Modeling Out-of-Order Processors for WCET Analysis*

Xianfeng Li
Dept. of Computer Sc. & Tech.
Peking University
lixianfeng@mprc.pku.edu.cn

Abhik Roychoudhury   Tulika Mitra
Dept. of Computer Science
National University of Singapore
{abhik,tulika}@comp.nus.edu.sg

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 pipeline, cache, branch prediction) to obtain safe and tight estimates. In this paper, we model *out-of-order superscalar* 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 [Eng02, LMR05, LMW99, PK89, Sha89, TFW00] 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 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

*Preliminary version of parts of this paper has previously been published as [LRM04].

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 [LS99b]. 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. [HLTW03] 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 [The04].

Our aim in this paper is to obtain a safe and tight WCET estimate for *out-of-order superscalar 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. Note that this cleverly avoids state space explosion in pipeline analysis — instead of calculating set of possible pipeline states (resulting from different instruction schedules) in a basic block, we calculate the time intervals at which the events which change pipeline state can occur. We then augment 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 superscalar 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 [LMR05]. Combining this branch prediction 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. For instruction cache behavior, we use a categorization based modeling where instructions are statically (and conservatively) categorized as hit/unknown in different control-flow contexts.

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 [PK89, Sha89]. 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* [TFW00] 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 [LMW99]. 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 [AMWH94, LMW99, TFW00] and pipeline [ZBN93, LHKM98, LS99b, SF99, Eng02], either individually or combined [HAM$^+$99, LBJ$^+$95]. Other important features, such as branch prediction, have received attention in recent years [CP00, LMR05].

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. Lim et al. [LBJ$^+$95] compute the WCET for RISC processors with pipelines and caches through an extension of Shaw's timing schema [Sha89]. Their work has been extended in [LHKM98] to model multiple-issue machines. The main difference between these works and our work is that [LHKM98] models the timing effects of *in-order* superscalar pipelines. The basic ingredients of their modeling have similarities with our modeling — they model dependencies, contentions and parallelism (owing to superscalarity)

among instructions. However, as they model in-order execution they can construct an order in which the timing effect of each instruction on other instructions can be determined. In our work, due to the modeling of out-of-order execution such an ordering is not possible — given two instructions i, j in a basic block the execution of i may delay instruction j in one instruction schedule, while in another schedule the execution of instruction j may delay instruction i. So, instead of enumerating all possible instruction schedules we initially estimate maximal delay (by other instructions) for each instruction, and iteratively tighten these estimates until they reach a fixed point.

Healy et al. [HAM+99] 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 [LS99a] combine instruction-level simulation with path analysis by allowing symbolic execution of instructions (whose operands are unknown). Schneider and Ferdinand [SF99] have applied abstract interpretation to model superscalar processors (Sun Sparc). This work models in-order superscalar pipelines by collecting all possible concrete pipeline states at a program point into an abstract pipeline state. This approach towards pipeline modeling has been later extended to model out-of-order pipelines — an issue which we now discuss.

Lundqvist and Stenström [LS99b] 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 [LS99b] 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 [Eng02] 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. [LTH02] present a work based on abstract interpretation to model Motorola ColdFire 5307 processor with in-order pipeline, cache and branch prediction. An extension of this approach is employed by Heckman et al. [HLTW03] 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 (thereby causing state space explosion). 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 [LS99b] makes it difficult to perform pipeline WCET analysis without an exhaustive enumeration of pipeline schedules. In this paper, we *avoid any enumeration of these schedules altogether*. 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 superscalar 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.

Finally, we note that our core method for estimating the timing effects of out-of-order pipelines is inspired by a performance analysis technique for real-time distributed systems [YW98] which analyzes a system consisting of several periodic tasks represented by task graphs. Each task consists of a partially ordered set of processes. 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. The algorithm estimates the worst case completion time of all the tasks. Even though [YW98] appears very different from our problem of out-of-order pipeline analysis, the two problems are conceptually similar since both problems need to analyze the timing behavior of processes with dependencies and resource contentions. However, there are some significant differences as well. The most important difference (in terms of constructing the estimation algorithm) is that in [YW98] 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 of out-of-order pipeline analysis. In [YW98] one can use the well-known result of Lehoczky et. al. [LSD89] to calculate response time, whereas this is not possible in our problem.

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

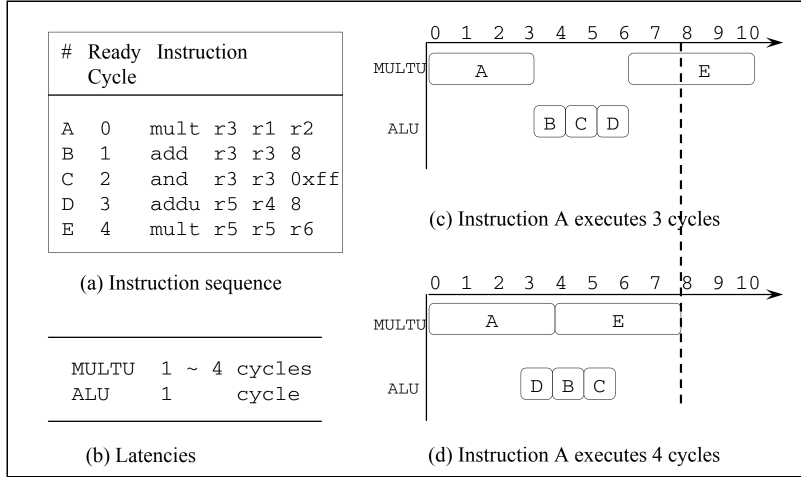

Figure 1: Timing Anomaly due to Variable-Latency Instructions

called *timing anomaly* [LS99b]. 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 occur due to different reasons: cache hit/miss for a load instruction, variable number of cycles taken by an arithmetic instruction like multiplication etc. Let us assume that the execution time of a sequence of instructions containing I is g_{max} (g_{min}) if I incurs a latency of l_{max} (l_{min}). The latencies of the other instructions in the sequence are fixed. A timing anomaly happens if either $(g_{max} - g_{min}) < 0$ or $(g_{max} - g_{min}) > (l_{max} - l_{min})$.

Figure 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

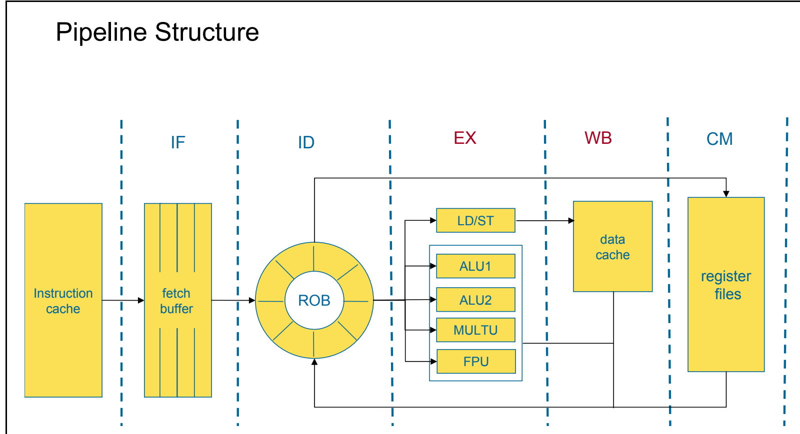

Figure 2: An Out-of-order Superscalar Processor Model

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 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

In this section, 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 shows an example of out-of-order superscalar 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 [BA97], which in turn is based on [Soh90]. The pipeline consist of five stages as shown in Figure 2.

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 4-entry I-buffer with at most 2 instructions fetched per cycle (degree of superscalarity = 2) for discussion in this paper. The quantity $IBuffer_size$ defines the size of the I-buffer.

2. **Decode & Dispatch (`ID`).** This stage decodes instructions in the fetch buffer and dispatches them into the re-order buffer (ROB) *in program order*. The ROB, an 8-entry buffer in our discussion, forms the core of the pipeline. The variable ROB_size

defines the size of the ROB. It functions as a combination of reservation stations and re-order buffer, which are implemented as separated components in some commercial processors. Instructions are stored in this buffer from the time they are dispatched to the time they are committed (see `CM` stage). We assume that at most two instructions can be decoded and dispatched per cycle.

3. **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. For load instructions, this stage only calculates the effective memory addresses, and the actual result will be read from the data cache in the next stage (the `WB` stage). If more than one instruction corresponding to a functional unit is ready for execution, the earliest instruction (in program order) is issued for execution. Functional units may or may not be pipelined. Again we assume that at most two instructions can be issued from the ROB to the 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, load instructions will get the operands from the data cache by using addresses calculated in the `EX` stage. We assume a perfect data cache. Results are written back to the ROB in this stage. For branch instructions, mispredictions will be resolved here as well. Note that in some real-life superscalar processors, only limited number of instructions can write back to the ROB per cycle. We assume unlimited write-back bandwidth for our processor model. However, the technique proposed in this paper can model limited write-back width (similar to our modeling of limited issue width).

5. **Commit (`CM`)**. This is the last pipeline stage where the oldest instructions that have completed previous stages write their output to the register file and free their ROB entries. Note that instructions commit *in program order*. That is, even if an instruction has completed its previous stages, it still has to wait for the earlier instructions to commit. In our experiments, we assume that the processor can commit at most two instructions per cycle.

In summary, in this paper we use a two-issue superscalar processor consisting of an in-order front-end (`IF` and `ID` stages), an out-of-order execution core (`EX` and `WB` stages), and an in-order back-end (`CM` stage).

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 [LMW99]. Such an integration of micro-architectural modeling with program path analysis has been employed

in existing works [TFW00]. 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 via resource contentions, except for the contentions ruled out by data dependencies. 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.

Given the execution time estimate of B 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) dependencies/contentions 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 only 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 Pipeline Modeling

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. 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, resource contentions and degree of superscalarity — the 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 & Dependence Relation). *The execution graph for a basic block B under a pipeline model is defined as*

$$G_B = (V_B, DR_B)$$

where V_B represents all possible combinations of instruction identifiers and pipeline stages for basic block B, and $DR_B \subseteq V_B \times V_B$ represents dependence relations among the nodes. For two nodes $u, v \in V_B$, we say that $(u, v) \in DR_B$ 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, $(u, v) \in DR_B \Rightarrow (v, u) \notin DR_B$.

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. For a node u with variable latency $min_lat_u \sim max_lat_u$, the node is annotated with an interval $[min_lat_u,\ max_lat_u]$. As some resources (e.g., floating point multiplier) in modern processors are fully pipelined, we also annotate such resources with **initiation intervals** (II). Initiation interval of a resource is defined as the number of cycles that must elapse between issuing two instructions to that resource. For example, a fully pipelined floating point multiplier can have a latency of 6 clock cycles and an initiation interval of 1 clock cycle. For a non-pipelined resource, II is the same as latency. Also, if we have multiple copies of the same resource (e.g., 2 ALUs), then we define the **multiplicity** of that resource.

The dependence relation in our execution graph is used to model the following dependencies.

- *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 finite-sized buffers and queues such as I-buffer or ROB.* For example, assuming a 4-entry I-buffer, there will be no entry available for $IF(I_{i+4})$ before the completion of $ID(I_i)$ (which removes I_i from the I-buffer). Therefore, there should be a dependence edge $ID(I_i) \rightarrow IF(I_{i+4})$. Similarly, with an 8-entry ROB, there should be a dependence edge $CM(I_i) \rightarrow ID(I_{i+8})$ 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.

- *Dependencies due to in-order execution in* `IF`*,* `ID`*, and* `CM` *pipeline stages.* For example, in a scalar processor (*i.e.*, degree of superscalarity = 1) there will be dependence edges $IF(I_i) \rightarrow IF(I_{i+1})$ because $IF(I_{i+1})$ can only start after $IF(I_i)$ completes. For a superscalar processor with n-way fetch (i.e., degree of superscalarity = n), we draw dependence edges $IF(I_i) \rightarrow IF(I_{i+n})$. This captures the fact that I_{i+n} cannot be fetched in the same cycle as I_i.

- *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 $WB(I_i) \rightarrow EX(I_j)$.

Apart from the dependence relation among the nodes in an execution graph (denoted by solid edges), we also define contention relations among the execution graph nodes. Contention relations model structural hazards in the pipeline. 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 $G_B = (V_B, DR_B)$ be its execution graph. We define a contention relation $CR_B \subseteq V_B \times V_B$ such that for two nodes $u, v \in V_B$, we say that $(u, v) \in CR_B$ iff nodes u and v may delay each other by contending for a resource, for example, functional unit or register write port.*

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

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

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 eight entries then clearly instructions I_i and I_{i+8} cannot coexist in the ROB to contend for a functional unit. The contention between two instructions obeys the following rules.

- If two instructions contend for a resource in the same clock cycle, the earlier instruction (according to program order) gets access to the resource, and
- Once an instruction gets access to a resource, 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 before 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.

Finally, we define a parallelism relation to model superscalarity, for example, multiple issues and multiple decodes.

Definition 3 (Parallelism Relation). *Let B be a basic block and $G_B = (V_B, DR_B)$ be its execution graph. We define a parallelism relation $PR_B \subseteq V_B \times V_B$ such that for two nodes $u, v \in V_B$, we say that $(u, v) \in PR_B$ iff*

- *nodes u, v denote the same pipeline stage (call it stg) of two different instructions I_i, I_j*
- *instructions I_i and I_j may start execution of this pipeline stage stg in parallel*

The second condition — instructions I_i and I_j may start execution of pipeline stage *stg in parallel* — requires some explanation. If the pipeline stage in question processes instructions in-order (the `IF`, `ID` and `CM` stages) this simply means that $|i - j|$ is less than the degree of superscalarity of the corresponding stage (i.e., the fetch, decode or commit width respectively). However, for the out-of-order pipeline stages the instructions are accommodated in the re-order buffer ROB; hence two instructions can start execution in parallel if they can co-exist in the ROB (i.e., $|i - j| < ROB_size$).

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); we assume that the degree of superscalarity is 2. 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

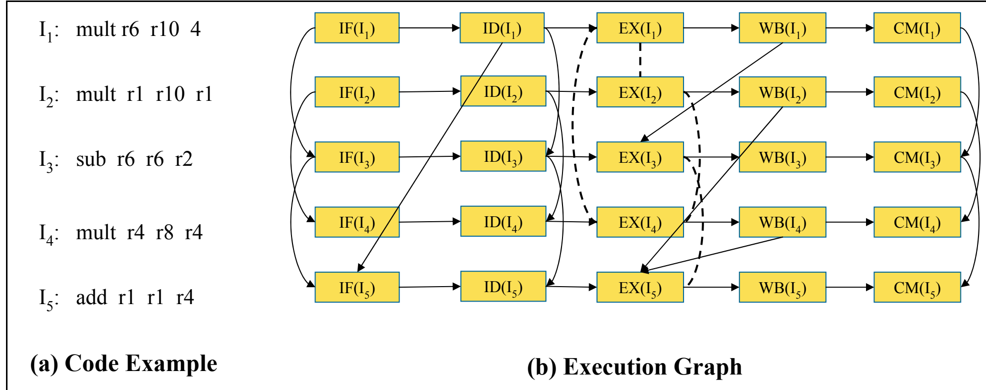

Figure 3: A basic block and its execution graph. The solid directed edges represent dependencies and the dashed undirected edges represent contention relations.

instructions I_1 and I_4 are both ready to execute and the multiplication unit is free, then $EX(I_1)$ will be issued for execution as I_1 appears before I_4 in program order; 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 Problem Definition Let B be a basic block consisting of a sequence of instructions $Code_B = I_1 \dots I_n$. 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.

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.

- $earliest[t_v^{ready}], latest[t_v^{ready}]$: Ready time of a node v is defined as the time when all its predecessors have completed execution. $earliest[t_v^{ready}]$ ($latest[t_v^{ready}]$) defines the earliest (latest) ready time of node v.

- $earliest[t_v^{start}], latest[t_v^{start}]$: Start time of a node v is defined as the time when it starts execution. A node may not be able to start execution when it becomes ready due to unavailability of a resource and/or limited degree of superscalarity. Therefore, $t_v^{start} \geq t_v^{ready}$.

- $earliest[t_v^{finish}], latest[t_v^{finish}]$: Finish time of a node v is defined as the time when it completes execution. Given a node v, we add the minimum (maximum) latency of v 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. $(u, v) \in CR_B$ and $instr_id(u) < instr_id(v)$. Recall that CR_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. $(u, v) \in CR_B$ and $instr_id(u) > instr_id(v)$.

- $peers(v)$: Set of nodes u s.t. $(u, v) \in PR_B$ and $instr_id(u) < instr_id(v)$. These nodes may start execution in parallel with v and appear earlier in program order. Hence these nodes can potentially delay the execution of v due to limited degree of superscalarity.

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

- $II(res), multiplicity(res)$: Initiation interval (the number of cycles that must elapse between issuing two instructions to the resource) and multiplicity (number of copies) of a resource res.

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 until there is no change in the bounds. Termination is guaranteed because of the following reason.

- We start with all pairs of instructions in the contention relation (i.e., every instruction can potentially delay every other instruction).
- At every step of the fixed-point computation, we remove pairs from this set — those instruction pairs that are shown to be separated in time.

As the number of instructions in a basic block is finite, the number of pairs initially in the contention relation is also finite. Furthermore, we remove at least one pair in every step of the fixed-point computation — so the fixed point computation *must terminate* in finitely many iterations; if the number of instructions in the basic block being estimated is n, the number of fixed-point iterations is bounded by n^2.

1 **foreach** *node* $v \in V$ **do**
2 $\quad earliest[t_v^{start}] := 0$; $earliest[t_v^{ready}] := 0$; $earliest[t_v^{finish}] := min_lat_v$;
$\quad latest[t_v^{start}] := \infty$; $latest[t_v^{ready}] := 0$; $latest[t_v^{finish}] := \infty$;
3 **repeat**
4 $\quad$ `LatestTimes(`G`); EarliestTimes(`G`);`
until *ready, start, and finish time of all nodes are unchanged*;
5 $WCET := latest[t_{CM(I_n)}^{finish}]$; /* I_n is the last instruction of the basic block */

Algorithm 1: WCET Estimation for Execution Graph G

Estimation Algorithm – the detailed picture Algorithm 1 gives the outline for computing the WCET given an execution graph corresponding to a basic block. The top level algorithm iteratively performs two operations: *LatestTimes* and *EarliestTimes*, which compute the upper and lower timing bounds of the 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), (b) its contenders, and (c) its peers.

We first consider the delay of v's start time by contenders later in program order (see Lines 6–9 of Algorithm 2). Note that the start time of node v can be delayed by *at most one* late contender. We exclude the contenders who have been identified to be separated from v (i.e., whose lifetimes cannot overlap with v). Given two nodes u and v in the execution graph, we simply set $separated(u, v)$ to true if $earliest[t_u^{ready}] \geq latest[t_v^{finish}]$ or $earliest[t_v^{ready}] \geq latest[t_u^{finish}]$.[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. Also, a late contender

[1]There exist more sophisticated techniques for finding nodes with disjoint lifetimes in a graph, e.g., see [MD92]. In our experiments we found that our simplified approach for identifying separated nodes substantially increases the efficiency of our WCET analysis.

1 $latest[t^{ready}_{IF(I_1)}] := 0;$ /* I_1 is the first instruction of the basic block */;
2 **foreach** *node* $v \in V$ *in topologically sorted order* **do**
3 $latest[t^{start}_v] := latest[t^{ready}_v]$;
4 /* Structural hazards */;
5 Let res be the resource that v is contending for;
6 $S_{late} := \{u \mid u \in late_contenders(v) \wedge \neg separated(u,v) \wedge earliest[t^{start}_u] < latest[t^{ready}_v]\}$;
7 **if** $S_{late} \neq \phi$ **then**
8 $tmp := \min\left(\max_{u \in S_{late}}(latest[t^{start}_u] + II(res)), latest[t^{ready}_v] + II(res) - 1\right)$;
9 $latest[t^{start}_v] := \max(tmp, \ latest[t^{start}_v])$;
10 $S_{early} := \{u \mid u \in early_contenders(v) \wedge \neg separated(u,v)\}$;
11 **if** $S_{early} \neq \phi$ **then**
12 $tmp := \min\left(\max_{u \in S_{early}}(latest[t^{start}_u] + II(res)), \ latest[t^{start}_v] + \lfloor \frac{|S_{early}| \times II(res)}{multiplicity(res)} \rfloor\right)$;
13 $latest[t^{start}_v] := \max(tmp, \ latest[t^{start}_v])$;
14 /* Superscalarity/parallelism */;
15 Let p be the degree of parallelism for the pipeline stage stg of node v;
16 **if** pipeline stage stg corresponds to in-order execution **then**
17 $S_{peer} := \{u \mid u \in peers(v) \wedge latest[t^{start}_u] > latest[t^{start}_v]\}$;
18 **if** $S_{peer} \neq \phi$ **then**
19 $latest[t^{start}_v] := \max_{u \in S_{peer}}(latest[t^{start}_u])$;
20 **else**
21 $S_{peer} := \{u \mid u \in peers(v) \wedge \neg separated(u,v)\}$;
22 **if** $|S_{peer}| \geq p$ **then**
23 $latest[t^{start}_v] := latest[t^{start}_v] + \lfloor \frac{|S_{peer}|}{p} \rfloor$;
24 $latest[t^{finish}_v] := latest[t^{start}_v] + max_lat_v$;
25 **foreach** *immediate successor* w *of* v **do**
26 $latest[t^{ready}_w] = max(latest[t^{ready}_w], \ latest[t^{finish}_v])$;

Algorithm 2: LatestTimes(G)

1 $earliest[t^{ready}_{IF(I_1)}] := 0;$ /* I_1 is the first instruction of the basic block */
2 **foreach** *node* $v \in V$ *in topologically sorted order* **do**
3 $earliest[t^{start}_v] := earliest[t^{ready}_v]$;
4 $earliest[t^{finish}_v] := earliest[t^{start}_v] + min_lat_v$;
5 **foreach** *immediate successor* w *of* v **do**
6 $earliest[t^{ready}_w] = max(earliest[t^{ready}_w], \ earliest[t^{finish}_v])$;

Algorithm 3: EarliestTimes(G)

$u \in late_contenders(v)$ cannot delay v after v is ready since v has higher priority. So, late contenders who do not satisfy the condition $earliest[t_u^{start}] < latest[t_v^{ready}]$ are excluded. The delay from a late contender u is bounded by u's latest start time plus the initiation interval of the corresponding resource $latest[t_u^{start}] + II(res)$. Here res is the resource being contended for (say a functional unit) and $II(res)$ is the initiation interval of resource res. In addition, u cannot delay v by more than the initiation interval of the corresponding resource; thus, we have another bound $latest[t_v^{ready}] + II(res) - 1$. The minimum of the two bounds is taken.

Apart from the delay due to late contenders of node v, we also need to estimate the delay in v's start time due to its early contenders. Note that the early contenders appear before v in program order. So in the worst case, all of them, except those proved to be separated from v (i.e., not overlapping with v's lifetime), can contend with v and delay its start time. This is captured in Lines 10–13 of Algorithm 2. First, the delay due to early contention cannot be beyond the time when all contenders have started execution and the initiation interval of the corresponding resources have elapsed; so

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

On the other hand, the maximum delay is also bounded by $\lfloor \frac{|S_{early}| \times II(res)}{multiplicity(res)} \rfloor$, where each early contender executes before node v.

Apart from contention, the execution of node v may get delayed due to limited degree of superscalarity such as issue width (Lines 15–23). If the pipeline stage has in-order execution, then v cannot start execution before its peers (Lines 16–19). For an out-of-order pipeline stage, if the the number of competing peers (captured by $|S_{peer}|$) is greater than the degree of superscalarity, then v may not be able start execution in the current clock cycle and it gets delayed (Lines 22–23). Note that this is a conservative delay estimate due to limited issue width. As we have already considered delay due to contention, node v is guaranteed to find its corresponding functional unit free. It simply has to find an issue slot by competing with the peers requesting the same or other functional unit types. Given issue width equal to p, this delay is bounded by $\lfloor \frac{|S_{peer}|}{p} \rfloor$.

The latest finish time of v is obtained by simply adding the maximum latency of the functional unit to $latest[t_v^{start}]$ (Line 24). 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 25–26 of Algorithm 2). Note that we have initialized latest ready time of all nodes to zero. 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.

The *EarliestTimes* algorithm (see Algorithm 3) computes the earliest ready, start, and finish times of all nodes in the execution graph. We only need to ensure that they are not over-estimated to ensure correctness. Therefore, we conservatively set the earliest start time to the earliest ready time.

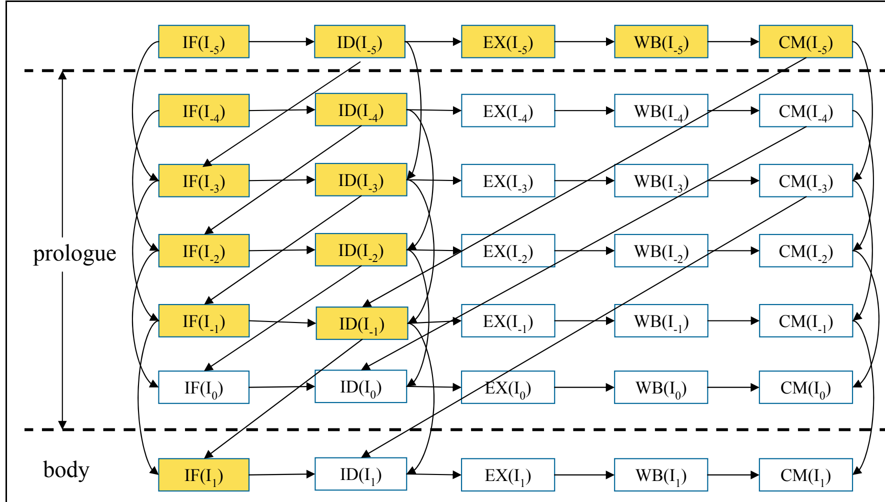

Figure 4: An Example Prologue

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 dependencies 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 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[3], 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.*

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

[3]We use smaller buffers here for illustration purposes only. The default configuration used for experiments is 4-entry I-buffer and 8-entry ROB.

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

```
1 foreach node u ∈ prologue (in reverse program order) do
2   if paths(u, IF(I1)) ≠ φ then
3     latest[t_u^finish] := − max_{π∈paths(u, IF(I1))} Σ_{x∈nodes(π)} min_lat_x; /* Inequality 2 */
4     latest[t_u^start] := latest[t_u^finish] − min_lat_u; latest[t_u^ready] := latest[t_u^start];
5   else
6     if u is a node corresponding to an in-order pipeline stage then
7       if ∃v ( paths(v, IF(I1)) ≠ φ ∧ u ∈ peers(v) ) then
8         latest[t_u^finish] := latest[t_v^finish];
9         latest[t_u^start] := latest[t_u^finish] − min_lat_u; latest[t_u^ready] := latest[t_u^start];
```

/* estimate remaining prologue nodes in a manner similar to Algorithm 2 */

Algorithm 4: Estimation of latest times of prologue nodes

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 $IF(I_1)$; here I_1 is the first instruction in the basic block whose WCET is being estimated. 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 $IF(I_1)$[4]. 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 $nodes(\pi)$ be the nodes in π appearing between u and $IF(I_1)$. Clearly

$$latest[t_u^{finish}] \leq latest[t^{ready}_{IF(I_1)}] - \sum_{x \in nodes(\pi)} min_lat_x \quad (1)$$

where min_lat_x is the minimum latency of node x. That is, the finish time of 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 prologue node u with a path to $IF(I_1)$) we can estimate the latest finish time of u as

$$latest[t_u^{finish}] \leq min_{\pi \in paths(u, IF(I_1))}(\; latest[t^{ready}_{IF(I_1)}] - \sum_{x \in nodes(\pi)} min_lat_x \;) \quad (2)$$

[4]All prologue nodes with a path to $IF(I_1)$ are shaded in Figure 4. In addition, some nodes without paths to $IF(I_1)$ are also shaded; we discuss more on this later.

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 3 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 4 of Algorithm 4).

Some prologue nodes do not have any path to $IF(I_1)$, but they are peers of some nodes with paths to $IF(I_1)$. For example in Figure 4 the node $IF(I_{-2})$, which has no path to $IF(I_1)$, is a peer of $IF(I_{-1})$. For in-order pipeline stages, if a node u waits for the completion of a node v, it also waits for the completion of any node in $peers(v)$. By exploiting this implicit constraint, latest times of nodes like $IF(I_{-2})$, $ID(I_{-2})$, $IF(I_{-4})$ and $ID(I_{-4})$ in Figure 4 are bounded by Lines 8-9 of Algorithm 4; thus, these nodes are also marked as *shaded* in Figure 4.

For the rest of prologue nodes (unshaded nodes in Figure 4), the latest time calculation is similar to Algorithm 2. Now, how are we taking into account the timing effects of the pre-prologue nodes on the prologue nodes? Suppose I_{-n} is the instruction just preceding the prologue. Then the latest finish time of $CM(I_{-n})$ is bounded by the following equation.

$$latest[t^{finish}_{CM(I_{-n})}] = - \max_{\pi \in paths(CM(I_{-n}),\ IF(I_1))} \sum_{x \in nodes(\pi)} min_lat_x$$

Clearly, for any pre-prologue predecessors or contenders, they must have completed execution by $latest[t^{finish}_{CM(I_{-n})}]$. Therefore, although we do not know the impact of pre-prologue nodes, we can safely use $latest[t^{finish}_{CM(I_{-n})}]$ as a safe bound. This observation is used for the following change — on Line 26 of Algorithm 2, the computed $latest[t^{ready}_w]$ is further bounded with the following equation.

$$latest[t^{ready}_w] = \max\left(latest[t^{ready}_w],\ latest[t^{finish}_{CM(I_{-n})}]\right)$$

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 pipelined execution.

Definition 4. *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 5. *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 the execution time 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 as follows.

Minimum value of overlap δ Let u be the node among $IF(I_1)$'s immediate predecessors with the longest (maximum) finish time. Then,

$$t^{ready}_{IF(I_1)} = t^{finish}_{u} \tag{5}$$

Let z be the CM node in the prologue to which every predecessor of $IF(I_1)$ has a path (for scalar pipeline, z is $CM(I_0)$; for pipeline of with degree of superscalarity equal to p, z is

$CM(I_{1-p})$), Clearly,

$$t_z^{ready} \;\geq\; t_u^{finish} + \left(\max_{\pi \in paths(u,z)} \sum_{x \in nodes(\pi)} min_lat_x \right) \tag{6}$$

This is because z can become ready only *after* its predecessors along the paths from u have executed. Therefore,

$$t_{CM(I_0)}^{finish} \geq t_z^{finish} \geq t_u^{finish} + \left(\max_{\pi \in paths(u,z)} \sum_{x \in nodes(\pi)} min_lat_x \right) + min_lat_z \tag{7}$$

From Equations (5) and (7), we get:

$$t_{CM(I_0)}^{finish} - t_{IF(I_1)}^{ready} \geq t_z^{finish} - t_{IF(I_1)}^{ready} \geq \left(\max_{\pi \in paths(u,z)} \sum_{x \in nodes(\pi)} min_lat_x \right) + min_lat_z \tag{8}$$

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

$$\begin{aligned} \delta \;&\geq\; \left(\max_{\pi \in paths(u,z)} \sum_{x \in nodes(\pi)} min_lat_x \right) + min_lat_z \\ &\geq\; \min_{u \overset{u}{\rightarrow} IF(I_1)} \left(\max_{\pi \in paths(u,z)} \sum_{x \in nodes(\pi)} min_lat_x \right) + min_lat_z \end{aligned} \tag{9}$$

In the above derivation of overlap δ, we use z instead of $CM(I_0)$ because for superscalar pipelines, some predecessors of $IF(I_1)$ have no paths to $CM(I_0)$. Node z is a peer node of $CM(I_0)$, and it often completes in the same cycle as $CM(I_0)$. Therefore, using z (instead of $CM(0)$) in overlap estimation rarely introduces extra pessimism.

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. A basic block B in general has multiple choices of prologues and epilogues. 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.

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 [HP96]. 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 follows from our earlier work [LMR05]. In particular, [LMR05] presents an Integer Linear Programming (ILP) based modeling where program path analysis as well as branch prediction are formulated as linear constraints; an ILP solver is used to maximize the objective function denoting program's execution time.

For instruction cache modeling, we use a categorization-based technique [AMWH94, TFW00] that independently classifies instruction accesses as *cache hit*, or *unknown*. An important issue in our instruction cache analysis is the impact of speculative execution (due to branch prediction) on instruction caching. To get safe estimates, we let speculative execution update the instruction cache *conditionally* in our instruction cache analysis. If an instruction is brought into the instruction cache conditionally, it will be classified as *unknown*.

The rest of this section will focus on integrating branch prediction and instruction cache analyses into the pipeline analysis. 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 [HP96].

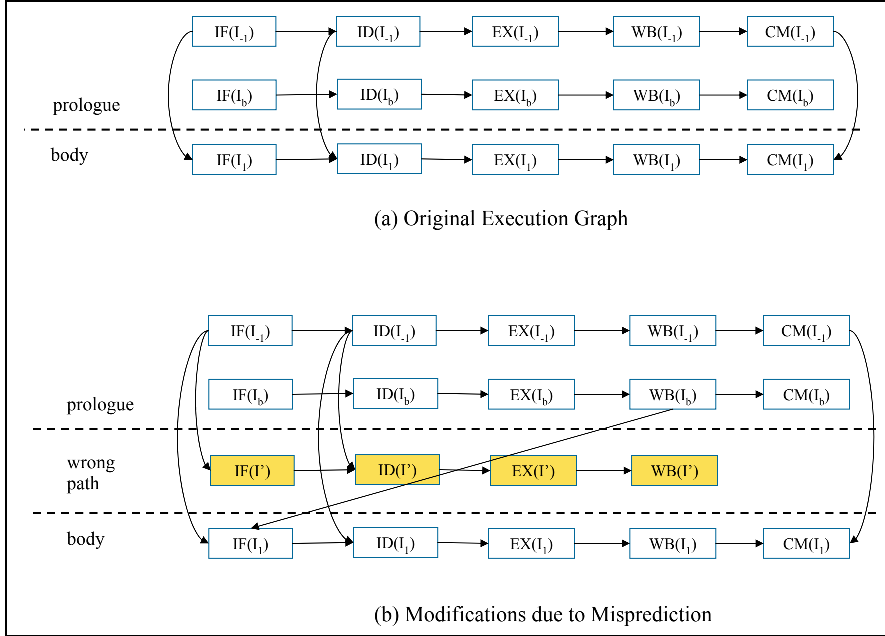

Figure 5: Execution Graph with Branch Prediction

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 referred to as *wrong path instructions*. In particular, we discuss the changes to execution graph nodes, dependence relation, contention relation and parallelism 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 I' is drawn for simplicity). There are no CM nodes for wrong path instructions as these instructions are not allowed to commit.

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, we do not need to change the execution graph. 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.

- $|Spec(b, X)| < ROB_size + IBuffer_size$ 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/parallelism relation Contention/parallelism among prologue, body and epilogue nodes remain unchanged. We also consider contention/parallelism of prologue and wrong path nodes in the estimation algorithm. Contention/parallelism 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 bounds.

6.2 Timing Estimation of a Basic Block in presence of Instruction Cache

So far we have assumed perfect instruction cache, that is, each instruction fetch takes a single clock cycle. We now discuss how we can capture the effects of instruction cache misses.

6.2.1 Categorization-based instruction cache analysis

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 of B_i 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. We now examine what may happen to a memory block $B_{i.j}$ when it is visited in the context of a loop level lp. For n-way set-associative cache, if lp contains less than n conflicting memory blocks of $B_{i.j}$, then the repeated visit to $B_{i.j}$ in lp will always be a cache hit; otherwise we classify it as "unknown", which means both hit and miss are possible. We define the following categorization function for this purpose.

$$cache_cat(B_{i.j},\ lp) = \begin{cases} hit, & \#conflicting\ memory\ blocks\ in\ lp\ < associativity \\ unknown, & otherwise \end{cases}$$

In a straightforward manner, we can take each loop level as the execution context, and perform the categorization of a basic block accordingly. However, a basic block may have identical categorizations in different loop levels, that is,

$$\forall j,\ cache_cat(B_{i.j},\ lp_1) = cache_cat(B_{i.j},\ lp_2)$$

In such cases, it is unnecessary to differentiate lp_1 and lp_2 as two contexts. Therefore, we can have fewer execution contexts than loop levels for categorization. We call such execution contexts as **cache scenario**s. All cache scenarios of B_i is denoted as Ω_i, which is a subset of the loop levels where B_i is contained. The number of times B_i is executed under a cache scenario ω, denoted as $N^{\omega}_{B_i}$, can be bounded in the following way. Recall that ω corresponds to a loop level, whose execution count is given as flow constraints. For ω_0, the cache scenario corresponding to the outermost loop level, we have the constraint

$$N^{\omega_0}_{B_i} \leq N_{\omega_0}$$

where N_{ω_0} is the execution count of the loop level of ω_0. For the rest of cache scenarios except the innermost one, we have constraints

$$N^{\omega_k}_{B_i} \leq N_{\omega_k} - \sum_{l<k} N^{\omega_l}_{B_i}$$

For the innermost cache scenario ω_k, it is constrained by

$$N^{\omega_k}_{B_i} = N_{B_i} - \sum_{l<k} N^{\omega_l}_{B_i}$$

6.2.2 Changes to timing estimation

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 that 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 we assign a one-cycle latency; otherwise an interval $[1,\ N]$ (where N is the cache miss penalty) is assigned to capture the possibilities of either hit or miss.

For the instructions in prologue and epilogue of B_i, we do not distinguish their cache scenarios. In other words, for prologue/epilogue instructions which are the first ones in their respective memory blocks, we conservatively assume their cache access results are unknown. Thus, we assign the interval $[1,\ N]$ to the IF stage for those instructions.

6.3 Putting it all together

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.

$$\text{Maximize} \quad T = \sum_{i=1}^{N} \sum_{j \to i} \sum_{\omega \in \Omega_i} t_{j \to i}^{c,\omega} * E_{j \to i}^{c,\omega} + t_{j \to i}^{m,\omega} * E_{j \to i}^{m,\omega} \tag{10}$$

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 $\omega \in \Omega_i$. Recall that Ω_i is the set of all cache scenarios of block B_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 [LMR05], 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 [LMR05] we give a detailed ILP modeling to bound $E_{j \to i}^{c}$ and $E_{j \to i}^{m}$, which we do not repeat here. The key idea is to bound the number of mispredictions at B_i by considering a global branch prediction scheme and finding the possible predictor states when control reaches at B_i.

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

$$E_{j \to i}^{c} = \sum_{\omega \in \Omega_i} E_{j \to i}^{c,\omega}; \quad E_{j \to i}^{m} = \sum_{\omega \in \Omega_i} E_{j \to i}^{m,\omega} \tag{11}$$

With the above two sets of constraints, $E_{j \to i}^{c,\omega}$ and $E_{j \to i}^{m,\omega}$ can be bounded.

Finally, the objective function in Equation 10 can be maximized by the ILP solver subject to (1) the control flow constraints and information about program loop bounds, (2) modeling of branch prediction [LMR05], (3) instruction cache modeling described in Section 6.2, and (3) the constraints presented in this section.

Program	Description	Bytes	#P	#BB	#BR	#LP	S
adpcm	Adaptive pulse code modulation	7296	16	140	43	18	N
dhry	Dhrystone benchmark	3144	16	98	34	8	Y
fdct	Fast Discrete Cosine Transform	2800	1	10	3	3	Y
fft	1024-point Fast Fourier Transformation	2216	1	27	8	5	Y
fir	FIR filter with Gaussian function	3824	7	69	13	8	N
ludcmp	LU decomposition algorithm	4728	2	60	17	11	N
matsum	Summation of two 100x100 matrices	232	1	5	2	2	Y
minver	Inversion of a floating point matrix	6144	3	102	31	17	N
qurt	Root computation of quadratic equations	1928	3	32	8	1	N
whet	Whetstone benchmark	2520	4	36	18	8	Y

Table 1: The Benchmark Programs

7 Experimental Evaluation

In this section, we evaluate the accuracy of our estimation technique with ten benchmarks (see Table 1). These programs have previously been used by other researchers for WCET analysis. Among them, `dhry`, `fdct`, `fft`, `matsum`, and `whet` were used by Li et al. [LMW99]; the others are from the real-time research group at Seoul National University [Rea] and the Real-Time Research Center at Mälardalen University [Mal].

In Table 1, column "Bytes" gives the size of the object code for each benchmark program. Here we do not count library code or other segments that are not included in our WCET analysis (data segments, stack, symbol table, etc). Column "#P" gives the number of procedures in each benchmark. Column "#BB" gives the total number of basic blocks in each program. Column "#BR" gives the number of conditional branches. Column "#LP" gives the number of loops. Finally, column "S" indicates whether the program has a single execution path or multiple execution paths.

For the single-path programs (`dhry`, `fdct`, `fft`, `matsum`, and `whet`) whose branch conditions are not dependent on input data, if the execution latencies of the instructions are fixed, we can precisely know their actual WCET by simulation. However, apart from `matsum`, all the single path programs contain variable latency instructions. Therefore, we cannot possibly use simulation to determine their WCET. Of course, for the multiple path programs (`adpcm`, `fir`, `ludcmp`, `minver`, `qurt`) the WCET estimation needs to consider variations in execution times due to multiple program paths as well variable latencies of instructions in each path.

7.1 Methodology

We use the SimpleScalar architectural simulation toolset [BA97] 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, degree of superscalarity, the latencies of the func-

tional 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 [CPL02], 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. We consider a two-issue superscalar processor consisting of an in-order front-end (IF and ID stages), an out-of-order execution core (EX and WB stages), and an in-order back-end (CM stage). The processor has a 4-entry instruction fetch buffer and 8-entry re-order buffer or ROB. It contains the following functional units: (a) two ALUs — each with single cycle latency, (b) a multiplication unit with variable latencies of $1 \sim 4$ cycles, (c) a floating point unit with variable latencies $1 \sim 12$ cycles, and (d) a load/store unit with a single cycle latency since we assume a perfect data cache.

The branch predictor is gshare [McF93, YP92]. It has a 128-entry branch history table (BHT). The three most recent branch history bits are XOR-ed with the seven least significant bits of the branch address to index into the BHT. 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 a constant as it depends on when the branch is resolved, i.e., the WB stage of the mispredicted branch is completed. We use 1KB two-way *set associative* instruction cache with 16 sets and 32 bytes line size. Thus `matsum` can be completely accommodated by the cache. The other programs have sizes ranging from two to seven times of the cache size, thus they will suffer from conflict misses. We assume that the cache miss penalty is 30 clock cycles. We conducted all our experiments on a 3 GHz Pentium IV PC with 2 GB main memory; the operating system running on the PC was Fedora Core 3 (Linux Kernel 2.6.12).

Program	Obs. WCET (cycles)	Est. WCET (cycles)	Ratio	Analysis Time (sec)	ILP Solving Time (sec)
adpcm	117835	166846	1.42	1.92	0.01
dhry	65498	85244	1.30	1.19	0.01
fdct	2691	2882	1.07	0.05	0.01
fft	749465	845295	1.13	0.23	0.01
fir	37862	42383	1.12	0.51	0.01
ludcmp	7024	8564	1.22	0.22	0.01
matsum	60406	60512	1.00	0.04	0.01
minver	4280	5259	1.23	0.77	0.01
qurt	1377	1709	1.24	0.50	0.01
whet	755354	803872	1.06	0.70	0.01

Table 2: Accuracy and running time of our out-of-order pipeline analysis

7.2 Results

We first present experimental results for pure pipeline analysis, that is, we assume perfect instruction cache and branch prediction. Under this configuration, we simply assume that each instruction fetch takes one clock cycle and every conditional branch is correctly predicted, that is, there is no pipeline stall caused by instruction cache miss or branch misprediction. Table 2 presents the observed WCET (*Obs. WCET*) and the estimated WCET (*Est. WCET*), as well as the ratio *Est. WCET / Obs. WCET*. The estimated WCET is not far from the observed WCET for most benchmarks. There are mainly two reasons for the overestimation — (1) the bounds on execution counts of basic blocks in the estimation are often higher than the actual execution counts during simulation (overestimation from program path analysis), and (2) the WCET estimation algorithm for the basic blocks introduces some amount of pessimism (overestimation from pipeline analysis).

To see how much overestimation is caused by our pipeline analysis, we use the execution counts of basic blocks observed in simulation as user constraints for analysis. Figure 6 compares the overall overestimation to pipeline-only overestimation. Benchmarks with single execution path are excluded from this figure as they suffer no overestimation from program path analysis. The `adpcm` benchmark has relatively higher overestimation. In this benchmark, the program path analysis contributes to a significant portion of the overestimation. For the other benchmarks shown in Figure 6, the WCET overestimation primarily comes from the pipeline analysis. However this overestimation does not constitute a large percentage of the observed WCET; so our estimated WCET is still close to the observed WCET as shown in Table 2.

Now we present the experimental results with the instruction caching and branch prediction enabled. These resuls are shown in Table 3. As can be seen from the ratio column, the overestimation increases due to the additional pessimism from branch prediction and instruction cache modeling. We can also observe some increase in WCET analysis time and

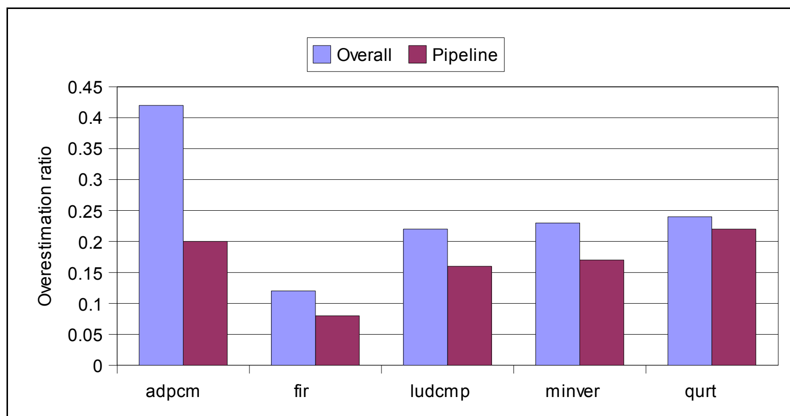

Figure 6: Overall and Pipeline Overestimations

ILP solving time, but the total estimation time (analysis + ILP solving) is less than 15 seconds for all our benchmarks.

Scalability of analysis Finally, we perform experiments to check whether our analysis scales up for different choices of the architectural parameters. In particular, we measure the changes in analysis time/accuracy owing to variation of cache configurations and re-order buffer (ROB) size. Figure 7 shows the variation of WCET overestimation under different cache configurations. Three cache configurations were used in these experiments. The first one is a 1KB cache (16 sets, 2-way, 32 bytes per cache line) with a cache miss penalty (CMP) of 30 clock cyles; the second one differs from the first one with a more aggressive CMP of 50 clock cycles; and the last one has a larger size of 4KB (32 sets, 4-way, 32 bytes per cache line). Except for the `qurt` benchmark, the variation of WCET overestimation due to variation in cache configuration is not significant.

At the core of an out-of-order processing pipeline is the re-order buffer or ROB. So, it is worth studying whether variations in ROB size affects our analysis time/accuracy. We found that the analysis accuracy is not significantly affected so we do not report these results here. The analysis time is however affected with variation in ROB size as can be seen from Table 4. Three sizes of the ROB are used in these experiments — a default 8-entry ROB, a 16-entry ROB, and a 32-entry ROB. The results show that the ROB size has a significant impact on analysis time. The main reason is that a larger ROB increases the size of each prologue/epilogue and also leads to many more prologues/epilogues for each basic block during pipeline analysis. The benchmark `fdct` is an exception — its analysis time increases only slightly with a larger ROB. This is due to the large basic blocks in fdct, which effectively limits the number of prologues/epilogues.

Program	Obs. WCET (cycles)	Est. WCET (cycles)	Ratio	Analysis Time (sec)	ILP Solving Time (sec)
adpcm	139346	227134	1.32	3.41	9.90
dhry	275177	436610	1.53	1.84	0.10
fdct	15006	16956	1.13	0.08	0.01
fft	944397	1146474	1.14	0.44	0.01
fir	77004	101333	1.26	0.79	0.31
ludcmp	17617	23818	1.28	0.43	0.12
matsum	62138	62734	1.01	0.07	0.01
minver	14221	21315	1.33	1.28	0.98
qurt	4114	6464	1.45	0.85	0.62
whet	760010	950818	1.14	1.24	0.01

Table 3: Accuracy and running time of our combined analysis for out-of-order pipeline, branch prediction and instruction cache.

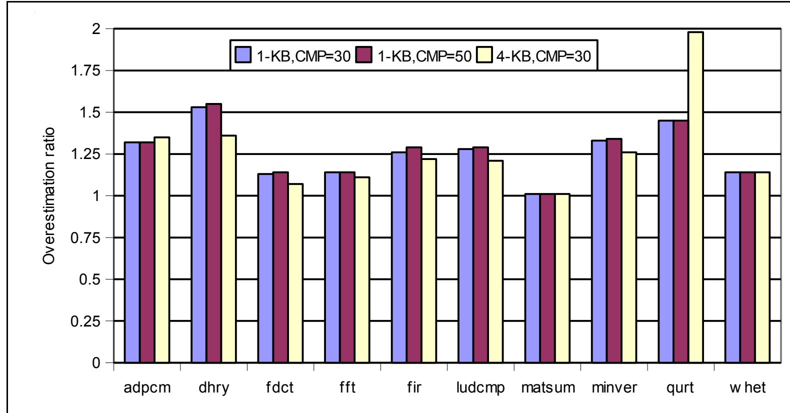

Figure 7: Variation of Overestimations under Different Cache Configurations

Program	8-entry ROB	16-entry ROB	32-entry ROB
adpcm	3.41	86.98	136.09
dhry	1.84	41.44	156.10
fdct	0.08	0.24	1.63
fft	0.44	3.25	177.06
fir	0.79	4.52	81.59
ludcmp	0.43	1.75	18.02
matsum	0.07	0.33	11.93
minver	1.28	8.35	168.59
qurt	0.85	7.34	156.78
whet	1.24	61.97	137.43

Table 4: Analysis Time (in Seconds) under Different ROB Sizes

8 Discussion

Timing anomaly complicates the Worst Case Execution Time (WCET) analysis of out-of-order pipelined execution. It invalidates the assumption that a global worst-case can be constructed by nicely composing local worst cases. On the other hand, an exhaustive enumeration of all possible local cases can be quite inefficient. In this paper, we have modeled an *out-of-order superscalar processor pipeline* for WCET analysis. The key idea behind our approach is to avoid exhaustive enumeration of instruction schedules 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 modeling 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.

Although the technique is proposed for out-of-order pipelines, it can also model many other architectural features. We first list the additional features that have been modeled and implemented in this paper.

Features that have been modeled Apart from out-of-order execution, we have modeled the following. *All our experiments include these features.*

- **Supercalarity** The *Parallelism Relation* introduced in this paper captures superscalar architectures.
- **Multiplicity of resources** Our processor contains two ALUs, and arithmetic instructions (except multiply and divide) can be issued to any of them.
- **Resource capacity** Some resources such as buffers and queues have limited capacity and thus introduce dependencies between instructions in an earlier pipeline stage and other instructions in a later stage. The examples include the fetch buffer and reorder buffer ($IF(I_i)$ depends on $ID(I_{i-2})$ if we have a 2-entry fetch buffer), and these dependencies have been captured in our execution graph.
- **Cache and Branch Prediction** We have modeled branch prediction and instruction caching as well as their interactions with the pipeline. Although each of the non-pipeline features need custom modeling techniques, the integration with the pipeline modeling has shown that the proposed technique for pipeline provides a good framework for incorporating other architectural features.

Features that can be modeled We give a list of miscellaneous features that can be modeled by the proposed technique (with some straightforward extensions to the execution graph or estimation algorithms). We have *not* implemented the modeling of these features in our WCET analyzer.

- **Multiple pipeline paths** An instruction may go though one of the several possible pipeline paths before these paths *merge* prior to instruction commit. In this case we

can conduct analysis for each individual pipeline paths until the merge point, where we bound the latest time at which the instruction can arrive at the merge point.

- **Other resource contentions** This paper focuses on contentions of functional units. In reality, contentions may arise in some other parts of the processor, for example, contentions for write-back ports, buses, etc. These contentions can be handled similarly provided the policy for resolving contentions is similar (i.e., break ties based on program order).

Finally, certain commercial processors may contain architectural features that do not fall into any of the above categories. Whether our proposed technique can model those new features with straightforward extensions is of course an open question.

Acknowledgments

This work was partially supported by University Research Council (URC) project R252-000-171-112 from National University of Singapore (NUS).

References

[AMWH94] R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 172–181, December 1994.

[BA97] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

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

[CPL02] CPLEX. The ILOG CPLEX Optimizer v7.5, 2002. Commercial software, `http://www.ilog.com`.

[Eng02] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, April 2002.

[HAM+99] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[HLTW03] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003.

[HP96] J.L. Hennessy and D.A. Patterson. *Computer Architecture- A Quantitative Approach*. Morgan Kaufmann, 1996.

[LBJ+95] 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):593–604, 1995.

[LHKM98] 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, December 1998.

[LMR05] X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems*, 29(1):27–58, January 2005.

[LMW99] 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):257–279, July 1999.

[LRM04] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 92–103, December 2004.

[LS99a] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, November 1999.

[LS99b] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 12–21, December 1999.

[LSD89] P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, 1989.

[LTH02] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Static Analysis Symposium (SAS)*, pages 294–309. Springer, September 2002.

[Mal] Malardalen Real-Time Research Centre. WCET Benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[McF93] S. McFarling. Combining branch predictors. Technical report, DEC Western Research Laboratory, June 1993.

[MD92] K. McMillan and D. Dill. Algorithms for interface timing verification. In *IEEE International Conference on Computer Design (ICCD)*, pages 48–51, October 1992.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, April 1989.

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

[SF99] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *International Workshop on Languages, Compilers and Tools for Embedded System (LCTES)*, pages 35–44, May 1999.

[Sha89] A.C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 1(2):875–889, July 1989.

[Soh90] G. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[TFW00] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analysis. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[The04] S. Thesing. *Safe and Precise Worst-Case Execution Time Prediction by Abstract Interpretation of Pipeline Models*. PhD thesis, University of Saarland, 2004.

[YP92] T.Y. Yeh and Y.N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 124–134, May 1992.

[YW98] T.Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, 1998.

[ZBN93] N Zhang, A Burns, and M Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.