THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRC9/05

Modeling Out-of-Order Processors for WCET Analysis

Xianfeng LI, Abhik ROYCHOUDHURY
and Tulika MITRA

September 2005

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Modeling Out-of-Order Processors for WCET Analysis *

Xianfeng Li Abhik Roychoudhury † Tulika Mitra
School of Computing, National University of Singapore, Republic of Singapore 117543

Abstract

Estimating the Worst Case Execution Time (WCET) of a program on a given processor is important for the schedulability analysis of real-time systems. WCET analysis techniques typically model the timing effects of micro-architectural features in modern processors (such as the pipeline, cache, branch prediction, etc.) to obtain safe and tight estimates. In this paper, we model out-of-order processor pipelines for WCET analysis. The analysis is, in general, 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. Our timing estimation technique for a basic block proceeds by a fixed-point analysis of the time intervals at which the instructions enter/leave a pipeline stage. To extend our estimation to whole programs, we use Integer Linear Programming (ILP) to combine the timing estimates for basic blocks. Timing effects of instruction cache and branch prediction are also modeled within our pipeline analysis framework. This forms a combined timing analysis framework that captures out-of-order pipeline, cache, branch prediction as well as the mutual interaction among these micro-architectural features. The accuracy of our analysis is demonstrated via tight estimates obtained for several benchmarks.

1 Introduction

Statically analyzing the Worst Case Execution Time (WCET) of a program is important for real-time software. Such timing analysis of software has been studied extensively [5, 13, 16, 23, 26, 29] due to its inherent importance in schedulability analysis. Usually it involves (a) path analysis to find out the infeasible paths in the program's control flow graph and (b) micro-architectural modeling to capture the timing effects of architectural

*Preliminary version of parts of this paper has previously been published as [15].

†Contact Author: abhik@comp.nus.edu.sg

features. WCET analysis techniques are conservative, i.e., they compute an upper bound of the program's actual worst case execution time. So, it is in general possible to ignore the effects of the underlying hardware by introducing pessimism. However, ignoring the micro-architectural features produces extremely loose timing bounds as modern processors employ advanced performance enhancing features such as the pipeline, cache, branch prediction, etc. To obtain a tight (yet safe) WCET estimate, we need to model the timing effects of micro-architectural features.

In the last decade, researchers have studied the effects of pipeline, cache and their interaction on program execution time. The assumptions used in most of these works are, unfortunately, only applicable to in-order pipelines where instructions are executed in program order. However, current high-performance processors employ out-of-order execution engines to mask latencies due to pipeline stalls; these stalls may happen due to data dependency, resource contentions, cache misses, branch mispredictions, etc. Even in the embedded domain, some recent processors employ out-of-order pipeline; examples include Motorola MPC 7410, PowerPC 755, PowerPC 440GP, AMD-K6 E and NEC VR5500 MIPS.

In this paper, we model the effects of out-of-order pipelines on the WCET of a program. The main difficulty in modeling such processors is the *timing anomaly* problem [19]. The implication of this problem is that the overall WCET of a program can exceed the estimate obtained by maximizing latencies of individual instructions. Consequently, all possible schedules of instructions with variable latencies need to be considered for estimating the WCET of even a single basic block. Recently, Heckman et al. [9] have modeled PowerPC 755 (an out-of-order processor) for timing analysis. In order to estimate the WCET of a basic block, they statically unfold the execution of the instructions using abstract pipeline states. Due to the presence of timing anomaly, if the latency of an instruction I cannot be determined statically, then upon execution of I a pipeline state will have several successor states corresponding to the various possible latencies of I. For complex pipelines, this can result in state space explosion [30].

Our aim in this paper is to obtain a safe and tight WCET estimate for out-of-order pipelines without enumerating all possible executions corresponding to variable latency instructions. This is achieved via a fixed-point analysis of the time intervals (instead of concrete time instances) at which the instructions of a basic block enter/leave different pipeline stages. Our technique is inspired by an iterative performance analysis technique for real-time distributed systems proposed by Yen and Wolf [32], which estimates the execution time of tasks with data dependences and resource contentions.

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.

We also extend our technique to include the effects of instruction cache and branch prediction. This leads to a micro-architectural modeling framework that captures the timing effects of out-of-order pipeline, instruction cache and branch prediction. These features have interactions among themselves, e.g., pipelined execution is affected by instruction cache and branch prediction behavior. Similarly, branch misprediction may positively or negatively affect the timing behavior of instruction cache by either pre-fetching useful blocks or evicting useful blocks due to wrong pre-fetching. We adopt our previously proposed ILP-based framework to model branch prediction, instruction cache and their interaction [14]. Combining this branch prediction and instruction cache modeling with out-of-order pipeline model is again non-trivial due to the presence of timing anomaly. However, we show that careful modifications of our estimation technique can safely and accurately capture the timing effects of the interaction between pipeline and other architectural features.

The rest of this paper is organized as follows. The next section surveys related work on WCET analysis. Section 3 describes the timing anomaly problem which makes WCET estimation of out-of-order execution difficult. In Section 4, we present our out-of-order processor model and give a brief overview of our WCET estimation method. Section 5 presents the detailed modeling of out-of-order pipelines for WCET estimation. Section 6 extends the technique in Section 5 to take into account the timing effects of branch prediction and instruction cache. Experimental results appear in Section 7. Finally Section 8 concludes the paper.

2 Related work

Research on WCET analysis was initiated more than a decade ago. Early activities can be traced back to [23, 26]. These works analyzed the program source code but did not consider hardware features such as cache or pipeline. Currently, there exist different approaches for combining program path analysis with micro-architectural modeling. One of them is a two-phased approach; it uses *abstract interpretation* [29] to categorize the execution time of the instructions and then applies Integer Linear Programming (ILP) to incorporate path constraints. The other one is an integrated approach proposed in the context of modeling

instruction caches [16]. It employs an ILP formulation using path constraints derived from the control flow graph as well as constraints on cache behavior.

We now summarize research on micro-architectural modeling for WCET analysis — in particular pipeline modeling. Most of the past works on micro-architectural modeling have focused on modeling instruction cache [1, 16, 28] and pipeline [33, 18, 20, 25, 5], either individually or combined [8, 11, 17]. Other important features, such as branch prediction, have received attention in recent years [3, 6, 13].

Pipelining is the core technique universally employed in modern processors and has been studied extensively for WCET analysis. Prior works in this area have successfully modeled in-order pipelines. Zhang et al. [33] model a simple pipeline structure with only two stages. Lim et al. [17] compute the WCET for RISC processors with pipelines and caches through an extension of Shaw's timing schema [26]. A case study with this approach for MIPS R3000/R3010 processors has been made by Hur et al. [11]. Their work has been extended in [18] to model multiple-issue machines. Healy et al. [8] present an integrated modeling of instruction cache and pipeline by first categorizing cache behavior of the instructions, and then using the cache information to analyze the performance of the pipeline. Lundqvist and Stenström [19] combine instruction-level simulation with path analysis by allowing symbolic execution of instructions (whose operands are unknown). Schneider and Ferdinand [25] have applied abstract interpretation to model superscalar processors (Sun Sparc).

Lundqvist and Stenström [20] discuss the issue of timing anomaly for processors with *out-of-order* execution engines. On such processors, a local worst case might not lead to the global worst case. For example, a cache miss could result in a shorter overall execution time than a cache hit. This observation makes micro-architectural modeling techniques mentioned earlier inapplicable to out-of-order processors. Lundqvist and Stenström [20] present a program modification approach to analyze WCET in the presence of out-of-order execution engines. The idea is to insert "synchronization" instructions before and after each variable latency instruction in the program to eliminate timing anomaly. However, synchronization instructions flush the pipeline incurring significant overhead. Moreover, their method requires software controlled caches, which may not be present in all processors. In his Ph.D. thesis, Engblom [5] has conducted a comprehensive study of various pipelines and presented a framework for modeling those pipelines. He studies timing anomaly, but does not explicitly propose any modeling technique for capturing the timing effects of out-of-order pipelines. Indeed, the conditions to ensure safe WCET estimation of pipelined execution developed in his work favor simpler in-order pipelines.

Recently WCET analysis has been employed for real-life modern processors. Langenbach

et al. [12] present a work based on abstract interpretation to model Motorola ColdFire 5307 processor with in-order pipeline, cache and branch prediction. A similar approach is employed by Heckman et al. [9] for modeling an out-of-order processor – PowerPC 755. Their modeling defines an abstract pipeline state as a set of concrete pipeline states and pipeline states with identical timing behavior are grouped together. Thus, if the latency of an instruction I cannot be statically determined, a pipeline state will have several successor states (resulting from the execution of I) corresponding to the various possible latencies of I. This style of modeling is used in order to easily integrate pipeline modeling with other micro-architectural features such as branch prediction and cache.

Our motivation towards studying pipeline modeling has come from a different angle. The problem of timing anomaly [20] makes it difficult to perform pipeline WCET analysis without an exhaustive enumeration of pipeline schedules. We are interested to see if this enumeration can be avoided. Therefore, we propose a pipeline modeling where the clock cycles at which an instruction I enters/leaves a pipeline stage S is tightly estimated as an interval (without enumerating the different clock cycles in which I enters/leaves S). We use this approach to model out-of-order pipelines and then extend it to integrate the timing effects of instruction cache and branch prediction.

The approach for modeling out-of-order pipelines presented in this paper does not depend on special processor features for controlling micro-architectural behaviors, neither does it need to modify the program object code. Also, our method carefully avoids enumerating all possible instruction schedules, thereby achieving efficiency in running time.

3 Timing Anomaly

Modern processors 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 dependences and/or resource contentions) with useful computations.

Out-of-order execution has a serious impact on WCET analysis due to a phenomenon called *timing anomaly* [20]. 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

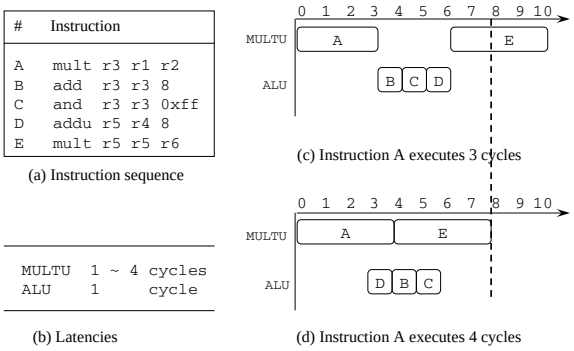


Figure 1: Timing Anomaly due to Variable-Latency Instructions

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 1 illustrates timing anomaly with an example. Instruction `B` depends on `A`, instruction `C` depends on `B`, and instruction `E` depends on `D`. Instructions `A` and `E` use `MULTU` functional unit ($1 \sim 4$ cycles latency) and the other instructions use the single cycle `ALU`.

We illustrate two possible execution scenarios. In the first scenario illustrated in Figure 1(c), `A` executes for three cycles: cycles $0 - 2$. Since `A` starts executing in cycle 0, it should have been ready for execution by cycle 0. Therefore, at the beginning of cycle 3, instructions `B`, `C`, `D` have all been fetched and decoded, that is, they are ready for execution. Furthermore, all of them are contending for the ALU. Instructions `B` and `C` execute on cycles 3 and 4, respectively. Even though `D` is ready for execution in cycle 3 itself, 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 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). Instruction `D` executes in cycle 3 allowing `E` to start execution in cycle 4. The overall execution time in this case is only 8 cycles. Thus, *a longer latency of instruction* `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 timing behavior of a sequence of instructions results from cache misses for all the instructions or 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

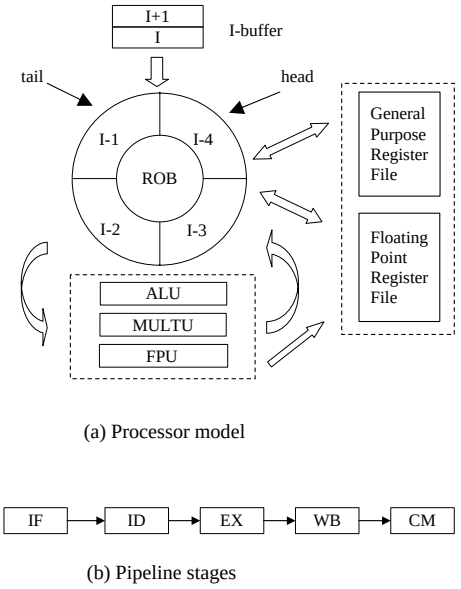


Figure 2: An Out-of-order Processor Model

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. In the next section, we explain the basic idea behind our approach that allows us to avoid such expensive enumeration.

4 Overview of Our Modeling

We discuss the out-of-order processor model considered by our WCET estimation method. This is followed by a brief overview of our pipeline modeling.

4.1 Our Processor Model

Figure 2(a) shows an example of out-of-order processor pipeline, which will be used for illustrating our estimation technique in this paper. This processor is a simplified version of the SimpleScalar `sim-outorder` simulator processor model [2], which in turn is based on [27]. The pipeline consist of five stages as shown in Figure 2(b).

1. **Instruction Fetch (`IF`)**. This stage fetches instructions from the memory *in program order* into the instruction fetch buffer I-buffer. Let us assume a 2-entry I-buffer for

discussion in this paper.

2. **Instruction Decode & Dispatch (`ID`)**. This stage decodes instructions in the I-buffer and dispatches them into a circular buffer, called the re-order buffer (ROB) *in program order*. The ROB, a 4-entry buffer in our discussion, forms the core of the pipeline. Instructions are stored in this buffer from the time they are dispatched to the time they are committed (see `CM` stage).

3. **Instruction Execute (`EX`)**. An instruction in the ROB is issued to its corresponding functional unit (FU) for execution when all its operands are ready and the functional unit is available. If more than one instruction corresponding to a functional unit is ready for execution, the earliest instruction (in program order) is issued for execution. 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, an instruction that has finished execution can always write back its results immediately. Clearly, instructions can write back results *out-of-order*, that is, in an order different from the program order.

5. **Commit (`CM`)**. This is the last pipeline stage where the oldest instruction that has completed the `WB` stage writes its output to the register file and frees its ROB entry. Note that the instructions commit *in program order*. That is, even if an instruction has completed its `WB` stage, it still has to wait for the earlier instructions to commit. At most one instruction can commit each cycle.

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

4.2 Our Pipeline Modeling

Given the control flow graph of a program, our WCET analysis method first derives a WCET estimate for each basic block. The basic block estimates are combined using Integer Linear Programming (ILP) to produce the program's WCET estimate [16]. Such an integration of micro-architectural modeling with program path analysis has been employed in existing works [29]. Formally, let $\mathcal{B}$ be the set of basic blocks of a program. Then, the program's WCET is given by the following objective function

$$\text{maximize} \sum_{B \in \mathcal{B}} N_B * c_B$$

where N_B is an ILP variable denoting the execution count of basic block B and c_B is a constant denoting the WCET estimate of basic block B. The linear constraints on N_B are developed from the flow equations based on the control flow graph. Thus for basic block B,

$$\sum_{B' \to B} E_{B' \to B} = N_B = \sum_{B \to B''} E_{B \to B''}$$

where $E_{B' \to B}$ ($E_{B \to B''}$) is an ILP variable denoting the number of times control flows through the control flow graph edge $B' \to B$ ($B \to B''$). Additional linear constraints are also provided to capture loop-bounds and any known infeasible path information.

How do we find the WCET estimate for a basic block B ? 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 occurs from maximum resource contention among instructions in B, that is, each instruction being delayed by maximum number of other instructions. We produce very coarse estimates for the time intervals at which instructions in B can start/finish execution by initially assuming that any instruction in B can delay the other instructions using the same functional unit, except for the contentions ruled out by data dependences. The estimates allow us to rule out certain contentions — if the earliest time at which instruction I is ready for execution occurs after the latest time at which J finishes, clearly I cannot delay J. 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 (where B's execution starts with an empty pipeline) is the maximum time between the fetch of B's first instruction and commit of B's last instruction.

Given the execution time estimate of B's execution starting with an empty pipeline, we

now have to find the constant c_B, basic block B's WCET estimate. We observe that the number of instructions before and after B which can directly affect the timing of B via dependence/contention is bounded by architectural parameters. Accordingly, we extend our timing estimation technique to operate on a basic block with prologue and epilogue (instructions before and after B which directly affect the timing of B). Time intervals for execution of instructions in prologue/epilogue are estimated conservatively by assuming maximum possible contentions. We also consider (a) data dependences between instructions in prologue and instructions in B, and (b) possible time overlap between instructions in B and prologue of B. In this way, we find the timing estimate of basic block B for all possible choices of prologues and epilogues. The maximum of these estimates is c_B, the estimated WCET of B.

In the preceding, we have given an overview of our modeling technique that captures the timing effects of out-of-order pipelines. The technical details of this modeling is presented in the next section. In Section 6, we extend our modeling to integrate the effects of branch prediction and instruction cache.

5 Out-of-Order Pipeline 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 to take into account the possible initial pipeline states and the instructions before/after the basic block.

5.1 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 4.1. 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 capturing 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 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 dependence 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 dependence relation among nodes in an execution graph (denoted by solid edges), we also define a contention relation among the execution graph nodes. We do not make the contention relation part of the execution graph so as to clearly identify what we mean by "paths" in the execution graph; *paths in the execution graph refer to chains of dependence edges.*

Definition 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, v denote the* `EX` *stages of two different instructions* `I` *and* `J` *respectively, and*
- *instructions* `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 clarify how the dependence and 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 dependence edges includes, in addition to traditional data dependencies, dependencies due to resource constraints and pipelined execution. We consider:

- Dependencies among pipeline stages of the same instruction. This is because an instruction must proceed from the first stage to the 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 such as full I-buffer or full 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 dependence edge $ID(I_i) \rightarrow IF(I_{i+2})$. Similarly, with a 4-entry ROB, there should be a dependence 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 dependence 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

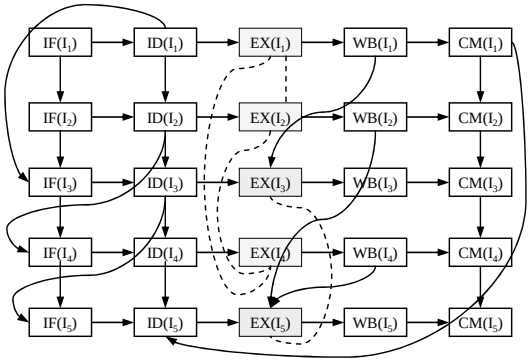
3. $|i - j| < ROB\_size$

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

```
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 3: A basic block and its execution graph. The solid directed edges represent dependencies and the dashed undirected edges represent contention relations.

- 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 functional 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 3 shows an example of execution graph. This graph is constructed from a basic block with five instructions as shown in Figure 3(a). In Figure 3(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. [7]. 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 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 $Code_B = I_1 \ldots I_n$. 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 B's 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 [32] 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 executing on 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 [32]; both these graphs capture 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, [32] captures periodic tasks whereas the instructions in our execution graph are not periodic. More importantly, in [32] 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 a 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, in general, $t_v^{start} \geq t_v^{ready}$.
- 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, for the non-EX stages $t_v^{finish} = t_v^{start} + 1$. 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, 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.

Estimation Algorithm – the big picture 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.

Estimation Algorithm – the detailed picture 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 pairs of nodes u, v. Basically, we find out pairs of nodes (u, v) whose lifetimes are guaranteed not to overlap; for these nodes we set $separated(u, v)$ to true. How do we find out pairs of nodes with 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

```
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 $G = (V, DE)$

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

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($G = (V, DE)$)

$earliest[t_v^{ready}] \geq latest[t_u^{finish}]$.[1] Thus, the tighter the time intervals obtained, the more is the number of 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 contending 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

[1] There exist more sophisticated techniques for finding nodes with disjoint lifetimes in a graph *e.g.* see [22]. In our experiments we found that our simplified approach for identifying separated nodes substantially increases the efficiency of our WCET analysis.

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 in Lines 7–10 of Algorithm 2. First, 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 $|S_{early}| \times max\_lat_v$ 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 (see lines 12–13 of Algorithm 2). In this way the *LatestTimes* algorithm estimates the latest ready/start/finish times of each node in the execution graph of the basic block being analyzed.

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*.

5.2 Estimation for a Basic Block with Context

In the last section, our technique for estimating the WCET of a basic block B is based on the simplifying assumptions that execution of instructions outside B does not interact with B'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.

The execution context of a basic block B is defined in terms of the instructions that directly affect the timing of B's execution. To model the execution time of a basic block B, we need to consider (1) contentions and data dependences among instructions prior to B and instructions in B, and (2) contentions between instructions in B and instructions after B[2]. The instructions before (after) a basic block B that *directly* affect the execution

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

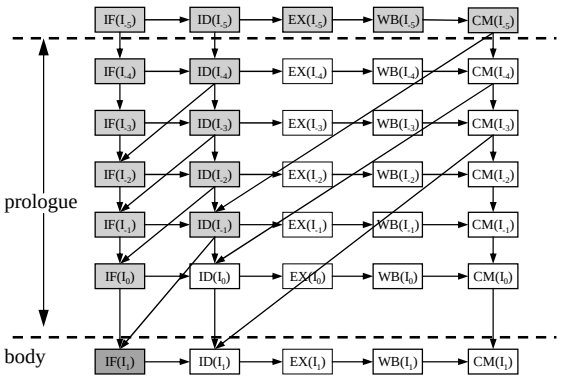


Figure 4: An Example Prologue

time of B constitute the contexts of B and are called the **prologue** (**epilogue**) of B. 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 enters the pipeline. Similarly, due to the 4-entry ROB, at most 4-1=3 instructions after B can contend with instructions in B. Of course, a basic block B may have multiple prologues and epilogues corresponding to the different paths along which B 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.*

Time Intervals for Prologue Nodes Figure 4 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) 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 times of these shaded prologue nodes are clearly bounded by $latest[t^{ready}_{IF(I_1)}] = 0$. Let u be a node in prologue with a path to $IF(I_1)$. Thus the finish time of u must be *before* the ready time of $IF(I_1)$. Consider any path π connecting u and $IF(I_1)$, and let $nodes(\pi)$

/* I_1 is the first instruction in the basic block $latest[t^{ready}_{IF(I_1)}] := 0$ */;
foreach *node* $u \in prologue$ **do**
  $shaded[u] := false$;
  if $paths(u, IF(I_1)) \neq \phi$ **then**
    $shaded[u] := true$;
    $latest[t^{finish}_u] := -\max_{\pi \in paths(u,IF(I_1))} \sum_{x \in nodes(\pi)} min\_lat_x$; /* Inequality 2 */
    $latest[t^{start}_u] := latest[t^{finish}_u] - min\_lat_u$; $latest[t^{ready}_u] := latest[t^{start}_u]$;
/* I_{-p} is the instruction just before the prologue */
$latest[t^{ready}_{CM(I_{-p})}] := -\max_{\pi \in paths(CM(I_{-p}),IF(I_1))} \sum_{x \in nodes(\pi)} min\_lat_x - 1$;
foreach *node* $u \in prologue$ *in topologically sorted order where* $shaded[u] = false$ **do**
  $latest[t^{ready}_u] := \max_{\{w|w \to u\}} \left(latest[t^{finish}_w]\right)$;
  $latest[t^{ready}_u] := \max\left(latest[t^{ready}_u], latest[t^{ready}_{CM(I_{-p})}]\right)$;
  $latest[t^{start}_u] := latest[t^{ready}_u] + max\_lat_u - 1$; /* conservative late contention */
  $S_{early} := early\_contenders(u)$;
  if $S_{early} \neq \phi$ **then**
    $tmp := \min\left(\max_{w \in S_{early}} \left(latest[t^{finish}_w]\right),\ \ latest[t^{start}_u] + |S_{early}| \times max\_lat_u\right)$;
    $latest[t^{start}_u] := \max\left(tmp, latest[t^{start}_u]\right)$;
  $latest[t^{finish}_u] := latest[t^{start}_u] + max\_lat_u$;

Algorithm 4: Estimation of latest times of prologue nodes

be the nodes in π appearing between u and $IF(I_1)$. Clearly

$$latest[t^{finish}_u] \leq latest[t^{ready}_{IF(I_1)}] - \sum_{x \in nodes(\pi)} min\_lat_x \tag{1}$$

where min_lat_x is the minimum latency of node x. That is, the finish time of shaded prologue node u cannot be later than the right-hand-side expression in Inequality 1 even assuming an ideal execution where each node along the path from u to v (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 1 holds for all paths between u and $IF(I_1)$. Therefore, for any shaded prologue node u (*i.e.* a node with a path to $IF(I_1)$) we can estimate the latest finish time of u as

$$latest[t^{finish}_u] \leq max_{\pi \in paths(u,IF(I_1))}(\ latest[t^{ready}_{IF(I_1)}] - \sum_{x \in nodes(\pi)} min\_lat_x\ ) \tag{2}$$

where $paths(u, IF(I_1))$ is the set of paths between u 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 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), 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, 3 with only one difference. Since the EX nodes in epilogue may have late contenders beyond the epilogue, 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 contentions from the epilogue nodes are taken into account in the estimation process.

Overlapped Execution For a basic block B with instructions $I_1, \ldots, I_n$ the execution time estimate of B 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 3. *The overlap δ between a basic block B and its preceding basic block B' is the period during which instructions from both the basic blocks are in the pipeline, that is*

$$\delta = t^{finish}_{CM(I_0)} - t^{ready}_{IF(I_1)} \tag{3}$$

where I_0 is the last instruction of block B' and I_1 is the first instruction of block B.

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. *For a basic block B with instructions $I_1, \ldots, I_n$, its execution time t_B is the interval from the time when the instruction immediate preceding B has finished commit to the time when B's last instruction has finished commit, that is*

$$t_B = t^{finish}_{CM(I_n)} - t^{finish}_{CM(I_0)} \tag{4}$$

where I_0 is the instruction immediately prior to B.

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 t_B for basic block B w.r.t. the time at which the first instruction I_1 of B is fetched, *i.e.* $t^{ready}_{IF(I_1)} = 0$. Thus $t_B = t^{finish}_{CM(I_n)} - \delta$. We can conservatively estimate t_B by finding the largest value of $t^{finish}_{CM(I_n)}$ and the smallest value of δ. The largest value of $t^{finish}_{CM(I_n)}$ is simply the quantity $latest[t^{finish}_{CM(I_n)}]$, calculated by our *LatestTimes* algorithm. The smallest value of the overlap δ is obtained from the following lemma.

Lemma 1.

$$\delta \geq \min_{\substack{u \\ u \to IF(I_1)}} \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \tag{5}$$

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

$$t^{ready}_{IF(I_1)} = t^{finish}_{u} \tag{6}$$

Clearly,

$$t^{ready}_{CM(I_0)} \geq t^{finish}_{u} + \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) \tag{7}$$

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

$$t^{finish}_{CM(I_0)} \geq t^{finish}_{u} + \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \tag{8}$$

From Equations (6) and (8), we get:

$$t^{finish}_{CM(I_0)} - t^{ready}_{IF(I_1)} \geq \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \tag{9}$$

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

$$\begin{aligned} \delta &\geq \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \\ &\geq \min_{\substack{u \\ u \to IF(I_1)}} \left(\max_{\pi \in paths(u, CM(I_0))} \sum_{x \in nodes(\pi)} min\_lat_x \right) + min\_lat_{CM(I_0)} \end{aligned} \tag{10}$$

□

Putting it all together Note that the execution time estimate t_B of a basic block B is obtained *for a specific prologue and a specific epilogue of* B. Since a basic block B in general has multiple choices of prologues and epilogues, they might result in different t_B. So, we

estimate B's execution time under all possible combinations of prologues and epilogues. The maximum of these estimates is used as B's WCET c_B. Let $\mathcal{P}$ and $\mathcal{E}$ be the set of prologues and epilogues for B.

$$c_B = \max_{p \in \mathcal{P},\ e \in \mathcal{E}} (t_B \text{ with prologue } p \text{ and epilogue } e)$$

c_B is used in defining the WCET of the program as the following objective function.

$$\text{maximize} \sum_{B \in \mathcal{B}} N_B * c_B$$

The quantity N_B denotes the execution count of basic block B and is a variable. $\mathcal{B}$ is the set of all basic blocks in the program. This objective function is maximized over the constraints on N_B 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.

This completes the description of our pipeline modeling. Detailed correctness proofs of our method appear in the appendix of `http://www.comp.nus.edu.sg/~lixianfe/thesis.pdf`

6 Integrating Branch Prediction and Instruction Cache

We have studied out-of-order pipelines for WCET analysis in Section 5. However, in current generation of processors, pipelining is always coupled with other micro-architectural features to reduce pipeline stalls [10]. In this section, we integrate the timing effects of branch prediction and instruction cache into our out-of-order pipeline modeling.

The modeling of branch prediction and instruction cache follows from our earlier work [14]. In particular, [14] presents an integrated Integer Linear Programming (ILP) based modeling where program path analysis as well as branch prediction/ instruction cache behavior are formulated as linear constraints; an ILP solver is used to maximize the objective function denoting program's execution time. In this paper, we directly use the ILP formulation of [14] to capture flow analysis, instruction cache and branch prediction effects.

In addition, we also need to consider the impact of instruction cache and branch prediction on our pipeline analysis. 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). We now describe these changes.

The rest of this section 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 the presence of pipeline, cache and branch prediction.

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

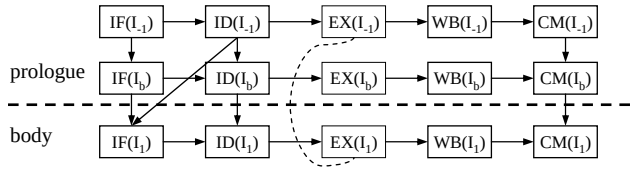
Clearly, if a branch is predicted correctly, then our pipeline analysis does not require any modification. However, a branch misprediction results in instructions along the wrong path being executed in the pipeline (without commit) and flushed out after the branch is resolved. This involves changes in the execution graph of a basic block. 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 at the end of its corresponding 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 [10].

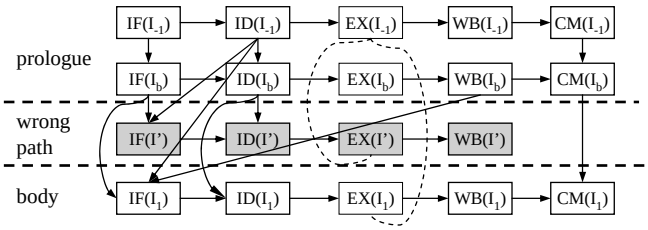
6.1.1 Changes to Execution Graph

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, dependence relation and contention relation among nodes. Consider the execution graph of a basic block B with a body, prologue and 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 5(a) and the modified execution graph fragment due to the misprediction of branch b is shown in Figure 5(b). In Figure 5(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.



(a) Original execution graph



(b) Modifications due to branch misprediction

Figure 5: Execution Graph with Branch Prediction

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. Consider a basic block B whose WCET is being estimated; thus the instructions in B contribute to the *body nodes* of the execution graph. Let b be a conditional branch instruction which is the last instruction of the prologue. Then, b is the last instruction of a basic block B' s.t. $B' \rightarrow B$ is an edge in the program's control flow graph. Let the outcome of branch b which leads to flow of control from block B' to block B be called X (non-taken or taken, denoted as 0 and 1, respectively). That is, if the prediction at branch b is X, no change to the execution graph is needed. Now, given a conditional branch b and its actual outcome X, we can identify the maximum sequence of wrong path instructions that can enter the pipeline if the prediction is $\neg X$. We call this sequence as $Spec(b, X)$. 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 (ROB) and the $IBuffer\_size$ is the size of the instruction fetch buffer (I-buffer).
- If another conditional branch b' is encountered along the wrong path, then the sequence $Spec(b, X)$ is terminated at b'.

Changes to dependence 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 dependence edges among the nodes in each category are drawn as usual. However, the dependence 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. We do not draw any dependence edges between wrong path nodes and body nodes. Instead we add a dependence 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 dependence 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 Algorithms 2, 3 to estimate the latest times and 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 (at the end of $WB(b)$). Therefore, the 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 prologue/epilogue (i.e., any branch in prologue/epilogue apart from the last branch of prologue) as follows. Consider any such branch b in the prologue/epilogue. We modify the execution graph such that correct as well 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". These 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 Cache

So far we have assumed perfect instruction cache, i.e., each instruction fetch takes 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 block (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 under the particular cache scenario ω. If it is a hit, then a single cycle is assigned; if it is a miss, a cache miss penalty N is assigned.

For the instructions in prologue and epilogue of B_i, however, we do not distinguish their cache scenarios. In other words, we conservatively assume that the cache access results of the instructions in prologue/epilogue are unknown. Thus, we assign the interval $[1, N]$ to the IF stage of prologue/epilogue instructions, which covers both the possibilities.

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 in a basic block hit or miss 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. We now describe this formulation.

6.3 Putting it all together

We describe an ILP formulation which integrates our analysis of pipelining, instruction cache 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 its cache scenarios and the prediction of its preceding branch. We denotes the set of possible cache scenarios at B_i as Ω_i. The number of possible cache scenarios can be 2^{n_i} in the worst case, where n_i is the number of memory blocks of B_i. However, only a few of these scenarios are possible in practice due to the constraints imposed by the program control flow. For better accuracy and less analysis time, it is necessary to exclude these infeasible scenarios. This is achieved by a simple preprocessing step which performs a least fixed-point computation. The preprocessing traverses the program control flow graph repeatedly by propagating and updating the possible cache scenarios at each basic block. The preprocessing terminates when 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.

$$\text{Maximize} \quad T = \sum_{i=1}^{N} \sum_{j \to i} \sum_{\omega \in \Omega_i} t_{j \to i}^{c,\omega} * E_{j \to i}^{c,\omega} + t_{j \to i}^{m,\omega} * E_{j \to i}^{m,\omega} \tag{11}$$

where $t_{j \to i}^{c,\omega}$ is the WCET of B_i executed under the following context: (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, and (3) B_i is executed under a cache scenario $\omega \in \Omega_i$. Also, $E_{j \to i}^{c,\omega}$ is the number of times that B_i is executed under this context. Similarly, $t_{j \to i}^{m,\omega}$ is the WCET of B_i executed under the following context: (1) B_i is reached from a preceding block B_j, (2) the branch at the end of B_j is mispredicted, and (3) B_i is executed under a cache scenario $\omega \in \Omega_i$. Again, $E_{j \to i}^{m,\omega}$ is the number of times that B_i is executed under this context.

Using our out-of-order pipeline analysis (Section 5) as well as the extensions proposed in Section 6.1, 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 $t_{j \to i}^{c,\omega}$ and $t_{j \to i}^{m,\omega}$ as constants. We now need to develop constraints to bound the ILP variables $E_{j \to i}^{c,\omega}$ and $E_{j \to i}^{m,\omega}$.

In our earlier work [14], we have proposed an ILP-based branch prediction modeling technique, which can bound the number of correct predictions and mispredictions. Let $E_{j \to i}^{c}$ and $E_{j \to i}^{m}$ be the number of correct predictions/mispredictions when control flow is transferred from B_j to B_i (in case block B_j does not have a conditional branch, $E_{j \to i}^{m}$ is simply set to zero). In [14] we give a detailed ILP modeling to bound $E_{j \to i}^{c}$ and $E_{j \to i}^{m}$. The modeling is involved, and more importantly, completely outside the scope of this paper. We do not repeat it here and instead refer the reader to [14] for details.

Now we observe that $E_{j \to i}^{c,\omega}$ and $E_{j \to i}^{m,\omega}$ are refined forms of $E_{j \to i}^{c}$ and $E_{j \to i}^{m}$ where block B_i's executions are further distinguished with cache scenarios at B_i. This leads to

$$E_{j \to i}^{c} = \sum_{\omega \in \Omega_i} E_{j \to i}^{c,\omega}; \quad E_{j \to i}^{m} = \sum_{\omega \in \Omega_i} E_{j \to i}^{m,\omega} \tag{12}$$

On the other hand, Li et al. [16] propose an ILP-based instruction cache modeling technique, which can bound the number of cache hits/misses for each memory block $B_{i.k}$ in B_i, denoted as $CH_{i.k}$ and $CM_{i.k}$, respectively. Recall that a cache scenario ω of B_i is an assignment of hit or miss to each of B_i's memory blocks. So we define new variables $\Omega_{i.k}^{hit}$ ($\Omega_{i.k}^{miss}$) as the set of those cache scenarios in Ω_i in which memory block $B_{i.k}$ results in a hit

Program	Description
des	Data Encryption Standard
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
matsum	Summation of two 100x100 matrices
minver	Inversion of a floating point matrix
qurt	Root computation of quadratic equations
whet	Whetstone benchmark

Table 1: The Benchmark Programs

(miss). Given Ω_i, the sets $\Omega_{i.k}^{hit}$ and $\Omega_{i.k}^{miss}$ can be computed straightforwardly. So we get

$$CH_{i.k} = \sum_{\omega \in \Omega_{i.k}^{hit}} \sum_{j \to i} (E_{j \to i}^{c,\omega} + E_{j \to i}^{m,\omega}); \quad CM_{i.k} = \sum_{\omega \in \Omega_{i.k}^{miss}} \sum_{j \to i} (E_{j \to i}^{c,\omega} + E_{j \to i}^{m,\omega}) \tag{13}$$

With the above two sets of constraints (12) and (13), $E_{j \to i}^{c,\omega}$ and $E_{j \to i}^{m,\omega}$ can be bounded, provided we can bound $CH_{i.k}$ and $CM_{i.k}$ variables. The constraints on CH and CM variables are obtained from the cache modeling of [16] as well as the integrated modeling of cache and branch prediction in our earlier work [14] (to take into account the effect of branch prediction on instruction cache). Once again, we do not repeat these constraints since they do not add any clarification to our pipeline analysis (which is the focus of this paper). The reader is referred to [14] for details.

Finally, the objective function in Equation 11 can be maximized by the ILP solver subject to (1) the control flow constraints, (2) modeling of branch prediction and instruction cache [14], and (3) the constraints presented in this section.

7 Experimental Evaluation

In this section, we evaluate the accuracy of our estimation technique for ten benchmarks listed in Table 1. These benchmarks have been used by other researchers for WCET analysis. Among them, `des`, `fdct`, `fft`, `isort` and `whet` have been used by Li et al. [16] for WCET analysis and the other benchmarks are from the real-time research group at Seoul National University [24]. Most of the benchmarks contain variable-latency arithmetic instructions. Few exceptions are `des`, `isort` and `matsum`; they do not contain any variable

latency arithmetic instructions.

7.1 Methodology

We use the SimpleScalar architectural simulation toolset [2] for our experiments. The SimpleScalar instruction set architecture (ISA) is a superset of MIPS ISA. We use the compiler provided by SimpleScalar toolset to generate executables corresponding to the benchmark programs. We wrote a prototype analyzer that accepts the SimpleScalar executable annotated with user-provided constraints such as loop bounds. It is parameterized with respect to the cache configurations, the latencies of the functional units as well as the number of entries in the I-buffer and the ROB. The estimation tool first disassembles the code, constructs the control flow graph of the program, estimates the WCETs for basic blocks, and finally generates the ILP constraints and the objective function for the program's WCET. The ILP formulation is solved by CPLEX [4], a commercial ILP solver. The WCET obtained through this analysis is the *Estimated WCET*.

Finding the *actual WCET* is difficult even for programs with few paths in the presence of out-of-order pipeline. This is because a program path with variable latency instructions can have many possible execution schedules and the exact worst case can only be determined by exhaustively evaluating the executions under all the schedules. In our experiments, we simulate the program using several data inputs that are likely to lead to longer execution times. We call the result obtained through simulation *Observed WCET*, which is guaranteed to be less than the actual WCET. The WCET value produced by our analysis, *Estimated WCET*, on the other hand is guaranteed to be more than the actual WCET. Thus

$$Estimated\ WCET \geq Actual\ WCET \geq Observed\ WCET$$

Ideally, we would like to compare the Estimated WCET with the actual WCET to find the accuracy of our analysis. Since we do not know the actual WCET, we conservatively compare the *Estimated WCET* with the *Observed WCET* to asses the accuracy of our WCET analysis. If the Estimated WCET is close to the Observed WCET, clearly this means that the Estimated WCET is close (or maybe even closer) to the actual WCET.

The processor configuration we have used is the following. It has a 4-entry I-buffer and 8-entry ROB. It contains the following variable latency functional units: (a) an integer multiplier with $1 \sim 4$ cycle latency, (b) a floating point adder with $1 \sim 2$ cycle latency, and (c) a floating point multiplier 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

Program	Obs. WCET	Est. WCET	Ratio	Analysis Time (sec)	ILP Solving Time (sec)
des	52181	68218	1.31	0.76	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
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 2: Accuracy and running time of our out-of-order pipeline analysis

assume single-cycle latency for load/store unit because we have not modeled data cache. The branch predictor we use is gshare [21, 31] where 2-bit branch history is XOR-ed with the four least significant bits of the branch address. Note that branch misprediction penalty is not specified here, as its effect has been accounted for in the pipelined execution. The penalty is bounded but not necessarily constant as it depends on when the branch is resolved, i.e., the WB stage of the mispredicted branch is completed. We assume 1-KB direct mapped instruction cache with 16 cache blocks (block size = 64 bytes). We assume that the cache miss penalty is 10 clock cycles. We run all the experiments on a 1.3 GHz Pentium IV PC with 1 GB memory.

7.2 Results

We first present experimental results for pure pipeline analysis, in other words, perfect instruction cache and branch prediction. Under this configuration, we simply assume that each instruction fetch takes single clock cycle and every conditional branch is correctly predicted, that is, there is no pipeline stall caused by the two events. Table 2 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	Obs. WCET	Est. WCET	Ratio	Analysis Time (sec)	ILP Solving Time (sec)
des	70342	89402	1.27	1.57	2.48
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
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 3: Accuracy and running time of our combined analysis for out-of-order pipeline, branch prediction and instruction cache.

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. All experiments were run on a 1.3 GHz Pentium IV PC with 1 GB memory.

Finally, we present the experimental results with the instruction cache and the branch prediction enabled. The resuls are shown in Table 3. As can be seen from the ratio column, the estimation is tight. Due to the integration of instruction cache and branch prediction modeling, the change in the pipeline analysis times is minimal (as compared to Table 2). The ILP solving times are however substantially different now since we model the effects of branch prediction and instruction cache as ILP constraints. The ILP solving times for our integrated modeling are given in the last column. Compared to solving times in (Table 2), there is an increase in the ILP solving time, which comes from the branch prediction modeling and the instruction cache modeling. Still the overall time overheads are quite tolerable for all the benchmarks.

8 Discussion

Timing anomaly complicates the Worst Case Execution Time (WCET) analysis on out-of-order processors by invalidating the assumption that local worst case always leads to global worst case. On the other hand, an exhaustive enumeration of all possible local cases is anticipated to be quite inefficient. In this paper, 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 can occur in pipelined execution. We have combined our pipeline modeling with instruction cache and branch prediction for WCET analysis. We have implemented our technique and experimentally validated its estimation accuracy against several standard benchmark programs used by other WCET research groups. In future, we plan to investigate the integration of data cache modeling into our WCET analysis framework.

References

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

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

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

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

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

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

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

[8] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), 1999.

[9] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7), July 2003.

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

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

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

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

[14] X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for timing analysis. *Journal of Real-Time Systems*, 29(1), 2005.

[15] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *IEEE Real-Time Systems Symposium*, 2004. http://www.comp.nus.edu.sg/∼abhik/pdf/rtss04-wcet.pdf.

[16] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3), 1999.

[17] S.-S. Lim, Y.H. Bae, G.T. Jang, B.-D. Rhee, S.L. Min, C.Y. Park, H. Shin, K. Park, and C.S. Kim. An accurate worst-case timing analysis technique for RISC processors. *IEEE Transactions on Software Engineering*, 21(7), 1995.

[18] S-S. Lim, J.H. Han, J. Kim, and S.L. Min. A worst case timing analysis technique for multiple-issue machines. In *IEEE Real Time Systems Symposium (RTSS)*, pages 334–345, 1998.

[19] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, 17(2-3), 1999.

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

[21] S. McFarling. Combining branch predictors. Technical report, DEC Western Research Laboratory, 1993.

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

[23] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-time Systems*, 1(2), 1989.

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

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

[26] A.C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 1(2), 1989.

[27] G. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3), 1990.

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

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

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

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

[32] T.Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11), 1998.

[33] N Zhang, A Burns, and M Nicholson. Pipelined processors and worst case execution times. *Journal of Real-Time Systems*, 5(4), 1993.