.28	0.79	0.01
isort	45763	60507	1.32	0.09	0.01
ludcmp	10682	14013	1.31	0.34	0.01
matmul	14181	18398	1.30	0.04	0.01
matsum	100813	111111	1.10	0.04	0.01
minver	6527	8550	1.31	0.99	0.01
qurt	1769	2200	1.24	0.72	0.01
whet	890104	1031485	1.16	0.96	0.01

Table 4.1: Accuracy of Out-of-Order Pipeline Analysis

pipelines will produce an estimated WCET $Est\_WCET_{in-order} > Obs\_WCET_{in-order}$. Thus, $Est\_WCET_{in-order}$, the estimation produced a simplified analysis of in-order pipelines, will be substantially larger than our estimated WCET $Est\_WCET_{out-of-order}$. This serves as an experimental validation of the need for an analysis method like ours.

Table 4.1 presents the observed WCET (column *Obs. WCET*) and the estimated WCET (column *Est. WCET*), as well as the ratio of the estimated WCET to the observed WCET. The estimated WCET is not far from the observed WCET for most benchmarks specially considering the fact that the difference between actual and observed WCET is unknown. There are mainly two reasons for the overestimation. (1) The bounds on execution counts of basic blocks in the estimation are often higher than the actual execution counts during simulation (overestimation from program path analysis). (2) The WCET estimation algorithm for the basic blocks introduces some amount of pessimism (overestimation from pipeline analysis). The pipeline analysis time and ILP solving time (counted in seconds) for the benchmarks are given by the last two columns. As we can see, both the pipeline analysis and the ILP solving take very little time.

4.4 Summary

Timing anomalies appearing in out-of-order processors complicate Worst Case Execution Time (WCET) analysis by invalidating the assumption that local worst case always lead to global worst case. On the other hand, an exhaustive enumeration of all possible local cases is anticipated to be quite inefficient. In this chapter, we have modeled an out-of-order processor pipeline for WCET analysis. The key idea behind our approach is to avoid exhaustive enumeration by bounding the time intervals at which the events in pipelined execution can occur. We have implemented our technique and experimentally validated its estimation accuracy against several standard benchmark programs used by other WCET research groups.

CHAPTER V

BRANCH PREDICTION ANALYSIS

In this chapter, we study another popular microarchitectural feature: *branch prediction*. Branch prediction is used to address control hazards [30] on pipelined processors. If a prediction is correct, the corresponding control hazard is overcome, otherwise a misprediction penalty is incurred. Apart from misprediction penalties, branch prediction also exerts indirect effects on the performance of other microarchitectural features, such as instruction cache. As the processor caches instructions along the mispredicted path, the instruction cache content is modified by the time the branch is resolved. This prefetching of instructions can have both constructive and destructive effects on cache performance and hence on WCET.

Clearly, we cannot assume perfect branch prediction for the purposes of WCET analysis. This assumption may result in an incorrect WCET (i.e., lower than the actual value), particularly, when a hard-to-predict conditional statement (if-then-else) is present inside a loop body and contributes substantially to a program's WCET. Alternatively, certain works assume that all branches in a program are mispredicted. This pessimism results in significant overestimation of the WCET as branch prediction accuracy is quite high for loop control branches.

Our effort in this chapter is to develop techniques to bound the occurrences of mispredictions and to model its interactions with an instruction cache. To dedicate to this task, we do not consider pipeline effects here. Thus all the variations on the execution times of basic blocks are merely caused by mispredictions and cache misses. The integration with pipeline modeling will be discussed in the next chapter.

We propose an Integer Linear Programming (ILP) based framework to model

branch prediction as well as its interaction with the instruction cache. We use ILP because the global nature of the behavior of branch prediction requires global program path information, which can be provided by the ILP based WCET calculation method used by us. Our branch prediction modeling is generic and parameterizable with respect to the currently used branch prediction schemes. Effects of branch misprediction on cache performance are integrated into our framework by extending previous work on instruction cache modeling [43]. Based on the branch prediction scheme and cache organization, our modeling derives linear constraints from the control flow graph of a program. These constraints are fed to an ILP solver for computing an upper bound on the program's execution time.

The rest of this chapter is organized as follows. In the next section we study dynamic branch prediction mechanisms for our modeling purpose. Then we present the modeling technique in Section 5.1. In Section 5.2 we show the combined analysis of branch prediction and instruction caching. In Section 5.3 we show by experimentation that our technique yields tight estimates. We conclude this chapter in Section 5.4.

5.1 Modeling Branch Prediction

In this section, we discuss the modeling of dynamic branch prediction schemes for WCET analysis. Recall that dynamic schemes make predictions according to the execution history. They commonly use a branch prediction table to store past branch outcomes and make predictions according to the stored information. They differ in the ways the prediction table being indexed. For the GAg scheme, a shift register called branch history register (BHR) which stores the the outcomes of n most recent branches is used as the index to the prediction table; the entry indexed by the BHR will provide the prediction and will be updated by the outcome of current branch. For the local scheme, the prediction table is indexed by the n lower order bits of the branch address. Some other schemes such as gshare and gselect use a combination of

the BHR and the address of the branch as index to the prediction table (details were given in Section 2.1.2). Our technique to be presented can model all above mentioned dynamic schemes.

5.1.1 The Technique

Issues in modeling branch prediction We proceed to examine the difficulties in modeling branch prediction for worst case execution time analysis. So far, microarchitectural features such as pipelining and instruction caching have been modeled for WCET analysis. In the presence of these features, the execution time of an instruction may depend on the past execution trace. For pipelining, these dependencies are typically local. That is, the execution time of an instruction may depend only on the past few instructions which are still in the pipeline. To model instruction caching and branch prediction, *global analysis* is required. This is because the effect of an instruction's execution on caches and branch predictors could affect the execution of remote instructions. However, there are two significant differences between the global analysis of the instruction caching and of branch prediction.

Both instruction caching and branch prediction maintain global data structures that record information about the past execution trace, namely the cache and the branch prediction table. For instruction caching, a given instruction can reside only in one row of the cache: if it is present, it is a cache hit; otherwise, it is a cache miss[1]. Local branch prediction is quite similar – outcomes of a given branch instruction are stored only in one fixed entry of the prediction table where predictions are made. However, for global branch prediction schemes, a given branch instruction may use different entries of the prediction table at different points of execution. Given a branch instruction I, a global branch prediction scheme uses the history $\mathcal{H}_I$ (which is the outcome of the last few branches before arriving at I) to decide the prediction table

[1]To be precise, in associative caches, an address can be present in only one cache *set*.

entry. Because it is possible to arrive at I with various histories, the prediction for I can use different entries of the prediction table at different points of execution.

The other difference between instruction caching and branch prediction modeling is obvious. In the case of instruction caching, if two instructions I and I' are competing for the same cache entry, then the flow of control either from I to I' or from I' to I will always cause a cache miss. However, for branch prediction, even if two branch instructions I and I' map to the same entry in the prediction table, the flow of control between them does not imply correct or incorrect prediction. Their competition for the same entry may have constructive or destructive effect in terms of branch prediction, depending on the outcome of the branches I and I'.

For ease of discussion, we take *GAg*, a global branch prediction scheme described in Section 2.1.2, as a modeling example. However, our modeling is generic and not restricted to *GAg* (as will be shown in Section 5.1.3). In fact, the default scheme in our experiments is the more popular *gshare* scheme.

Control Flow Graph (CFG) The starting point of our analysis is the control flow graph of the program, from which we can derive program flow constraints, as described in Section 2.3.1.

Defining WCET In Section 2.3.1, the WCET is defined as Equation 2.1 under the assumption that the cost of a basic block is a constant. Now in the presence of branch prediction, the cost of a basic block under a misprediction is higher than its cost under a correct prediction. Thus the WCET should be modified to reflect this difference. Suppose the cost of B_i under the correct prediction is $cost_i$ and a misprediction penalty is bmp, then the cost under the misprediction is $cost_i + bmp$. Let bm_i be the misprediction count of B_i (thus B_i is correctly predicted $v_i - bm_i$

times). The objective function for the total execution time of the program becomes:

$$Time = \sum_{i=1}^{N} (cost_i * (v_i - bm_i) + (cost_i + bmp) * bm_i)$$

The first term is the sum of execution times under correct predictions and the second term is the sum of execution times under mispredictions. The WCET objective function can be written as the following form.

$$Time = \sum_{i=1}^{N} (cost_i * v_i + bmp * bm_i) \tag{5.1}$$

To find the worst case execution time, we need to maximize the above objective function. For this purpose, we need to derive constraints on bm_i.

Introducing History Patterns To predict the direction of the branch in B_i, first, the index into the prediction table is computed. In the case of *GAg*, this index is the outcome of the last k branches before B_i is executed and recorded in the Branch History Register (BHR) with k bits. Thus, if $k = 2$ and the last two branches are taken (1) followed by not taken (0), then the index will be 10. We define annotated execution counts and misprediction counts v_i^π and bm_i^π, corresponding to the execution of B_i with BHR $= \pi$ when B_i is reached. Similarly, $e_{i\to j}^\pi$ denotes the number of times the edge $e_{i\to j}$ is passed with $BHR = \pi$ at the beginning of basic block B_i

$$bm_i^\pi \le v_i^\pi;\ \ e_{i\to j} = \textstyle\sum_\pi e_{i\to j}^\pi;\ \ bm_i = \textstyle\sum_\pi bm_i^\pi;\ \ v_i = \textstyle\sum_\pi v_i^\pi.$$

For each B_i and history π, we find out whether it is possible to reach B_i with history π. This information can be obtained via a terminating least fixed point analysis on the control flow graph. Clearly, if it is not possible to reach B_i with π, then $e_{i\to j}^\pi = v_i^\pi = bm_i^\pi = 0$.

Control flow among history patterns To provide an upper bound on bm_i^π, we first define constraints on v_i^π (since $bm_i^\pi \le v_i^\pi$). This is done by modeling the change in history along the control flow graph.

Definition 5.1. *Let $label(i \rightarrow j)$ be an annotation on an edge $i \rightarrow j$ of the CFG, which is given a value according to the following rules:*

$$\begin{aligned} label(i \rightarrow j) = \quad & U \text{ if } i \rightarrow j \quad \text{implies unconditional flow} \\ & 1 \text{ if } i \rightarrow j \quad \text{implies branch at } i \text{ is taken} \\ & 0 \text{ if } i \rightarrow j \quad \text{implies branch at } i \text{ is non-taken} \end{aligned}$$

Definition 5.2. *Let π be a history pattern with k bits (the width of the Branch History Register) at B_i. It is composed of the sequence of outcomes of the most recent k branches with the latest outcome at the rightmost bit. The change in history pattern along $i \rightarrow j$ is given by:*

$$\begin{aligned} \Gamma(\pi, i \rightarrow j) = \quad & \pi \quad && \text{if } label(i \rightarrow j) = U \\ & left(\pi, 0) \quad && \text{if } label(i \rightarrow j) = 0 \\ & left(\pi, 1) \quad && \text{if } label(i \rightarrow j) = 1 \end{aligned}$$

where $left(\pi, 0)$ ($left(\pi, 1)$) shifts pattern π to the left by one bit (the old leftmost bit is therefore discarded) and puts 0 (1) as the rightmost bit.

Now, B_i can execute with history π only if there exists B_j executing with history π' such that $\Gamma(\pi', j \rightarrow i) = \pi$. Note that for any such incoming edge $j \rightarrow i$, there can be two history patterns π' such that $\Gamma(\pi', j \rightarrow i) = \pi$. For example, if $label(j \rightarrow i) = 1$, then $\Gamma(011, j \rightarrow i) = \Gamma(111, j \rightarrow i) = 111$. Therefore, from the inflows of B_i's execution with history π we get:

$$v_i^{\pi} = \sum_{j} \sum_{\pi \,=\, \Gamma(\pi', j \rightarrow i)}^{\pi'} e_{j \rightarrow i}^{\pi'}$$

Similarly, from the outflows of B_i's execution with history π, we get:

$$v_i^{\pi} = \sum_{j} e_{i \rightarrow j}^{\pi}$$

Repetition of a history pattern Let us assume a misprediction of the branch in B_i with history π. This means that certain blocks (perhaps B_i itself) were executed with history π such that the outcomes of these branches created a prediction different from the current outcome of B_i. Thus, to model mispredictions, we need to capture repeated occurrences of a history π during the program's execution. For this purpose, we define $p^{\pi}_{i \rightsquigarrow j}$.

Definition 5.3. *Let B_i and B_j be two basic blocks with branch instructions and π be a history pattern. Then $p^{\pi}_{i \rightsquigarrow j}$ is the number of times a path is taken from B_i to B_j s.t.*

- *π never occurs at a node with a branch instruction between B_i and B_j.*
- *If $B_i \neq$ start block, then π occurs at B_i*
- *If $B_j \neq$ end block, then π occurs at B_j*

Intuitively, $p^{\pi}_{i \rightsquigarrow j}$ denotes the number of times control flows from B_i to B_j s.t. the πth row of the prediction table is only used for branch prediction at B_i and B_j, and is never accessed in between. In these scenarios, the outcome of B_i may cause a misprediction at B_j. Furthermore, $p^{\pi}_{start \rightsquigarrow i}$ ($p^{\pi}_{i \rightsquigarrow end}$) models the number of times the π th row of the prediction table is looked up for the first (last) time at B_i.

When the πth row is used for branch prediction at B_i, either the πth row is used for the first time (denoted by $p^{\pi}_{start \rightsquigarrow i}$) or the πth row was used for branch prediction last time in some block $B_j \neq B_{start}$. Similarly, for every use of the πth row of the prediction table at B_i, either it is the last use (denoted by $p^{\pi}_{i \rightsquigarrow end}$) or it will be used the next time in $B_j \neq B_{end}$. Since v^{π}_i denotes the number of times B_i uses the πth row of the prediction table, we have:

$$v^{\pi}_i = \sum_j p^{\pi}_{j \rightsquigarrow i} = \sum_j p^{\pi}_{i \rightsquigarrow j}$$

Also, there can be at most one first use, and at most one last use of the π th row of the prediction table during program execution. Therefore, we get:

$$\sum_i p^{\pi}_{start \rightsquigarrow i} \leq 1 \quad \text{and} \quad \sum_i p^{\pi}_{i \rightsquigarrow end} \leq 1$$

Introducing branch outcomes To model mispredictions, we not only need to model the repetition of history patterns, but also branch outcomes. A misprediction occurs on differing branch outcomes for the same history pattern. Therefore, we partition the paths contributing to the count $p^{\pi}_{i \rightsquigarrow j}$ based on the branch outcome at B_i: $p^{\pi,1}_{i \rightsquigarrow j}$ and $p^{\pi,0}_{i \rightsquigarrow j}$, which denote the execution count of those paths that begin with the outgoing edge of B_i labeled 1 (i.e., outcome 1) and 0, respectively. By definition:

$$p^{\pi}_{i \rightsquigarrow j} = p^{\pi,1}_{i \rightsquigarrow j} + p^{\pi,0}_{i \rightsquigarrow j}$$

$$\textstyle\sum_j p^{\pi,1}_{i \rightsquigarrow j} = e^{\pi}_{i \rightarrow k} \quad \text{and} \quad \sum_j p^{\pi,0}_{i \rightsquigarrow j} = e^{\pi}_{i \rightarrow l}$$

where $label(i \rightarrow k) = 1$ and $label(i \rightarrow l) = 0$. In other words, $i \rightarrow l$ and $i \rightarrow k$ are the outgoing edges of basic block B_i with labels 0 and 1, respectively.

Modeling mispredictions For simplicity of exposition, let us assume that each row of the prediction table contains a one-bit prediction: 0 denotes a prediction that the branch will not be taken, and 1 denotes a prediction that the branch will be taken. However, our technique for estimating mispredictions is generic. It can be extended if the prediction table maintains more than one bit per entry. In particular, a recent work [4] has modeled a n-bit saturating counter (in each row of the prediction table).

Recall that bm^{π}_i denotes the number of mispredictions of the branch in B_i when it is executed with history pattern π. There can be two scenarios in which B_i is mispredicted with history π:

- Case 1: Branch of B_i is taken

  Since the actual control flow will go through the taken edge $i \rightarrow k$, we denote

the misprediction count of this case as $em^{\pi}_{i\to k}$. Obviously, $em^{\pi}_{i\to k} \le e^{\pi}_{i\to k}$. On the other hand, when a branch at B_i is mispredicted as not taken, the prediction in row π of the prediction table must be 0 (not taken). This is possible only if another block B_j is executed with history π and outcome 0 and history π never appears between B_j and B_i. The total number of such inflows into B_i is at most $\sum_j p^{\pi,0}_{j\rightsquigarrow i}$. Combine above observations, we have:

$$em^{\pi}_{i\to k} \;\le\; min\left(e^{\pi}_{i\to k},\;\; \sum_j p^{\pi,0}_{j\rightsquigarrow i}\right) \tag{5.2}$$

- Case 2: Branch of B_i is not taken

  Since the actual control flow will go through the not taken edge $i \to l$, we denote the misprediction count of this case as $em^{\pi}_{i\to l}$. Following the reasoning in Case 1, we have the following bound on $em^{\pi}_{i\to l}$:

$$em^{\pi}_{i\to l} \;\le\; min\left(e^{\pi}_{i\to l},\;\; \sum_j p^{\pi,1}_{j\rightsquigarrow i}\right) \tag{5.3}$$

Thus, the misprediction count bm^{π}_i is the sum of the two cases:

$$bm^{\pi}_i = em^{\pi}_{i\to k} + em^{\pi}_{i\to l} \tag{5.4}$$

Additionally, let $em_{i\to j}$ be the misprediction count for control flow transfers along the edge $i \to j$ under all branch histories, we can bound it by the following equation[2]:

$$em_{i\to j} = \sum_{\pi} em^{\pi}_{i\to j} \tag{5.5}$$

Putting it all together We have derived linear inequalities on v_i (execution count of B_i) and bm_i (misprediction count of B_i). We now maximize the objective function (denoting the execution time of the program), subject to these constraints using an (integer) linear programming solver. This gives an upper bound of the program's WCET.

[2]This is unnecessary here, but the variables and constraints will be useful in the next chapter

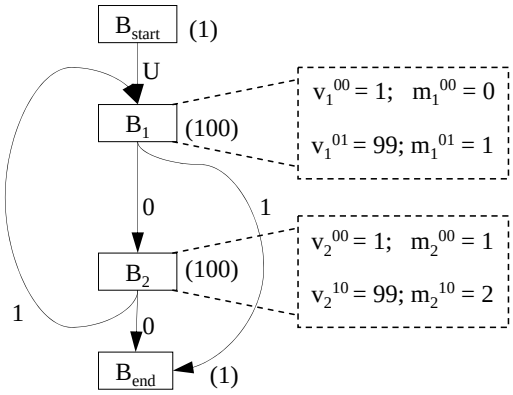


Figure 5.1: Example of the Control Flow Graph

5.1.2 An Example

In this part, we illustrate our WCET estimation technique with a simple example. Consider the control flow graph in Figure 5.1. The start and end blocks are called B_{start} and B_{end} respectively. All edges of the graph are labeled. Recall that the label U denotes unconditional control flow and the label 1 (0) denotes control flow by taking (not taking) a conditional branch. We assume that a two-bit history pattern is maintained *i.e.*, the prediction table has four rows for the four possible history patterns: 00, 01, 10, 11. Also, each row of the prediction table contains one bit to store the last outcome for that pattern: 0 for not taken and 1 for taken.

Flow constraints and loop bounds The *start* and *end* nodes execute only once. Hence

$$v_{start} = v_{end} = 1 = e_{start \to 1} = e_{2 \to end} + e_{1 \to end}$$

From the inflows and outflows of blocks 1 and 2, we get:

$$v_1 = e_{start \to 1} + e_{2 \to 1} = e_{1 \to 2} + e_{1 \to end}$$

$$v_2 = e_{1 \to 2} = e_{2 \to end} + e_{2 \to 1}$$

Furthermore, the edge $2 \rightarrow 1$ is a loop, and its bound must be given. In our method, this bound is either computed offline or provided by the user. Let us consider a loop bound of 100. Then,

$$e_{2\rightarrow 1} < 100$$

Defining WCET Let us assume a branch misprediction penalty of three clock cycles. The WCET of the program is obtained by maximizing:

$$Time = 2v_{start} + 2v_1 + 4v_2 + 2v_{end} + 3bm_1 + 3bm_2$$

assuming $t_{start} = t_1 = 2$, $t_2 = 4$ and $t_{end} = 2$. Recall that t_i is the execution time of block i (assuming perfect prediction); bm_i is the number of mispredictions of block i. There are no mispredictions for executions of *start* and *end* blocks, since they do not have branches.

Introducing History Patterns We find out the possible history patterns π for each basic block B_i via static analysis of the control flow graph. The initial history at the beginning of program execution is assumed to be 00. In our example, the possible history patterns for the different basic blocks are as follows:

B_{start}: {00}

B_1: {00, 01}

B_2: {00, 10}

B_{end}: {00, 01, 11}

We now introduce the variables v_i^π and bm_i^π: the execution count and misprediction count of block i with history π.

$$v_{start} = v_{start}^{00} = 1 \qquad bm_{start} = 0$$

$$v_1 = v_1^{00} + v_1^{01} \qquad bm_1 = bm_1^{00} + bm_1^{01}$$

$$v_2 = v_2^{00} + v_2^{10} \qquad bm_2 = bm_2^{00} + bm_2^{10}$$

$$v_{end} = v_{end}^{00} + v_{end}^{01} + v_{end}^{11} = 1 \qquad bm_{end} = 0$$

$$bm_1^{00} \leq v_1^{00} \qquad bm_1^{01} \leq v_1^{01}$$

$$bm_2^{00} \leq v_2^{00} \qquad bm_2^{10} \leq v_2^{10}$$

We also define variables of the form $e_{i\to j}^{\pi}$ as follows (by using the set of patterns possible at each basic block):

$$e_{start\to 1} = e_{start\to 1}^{00}$$

$$e_{1\to 2} = e_{1\to 2}^{00} + e_{1\to 2}^{01} \qquad e_{1\to end} = e_{1\to end}^{00} + e_{1\to end}^{01}$$

$$e_{2\to 1} = e_{2\to 1}^{00} + e_{2\to 1}^{10} \qquad e_{2\to end} = e_{2\to end}^{00} + e_{2\to end}^{10}$$

Control flow among history patterns We now derive the constraints on v_i^{π} based on the flow of the pattern π. Let us consider the inflows and outflows of block 1 with history 01. From the inflows we get:

$$v_1^{01} = e_{2\to 1}^{00} + e_{2\to 1}^{10}$$

Note that the inflow from block *start* to block 1 is automatically disregarded in this constraint since it cannot produce a history 01 when we arrive at block 1. Also, for the inflows from block 2 the history at block 2 can be either 00 or 10. Both of these patterns produce history 01 at block 1 when control flows via the edge $2 \to 1$ *i.e.*, $\Gamma(00, 2 \to 1) = \Gamma(10, 2 \to 1) = 01$ from Definition 5.2.

From the outflows of the executions of block 1 with history 01 we have:

$$v_1^{01} = e_{1\to 2}^{01} + e_{1\to end}^{01}$$

Constraints for inflows/outflows of block 1 with history 00, block 2 with history 00, and block 2 with history 10 are derived similarly.

Repetition of history pattern To model the repetition of a history pattern along a program path, variables $p^{\pi}_{i \rightsquigarrow j}$ are introduced (refer to Definition 5.3). We now present the constraints for the pattern 01. Corresponding to the first and last occurrence of the history pattern 01, we get:

$$p^{01}_{start \rightsquigarrow 1} \leq 1 \quad \text{and} \quad p^{01}_{1 \rightsquigarrow end} \leq 1$$

Corresponding to the repetition of the pattern 01, the constraints are as follows:

	Exec. with pattern 01	Inflow from last occurrence of 01	Outflow to next occurrence of 01
v^{01}_1		$= p^{01}_{1 \rightsquigarrow 1} + p^{01}_{start \rightsquigarrow 1}$	$= p^{01}_{1 \rightsquigarrow 1} + p^{01}_{1 \rightsquigarrow end}$

Similarly, we provide constraints for the other patterns.

Introducing branch outcomes For each $p^{\pi}_{i \rightsquigarrow j}$, we define the variables $p^{\pi,0}_{i \rightsquigarrow j}$ and $p^{\pi,1}_{i \rightsquigarrow j}$ via the equation $p^{\pi}_{i \rightsquigarrow j} = p^{\pi,0}_{i \rightsquigarrow j} + p^{\pi,1}_{i \rightsquigarrow j}$. More importantly, we relate $p^{\pi}_{i \rightsquigarrow j}$ variables to $e^{\pi}_{i \rightarrow j}$ variables via $p^{\pi,0}_{i \rightsquigarrow j}$ and $p^{\pi,1}_{i \rightsquigarrow j}$. For example we have $p^{10,1}_{2 \rightsquigarrow 2} + p^{10,1}_{2 \rightsquigarrow end} = e^{10}_{2 \rightarrow 1}$ in Figure 5.1. In our simple example, we only derive trivial constraints in this category. In general, a sum of $p^{\pi,1}_{i \rightsquigarrow j}$ (or $p^{\pi,0}_{i \rightsquigarrow j}$) variables equals an $e^{\pi}_{i \rightarrow j}$ variable.

Modeling mispredictions Let us now derive the constraints for bm^{01}_1, the number of mispredictions of block 1 with history 01. For this, we consider two cases corresponding to the outcome of the branch at block 1.

- Case 1: The branch at block 1 is taken, and the last branch using the 01 row of the predictor table is not taken.

  The number of times the branch at block 1 under history 01 is taken is $e^{01}_{1 \rightarrow end}$. The number of times the last branch (before arriving at block 1) using the 01 row of the predictor table is not taken is $p^{01,0}_{start \rightsquigarrow 1} + p^{01,0}_{1 \rightsquigarrow 1}$. Note that the other

block (block 2) is not considered since block 2 cannot be reached with pattern 01.

Thus, the misprediction count $em^{01}_{1\to end}$ for this case is

$$em^{01}_{1\to end} \;\leq\; min\left(e^{01}_{1\to end},\;\; p^{01,0}_{start\rightsquigarrow 1} + p^{01,0}_{1\rightsquigarrow 1}\right)$$

- Case 2: The branch at block 1 under history 01 is not taken, and the last branch using the 01 row of the predictor table is taken. This happens $em^{01}_{1\to 2}$ times where

$$em^{01}_{1\to 2} \;\leq\; min\left(e^{01}_{1\to 2},\;\; 0\right) = 0$$

Note that 0 appears in above formula as in this particular example, no earlier branch using the 01 row of the predictor table with outcome taken can reach block 1.

By combining the above two cases, we get the misprediction number for bm^{01}_1:

$$bm^{01}_1 = em^{01}_{1\to end} + em^{01}_{1\to 2}$$

Other misprediction constraints are:

$$\begin{aligned}
bm^{00}_1 &= em^{00}_{1\to end} + em^{00}_{1\to 2} \\
&\leq min\left(e^{00}_{1\to end}\;,\; p^{00,0}_{start\rightsquigarrow 1}\right) + min\left(e^{00}_{1\to 2}\;,\; 0\right) \\
bm^{00}_2 &= em^{00}_{2\to 1} + em^{00}_{2\to end} \\
&\leq min\left(e^{00}_{2\to 1}\;,\; p^{00,0}_{1\rightsquigarrow 2}\right) + min\left(e^{00}_{2\to end}\;,\; 0\right) \\
bm^{10}_2 &= em^{10}_{2\to 1} + em^{10}_{2\to end} \\
&\leq min\left(e^{10}_{2\to 1}\;,\; p^{10,0}_{start\rightsquigarrow 2}\right) + min\left(e^{10}_{2\to end}\;,\; p^{10,1}_{2\rightsquigarrow 2}\right)
\end{aligned}$$

They correspond to the constraints on bm^{π}_i in Section 5.1.1. Maximizing the objective function with respect to all these constraints gives the program's WCET.

The execution counts of basic blocks as well as their misprediction counts computed by the ILP solver are given in Figure 5.1.

5.1.3 Retargetability

We now discuss how our modeling can be used to capture the effects of various local and global branch prediction schemes. Our modeling of branch prediction is independent of the definition of the prediction table index, so far called the history pattern π. All our constraints only assume the following: (a) the presence of a global prediction table, (b) the index π into this prediction table, and (c) every time the π th row is looked up for branch prediction, it is updated subsequent to the branch outcome. These constraints continue to hold even if π does not denote the history pattern (as in the *GAg* scheme).

In fact, the different branch prediction schemes differ from each other primarily in how they index into the prediction table. Thus, to predict a branch I, the index computed is a function of: (a) the past execution trace (history) and (b) the address of the branch instruction I. In the *GAg* scheme, the index computed depends solely on the history and not the branch instruction address. Other global prediction schemes (*gshare*, *gselect*) use both the history and the branch address, while local schemes use only the branch address.

To model the effect of other branch prediction schemes, we only alter the meaning of π, and show how π is updated with the control flow (the Γ function of Definition 5.2). This of course affects the possible prediction table indices that can be looked up at a basic block B_i. *No change is made to the linear constraints* (parameterized w.r.t. possible prediction table indices at each basic block) described in the previous subsection. These constraints then bound a program's WCET (under the new branch prediction scheme).

Other global schemes We now discuss two other global prediction schemes: *gshare* and *gselect* [51, 73]. In *gshare*, the index π used for a branch instruction I is defined

as

$$\pi = history_m \oplus address_n(I)$$

where m, n are constants, $n \geq m$, $\oplus$ is XOR, $address_n(I)$ denotes the lower order n bits of I's address, and $history_m$ denotes the most recent m branch outcomes (which are XOR-ed with higher-order m bits of $address_n(I)$). The updating of π due to control flow is modeled by the function:

$$\Gamma_{gshare}(\pi, i \rightarrow j) = \Gamma(history_m, i \rightarrow j) \oplus address_n(j)$$

where $i \rightarrow j$ is an edge in the control flow graph, $address_n(j)$ is the least significant n bits of the branch instruction in basic block j, and Γ is the function on the history patterns described in Definition 5.2.

The modeling of the *gselect* prediction scheme is similar. Here, the index π into the prediction table is defined as:

$$\pi = history_m \bullet address_n(j)$$

where m and n are some constants and $\bullet$ denotes concatenation. The updating of π due to control flow is given by function $\Gamma_{gselect}$

$$\Gamma_{gselect}(\pi, i \rightarrow j) = \Gamma(history_m, i \rightarrow j) \bullet address_n(j)$$

Again, $i \rightarrow j$ is an edge in the control flow graph and Γ is the function described in Definition 5.2.

Local prediction schemes In local schemes, the index π into the prediction table for predicting the outcome of instruction I is $\pi = address_n(I)$. Here, n is a constant and $address_n(I)$ denotes the least significant n bits of the address of branch instruction I.

Updating of the index π due to control flow is given by $\Gamma_{local}(\pi, i \rightarrow j) = address_n(j)$. Here, $i \rightarrow j$ is an edge in the control flow graph and $address_n(j)$ is

the least significant n bits of the last instruction in basic block j. If block j contains a branch instruction I, it must be the last instruction of j. Thus, the least significant n bits of the address of I are used to index into the prediction table (as demanded by local schemes). If j does not contain any branch instruction, then the index computed is never used to lookup the prediction table. Clearly, since each block j always uses the same index π into the prediction table, index π is used at basic block j if and only if π denotes the least significant n bits of the address of the branch instruction of block j (if any).

5.2 Integration with Instruction Cache Analysis

Our branch prediction analysis is a Integer Linear Program (ILP) based approach. In this section, we will show how to integrate it with another ILP based instruction cache analysis. The key point for combined analysis of multiple microarchitectural features is to capture their interactions. In the context of branch prediction and instruction caching, the interaction is unidirectional. The speculative execution via branch prediction can alter the behavior of the instruction cache, i.e., instructions can be prefetched into or displaced from the cache due to speculative execution. On the other hand, the instruction cache does not change the branch prediction outcome as an cache access does not access or change the state of the branch predictor. This indicates that our branch prediction technique needs not to be changed in the combined analysis.

To discuss the combined analysis, we first review on the instruction cache analysis technique proposed by Li et al. [43]. Then we make modifications to it to take into account the effect of speculative execution.

5.2.1 Instruction Cache Analysis

We recapitulate the earlier instruction cache modeling [43]. A basic block B_i is partitioned into n_i l-blocks[3] denoted as $B_{i.1}, B_{i.2}, \ldots, B_{i.n_i}$. Let $cm_{i.j}$ be the total cache misses for l-block $B_{i.j}$ and *cmp* be the constant denoting the cache miss penalty. Then, the total execution time is:

$$Time = \sum_{i=1}^{N} (cost_i \times v_i + bmp \times bm_i + \sum_{j=1}^{n_i} cmp \times cm_{i.j}) \tag{5.6}$$

For simplicity of exposition, let us assume a direct mapped cache. For each cache line c, a *Cache Conflict Graph (CCG)* G_c [43] is constructed. The nodes of G_c are the l-blocks mapped to c. An edge $B_{i.j} \rightsquigarrow B_{u.v}$ exists in G_c iff there exists a path in the CFG s.t. control flows from $B_{i.j}$ to $B_{u.v}$ without going through any other l-block mapped to c. In other words, there is an edge between l-blocks $B_{i.j}$ to $B_{u.v}$ if $B_{i.j}$ can be present in the cache when control reaches $B_{u.v}$.

Let $r_{i.j \rightsquigarrow u.v}$ be the execution count of the edge between l-blocks $B_{i.j}$ and $B_{u.v}$ in a CCG. Now, the execution count of l-block $B_{i.j}$ equals the execution count of basic block B_i. Also, at each node of the CCG, the inflow equals the outflow and both equal the execution count of the node. Therefore,

$$v_i = \sum_{u.v} r_{i.j \rightsquigarrow u.v} = \sum_{u.v} r_{u.v \rightsquigarrow i.j} \tag{5.7}$$

The cache miss count $cm_{i.j}$ equals the inflow from *conflicting* l-blocks in the CCG (whether two l-blocks are conflicting or non-conflicting is statically determined by portions of their instruction addresses, which are used as tags in cache line). Thus, we have:

$$cm_{i.j} = \sum_{\substack{u.v \\ B_{u.v} \ conflicts \ B_{i.j}}} r_{u.v \rightsquigarrow i.j} \tag{5.8}$$

[3]A line-block, or l-block, is a sequence of instructions in a basic block that belong to the same instruction cache line.

5.2.2 Changes to Instruction Cache Analysis

Effects of speculative execution on caching WCET analysis as described in the previous section does not take into account the effect of branch misprediction on instruction cache performance. When a branch is predicted, instructions are fetched and executed from the predicted path. If all the branches are predicted correctly, then the analysis described in previous section will give accurate results. Now, consider a branch that is mispredicted. The processor will fetch and execute instructions along the mispredicted path till the branch is resolved. There can be two scenarios during mispredicted path execution: (1) there is no cache miss, and (2) there is at least one cache miss. In the first scenario, the misprediction has no effect on the instruction cache. However, in the second scenario, the instruction cache content is modified when the processor resumes execution from the correct path. Various studies have concluded that depending on the application, this wrong-path prefetching can have a constructive or a destructive effect on the instruction cache's performance [12, 59]. Our goal here is to model this wrong-path cache effect for WCET analysis.

We make two standard assumptions. First, we assume that the processor allows only *one unresolved branch* at any point of time during execution. Thus, if another branch is encountered during speculative execution, the processor simply waits till the previous branch is resolved. We also assume that the instruction cache is *blocking* (i.e., it can support only one pending cache miss). This is indeed the case in almost all commercial processors.

We introduce some notations for the subsequent parts. We use $[B_{i.j}]$ to denote the cache line to which l-block $B_{i.j}$ maps. The shorthand $B_{i.j} \cong B_{u.v}$ is used to denote that l-blocks $B_{i.j}$ and $B_{u.v}$ map to the same cache line. Thus $B_{i.j} \cong B_{u.v}$ iff $[B_{i.j}] = [B_{u.v}]$.

The effects of speculation on instruction cache performance can be categorized as follows:

1. An l-block $B_{i.j}$ misses during normal execution since it is displaced by another l-block $B_{u.v} \cong B_{i.j}$ during speculative execution (**destructive effect**).

2. An l-block $B_{i.j}$ hits during normal execution, since it is pre-fetched during speculative execution (**constructive effect**).

3. A pending cache miss of $B_{i.j}$ during speculative execution along the wrong path causes the processor to stall when the branch is resolved. How long the stall lasts depends on the portion of cache miss penalty which is masked by the branch misprediction penalty. If the speculative fetching is completely masked by branch penalty, then there is no delay incurred.

The last situation cannot be simply deemed constructive or destructive, although a delay often happens in that case. The cost of the delay may be offset later by a cache hit to the l-block.

The following changes to the Cache Conflict Graph (CCG) capture both the constructive and destructive effects of speculative execution on cache.

Additional nodes in Cache Conflict Graph We add all the l-blocks fetched along the mispredicted path to their respective cache conflict graphs. Given a conditional branch b, its actual outcome X (not taken or taken, denoted as 0 and 1, respectively) and misprediction penalty *bmp* (a constant number of clock cycles), we can identify the set of l-blocks accessed along the mispredicted path, called *Spec(b,X)*. Clearly, the cost of executing the blocks in $Spec(b, X)$ cannot exceed *bmp*. If one or more blocks cause cache misses, then not all the l-blocks in $Spec(b, X)$ can execute. Those l-blocks executed along the mispredicted path are called *ml-blocks* and are annotated with the corresponding basic block containing the branch instruction and the actual outcome. For example, if $B_{i.j} \in Spec(b, X)$, then the corresponding ml-block is denoted by $B_{i.j}^{b,X}$. Note that it is possible to have multiple ml-blocks corresponding

to an l-block. For an l-block $B_{i.j}$, all its ml-blocks are added to the CCG of the cache line it maps to.

Additional edges in Cache Conflict Graph We now need to add additional edges in the cache conflict graphs. Given a CCG, we add edges between ml-blocks and the normal l-blocks; we also add edges between ml-blocks. For an ml-block $B_{i.j}^{b,X}$, we add edges to/from all the other l-blocks $B_{u.v}$ in the CCG of cache line $[B_{i.j}]$ and their corresponding ml-blocks as follows:

1. $B_{u.v} \rightsquigarrow B_{i.j}^{b,X}$ if there exists a path from $B_{u.v}$ to $B_{i.j}$ through branch b that does not contain any other l-block mapped to $[B_{i.j}]$. This models the flow from the last normal use of the cache line to the ml-block.

2. $B_{i.j}^{b,X} \rightsquigarrow B_{u.v}^{b,X}$ if $B_{u.v}$ is the next use of the cache line $[B_{i.j}]$ in $Spec(b, X)$ after $B_{i.j}$. This models the flow from the ml-block to the next possible use of the cache line along the mispredicted path.

3. $B_{i.j}^{b,X} \rightsquigarrow B_{u.v}$ if there exists a path from branch b with outcome X to $B_{u.v}$ that does not contain any other l-block mapped to $[B_{i.j}]$.

4. In addition, in case 3, if the path to $B_{u.v}$ goes through branch b' and $B_{u.v} \in Spec(b', Y)$ (b' can be the same as or different from b), then we also add $B_{i.j}^{b,X} \rightsquigarrow B_{u.v}^{b',Y}$. The edges in cases 3 and 4 model the flow from the ml-block to the next possible use of the cache line after the branch is resolved.

Figure 5.2 illustrates these cases. The shaded rectangles are the ml-blocks and the unshaded ones are the normal l-blocks. The third and fourth type of edges require some explanation. If there are multiple l-blocks along the speculative path that map to a particular cache line, then we conservatively add outgoing edges from all of them to the first use of the cache line in the correct path (or another speculative path). This is because any one of these l-blocks can be in the cache when the branch is

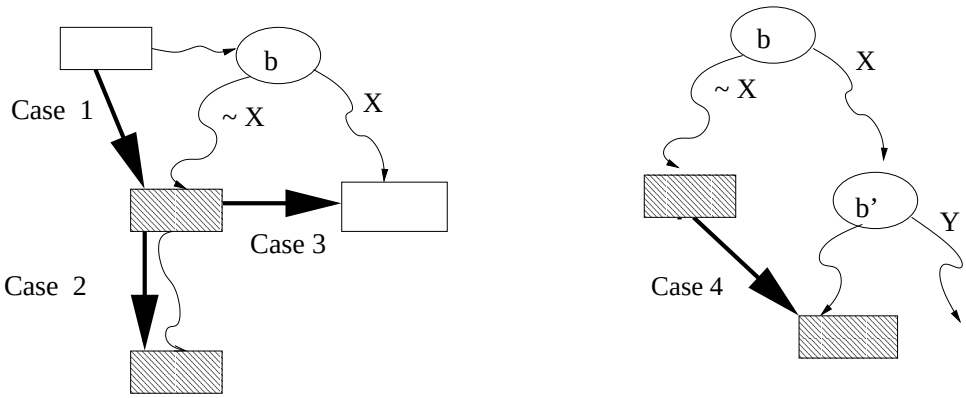


Figure 5.2: Additional edges in the Cache Conflict Graph due to Speculative Execution. The l-blocks are shown as rectangular boxes, and the ml-blocks among them are shaded.

resolved; exactly which one will be in the cache when the branch is resolved depends on the exact values of *bmp*, *cmp* and the execution time of the individual basic blocks.

Figure 5.3 illustrates the modifications to the CCG with an example. The control flow graph is shown in Figure 5.3(a). Let us assume that l-blocks $B_{0.1}$, $B_{1.2}$ and $B_{3.1}$ belong to the same cache line. Then, the original CCG for that cache line is shown in Figure 5.3(b). A dummy start node and an end node are added to each CCG to make the initial and terminal flow equations correct.

The modifications to the CCG due to wrong-path prefetching is shown in Figure 5.3(c). We add two ml-block $B_{3.1}^{2,1}$ and $B_{1.2}^{3,0}$ corresponding to the mispredictions at node B_2 and node B_3, respectively. Note that we do not add any node corresponding to a 0 outcome at branch B_2 and a 1 outcome at branch B_3. This is because with a 0 outcome at branch B_2, the mispredicted path fetches basic block B_2 which does not contain any l-block that maps to the cache line, and similarly for B_3 with outcome 1. Among the additional edges, $B_{1.2} \rightsquigarrow B_{3.1}^{2,1}$ and $B_{3.1} \rightsquigarrow B_{1.2}^{3,0}$ belong to the first type. The edges $B_{3.1}^{2,1} \rightsquigarrow B_{3.1}$ and $B_{3.1}^{2,1} \rightsquigarrow B_{3.1}^{2,1}$ belong to the third and fourth type respectively.

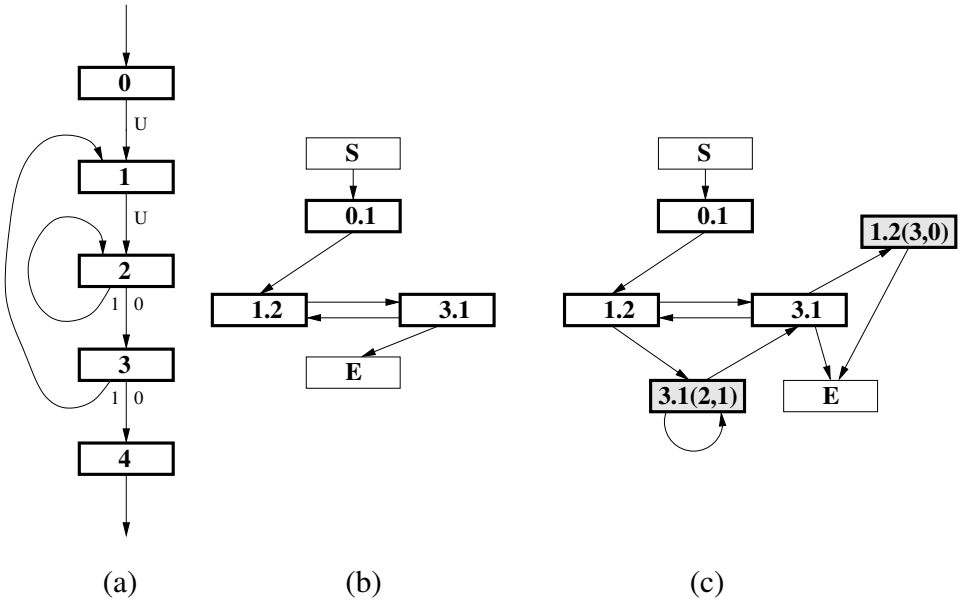


Figure 5.3: Changes to Cache Conflict Graph (Shaded nodes are ml-blocks)

Figure 5.3 shows the modeling of the constructive effect of wrong path prefetching. In the original CCG, there is an edge $B_{1.2} \rightsquigarrow B_{3.1}$ and that is the only path between the two nodes. Therefore, every time control reaches from $B_{1.2}$ to $B_{3.1}$, it is a cache miss. In the modified CCG in Figure 5.3(c), there is another path from $B_{1.2}$ to $B_{3.1}$ via the ml-block $B_{3.1}^{2,1}$. First, there is no cache miss along $B_{3.1}^{2,1} \rightsquigarrow B_{3.1}$ as they are physically the same l-block. Second, the cache miss along $B_{1.2} \rightsquigarrow B_{3.1}^{2,1}$ is partially masked by the branch misprediction delay. Thus, this kind of *prefetching* is constructive to the execution.

Additional constraints on ml-blocks The execution count of a normal l-block is equal to the execution count of the basic block it belongs to. However, for an ml-block $B_{i.j}^{b,X}$, this count is dependent on the number of mispredictions at branch b where the actual outcome is X (X is 0 or 1). To derive this execution count, note that the number of ml-blocks missed due to a single misprediction is $\left\lceil \frac{bmp}{cmp} \right\rceil$ where bmp (cmp) denotes branch misprediction penalty (cache miss penalty). In accordance

with most modern processors, we assume $bmp < cmp$ and therefore $\left\lceil \frac{bmp}{cmp} \right\rceil = 1$. This assumption is, however, not required, and our modeling can be easily extended. Given $bmp < cmp$, a single misprediction can result in at most one cache miss along the mispredicted path. Let $Spec(b, X) = \langle B_{u_1.v_1}, \ldots, B_{u_k.v_k} \rangle$. Therefore, the execution count of the ml-block $B^{b,X}_{u_i.v_i}$ is:

$$bm^X_b - \sum_{l=1}^{i-1} cm^{b,X}_{u_l.v_l}$$

where bm^X_b is the number of mispredictions at branch b with outcome X (obtained from the modeling of branch prediction) and $cm^{b,X}_{u_l.v_l}$ is the number of cache misses for the ml-block $B^{b,X}_{u_l.v_l}$. Constraints on $cm^{b,X}_{u_l.v_l}$ are obtained from the CCG as shown in Equation 5.8 (refer to page 93). Constraints on bm^X_b are obtained from our modeling of branch prediction described in Section 5.1.1.

Objective function The objective function is:

$$\begin{aligned} Time &= \sum_{i=1}^{N} (cost_i \times v_i + bmp \times bm_i + \sum_{j=1}^{n_i} cmp \times cm_{i.j}) \\ &\quad + \sum_{\substack{Cond.\ branch\ b \\ X \in 0,1}} mp\_delay(b, X) \end{aligned} \tag{5.9}$$

The three subterms of the first term are the ideal execution time, the branch penalty and the cache penalty, respectively. The last term, $mp\_delay(b, X)$ is the delay that the processor has waited for pending cache misses (arising during mispredictions) after mispredictions have been resolved. As the assumption $bmp < cmp$ holds, the criteria for such a delay to happen are: (a) a cache miss happens during a misprediction, and (b) this cache miss is not completely masked by the misprediction (still pending when the branch is resolved). Recall that $Spec(b, X) = \langle B_{u_1.v_1}, \ldots, B_{u_k.v_k} \rangle$. We define:

$$mp\_delay(b, X) = \sum_{i=1}^{k} (cm_{u_i.v_i}^{b,X} \times delay_{u_i.v_i}^{b,X})$$

$$delay_{u_i.v_i}^{b,X} = cmp - (bmp - \sum_{l=1}^{i-1} cost_{u_l.v_l})$$

where $cost_{u_l.v_l}$ is the ideal execution time of the *l-block* $B_{u_l.v_l}$. Also, $delay_{u_i.v_i}^{b,X}$ is the delay introduced due to the cache miss of $B_{u_i.v_i}$ along the mispredicted path of branch b (where the actual outcome is X). This delay is not a constant, as part of the cache miss penalty *cmp* can be masked, depending on the location of the cache miss in the mispredicted path.

5.3 Experimental Evaluation

In this section we experimentally evaluate our branch prediction analysis as well as the combined analysis with instruction caching.

Since we want to examine the effects of instruction caching and branch prediction, we exclude the impact of other factors, such as pipelining, data caching, data dependencies, etc. In our experiments, we assume a perfect processor pipeline with no stalls due to data dependencies. This allows each instruction to take a fixed number of clock cycles to execute. The only timing overhead is introduced by instruction cache misses and branch mispredictions of conditional branches.

We use the same set of benchmarks as in Chapter 4, and we compare our estimation against the simulation on SimpleScalar. Our analyzer is parameterized with respect to the prediction scheme, the predictor table size, the misprediction penalty, the cache configuration and the cache miss penalty. The default parameters in our experiments are as follows: (1) branch prediction scheme is *gshare*; (2) the two-bit branch history is XOR-ed with the four least significant bits of the branch address; (3) the branch misprediction penalty and cache miss penalty are five and 10 clock cycles respectively; (4) the instruction cache is a 1k direct-mapped cache with 16

Program	Obs. WCET	Est. WCET	Ratio
des	53047	58022	1.09
dhry	128420	131024	1.02
fdct	2513	2513	1.00
fft	219192	229406	1.04
fir	29412	33145	1.12
isort	47120	47251	1.00
ludcmp	9250	9731	1.05
matmul	15084	15184	1.01
matsum	101821	101821	1.00
minver	6259	6653	1.06
qurt	1296	1536	1.19
whet	537125	571615	1.06

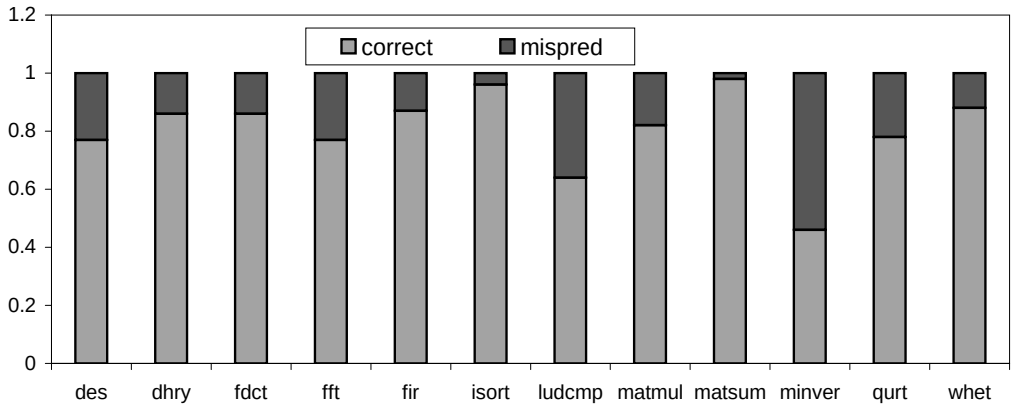
Table 5.1: Modeling Gshare Branch Prediction Scheme for WCET Analysis.

cache lines, and each line has 64 bytes. Experiments on the impact of changing the parameters are reported later in the section.

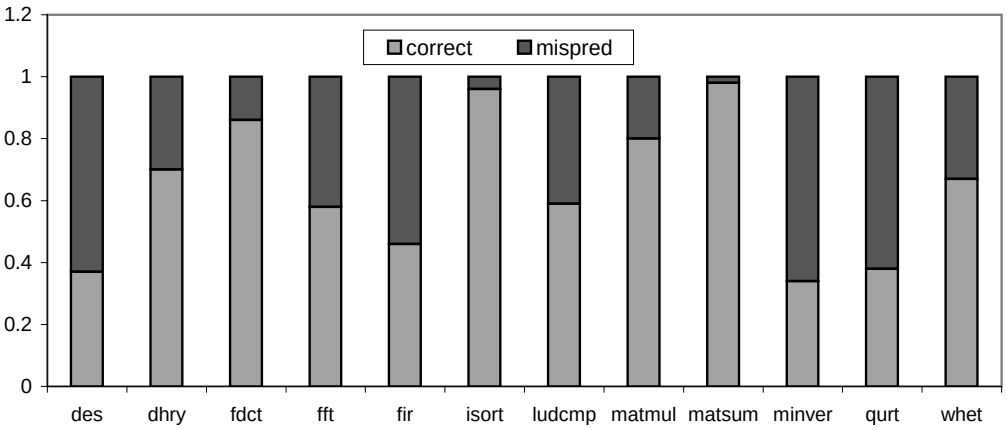
We first justify the need of modeling branch prediction. Figure 5.4 shows the correct predictions and mispredictions in observation as well as in estimation. On chart (a), we can see that for the majority of the benchmarks, more than eighty percent of the branches are correctly predicted, which means if we do not model branch predictions, these dominantly correct predictions will be pessimistically taken as mispredictions. On chart (b), we can see that our analysis indeed captures considerable correct predictions.

The results of branch prediction modeling are reported in Table 5.1. It shows the observed WCET (column **Obs. WCET**) obtained from SimpleScalar and the estimated WCET (column **Est. WCET**) obtained from our ILP based technique. We use the popular gshare prediction scheme in these experiments. We also evaluate the accuracy of our estimation technique by presenting the ratio of the estimated WCET over the observed WCET in the **Ratio** column in Table 5.1.

Table 5.2 gives the detailed results for the three branch prediction schemes: *gshare*, *GAg* and *local*. Note the WCETs are in clock cycles while mispredictions are in counts.



(a) Observation



(b) Estimation

Figure 5.4: The Importance of Modeling Branch Prediction: Mispredictions in Observation and Estimation

Pgm.	WCET					
	gshare		GAg		local	
	Obs.	Est.	Obs.	Est.	Obs.	Est.
des	53047	58022	52942	58006	54207	57671
dhry	128420	131024	127425	129385	126405	127035
fdct	2513	2513	2508	2508	2493	2493
fft	219192	229406	225932	249747	229552	229665
fir	29412	33145	31177	34617	29622	33337
isort	47120	47251	46185	47741	47135	47246
ludcmp	9250	9731	9265	9715	9265	9715
matmul	15084	15184	15179	15179	15064	15064
matsum	101821	101821	101826	101826	101806	101806
minver	6259	6653	6254	6631	6199	6631
qurt	1296	1536	1256	1461	1391	1461
whet	537125	571615	571580	571610	571570	571580
Pgm.	**Mispredictions**					
	gshare		GAg		local	
	Obs.	Est.	Obs.	Est.	Obs.	Est.
des	574	1519	553	1509	806	1438
dhry	2603	3170	2404	2800	2200	2406
fdct	8	8	7	7	4	4
fft	3094	5140	4442	9205	5166	5193
fir	183	770	536	1074	225	820
isort	391	400	204	596	394	399
ludcmp	105	119	108	116	108	116
matmul	204	224	223	223	200	200
matsum	203	203	204	204	200	200
minver	122	150	121	146	110	146
qurt	25	72	17	57	44	57
whet	3752	10650	10643	10649	10641	10643

Table 5.2: Observed and Estimated WCET and Misprediction Counts of Gshare, GAg and Local Schemes.

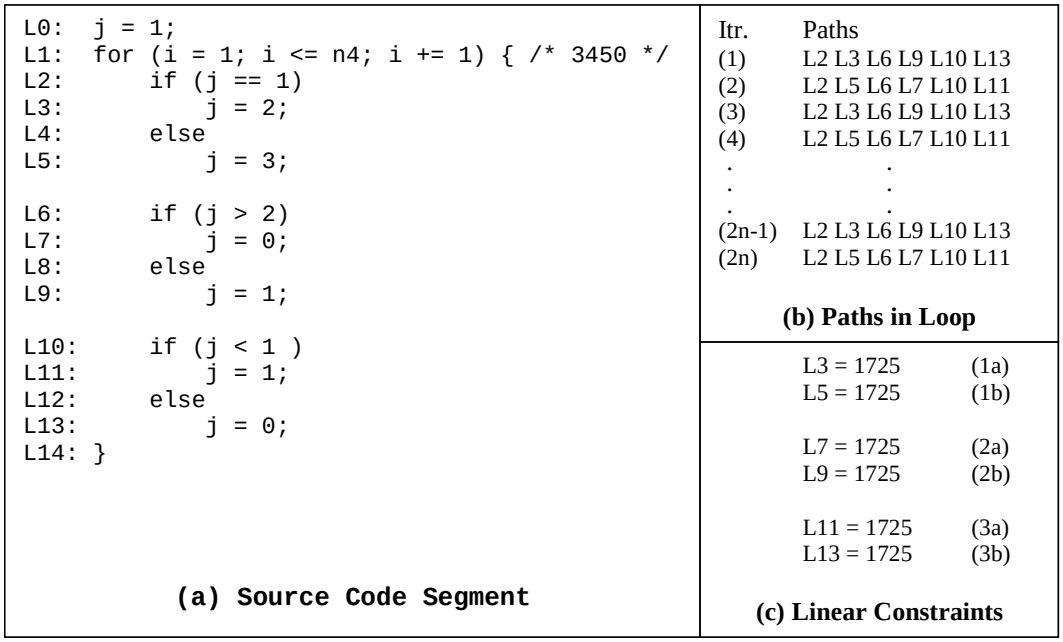


Figure 5.5: A Fragment of the Whetstone Benchmark

Difficulty in Exploiting Temporal Path Information One reason for the overestimation of misprediction counts is the aggregate nature of the ILP approach. The ILP approach only allows us to provide linear constraints on basic block execution counts. However, path information (even if provided by the user) *cannot* be exploited by the ILP solver. For example, let us study a program segment of the `whet` benchmark given in Figure 5.5. Figure 5.5(a) is a loop body with loop iteration counts annotated. There are three `if-then-else` constructs embedded in the loop body. By taking a closer look, we can figure out that the outcomes of these branches are not dependent on the input data. The paths the loop body can take in each iteration is given in Figure 5.5(b). We can see there are only two paths and they alternate during the iterations. However, this temporal information cannot be fed into the ILP solver. Instead, the ILP solver uses the constraints in Figure 5.5(c) to implicitly consider any path satisfying these constraints. All such paths are considered in the ILP solver's quest to maximize branch predictions (leading to overestimation).

So far, we have presented the experimental results for branch prediction modeling.

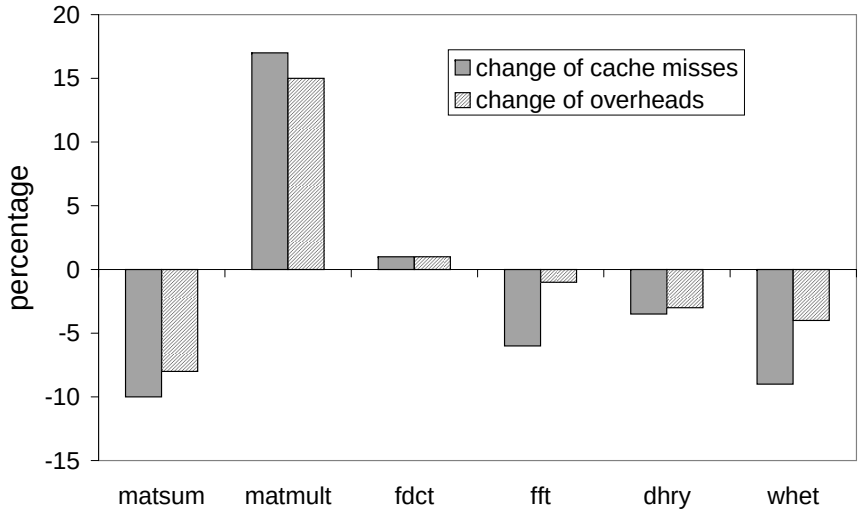


Figure 5.6: Change (in Percentage) of Cache Misses and Overall Penalties in Combined Modeling to Those in Individual Modelings

We now discuss the integrated modeling of instruction caching and branch prediction. First, we illustrate the importance of combined modeling of cache and speculation for WCET analysis by comparing it against a naive technique which models both caching and speculation but ignores the cache-speculation interaction. Figure 5.6 shows this comparison with benchmarks for which we can find the actual WCET (and the corresponding cache miss and branch misprediction overheads).

The first group of bars indicate the percentage increase/decrease in cache misses due to the effect of branch prediction on cache behavior. For `matmult` and `fdct`, there are more cache misses in the combined modeling than in the naive modeling, indicating that the destructive effects of speculation are more significant than the constructive effects. For other programs, the constructive effects outperform the destructive effects, thereby decreasing the number of cache misses. The second group of bars shows the percentage change in total timing overhead of cache misses and branch mispredictions due to cache-speculation interaction. The timing overhead shows similar behavior as cache misses. The results show that if naive modeling is used (i.e., the effect of branch prediction on caching is not modeled), the WCET

Pgm.	WCET			Mispred		Cache miss	
	Obs.	Est.	Ratio	Obs.	Est.	Obs.	Est.
des	87436	96437	1.10	574	1460	3255	3497
dhry	218684	232523	1.06	2603	2514	8125	9639
fdct	8798	8803	1.00	8	8	626	626
fft	219428	229651	1.04	3094	5139	21	25
fir	65223	67370	1.03	183	370	3506	3451
isort	48685	48836	1.00	391	400	107	109
ludcmp	11328	13612	1.20	105	115	192	383
matsum	105504	105917	1.00	203	203	307	409
matmul	25155	25679	1.02	204	215	945	975
minver	8279	9461	1.14	122	140	183	279
qurt	1925	2242	1.16	25	73	57	63
whet	545544	581557	1.06	3752	10580	765	986

Table 5.3: Combined Analysis of Branch Prediction and Instruction Caching

can either be overestimated (as the downward bars indicate), or, more seriously, be underestimated (as the upward bars indicate).

The results for combined modeling of instruction caching and branch prediction are given in Table 5.3. Note that the numbers for the `WCET` columns are in processor cycles while the `Mispred.` and `Cache miss` columns denote misprediction and cache miss counts. As we can see from the `ratio` column, most benchmarks have tight estimated bounds.

Modern processors have deep pipelines and an increasing gap between processor speed an memory latency. Deeper pipelining leads to a larger misprediction penalty (in terms of clock cycles). The increasing processor-memory speed gap results in a longer cache miss penalty. Due to this trend of hardware advancement, we examine the accuracy of our WCET analysis with more aggressive parameters by doubling the *bmp* (from the default five clock cycles to 10 clock cycles) and the *cmp* (from 10 clock cycles to 20 clock cycles). From the chart in Figure 5.7 we can see that for each benchmark, the Est/Obs WCET ratio does not change significantly under different penalty settings. This indicates that the accuracy of our analysis can be applied to processors with high misprediction penalties and/or cache miss penalties.

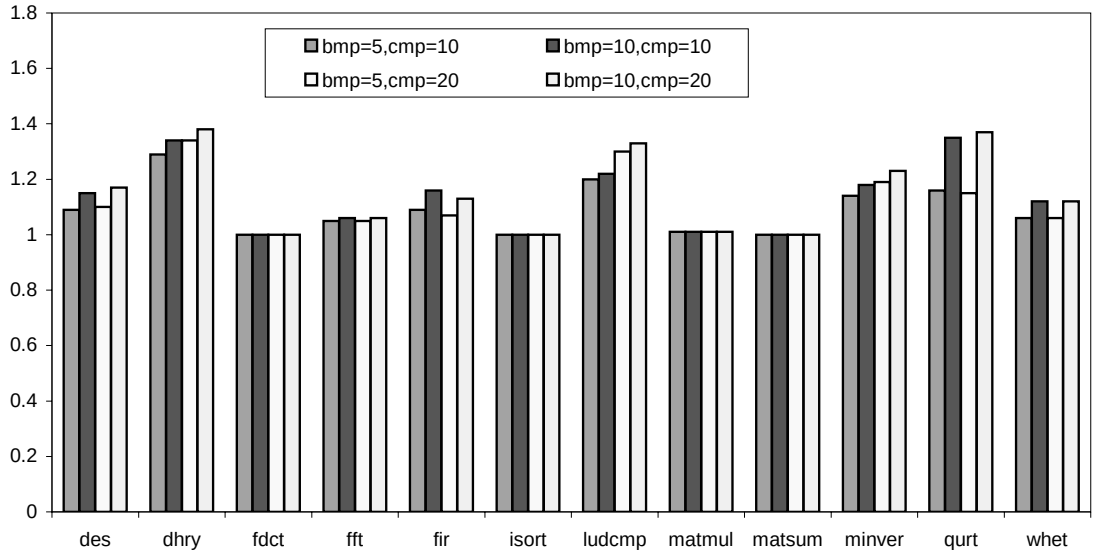


Figure 5.7: Est./Obs. WCET Ratio under Different **Misprediction Penalties** and **Cache Miss Penalties**

Program	**Complexity**			**Time (seconds)**	
	Bytes	Blocks	Branches	Analysis	Solving
des	3776	59	23	1.21	1.68
dhry	3144	98	34	1.47	2.48
fdct	5744	20	9	0.06	0.02
fft	2472	29	19	0.10	0.09
fir	3848	70	39	1.44	1.47
isort	152	7	4	0.02	0.01
ludcmp	5152	65	44	0.49	0.13
matmul	264	7	3	0.01	0.01
matsum	192	5	2	0.01	0.01
minver	6672	109	74	1.69	3.80
qurt	1944	33	20	0.76	0.63
whet	2512	40	22	0.40	0.55

Table 5.4: Program Complexity and Processing Time

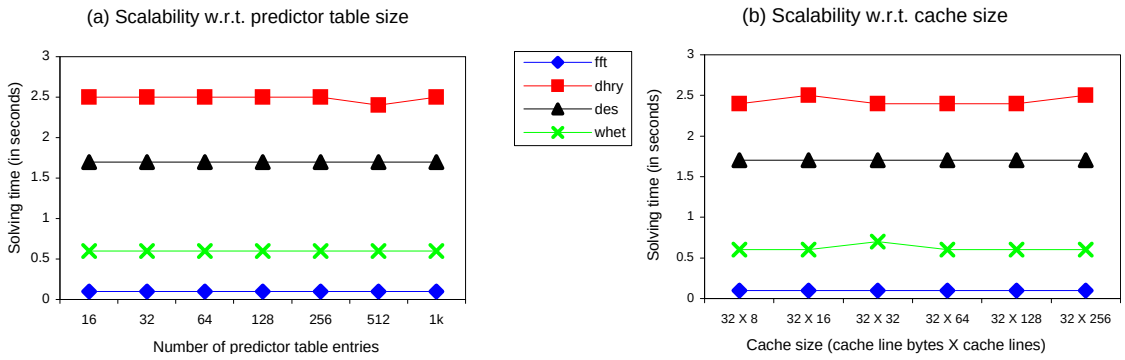


Figure 5.8: Scalability with Increasing Branch Prediction Table Size and Cache Size

Finally, we look at the performance of the analysis. The complexities of the programs and their solving times are given in Table 5.4. The complexity of a program is presented by its size in terms of bytes, its number of basic blocks as well as its conditional branches. This is only an approximate measure of the complexity. The column `Analysis` gives the times that our analyzer takes for performing control flow analysis, branch prediction modeling and the ILP problem formulation; and the `Solving` column gives the ILP solving times by CPLEX. Here, the gshare scheme is used with the default parameters used earlier. As we can see, the analysis times and ILP solving times of all benchmarks are within seconds.

We now consider the variation of ILP solution time for some benchmarks with larger predictor table sizes (the gshare scheme) and cache sizes. In Figure 5.8(a), the branch prediction table sizes vary from 16 to 1024 entries. Recall that in *gshare*, the branch instruction address is XOR-ed with the global branch history bits. In practice, the *gshare* scheme uses a smaller number of history bits than address bits, and XORs the history bits with the selected portion of the address [51]. The number of history bits is usually small as the correlation among remote branches is very weak in most cases. In our case, the history is two bits. Figure 5.8(a) shows that the ILP solving times do not change substantially. The reason is that with a fixed history size and an increased prediction table size, the number of cases where two or more

branches have the same pattern starts to decrease. Since the constraints for each individual pattern are independent of the other patterns, the complexity of the ILP problem largely depends on how many branches can execute with the same pattern. Thus, ILP solution time does not increase significantly with the increase in size of the branch prediction table.

Figure 5.8(b) shows the solving times when the instruction cache size is varied. Again, we observe that the solving time does not change substantially. One of the reasons is that the constraints for each cache line is independent of the other cache lines. Thus, increasing the number of cache lines does not change the structure of the ILP problem.

5.4 Summary

In this chapter, we presented a framework to model dynamic branch predictions for WCET analysis. Our modeling can be targeted to various dynamic branch prediction schemes (which are used in both general-purpose and embedded processors [32, 53]). This ILP-based modeling is conveniently integrated with the ILP-based program path analysis. We also extended the branch prediction modeling to a combined analysis of branch prediction and instruction caching. The destructive/constructive effects of branch prediction on cache behavior are captured uniformly. Using our technique, we have obtained tight timing estimates for benchmark programs under various branch prediction schemes. This technique also scales up with regard to the increased size of the two hardware features: branch prediction table with larger size or larger instruction cache.

CHAPTER VI

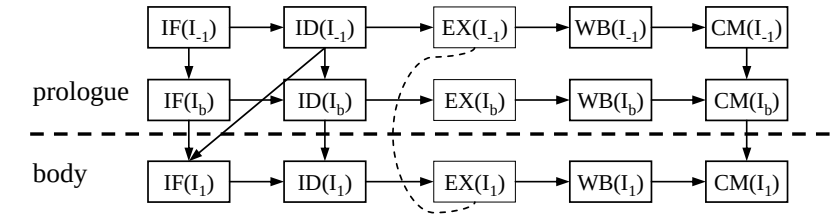
ANALYSIS OF PIPELINE, BRANCH PREDICTION AND INSTRUCTION CACHE

We have studied out-of-order pipelines for WCET analysis in Chapter 4 and branch prediction as well as its interaction with instruction caches in Chapter 5. In this chapter we integrate the timing effects of branch prediction and instruction caches with our out-of-order pipeline modeling. To achieve this, we need to study the impact of instruction caches and branch prediction on the pipeline. This involves changes in our estimation algorithm as well as the execution graph for each basic block (since a branch misprediction may execute additional code speculatively). These changes are now described.

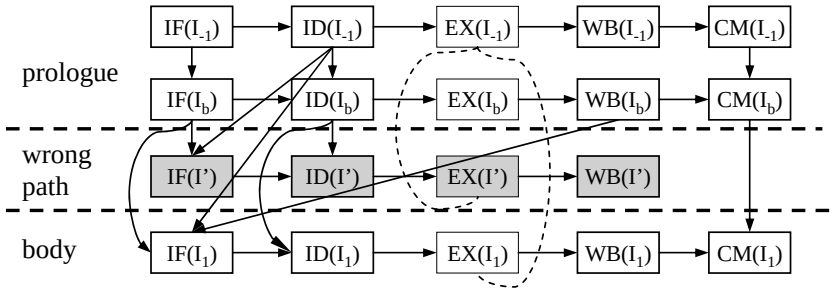
The rest of this chapter is organized as follows. First we describe how the WCET estimation of a basic block is affected by branch prediction (Section 6.1), and instruction cache (Section 6.2). Then, in Section 6.3 we describe the ILP formulation for WCET estimation of the whole program in presence of pipeline, cache and branch prediction. In Section 6.4 we show by experimentation that the combined analysis yields tight estimates. We conclude this chapter in Section 6.5.

6.1 Timing Estimation of a Basic Block in Presence of Branch Prediction

Clearly, if a branch is predicted correctly, then our pipeline analysis does not require any modification. However, a branch misprediction results in instructions along the wrong path being executed in the pipeline (without commit) and flushed out after the branch is resolved. This involves changes in the execution graph of a basic block.



(a) Original execution graph



(b) Modifications due to branch misprediction

Figure 6.1: Execution Graph with Branch Prediction

Before describing these changes, we make the following assumptions.

Assumptions First, we assume that the processor allows only *one unresolved branch* at any point of time during execution. Thus, if another branch is encountered during speculative execution, the processor simply waits till the previous branch is resolved. Second, we assume that the outcome of a branch is resolved upon the completion of its WB stage. If it is a misprediction, the wrong path instructions are flushed out and the processor resumes execution along the correct path immediately. Last, we assume that the branch prediction takes place at the end of the fetch stage. That is, the target address is available at the end of the fetch stage irrespective of whether a branch is predicted as taken or non-taken. In reality, this is easy for a non-taken prediction; but for a taken prediction, extra resources, such as branch target buffer, are needed to achieve this goal [30].

6.1.1 Changes to Execution Graph

We now describe the changes to the execution graph of a basic block in order to account for instructions executed due to branch misprediction; these instructions are also referred to as *wrong path instructions*. In particular, we discuss the changes to execution graph nodes, dependency relation and contention relation among nodes. Consider the execution graph of a basic block B with a body, a prologue and an epilogue. If the last instruction of the prologue is a branch b, we include instructions along the mispredicted path of b; otherwise no change is made to the execution graph.

A fragment of an execution graph without misprediction is shown in Figure 6.1(a) and the modified execution graph fragment due to the misprediction of branch b is shown in Figure 6.1(b).

Additional nodes in the execution graph A mispredicted branch brings the instructions along the wrong path into the pipeline. In order to capture their effect on the execution of normal instructions, we construct nodes corresponding to these wrong path instructions in the execution graph. Given a conditional branch b and its actual outcome X (non-taken or taken, denoted as 0 and 1, respectively), we can identify the maximum sequence of wrong path instructions that can enter the pipeline, called $Spec(b, X)$. The length of this sequence is bounded by two factors.

- $|Spec(b, X)| \leq ROB\_size + IBuffer\_size$ where ROB_size is the size of the re-order buffer and the $IBuffer\_size$ is the size of the Instruction fetch buffer
- If another conditional branch b' is encountered along the wrong path, then the sequence $Spec(b, X)$ is terminated at b'.

In Figure 6.1(b), the shaded nodes are the wrong path nodes (only one instruction is drawn for simplicity). There are no CM nodes for wrong path instructions as these instructions are not allowed to commit.

Changes to dependency relation Due to the changes in the execution graph nodes, the nodes can now be categorized as (a) prologue nodes (b) wrong path nodes (c) body nodes (this is the basic block being analyzed) and (d) epilogue nodes. The dependency edges among the nodes in each category are drawn as usual. However, the dependency edges among nodes in different categories require some explanation. First, we observe that the lifetimes of the wrong-path nodes and body nodes are **disjoint**. Hence we do not draw any dependency edges between wrong path nodes and body nodes. Instead we add a dependency edge between $WB(b)$ and $IF(I_1)$ where b is the branch in the prologue whose misprediction we are considering, and I_1 is the first instruction in the basic block being analyzed. This reflects the fact that instructions in the correct path (the body nodes) are fetched after the mispredicted branch is resolved. The dependency edges between the prologue and body nodes are drawn as usual, that is, they are not affected by the insertion of the wrong path nodes. This is because we do not make any assumptions about when the mispredicted branch is resolved.

Changes to contention relation Contention relation among prologue, body and epilogue nodes remain unchanged. We also consider contention of prologue and wrong path nodes in the estimation algorithm. Contention of body and wrong path nodes are not considered since the body nodes and wrong path nodes are guaranteed to have disjoint lifetimes.

6.1.2 Changes to Estimation Algorithm

As before, we use Algorithm 4 to estimate latest times of prologue nodes; earliest times of prologue nodes are conservatively estimated to $-\infty$. We still use Algorithm 2 to estimate the latest times and the Algorithm 3 to estimate the earliest times of the body and epilogue nodes in the modified execution graph. For the wrong path nodes, we use Algorithms 2, 3 to estimate the latest/earliest times but with one important

change. We observe that the wrong path nodes are flushed after branch b is resolved. Therefore, the latest ready, start, and finish times of all the wrong path nodes are additionally bounded by $latest[t^{finish}_{WB(b}]$.

6.1.3 Handling Prediction of Other Branches

So far we have discussed how to handle a mispredicted branch at the end of the prologue (i.e., if the last instruction before the current basic block is a mispredicted branch). However, the prologue and epilogue can contain multiple conditional branches if the basic blocks are too small. One possibility is to consider both the scenarios (correct and misprediction) for these conditional branches. However, this would require considering a large number of possibilities and is clearly very inefficient.

We observe that only the last conditional branch in the prologue has significant impact on the execution time of a basic block. Therefore, for this branch, we consider both the correct prediction and the misprediction scenarios and compute the execution time of the basic block accordingly. This leads to two possible WCET estimates of the basic block under the two scenarios.

We avoid enumerating correct/wrong prediction of other branches in the prologue or epilogue (*i.e.* any branch in the prologue or epilogue apart from the last branch of prologue) as follows. Consider any such branch b in the prologue or epilogue. We modify the execution graph such that correct as well as wrong prediction of b is considered. This is done by defining the special edge from the $WB(b)$ to the IF stage of the first instruction along the correct path as a **conditional** edge. This conditional edge is considered during the estimation of the latest times; but it is ignored in the estimation of earliest times. Similarly, all the wrong path nodes due to misprediction of b and their contentions are also considered to be **conditional**. They are considered during latest times calculations but are ignored for earliest times calculations. The intuition behind this approach is to take both possibilities of prediction (correct/wrong

prediction) into account so as to compute safe upper and lower bounds.

6.2 Timing Estimation of a Basic Block in Presence of Instruction Caching

We now perform combined analysis of pipelining, branch prediction and instruction caching. In our earlier discussions, we have assumed that there is no instruction cache and each instruction fetch takes a single clock cycle. We now discuss how we can capture the effects of instruction cache misses.

Given a cache configuration, a basic block B_i can be partitioned into a fixed number of memory blocks, with instructions in each memory block being mapped to the same cache line (cache accesses of instructions other than the first one in a memory block are always cache hits). Let the memory blocks be denoted as $B_{i.1}, B_{i.2}, \ldots, B_{i.n_i}$, where n_i is the number of memory blocks in B_i; *a cache scenario of B_i is defined as a mapping of hit or miss to each of the memory blocks of B_i.*

Now we study the changes to be made to the estimation of B_i under a particular cache scenario ω. First, it is obvious that the instruction cache only affects the latency of the instruction fetch (IF) stage, but does not affect data dependencies or contentions, thus no changes need to be made to the execution graph. Second, there is a slight change to the estimation algorithm. Recall when instruction cache was not modeled, the IF stage was assigned a single-cycle latency. Now the latency of IF stage is determined by the cache access result of an instruction. If it is a hit, then a single cycle is assigned; if it is a miss, a cache miss penalty, N is assigned; otherwise the access result is unknown and an interval $[1, N]$ is assigned, which covers both possibilities.

Note that for the context instructions (prologue and epilogue) of B_i we do not distinguish their cache scenarios. In other words, we conservatively assume the cache access results of the first instructions of the memory blocks in prologue/epilogue

are unknown. Thus, we assign the interval $[1, N]$ to the IF stage of each of those instructions. This policy is based on the observation that cache hits/misses in context instructions do not affect the execution time of B_i significantly.

In the preceding, we have clarified how to account for timing effects of instruction cache if we know the cache scenario, that is, whether the memory blocks of a basic block is in the cache. In reality, we need to consider all possible cache scenarios and bound the number of occurrences of the different cache scenarios under which a basic block may be executed. We accomplish this via Integer Linear Programming. In particular, we introduce ILP variables to capture the number of occurrences of any basic block B_i under (a) correct/wrong prediction of the preceding branch (b) a specific cache scenario to denote hit/miss of memory blocks of B_i. Constraints are imposed on these ILP variables to bound their values, thereby obtaining an estimate of the program's WCET.

6.3 Putting It All Together

We now describe the ILP formulation which integrates our analyses of pipelining, instruction caching and branch prediction. Let $B_1, \ldots, B_N$ be the set of basic blocks of the program whose WCET we are estimating. Now the execution of B_i is associated with the prediction of its preceding branch and its cache scenario. We denote the set of possible cache scenarios at B_i as Ω_i. For the possible cache scenarios Ω_i of B_i, the number of cache scenarios can be 2^{n_i} in the worst case, where n_i is the number of memory blocks of B_i. However, constrained by the program control flow, only a few scenarios are possible in reality. For better accuracy and less analysis time, it is necessary to exclude those infeasible ones. This can be achieved by a preprocessing step. The preprocessing traverses the program control flow graph by propagating and updating the cache states; at the entry of each basic block, distinct cache scenarios are collected. The preprocessing terminates until no new scenarios are found at any

basic block.

Considering the possible cache scenarios and correct/wrong prediction of the preceding branch for a basic block, the ILP objective function denoting a program's WCET is now written as follows. Note that Ω_i denotes the set of possible cache scenarios at B_i.

$$Time = \sum_{i=1}^{N} \sum_{j \to i} \sum_{\omega \in \Omega_i} \left(costc^{\omega}_{j \to i} \times ec^{\omega}_{j \to i} + costm^{\omega}_{j \to i} \times em^{\omega}_{j \to i} \right) \tag{6.1}$$

where $costc^{\omega}_{j \to i}$ is the WCET of B_i executed under the following contexts: (1) B_i is reached from a preceding block B_j (2) the branch prediction at the end of B_j is correct or B_j does not have a conditional branch (3) B_i is executed under a cache scenario $\omega \in \Omega_i$; $ec^{\omega}_{j \to i}$ is the number of times that B_i is executed under these contexts. Similarly, $costm^{\omega}_{j \to i}$ is the WCET of B_i executed under the following contexts: (1) B_i is reached from a preceding block B_j (2) the branch at the end of B_j is mispredicted (3) B_i is executed under a cache scenario $\omega \in \Omega_i$; $em^{\omega}_{j \to i}$ is the number of times that B_i is executed under these contexts.

Using our out-of-order pipeline analysis in Chapter 4 as well as the extensions proposed in Section 6.1 and Section 6.2, we can estimate the WCET of a basic block provided the correct/wrong prediction of the preceding branch and the cache scenario is known. In other words, we can estimate $costc^{\omega}_{j \to i}$ and $costm^{\omega}_{j \to i}$ as constants. We now need to develop constraints to bound the ILP variables $ec^{\omega}_{j \to i}$ and $em^{\omega}_{j \to i}$.

In Chapter 5, we have proposed an ILP-based branch prediction modeling technique, which can bound correct branch predictions and misprdictions. For instance, the count of mispredictions of a basic block B_i along one edge $i \to j$ was denoted as a variable $em_{j \to i}$ and was bounded by constraints (5.2)-(5.5) in Section 5.1.1. The correct predictions along the same edge, which was not explicitly defined in Chapter 5, can be straightforwardly defined as $ec_{j \to i} = e_{j \to i} - em_{j \to i}$.

Now we observe that $ec^{\omega}_{j \to i}$ and $em^{\omega}_{j \to i}$ are refined forms of $ec_{j \to i}$ and $em_{j \to i}$ where block B_i's executions are further distinguished with cache scenarios at B_i. This leads

to the following equations.

$$ec_{j\to i} = \sum_{\omega\in\Omega_i} ec^{\omega}_{j\to i}; \quad em_{j\to i} = \sum_{\omega\in\Omega_i} em^{\omega}_{j\to i} \tag{6.2}$$

On the other hand, with the ILP-based instruction cache modeling by Li et al. [43] as well as our modification to it in Section 5.2, we can further bound the occurrences of cache scenarios by relating them to the cache hits/misses of memory blocks. Recall in Section 5.2, the cache miss count for a memory block $B_{i.k}$ was denoted as $cm_{i.k}$ (cache hit count for $B_{i.k}$, which was not explicitly defined, can be straightforwardly defined as $ch_{i.k} = v_i - cm_{i.k}$, where v_i is the execution count for both the basic block B_i and the memory block $B_{i.k}$). Since a cache scenario ω of B_i is an assignment of hit or miss to each of B_i's memory blocks, we can partition Ω_i, the set of possible cache scenarios at B_i as

$$\Omega_i = \Omega^h_{i.k} \cup \Omega^m_{i.k} \quad \Omega^h_{i.k} \cap \Omega^m_{i.k} = \phi$$

$\Omega^h_{i.k}$ ($\Omega^m_{i.k}$) is the set of those cache scenarios in Ω_i in which memory block $B_{i.k}$ results in a hit (miss). Given Ω_i, the sets $\Omega^h_{i.k}$ and $\Omega^m_{i.k}$ can be computed straightforwardly. We can now state the following.

$$ch_{i.k} = \sum_{\omega\in\Omega^h_{i.k}} \sum_{j\to i} \left(ec^{\omega}_{j\to i} + em^{\omega}_{j\to i}\right); \quad cm_{i.k} = \sum_{\omega\in\Omega^m_{i.k}} \sum_{j\to i} \left(ec^{\omega}_{j\to i} + em^{\omega}_{j\to i}\right); \tag{6.3}$$

With constraints in Equations 6.2 and 6.3, $ec^{\omega}_{j\to i}$ and $em^{\omega}_{j\to i}$ can be effectively bounded.

Finally, the objective function in Equation 6.1 can be maximized by the ILP solver subject to (1) the control flow constraints, (2) the branch prediction modeling constraints, (3) the instruction cache modeling constraints, and (4) the constraints presented in this section.

6.4 Experimental Evaluation

In this section, we evaluate the accuracy of our estimation technique for the same benchmarks used in the earlier chapters. The configurations of the pipeline, the

Program	WCET		Ratio	Analysis	Solving
	Obs.	Est.		time (sec.)	time (sec.)
des	70342	89402	1.27	1.57	2.48
dhry	178033	224300	1.26	3.22	3.80
fdct	14635	15368	1.05	0.30	0.06
fft	1213159	1385034	1.14	1.21	0.21
fir	50600	64305	1.27	2.40	1.66
isort	46492	61733	1.33	0.13	0.01
ludcmp	12560	16143	1.29	1.24	0.27
matmul	15198	19097	1.26	0.05	0.01
matsum	101655	111758	1.10	0.04	0.01
minver	8518	11032	1.30	2.43	4.27
qurt	2147	2767	1.29	1.36	0.81
whet	890353	1063710	1.19	1.19	0.86

Table 6.1: Combined Analysis of Out-of-Order Pipelining, Branch Prediction and Instruction Caching

instruction cache and the branch predictor are the same as in the earlier chapters except for the branch predictor no branch misprediction penalty is explicitly given. This is because the misprediction penalties are now accounted for by the pipelined execution, and how long a misprediction of a branch b lasts is dependent on when $WB(b)$ completes, which means the misprediction penalties are no longer constants (as assumed in Chapter 5).

The experimental results are given in Table 6.1. As the **ratio** column indicates, the combined analysis yields tight estimations. To further illustrate the effectiveness of the technique for the combined analysis, we compare the overestimation to that in the pure pipeline modeling in Figure 6.2. It clearly shows that the combined analysis does not produce significantly more overestimation than the pure pipeline modeling does. This justifies our modifications to the pipeline analysis algorithms which take into account mispredictions and cache misses. We also observed that for some of the benchmarks such as `des`, `fdct`, and `fft`, the overestimation in the combined analysis is even less than the pure pipeline modeling. This can be explained that with the occurrences of cache misses and branch mispredictions in the pipelined execution,

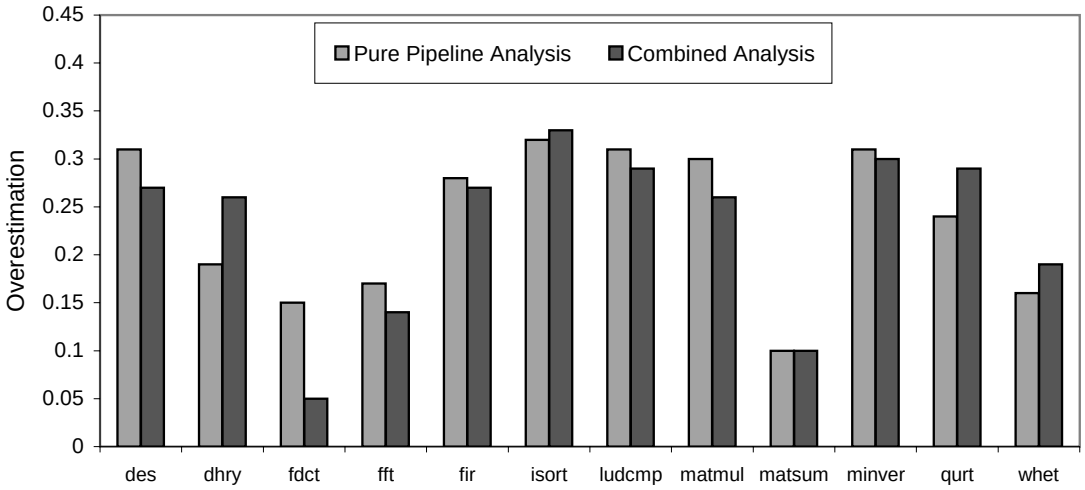


Figure 6.2: Overestimations in the Pure Pipeline Analysis and Overestimations in the Combined Analysis

instructions might have lower chances to contend with each other. For example, with a cache miss, it might be possible to determine by our estimation algorithms that for an instruction I preceding the cache miss and another one I' following it, $separated[EX(I), EX(I')] = false$, which means I and I' cannot contend with each other in any case. This fact might not hold without the cache miss. As we learned from Chapter 4, less contentions can lead to more accurate estimates.

On the other hand, indicated by the **Analysis time** and **Solving time** columns, the combined analysis can be performed very efficiently. For example, the increase of ILP solving time to that in the branch prediction analysis (refer to Table 5.4) is not very significant. This is because the main difference between the combined analysis and branch prediction analysis is how the cost of a basic block is determined, while the differences between the ILP problems in the two analyses are not so substantial.

6.5 Summary

This chapter presents a combined analysis of out-of-order pipelining, branch prediction and instruction caching. We achieve this by studying the timing effects of branch

prediction and instruction caching on the pipeline, and we extend the pipeline analysis algorithms to capture these effects. For branch prediction, we add additional nodes and edges corresponding to speculatively executed code into the execution graph, and make slight changes to the estimation algorithms to take care of the difference of speculative execution. For instruction caching, the changes made to pipeline analysis is even more straightforward: the only change is that the original single-cycle latencies of the instruction fetches are changed to latencies corresponding to cache hits, cache misses or both possibilities. The insignificant modifications made to the pure pipeline analysis suggest a good extensibility of our pipeline analysis framework.

CHAPTER VII

CONCLUSION

This chapter concludes the thesis. In Section 7.1 we summarize the contribution of this thesis and in Section 7.2 we discuss some future directions.

7.1 Summary of the Thesis

Worst Case Execution Time (WCET) prediction has been a fundamental problem for hard real-time systems. Typically, the WCET of a task is hard to predict by running the task because all possible sets of data input have to be evaluated to guarantee that the worst case is covered. As a result, static Worst Case Execution Time analysis, which predicts the maximum running time of the program without actually running it, has become a promising alternative approach and extensive research has been conducted in this direction. In general, it consists of three subtasks: (1) program path analysis, which identifies feasible/infeasible program paths; (2) microarchitecture modeling, which models the timing effects of hardware features to determine instruction timing; and (3) WCET calculation, which calculates the WCET of the program with program path information and instruction timing information. Among them, microarchitecture modeling has become an increasingly important yet difficult task mainly because modern processors have employed aggressive microarchitectural features for the quest of higher performance.

In this thesis, we study the core microarchitectural features of modern processors, namely out-of-order pipelines, dynamic branch predictions and instruction caching. We have developed a microarchitecture modeling framework which models the above three features in combination. The framework consists of two levels: the local level

analyses estimate the worst case execution time of a basic block under a specific execution context, while the global level analyses are responsible for identifying execution contexts for basic blocks and bounding the occurrences of these contexts. This way, we can estimate the WCET of the whole program by summing up the execution times of basic blocks under different execution contexts. Under this framework, we have developed analytical techniques for the individual microarchitectural features and have proposed a method for combining all them together.

First, for out-of-order pipelines, we have proposed an innovative technique to address a phenomenon called *timing anomaly* [50]. In the presence of timing anomaly, techniques which generally take the local worst case for WCET estimation no longer guarantee safe bounds. This prompts the need to consider all possible local cases and their subsequent executions. However, a naive approach which enumerates the possible cases individually is often expensive in terms of both the analysis time and resource needs. Our technique avoids enumeration for individual cases. The key point of this technique is a fixed-point analysis of time intervals at which the instructions can enter/leave the pipeline stages. Experimental results have shown that this technique yields accurate results and works efficiently.

Second, for dynamic branch predictions, we have proposed an Integer Linear Programming based framework to bound branch mispredictions. The branch prediction analysis is integrated with the ILP based WCET calculation. We follow this strategy because branch prediction exhibits a strong global nature, that is, the prediction is based on the executions of earlier branches whose distance to current branch could be either near or far away. As a result, global program flow information is needed, which can be provided by the ILP based WCET calculation. This ILP-based framework is parameterized and can be straightforwardly targeted to a variety of branch prediction schemes. Apart from branch prediction modeling, we have also studied the effect of speculative execution (via branch prediction) on instruction caching. The effect is

captured by modifications to an existing ILP based instruction cache analysis [43].

Last, we have combined the analyses of the three features: out-of-order pipelines, branch prediction and instruction caching. We do so by studying the timing effects of branch prediction and instruction caching on the pipeline and making modifications to the pure pipeline analysis algorithms to capture their effects. The modifications are not substantial and the combined analysis works efficiently, suggesting a good extensibility of our framework for modeling more microarchitectural features.

7.2 Future Work

We have identified the following directions to be pursued in the future.

Integration with program path analysis There has been substantial program path analysis work in the literature. By excluding infeasible paths with the help of program path analysis, the accuracy of WCET analysis can be significantly improved. We will try to adopt some program path analysis techniques into our framework.

Data cache analysis Data cache is another important feature in current processors. Unlike instruction cache, whose behavior is only determined by the program flow, the behavior of data cache is affected by both the program flow and data values. As a result, techniques which exploit control flow information for instruction cache analysis are not sufficient in the context of data cache; and we need to develop new methods to model it.

Analysis for real-life processors We would like to extend our work to real-life superscalar processors which essentially have the three components we have addressed. Working on a real-life processor is more challenging and the experience of working on it may inspire us with new ideas or improvements to existing ones.

WCET optimization Its purpose is to reduce the estimated WCET by program transformation. There has been some research activities in this direction. Zhao et al. [76, 75] optimize the WCET of a program by code positioning or by optimizing the worst case path using compiler optimizations like path duplication and loop unrolling. Bodin and Puaut [5] propose a WCET-oriented static branch prediction algorithm for processors supporting compiler-directed branch prediction.

Integrating the timing analyzer with the compiler Timing analyzer needs both high level source code information and low level object code information. For example, the analyzer users may want to give program path information at the source code level and want the compiler to transform it into object code level representation. The compiler can also yield its results of data flow analysis for the timing analyzer, such as loop bounds or infeasible paths. The key issue for integrating the timing analyzer with the compiler is to develop a standard interface between the two parties, such that when the timing analyzer is targeted to a new compiler, both the modifications to the compiler and the timing analyzer are minimal and can be easily done.

APPENDIX A

PROOFS FOR THE PIPELINE ANALYSIS ALGORITHMS

In Chapter 4, we have presented how the algorithms (Algorithm 1, 2, 3 and 4) produce estimates for the worst case costs of the basic blocks. Intuitively, they start with conservative timing intervals for the executions of instructions and iteratively tighten the intervals until a fixed-point is reached. In this appendix, we give a formal proof for their correctness, that is, the calculated intervals indeed cover all possible execution times of instructions and the worst case costs of basic blocks are not underestimated.

A.1 Proofs for the Context-Free Estimation

In this section we prove the correctness for the algorithms in Section 4.2.1 where we do not consider the execution context of a basic block. We want prove that the estimated WCET for a basic block is no less than any possible execution times of that basic block by showing that the *latest* times and *earliest* times calculated by Algorithm 2 and Algorithm 3 for the execution graph nodes are indeed the upper and lower bounds of their corresponding execution times.

Lemma A.1. *Let u and v be two contending nodes in the execution graph with $u \in late\_contenders(v)$, and let S_{late} be the set of late contenders of v computed by Algorithm 2. If in a particular run, u delays v, and the relationship of the execution times with the calculated earliest and latest times by our algorithms is that*

$\forall w \in \{u, v\}$,

$$\begin{aligned} earliest[t_w^{ready}] &\leq t_w^{ready} \leq latest[t_w^{ready}] \\ earliest[t_w^{start}] &\leq t_w^{start} \leq latest[t_w^{start}] \\ earliest[t_w^{finish}] &\leq t_w^{finish} \leq latest[t_w^{finish}] \end{aligned} \tag{A.1}$$

then $u \in S_{late}$, which means the actual late contender delaying v is in the calculated set of late contenders.

Proof. Since u is a late contender delaying v, $t_u^{start} < t_v^{ready} < t_u^{finish}$. Now we prove the lemma in two steps. First, we show that $separated(u, v) = false$. By definition, $separated(u, v) = true$ must satisfy the following inequalities

$$earliest[t_u^{ready}] \geq latest[t_v^{finish}] \ \vee \ earliest[t_v^{ready}] \geq latest[t_u^{finish}]$$

Now we prove that neither of them can be true. With $t_u^{ready} < t_v^{ready}$ and (A.1),

$$earliest[t_u^{ready}] \leq t_u^{ready} < t_v^{ready} < t_v^{finish} \leq latest[t_v^{finish}]$$

Similarly,

$$earliest[t_v^{ready}] \leq t_v^{ready} < t_u^{finish} \leq latest[t_u^{finish}]$$

Combine the above two, $separated(u, v) = false$.

Second, we show that $earliest[t_u^{start}] < latest[t_v^{ready}]$. This is true because

$$earliest[t_u^{start}] \leq t_u^{start} < t_v^{ready} \leq latest[t_v^{ready}]$$

Therefore, following the calculation of S_{late} in Algorithm 2, $u \in S_{late}$.

□

Lemma A.2. *Let v be a node in the execution graph and let S_{early} be its early contenders computed by Algorithm 2. If in a particular run, the actual early contenders delaying v are S'_{early}, and the inequalities in (A.1) are true for v and S'_{early} here, then $S'_{early} \subseteq S_{early}$, which means the actual early contenders delaying v are included in the set of early contenders calculated by Algorithm 2.*

Proof. Since every $u \in S'_{early}$ is an early contender delaying v, $t_u^{ready} < t_v^{finish}$ and $t_v^{ready} < t_u^{finish}$. With (A.1), we have

$$earliest[t_u^{ready}] \leq t_u^{ready} < t_v^{finish} \leq latest[t_v^{finish}]$$

and

$$earliest[t_v^{ready}] \leq t_v^{ready} < t_u^{finish} \leq latest[t_u^{finish}]$$

which means $separated(u, v) = false$. Thus $u \in S_{early}$ and $S'_{early} \subseteq S_{early}$.

□

Theorem A.3. *For every node v in the execution graph, the following relationship between the actual execution times of v and its earliest/latest times calculated by Algorithms 2 and 3 in each iteration of Algorithm 1 is true.*

$$earliest[t_v^{ready}] \leq t_v^{ready} \leq latest[t_v^{ready}] \tag{A.2}$$

$$earliest[t_v^{start}] \leq t_v^{start} \leq latest[t_v^{start}] \tag{A.3}$$

$$earliest[t_v^{finish}] \leq t_v^{finish} \leq latest[t_v^{finish}] \tag{A.4}$$

Proof. We prove it by induction. Assume (A.2 - A.4) are true for all nodes in previous iterations and for the nodes earlier than v in topologically sorted order in current iteration. We show that (A.2 - A.4) are also true for v in current iteration.

Obviously, the base case is true since the latest times are initialized as ∞ and earliest times are initialized as 0 or minimum latencies (for finish events). For the induction case, we take the latest times for discussion.

For v's ready time, let its predecessors be $DE(v) = \{u \mid (u, v) \in DE\}$, then $t_v^{ready} = \max_{u \in DE(v)} \left(t_u^{finish}\right)$. On the other hand, by Algorithm 2 (Lines 12 - 13), $latest[t_v^{ready}] = \max_{u \in DE(v)} \left(latest[t_u^{finish}]\right)$. By induction, $\forall u \in DE(v)$, $t_u^{finish} \leq latest[t_u^{finish}]$, therefore

$$t_v^{ready} = \max_{u \in DE(v)} \left(t_u^{finish}\right) \leq \max_{u \in DE(v)} \left(latest[t_u^{finish}]\right) = latest[t_v^{ready}]$$

For v's start time, let the late contender delaying v, if any, be w and its delay to v be d_1 cycles; let the early contenders delaying v, if any, be S'_{early} and their delays to v be d_2 cycles (note d_1 must happen before d_2 as w can only delay v by starting execution before v is ready). Then $t_v^{start} = t_v^{ready} + d_1 + d_2$. For d_1,

$$t_v^{ready} + d_1 \leq \min\left(t_w^{finish},\ t_v^{ready} + max\_lat_v - 1\right)$$

According to Lemma A.1, $w \in S_{late}$, along with the induction assumption, we can derive the following from above inequality

$$t_v^{ready} + d_1 \leq \min\left(\max_{u \in S_{late}}\left(latest[t_u^{finish}]\right),\ latest\left[t_v^{ready}\right] + max\_lat_v - 1\right)$$

which means

$$t_v^{ready} + d_1 \leq latest[t_v^{start}]' \tag{A.5}$$

where $latest[t_v^{start}]'$ is the intermediate latest start time computed on Line 6 in Algorithm 2. Next, for d_2, suppose each $u \in S'_{early}$ delays v for d_u cycles (where $d_u \leq max\_lat_u = max\_lat_v$). Then $d_2 = \sum_{u \in S'_{early}} d_u \leq \left|S'_{early}\right| \times max\_lat_v$. According to Lemma A.2, $S'_{early} \subseteq S_{early}$. Thus

$$d_2 \leq |S_{early}| \times max\_lat_v \tag{A.6}$$

Now we examine t_v^{start} under two cases: $d_2 = 0$ and $d_2 > 0$.

In the first case, $t_v^{start} = t_v^{ready} + d_1$, and according to (A.5), $t_v^{start} \leq latest[t_v^{start}]'$. Compare to the $latest[t_v^{start}]$ calculated on Line 10 in Algorithm 2, $t_v^{start} \leq latest[t_v^{start}]$.

In the second case, one implication is that t_v^{start} cannot be later than the finish time of the last one who delays v, thus

$$t_v^{start} \leq max_{u \in S'_{early}}\left(t_u^{finish}\right)$$

Since $S'_{early} \subseteq S_{early}$ and $t_u^{finish} \leq latest[t_u^{finish}]$ (by induction), we can derive from above inequality the following

$$t_v^{start} \leq max_{u \in S_{early}}\left(latest\left[t_u^{finish}\right]\right) \tag{A.7}$$

On the other hand, by applying (A.5) and (A.6),

$$\begin{aligned} t_v^{start} &= t_v^{ready} + d_1 + d_2 \\ &\leq latest[t_v^{start}]' + d_2 \\ &\leq latest[t_v^{start}]' + |S_{early}| \times max\_lat_v \end{aligned} \tag{A.8}$$

Combine (A.7) and (A.8),

$$t_v^{start} \leq \min\left(max_{u \in S_{early}}\left(latest\left[t_u^{finish}\right]\right),\ \ latest[t_v^{start}]' + |S_{early}| \times max\_lat_v\right) \tag{A.9}$$

in which the right hand side corresponds to *tmp* on Line 9 in Algorithm 2. Compare to $latest[t_v^{start}]$ calculated on Line 10, $t_v^{start} \leq latest[t_v^{start}]$.

For v's finish time, suppose v executes for lat_v ($\leq max\_lat_v$) cycles, $t_v^{finish} \leq latest[t_v^{finish}]$ simply because

$$\begin{aligned} t_v^{finish} &= t_v^{start} + lat_v \\ &\leq latest[t_v^{start}] + max\_lat_v \end{aligned}$$

Thus we have proved that the latest times calculated by Algorithm 2 indeed provide upper bounds for the actual execution times of the nodes in the execution graph. Similarly, we can prove that the earliest times calculated by Algorithm 3 indeed provide lower bounds.

□

With Theorem A.3, we can claim that the WCET of a basic block estimated by the algorithms (1, 2 and 3) in Section 4.2.1 is a safe upper bound of the possible execution times of that basic block. This is because the estimated WCET is $latest\left[t_{CM(I_n)}^{finish}\right]$, which by Theorem A.3 is no less than the actual execution time, $t_{CM(I_n)}^{finish}$.

A.2 Proofs for the Context-Inclusive Estimation

In this section we prove the correctness for the algorithms in Section 4.2.2 where we take the execution context of a basic block into account. We want to prove that the estimated WCET for a basic block is no less than any possible execution times of that basic block. Recall the execution time is estimated as $latest\left[t_{CM(I_n)}\right]^{finish} - \delta$. If we can show that $latest\left[t_{CM(I_n)}\right]^{finish}$ is not underestimated and δ, the overlap, is not overestimated, then the estimated execution time is correct. The correctness of overlap estimation has been guaranteed by Theorem 4.1. Therefore we only need to prove the correctness of the estimation of $latest\left[t_{CM(I_n)}\right]^{finish}$. We do this by proving that for any node (prologue, body or epilogue), the estimated latest and earliest times are indeed upper and lower bounds for its actual execution times.

We first show that the execution times of the prologue nodes are correctly bounded. Algorithm 4.3 estimating the prologue nodes consists of two parts: one part for the estimation of the shaded nodes, which have paths to $IF(I_1)$, the fetch of the first instruction in the body; and the other part for the rest prologue nodes. The correctness of the first part has already been guaranteed by Inequality 4.2. The second part is very similar to Algorithm 2, with one extra bound $latest\left[t^{ready}_{CM(I_{-p})}\right]$ on Line 10 and a maximized estimated delay from the late contender on Line 11. Now we only need to prove the correctness for the two differences because the proof for the rest of the algorithm can follow that in the previous section.

Lemma A.4. *Suppose for each prologue node preceding an unshaded node v in topologically sorted order, its latest and earliest times provide upper and lower bounds for its execution times. Then, v's latest ready time calculated by Lines 9 and 10 in Algorithm 4 gives an upper bound on its actual ready time. That is, $t^{ready}_v \leq latest\left[t^{ready}_v\right]$*

Proof. Let the immediate predecessors of v (those with a dependence edge to v), denoted as $DE(v)$, be partitioned into two parts: those in the prologue, denoted as

$DE_1(v)$; and those in the pre-prologue (which are not known), denoted as $DE_2(v)$. Thus,

$$\begin{aligned} t_v^{ready} &= \max_{uDE(v)} \left(t_u^{finish}\right) \\ &= \max\left(\max_{u \in DE_1(v)} \left(t_u^{finish}\right), \max_{u \in DE_2(v)} \left(t_u^{finish}\right) \right) \end{aligned} \tag{A.10}$$

First,

$$\max_{u \in DE_1(v)} \left(t_u^{finish}\right) \leq \max_{u \in DE_1(v)} \left(latest\left[t_u^{finish}\right]\right) \tag{A.11}$$

Second, all nodes in $DE_2(v)$ are pre-prologue nodes and they should have completed execution when the last pre-prologue node $CM(I_{-p})$ becomes ready, therefore

$$\max_{u \in DE_2(v)} \left(t_u^{finish}\right) \leq t_{CM(I_{-p})}^{ready} \leq latest\left[t_{CM(I_{-p})}^{ready}\right] \tag{A.12}$$

Combine (A.10), (A.11) and (A.12),

$$t_v^{ready} \leq \max\left(\max_{u \in DE_1(v)} \left(latest\left[t_u^{finish}\right]\right), latest\left[t_{CM(I_{-p})}^{ready}\right] \right)$$

The above right hand side is equal to the $latest\left[t_v^{ready}\right]$ calculated by Lines 9 and 10. Therefore we proved $t_v^{ready} \leq latest\left[t_v^{ready}\right]$.

□

The correctness of Line 11 for bounding delay from an early contender is obvious – the maximum delay $max\_lat_v - 1$ is assumed.

Theorem A.5. *For every node v in the execution graph including the prologue, body and epilogue, Inequalities (A.2 - A.4) are satisfied. In other words, the estimated latest and earliest times indeed provide upper and lower bounds for the actual execution times.*

Proof. For the prologue nodes, the correctness of the only differences between Algorithm 2 and Algorithm 4 has been proved by Lemma A.4, and the proof for the rest of Algorithm 4 is the same as the proof for Algorithm 2 in last section. Similarly, the

estimation algorithms for body nodes and epilogue nodes are exactly the same as in last section whose correctness has already been proved. Thus Inequalities (A.2 - A.4) hold.

□

It can be proved straightforwardly from Theorem A.5 that $latest\left[t^{ready}_{CM(I_n)}\right]$ is an upper bound to the actual $t^{ready}_{CM(I_n)}$. Since the estimated overlap δ has been proved earlier to be a lower bound to the actual overlap, the estimated execution time $latest\left[t^{finish}_{CM(I_n)}\right] - \delta$ is an upper bound to the actual execution time.

REFERENCES

[1] Aho, A., S. R. U. J., *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Altenbernd, P., "On the false path problem in hard real-time programs," in *8th Euromicro Workshop on Real Time Systems (WRTS)*, 1996.

[3] Arnold, R., Mueller, F., Whalley, D., and Harmon, M., "Bounding worst-case instruction cache performance," in *IEEE Real-Time Systems Symposium*, 1994.

[4] Bate, I. and Reutemann, R., "Worst-case timing analysis for dynamic branch predictors," in *30th EuroMicro Conference*, 2004.

[5] Bodin, F. and Puaut, I., "A WCET-oriented static branch prediction scheme for real-time systems," in *Proc. of the 17th Euromicro Conference on Real-Time Systems*, (Palma de Mallorca, Spain), July 2005.

[6] Burger, D. and Austin, T., "The SimpleScalar Tool Set, Version 2.0," Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[7] Char, B., Geddes, K., Gonnet, G., Leong, B., Monagan, M., and Watt, S., *Maple V Language Reference Manual*. Springer-Verlag, 1991.

[8] Chen, K., Malik, S., and August, D., "Retargatable static software timing analysis," in *IEEE/ACM Intl. Symp. on System Synthesis (ISSS)*, 2001.

[9] Colin, A. and Puaut, I., "Worst case execution time analysis for a processor with branch prediction," *Journal of Real time Systems*, May 2000.

[10] Colin, A. and Puaut, I., "A modular and retargetable framework for tree-based WCET analysis," in *Proc. of the 13th Euromicro Conference on Real-Time Systems*, (Delft, The Netherlands), pp. 37–44, June 2001.

[11] Colin, A. and Puaut, I., "A modular and retargetable framework for tree-based WCET analysis," Tech. Rep. 0, IRISA, March 2001.

[12] Combs, J., Combs, C., and Shen, J., "Mispredicted path cache effects," in *In Euro-Par Conference*, 1999.

[13] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., *Introduction to Algorithms (Second Edition)*. MIT Press, 2001.

[14] Cousot, P. and Cousot., R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.," in *ACM Symposium on Principles of Programming Languages*, 1977.

[15] CPLEX, "The ILOG CPLEX Optimizer v7.5," 2002. Commercial software, http://www.ilog.com.

[16] Engblom, J., *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, 2002.

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

[18] Engblom, J. and Ermedahl, A., "Modeling complex flows for worst-case execution time analysis," in *IEEE Real-Time Systems Symposium*, 2000.

[19] Engblom, J., Ermedahl, A., and Altenbernd, P., "Facilitating worst-case execution times analysis for optimized code," in *Proceedings of the 10th Euromicro Real-Time Systems Workshop*, 1998.

[20] Ermedahl, A. and Gustafsson, J., "Deriving annotations for tight calculation of execution time," in *European Conference on Parallel Processing*, 1997.

[21] Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., and Wilhelm, R., "Reliable and precise WCET determination for a real-life processor," in *Intl. Workshop on Embedded Software (EmSoft)*, 2001.

[22] Ferdinand, C. and Wilhelm, R., "Fast and Efficient Cache Behavior Prediction for Real-Time Systems," *Real-Time Systems*, vol. 17, no. (2/3), 1999.

[23] Fields, B., Bodik, R., and Hill, M., "Slack: Maximizing performance under technological constraints," in *29th ACM Annual International Symposium on Computer architecture*, 2002.

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

[25] Healy, C., Sjodin, M., Rustagi, V., and Whalley, D., "Bounding loop iterations for timing analysis," in *IEEE Real-time Appplications Symposium (RTAS)*, 1998.

[26] Healy, C., Sjodin, M., Rustagi, V., Whalley, D., and Engelen, R., "Supporting timing analysis by automatic bounding of loop iterations," *Real-Time Systems*, vol. 18, no. 2/3, pp. 129–156, 2000.

[27] Healy, C. and Whalley, D., "Automatic detection and exploitation of branch constraints for timing analysis," *IEEE Transaction on Software Engineering*, vol. 28, no. 8, 2002.

[28] Healy, C., Whalley, D., and Harmon, M., "Integrating the timing analysis of pipelining and instruction caching," in *IEEE Real-Time Systems Symposium (RTSS)*, 1995.

[29] Heckmann, R., Langenbach, M., Thesing, S., and Wilhelm, R., "The Influence of Processor Architecture on the Design and the Results of WCET Tools," *Proceedings of the IEEE*, vol. 91, July 2003.

[30] Hennessy, J. and Patterson, D., *Computer Architecture- A Quantitative Approach*. Morgan Kaufmann, 1996.

[31] Hur, Y., Bae, Y. H., Lim, S.-S., Kim, S.-K., Rhee, B.-D., Min, S. L., Park, C. Y., Shin, H., and Kim, C. S., "Worst case timing analysis of RISC processors: R3000/r3010 case study," in *IEEE Real-Time Systems Symposium (RTSS)*, 1995.

[32] Inc., S., "SiByte SB-1 MIPS64 embedded CPU Core," in *Embedded Processor Forum*, 2000.

[33] Kirner, R. and Puschner, P., "Transformation of path information for WCET analysis during compilation," in *13th Euromicro Conference on Real-Time Systems*, 2001.

[34] Kirner, R. and Puschner, P., *Extending Optimising Compiliation to Support Worst-Case Execution Time Analysis*. PhD thesis, Vienna University of Technology, 2003.

[35] Kligerman, E. and Stoyenko, A. D., "Real-time euclid: a language for reliable real-time systems," *IEEE Trans. Softw. Eng.*, vol. 12, no. 9, pp. 941–949, 1986.

[36] Langenbach, M., Thesing, S., and Heckmann, R., "Pipeline modeling for timing analysis," in *Static Analysis Symposium (SAS)*, 2002.

[37] Li, X., Mitra, T., and Roychoudhury, A., "Accurate timing analysis by modeling caches, speculation and their interaction," in *ACM Design Automation Conf. (DAC)*, 2003.

[38] Li, X., Mitra, T., and Roychoudhury, A., "Modeling control speculation for timing analysis," *Journal of Real-Time Systems*, vol. 29, no. 1, 2005.

[39] Li, X., Roychoudhury, A., and Mitra, T., "Modeling out-of-order processors for software timing analysis," in *IEEE Real-Time Systems Symposium*, 2004.

[40] Li, Y.-T. S. and Malik, S., "Performance analysis of embedded software using implicit path enumeration," in *Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[41] Li, Y.-T. S., Malik, S., and Wolfe, A., "Efficient microarchitecture modeling and path analysis for real-time software," in *Proceeding of the IEEE Real-Time Systems Symposium*, 1995.

[42] Li, Y.-T. S., Malik, S., and Wolfe, A., "Cache modeling for real-time software: Beyond direct mapped instruction caches," in *Proceeding of the IEEE Real-Time Systems Symposium*, 1996.

[43] Li, Y.-T. S., Malik, S., and Wolfe, A., "Performance estimation of embedded software with instruction cache modeling," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 3, 1999.

[44] Lim, S.-S., Bae, Y., Jang, G., Rhee, B.-D., Min, S., Park, C., Shin, H., Park, K., and Kim, C., "An accurate worst-case timing analysis technique for

RISC processors," *IEEE Transactions on Software Engineering*, vol. 21, no. 7, 1995.

[45] Lim, S.-S., Bae, Y., Jang, G., Rhee, B., Min, S., Park, C., Shin, H., Park, K., and Kim, C., "An accurate worst case timing analysis technique for risc processors," in *IEEE Real-Time Systems Symposium*, 1994.

[46] Lim, S.-S., Han, J., Kim, J., and Min, S., "A worst case timing analysis technique for multiple-issue machines," in *IEEE Real Time Systems Symposium (RTSS)*, pp. 334–345, 1998.

[47] Liu, Y. and Gomez, G., "Automatic time-bound analysis for a higher-order language," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 1998.

[48] Liu, Y. and Gomez, G., "Automatic accurate cost-bound analysis for high-level languages," *IEEE Transactions on Computers*, vol. 50, no. 12, 2001.

[49] Lundqvist, T. and Stenström, P., "An integrated path and timing analysis method based on cycle-level symbolic execution," *Journal of Real-Time Systems*, vol. 17, no. 2-3, 1999.

[50] Lundqvist, T. and Stenström, P., "Timing anomalies in dynamically scheduled microprocessors," in *IEEE Real-Time Systems Symposium*, 1999.

[51] McFarling, S., "Combining branch predictors," tech. rep., DEC Western Research Laboratory, 1993.

[52] McMillan, K. and Dill, D., "Algorithms for interface timing verification," in *IEEE International Conference on Computer Design*, 1992.

[53] Microelectronics, I., "PowerPC 440GP Embedded Processor," in *Embedded Processor Forum*, 2001.

[54] Mitra, T., Roychoudhury, A., and Li, X., "Timing analysis of embedded software for speculative processors," in *ACM SIGDA International Symposium on System Synthesis (ISSS)*, 2002.

[55] Mueller, F. and Whalley, D. B., "Fast instruction cache analysis via static cache simulation," in *Simulation Symposium*, 1995.

[56] Mueller, F., *Static Cache Simulation and its Applications*. PhD thesis, The Florida State University, 1994.

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

[58] Park, C. and Shaw, A., "Experiments with a program timing tool based on source-level timing schema," *IEEE Transactions on Computers*, vol. 24, no. 5, 1991.

[59] Pierce, J. and Mudge, T., "Wrong-path instruction prefetching," in *In ACM Intl. Symp. on Microarchitectures(MICRO)*, 1996.

[60] Price, C., "MIPS IV Instruction Set, revision 3.1," 1995.

[61] Puschner, P. and Koza, C., "Calculating the maximum execution time of real-time programs," *Journal of Real-time Systems*, vol. 1, no. 2, 1989.

[62] Puschner, P., "Worst-case execution time analysis at low cost," *Control Engineering Practice*, vol. 6, pp. 129–135, Jan. 1998.

[63] Real-Time Research Group at Seoul National University, "SNU Real-Time Benchmarks." http://archi.snu.ac.kr/RESEARCH/index.html.

[64] Schneider, J. and Ferdinand, C., "Pipeline behavior prediction for superscalar processors by abstract interpretation," in *ACM Intl. Workshop on Languages, Compilers and Tools for Embedded System (LCTES)*, 1999.

[65] Schrijver, A., *Theory of Linear and Integer Programming.* John Wiley Ltd., 1986.

[66] Shaw, A., "Reasoning about time in higher level language software," *IEEE Transactions on Software Engineering*, vol. 1, no. 2, 1989.

[67] Sohi, G., "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE Transactions on Computers*, vol. 39, no. 3, 1990.

[68] Stappert, F., Ermedahl, A., and Engblom, J., "Efficient longest executable path search for programs with complex flows and pipeline effects," Tech. Rep. 2001-012, Uppsala University, 2001.

[69] Sultan, A., *Linear Programming, An Introduction with Applications.* Academic Press Inc., 1986.

[70] Theiling, H. and Ferdinand, C., "Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[71] Theiling, H., Ferdinand, C., and Wilhelm, R., "Fast and precise WCET prediction by separated cache and path analysis," *Journal of Real Time Systems*, May 2000.

[72] Thesing, S., *Safe and Precise Worst-Case Execution Time Prediction by Abstract Interpretation of Pipeline Models.* PhD thesis, University of Saarland, 2004.

[73] Yeh, T. and Patt, Y., "Alternative implementations of two-level adaptive branch prediction," in *ACM Intl. Symp. on Computer Architecture (ISCA)*, 1992.

[74] Yen, T. and Wolf, W., "Performance estimation for real-time distributed embedded systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 11, 1998.

[75] Zhao, W., Kreahling, W., Whalley, D., Healy, C., and Mueller, F., "Improving WCET by optimizing worst-case paths," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2005.

[76] Zhao, W., Whalley, D., Healy, C., and Mueller, F., "WCET code positioning," in *IEEE Real-Time Systems Symposium*, 2004.