Modeling Out-of-Order Processors for Software Timing Analysis

Xianfeng Li Abhik Roychoudhury Tulika Mitra
School of Computing, National University of Singapore
{lixianfe,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 microarchitectural features in modern processors (such as the pipeline, caches, branch prediction, etc.) to obtain safe but tight estimates. In this paper, we model out-of-order processor pipelines for WCET analysis. This analysis is, in general, difficult even for a basic block (a sequence of instructions with single-entry and single-exit point) if some of the instructions have variable latencies. This is because the worst case execution time 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 is inspired by an existing performance analysis technique for tasks with data dependences and resource contentions in real-time distributed systems. We extend our analysis by modeling the interaction among consecutive basic blocks as well as the effect of instruction cache. Finally, we employ Integer Linear Programming (ILP) to compute the WCET of an entire program. 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. Due to its inherent importance in schedulability analysis, such timing analysis of software has been studied extensively [3, 10, 11, 18, 21, 23]. Usually the analysis involves path analysis to find out infeasible paths in the program's control flow graph and microarchitectural modeling. Note that WCET analysis techniques are conservative, that is, they compute an upper bound of the program's actual worst case execution time. So, it is possible to ignore the effects of the underlying hardware by introducing pessimism. However, ignoring the microarchitectural features produces extremely loose timing bounds as modern processors employ advanced performance enhancing features such as the pipeline, cache, branch prediction, etc. To obtain a tight (yet safe) WCET estimate, we need to model the timing effects of microarchitectural features.

In the recent past, researchers have studied the effects of pipeline, cache and their interaction on program execution time [11, 14, 20, 23]. These works are based on assumptions that are 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 resource contentions, cache misses, branch mispredictions, and so on. Even in the embedded domain, some recent processors employ out-of-order pipeline; examples include Motorola MPC 7410, 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 of modeling such processors is due to the *timing anomaly* problem [15]. It shows 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. [7] modeled an out-of-order processor – PowerPC 755. In order to estimate the WCET of a basic block, they statically simulate the execution of the instructions using abstract pipeline states. However, due to the presence of timing anomaly, a pipeline state has to be split into multiple pipeline states if the latency of an instruction is unknown, for example.

In this paper, we show how to obtain a safe and tight WCET estimate for out-of-order pipeline without enumerating all possible instruction schedules. Our technique is inspired by an iterative performance analysis technique for real-time distributed systems proposed by Yen and Wolf [25], which estimates the execution time of tasks with data dependences and resource contentions. We exploit and augment their technique for estimating the WCET of a basic block by treating individual instruc-

tions as tasks. Clearly, there are data dependences between instructions in a program; resource contention is defined in terms of two instructions requiring the same functional unit. We then extend our solution for estimating the WCET of a basic block to arbitrary programs with complex control flows. The extension involves three steps. First, we construct a control flow graph (CFG) from an object code file. We apply the timing estimation technique to basic blocks. Next, we bound the timing effects of instructions preceding or succeeding the basic block. Finally, Integer Linear Programming (ILP) technique is employed to estimate the WCET of the entire program. We also extend our technique to include the effect of instruction cache.

The rest of this paper is organized as follows. Section 2 surveys related work on WCET analysis. Section 3 explains the technical difficulties in modeling out-of-order execution engines. The next two sections present our estimation technique: estimation for a basic block (Section 4) followed by the WCET estimation of a complete program (Section 5). Integrations of instruction cache and branch prediction with our pipeline analysis are discussed in Section 6. Experimental results are presented in Section 7. Concluding remarks appear in Section 8.

2. Related work

Research on WCET analysis was initiated more than a decade ago. Early activities can be traced back to [18, 21]. These works analyzed the program source code and 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* [23] to categorize the execution time of the instructions and then applies the Integer Linear Programming (ILP) to incorporate path constraints. The other one is an integrated approach proposed in the context of modeling instruction caches [11]. It employs an ILP formulation using path constraints derived from the control flow graph as well as constraints on cache behavior. A major concern with the ILP-only approach is the scalability of ILP problem size and/or solution times [24].

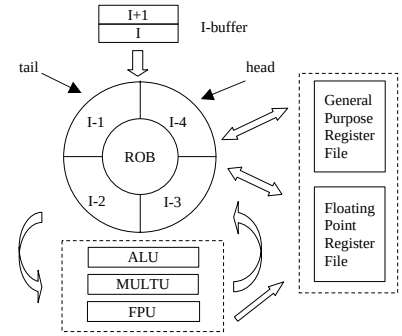
Pipelining is the core technique universally employed in modern processors and has been studied extensively for WCET analysis. Prior works in this area have successfully modeled in-order pipelines. Zhang et al. [26] modeled a simple pipeline structure with only two stages. Lim et al. [12] computed the WCET for RISC processors with pipelines and caches through an extension to Shaw's timing schema [21]. A case study with this approach for MIPS R3000/R3010 processors was made by Hur et al. [9]. Their work has been extended in [13] to model multiple-issue machines. Healy et al. [6] presented another approach for modeling processors with both instruction cache and pipeline. They first categorized cache behaviors of the instructions, and then used the cache information to analyze the performance of the pipeline. Lundqvist and Stenström [15] combined instruction level simulation with path analysis by allowing symbolic execution of instructions (whose operands are unknown). Schneider and Ferdinand [20] applied *abstract interpretation* for modeling superscalar processors (SUNSPARC 1). Recently WCET analysis on real modern processors have emerged. Langenbach et. al. [7] presented a work based on abstract interpretation on Motorola ColdFire 5307 which has pipelines, caches and branch prediction etc. The abstract pipeline state they used was just a set of concrete pipeline states and they reported that the amount of pipeline states tended to be tolerable. Heckman et. al. [7] modeled an out-of-order processor – PowerPC 755. Their way of handling out-of-order execution was to consider all interleavings of instructions. Due to the increased complexity, they suggested that out-of-order execution be limited.

Lundqvist and Stenström [16] observed 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 [16] presented 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. Recently, Engblom [3] conducted a comprehensive study of various pipelines and presented a framework for modeling those pipelines. Even though he studied timing anomalies, he did not propose any specific solution for modeling out-of-order processors.

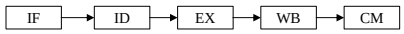
The approach for modeling out-of-order pipelines presented in this paper does not depend on special processor features for controling micro-architectural behaviors, neither does it need to modify the program object code. Also, this interval based technique is free of enumerating all possible interleavings of instructions, therefore it achieves high efficiency.

3. Difficulties in modeling out-of-order execution

Modern processors employ out-of-order execution where the instructions can be scheduled for execu-



(a) Processor model



(b) Pipeline stages

Figure 1: Our out-of-order processor pipeline model

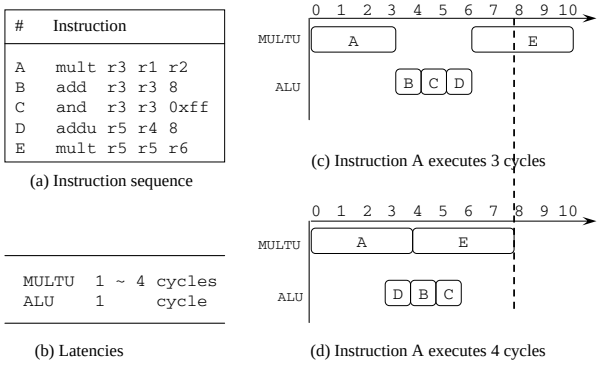
tion in an order different from the original program order. In such a processor, an instruction can execute if its operands are ready and the corresponding functional unit is available, irrespective of whether earlier instructions have started execution or not. The out-of-order execution improves the processor's performance significantly as it replaces pipeline stalls (due to dependences and/or resource contentions) with useful computations. However, the out-of-order execution mechanism makes WCET analysis difficult. We first introduce an out-of-order processor pipeline to illustrate these difficulties.

3.1. An out-of-order processor pipeline

Figure 1(a) shows an example of an out-of-order processor pipeline, which we will use to illustrate our estimation technique later in the paper. This pipeline is a simplified version of the SimpleScalar `sim-outorder` simulator pipeline [1], which in turn is based on [22]. The pipeline consist of five stages as shown in Figure 1(b).

1. **Instruction Fetch (`IF`)**. This stage fetches instructions from the memory *in program order* into the instruction fetch buffer `I-buffer`. Let us assume a 2-entry `I-buffer` for discussion.
2. **Instruction Decode & Dispatch (`ID`)**. This stage decodes instructions in the `I-buffer` and dispatches them into a circular buffer, called the re-order buffer `ROB` *in program order*. The `ROB`, a 4-entry buffer in our example, forms the core of the pipeline. Instructions are stored in this buffer from the time they are dispatched to the time they are committed (see `CM` stage).
3. **Instruction Execute (`EX`)**. An instruction in the `ROB` is issued to its corresponding functional unit for execution when all its operands are ready and the functional unit is available. If more than one instructions corresponding to a function unit are ready for execution, the earliest instruction is issued for execution. We assume that the functional units are not pipelined, that is, an instruction can be issued to a functional unit F only after the previous instruction occupying F has completed execution. We also assume that the number of instructions issued in a clock cycle is only bounded by the number of functional units. The `EX` stage exhibits true *out-of-order* behavior as an instruction can start execution irrespective of whether earlier instructions have started execution or not.
4. **Write Back (`WB`)**. In this stage, instructions that have finished execution forward their results to awaiting instructions, if any, in the `ROB`. If all the operands of an awaited instruction become ready, the instruction will be among the candidates scheduled for execution in the next cycle. We assume that there is no contention in the `WB` stage, that is, an instruction that has finished execution can always write back its results immediately. Clearly, instructions can write back results in *out-of-order* as well.
5. **Commit (`CM`)**. This is the last stage where the oldest instruction that has completed the `WB` stage writes its output to the register file and frees its `ROB` entry. Note that the instructions commit *in program order*. That is, even if an instruction has completed its `WB` stage, it still has to wait for the earlier instructions to commit. At most one instruction can commit each cycle.

In summary, in our processor model, `EX` and `WB` are the two pipeline stages where instructions can proceed out-of-order. There is resource contention only in the `EX` stage where instruction may compete for functional units.



(a) Instruction sequence

(b) Latencies

(c) Instruction A executes 3 cycles

(d) Instruction A executes 4 cycles

Figure 2: Example of timing anomaly due to variable latency instructions

3.2. Timing anomaly

The out-of-order execution has a serious impact on WCET analysis in the form of *timing anomaly* ob-

served by Lundqvist and Stenström [16]. Let us consider a variable latency instruction I with two possible latencies l_{min} and l_{max} such that $l_{max} > l_{min}$. 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 2 illustrates timing anomaly with an example. In the code fragment, instruction B depends on A, instruction C depends on B, and instruction E depends on D. Instructions A and E use the MULTU functional unit with latency of $1 \sim 4$ cycles and the other instructions use the single cycle ALU functional unit. We illustrate two possible execution scenarios. In the first scenario illustrated in Figure 2(c), instruction A executes for three cycles. Therefore, instructions B and C execute on cycle 3 and 4 respectively, respectively. Instruction D is ready for execution in cycle 3 itself, but it can only be scheduled for execution in cycle 5 after B and C (which appear earlier in program order). The overall execution time in this case is 10 cycles. In the second scenario as illustrated in Figure 2(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). Therefore D executes in clock cycle 3 allowing E to start execution in clock cycle 4. The overall execution time in this case is only eight cycles. Thus, *a longer latency of* A *results in a shorter overall execution time*.

In the presence of timing anomaly, techniques which generally take the local worst case for WCET estimation no longer guarantee safe bounds. For example, assuming the longest latency for variable-latency arithmetic instructions is not safe for WCET estimation of out-of-order processors. 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 take up to K^N rounds of estimations. To address this problem, Lundqvist and Stenström[16] proposed a program modification approach to enable safe local decision making for WCET analysis with out-of-order processors. They insert synchronization instructions before and after every variable-latency instruction in the code. A synchronization instruction flushes the pipeline, thereby enforcing a predictable pipeline state. However, pipeline flushes waste many clock cycles and lead to imprecise analysis.

4. Estimating the execution time of a basic block

Our effort in this section is to develop an algorithm for estimating the WCET of a basic block on our out-of-order processor pipeline. Instructions in a basic block are executed sequentially, that is, there is no non-determinism in terms of control flow transfer. In order to focus on pipeline modeling, we will initially assume that there are no cache misses or branch mispredictions. Later, we show how to integrate cache and branch prediction with our pipeline analysis.

Our approach avoids explicit enumeration of possible instruction schedules. It first formulates the problem in terms of an execution graph where the edges represent the data dependences as well as dependences among different pipeline stages. Secondly, it maintains the start and completion times of the pipeline stages corresponding to each instruction as conservative intervals. For example, the start time of a particular pipeline stage for an instruction is estimated as $[l, u]$, where l and u denote the earliest and latest possible start times, respectively. Clearly, the WCET of an instruction trace is then the latest possible finish time of the last instruction's commit stage. Our algorithm starts with very loose bounds on the intervals and iteratively tightens the bounds.

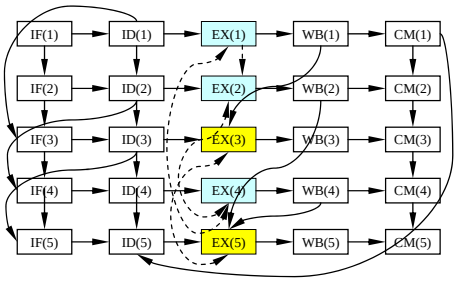
4.1. Problem formulation

Execution graph Given a straight-line code, each node in the execution graph represents a tuple, an instruction identifier and a pipeline stage, denoted as *stage(instruction identifier)*, e.g. IF(I) is the fetch stage of instruction I. If the code contains N instructions and the pipeline contains P stages, then the number of nodes in the execution graph is $N \times P$. Each node is associated with the latency of the corresponding pipeline stage. All the pipeline stages, except EX, have single cycle latency. If instruction I has variable latency, then EX(I) is annotated with $[l, u]$, where l and u are lower and upper bounds on latency, respectively. Solid edges in the graph represent dependences among the nodes. A dependence edge $u \rightarrow v$ from node u to v indicates that node v can start only after node u completes. There exists four types of dependences.

- Dependences among pipeline stages of the same instruction. This is because an instruction must proceed from the first stage to the last, for example, ID(I) must follow IF(I).
- Dependences due to in-order execution in IF, ID, and CM pipeline stages. Different instructions should proceed in program order through these pipeline stages, for example, IF(I+1) can only start after IF(I).
- Data dependences among instructions. If instruction I produces a result that instruction J uses, then we have an edge from WB(I) to EX(J). As we assume perfect branch prediction, we do not need to model control dependence edges even if the code contains branch instructions.

```
1: mult r6  r10  4
2: mult r1  r10  r1
3: sub  r6  r6   r2
4: mult r4  r8   r4
5: add  r1  r1   r4
```

(a) Code example



(b) Execution graph of the code

Figure 3: A sequence of instructions and the corresponding execution graph

- Dependences due to full `I-buffer` or `ROB`. For example, assuming that the `I-buffer` has two entries, there will be no entry available for `IF(I+2)` before `ID(I)` completes (which frees its entry from the `I-buffer`). Similarly, by assuming a 4-entry `ROB`, there will be an edge from `CM(I)` to `ID(I+4)` as `CM(I)` frees up an entry 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.

Dashed edges are drawn to reflect functional unit contentions among instructions. Thus, if instruction `I` can possibly delay the execution of instruction `J` by contending for a functional unit, then a dashed edge is drawn from `EX(I)` to `EX(J)`. It is not necessary that `I` appears before `J` in program order. If `I` and `J` can delay each other, then we will have a bi-directional dashed edge between `EX(I)` and `EX(J)`. In our example pipeline, resource contention happens only in the `EX` stage. There cannot be any contention between instructions with data dependences or between instructions that can never coexist in the `ROB` (their distance is greater than or equal to the capacity of the `ROB`).

Figure 3 shows an example of execution graph. Edges $\texttt{WB}(1) \rightarrow \texttt{EX}(3)$, $\texttt{WB}(2) \rightarrow \texttt{EX}(5)$, and $\texttt{WB}(4) \rightarrow \texttt{EX}(5)$ reflect data dependences. The dashed edges represent contention for functional units. For example, the bi-directional dashed edge between `EX(1)` and `EX(4)` implies: (a) if instructions 1 and 4 are both ready to execute and the functional unit `MULTU` is free, then `EX(1)` will have higher priority, and (b) if `EX(4)` has already started before `EX(1)` was ready, then `EX(4)` will be allowed to complete and thereby delay `EX(1)`. There is no preemption of `EX(4)` by `EX(1)`. Note that the dashed edge $\texttt{EX}(1) \rightarrow \texttt{EX}(2)$ is uni-directional. This is because the source operand registers of instruction 1 are a subset of those of instruction 2. This implies that instruction 1 will always be ready before instruction 2. Moreover, as instruction 1 has higher priority than instruction 2 due to program order, `EX(2)` cannot delay `EX(1)`.

Our execution graph is similar to the dynamic dependence graph among instructions of Fields et al. [5]. In their work, the dependence graph is obtained from a concrete simulation run, i.e., a trace of dynamic instructions. Therefore, the actual resource contentions exercised in that particular run are known and the nodes are annotated with the execution latency as well as the wait time for a functional unit. They study how much each instruction can be delayed (the slack) without increasing the execution time of the run. Our execution graph is static and all possible resource contentions between instructions are represented in the execution graph for purposes of static analysis.

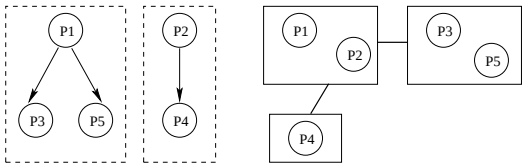
Problem definition Given a straight-line code consisting of a sequence of instructions $I_1, I_2, \ldots, I_N$ and the corresponding execution graph, we need to estimate its WCET, that is, the maximum completion time of the node $\texttt{CM}(I_N)$ assuming $\texttt{IF}(I_1)$ starts in time zero. Note that this problem is *not* equivalent to finding the longest path from $\texttt{IF}(I_1)$ to $\texttt{CM}(I_N)$ in the execution graph (taking the maximum latency of each pipeline stage). The execution time of a path in the execution graph is not a simple summation of the latencies of the individual nodes because of two reasons.

- The total time spent in making the transition from $\texttt{ID}(I)$ to $\texttt{EX}(I)$ is dependent on the contention from the 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 dependences, including data dependences.

4.2. A related problem

Given the problem formulation, we use an iterative algorithm to estimate the WCET of a sequence of instructions. The algorithm is safe, that is, it never produces an under-estimation. It begins with a very coarse approximation for the start and completion times of the nodes. These approximations are refined iteratively until (a) a fixed point is reached, or (b) a prescribed number of iterations is executed. The basic structure of our algorithm is inspired by a performance analysis technique for real-time distributed systems [25].

For the related problem, Yen and Wolf analyze a system consisting of several periodic tasks represented by task graphs as in Figure 4(a). Each task consists of a partially ordered set of processes, and process has lower and upper bounds on its computation time. The hardware architecture



(a) Task graphs (b) Distributed system

Figure 4: The distributed system (by Yen and Wolf)

consists of a set of Processing Elements (PE) connected via communication edges as in Figure 4(b). Processes are allocated to the PEs and priorities are assigned among the processes assigned to the same PE. A process P is scheduled to execute on a processor E if (1) all of P's predecessors have completed execution, and (2) no higher priority process in running on E. P can possibly preempt a lower priority process to start execution; on the other hand, P may itself get preempted by higher priority processes during its computation. The algorithm estimates the worst case completion time of all the tasks.

The problem addressed by Yen and Wolf's algorithm is similar to our analysis problem in some key aspects. For convenience, let us call their problem $\mathcal{P}$ and our problem $\mathcal{P}'$, respectively. The similarities are:

- The execution graph of $\mathcal{P}'$ is similar to the task graph of $\mathcal{P}$. In $\mathcal{P}$, processes have variable execution times; In $\mathcal{P}'$, EX stages have variable latencies.
- Edges in both graphs represent dependences among nodes. Moreover, there are resource contentions in both problems. In $\mathcal{P}$, contending processes are assigned fixed priorities by the system designer whereas in $\mathcal{P}'$, the program order determines priorities of instructions.

The similarities between the two problems render the framework of the algorithm in $\mathcal{P}$, which iteratively refines timing bounds of processes, applicable to $\mathcal{P}'$. However, there are some significant differences as well.

- In $\mathcal{P}$, tasks are periodic while the execution graph in $\mathcal{P}'$ is non-periodic.
- In $\mathcal{P}$, a higher priority process hp may delay a lower priority process lp by preemption; but lp cannot delay hp. However, in $\mathcal{P}'$, 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.

These 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 for the two problems.

4.3. Our approach

As discussed in Section 4.1, our problem is not equivalent to finding the longest path in the execution graph due to resource contentions and dependences. Dependences are taken care of by using a modified longest path algorithm that traverses the nodes in topologically sorted order. This topological traversal ensures that when a node is visited, the completion times of all its predecessors are known. To model the effect of resource contentions, we conservatively estimate an upper bound on the delay due to contentions for a functional unit by other instructions. A single pass of the modified longest path algorithm computes loose bounds on the lifetime of each node. These bounds are used to identify nodes with disjoint lifetimes. These nodes are not allowed to contend in the next pass of the longest path search to get tighter bounds. These two steps repeat till either there is no change in the bounds or a pre-defined number of iterations have elapsed.

Notations Before we discuss the algorithm, we explain the notations used.

- t_i^{ready}: Ready time of node *i* is defined as the time when all its predecessors have completed execution.
- t_i^{start}: Start time of node a *i* is defined as the time when it starts execution. For all the nodes except EX(I), $t_i^{start} = t_i^{ready}$. EX(I) may not be able to start execution when it becomes ready if another instruction is using the corresponding functional unit, or some higher priority instructions (earlier than I in program order) are also ready. In general, $t_i^{start} \geq t_i^{ready}$.
- t_i^{finish}: Finish time of a node *i* s defined as the time when it completes execution. Pipeline stages other than EX need only one cycle to execute. Therefore, $t_i^{finish} = t_i^{start} + 1$. For EX stage, we add the minimum (maximum) latency of the functional unit to t_i^{start} when we compute its *earliest* (*latest*) finish time.
- $separated(i, j)$: If the executions of the two nodes i and j cannot overlap, then $separated(i, j)$ is assigned to *true*; otherwise, they might overlap and it is assigned to *false*.
- $early\_contenders$(EX(I)): This is the set of EX nodes s.t. EX(J) $\in$ $early\_contenders$(EX(I)) if J appears before I in program order and there is a dashed edge denoting resource contention from EX(J) to EX(I) in the execution graph.
- $late\_contenders$(EX(I)): This is the set of EX nodes s.t. EX(J) $\in$ $late\_contenders$(EX(I)) if J appears after I in program order and there is a dashed edge denoting resource contention from EX(J) to EX(I) in the execution graph.

```
1  maxsep[., .] = ∞; step = 0;
2  foreach node i in G do
3      earliest[t_i^start] := 0; latest[t_i^finish] := ∞; latest[t_i^start] := ∞; earliest[t_i^finish] := MIN_LATENCY_i;
4  repeat
5      LatestTimes(G); EarliestTimes(G);
6      foreach i in G in topological order do
7          MaxSeparations(i);
8      step := step + 1 ;
   until maxsep[., .] is unchanged or step > limit;
9  WCET = latest[t_sink^finish]; /* sink is node in G representing commit of last instruction */
```

Figure 5: WCET Estimation of Execution Graph G

```
1  latest[t_src^ready] := 0; /* src is the root node of G */
2  foreach node i in G in topologically sorted order do
3      latest[t_i^start] := latest[t_i^ready];
4      S_late := late_contenders(i) ⋂ {j | ¬separated(i, j) ⋀ earliest[t_j^start] < latest[t_i^ready]};
5      if S_late ≠ φ then
6          latest[t_i^start] := min(max_{j∈S_late}(latest[t_j^finish]), latest[t_i^ready] + MAX_LATENCY_i − 1);
7      S_early := early_contenders(i) ⋂ {j | ¬separated(i, j)} ;
8      if S_early ≠ φ then
9          tmp := min(max_{j∈S_early}(latest[t_j^finish]), latest[t_i^start] + |S_early| × MAX_LATENCY_i);
10         latest[t_i^start] := max(tmp, latest[t_i^start]);
11     latest[t_i^finish] := latest[t_i^start] + MAX_LATENCY_i ;
12     foreach immediate successor k of i do
13         latest[t_k^ready] = max(latest[t_k^ready], latest[t_i^finish]);
```

Figure 6: LatestTimes(G)

WCET computation Figure 5 presents the algorithm for computing the WCET given an execution graph. The top level framework presented in Figure 5 is similar to [25]. However, as mentioned in Section 4.2, computation of latest times and earliest times (Figure 6 and 7) are different in our case. The top level algorithm iteratively performs two tasks: timing bounds computation and separations analysis. The first task is done by *LatestTimes* and *EarliestTimes*, which compute the upper and lower timing bounds of the nodes. The second task is done by *MaxSeparations*, which identifies pairs of nodes whose lifetimes are separated. The tighter the bounds obtained in the first task, 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 by the second task, the tighter are the timing intervals computed by the first task in subsequent iterations due to lesser number of competing nodes.

Figure 6 computes the latest ready, start, and finish times for each node of the execution graph. The latest start time of node i, denoted as $latest[t_i^{start}]$, is computed according to (a) its latest ready time $latest[t_i^{ready}]$ (which is obtained from the latest finish times of its predecessors), and (b) its contenders. We first consider the delay of i's start time by contenders later in program order, denoted as $late\_contenders(i)$. Obviously, a late contender j cannot start delaying i after i is ready because i has higher priority. Therefore late contenders who do not satisfy the condition $earliest[t_j^{start}] < latest[t_i^{ready}]$ are excluded. We also exclude the contenders who have been identified by *MaxSeparations* to be separated from i. The delay from a late contender j is bounded by j's latest finish time $latest[t_j^{finish}]$. In addition, j cannot delay i by more than its maximum latency; thus, we have another bound $latest[t_i^{ready}] + MAX\_LATENCY_i - 1$ where $MAX\_LATENCY_i = MAX\_LATENCY_j$ is the maximum latency of the contended functional unit. The minimum of the two bounds is taken. Note that the start of node i can be delayed by at most one late contender.

Apart from the delay due to late contenders of i, we also need to estimate the delay in i's start time due to its early contenders. Note that the early contenders appear before i in program order. So in the worst case, all of them, except those proved to be separated from i by the *MaxSeparations* algorithm, can contend with i and delay its start time. This is captured in lines 7–10 of Figure 6. The latest finish time of i is obtained by simply adding the maximum latency of the functional unit to $latest[t_i^{start}]$ (line 11). This is because an instruction cannot get preempted once it has started execution on a functional unit. The immediate successors of i get their latest ready times updated if i's latest finish time is higher than the current approximation of their latest ready times (see line 12–13 of Figure 6).

Similarly, Figure 7 computes the earliest ready, start, and finish times of all nodes in the execution graph. The main

```
1  earliest[t_src^ready] := 0;  /* src is the root node of G */
2  foreach node i in G in topologically sorted order do
3      earliest[t_i^start] := earliest[t_i^ready];
4      S_late := late_contenders(i) ⋂ {j | ¬separated(i,j) ⋀ latest[t_j^start] < earliest[t_i^ready] < earliest[t_j^finish]};
5      S_early := early_contenders(i) ⋂ {j | ¬separated(i,j) ⋀ latest[t_j^start] ≤ earliest[t_i^ready] < earliest[t_j^finish]};
6      S := S_late ⋃ S_early;
7      if S ≠ φ then
8          earliest[t_i^start] := max(max_{j∈S}(earliest[t_j^finish]), earliest[t_i^ready]);
9      earliest[t_i^finish] := earliest[t_i^start] + MIN_LATENCY_i ;
10     foreach immediate successor k of i do
11         earliest[t_k^ready] = max(earliest[t_k^ready], earliest[t_i^finish]);
```

Figure 7: EarliestTimes(G)

difference is that we allow a node j to contend and thereby delay the earliest start time of a node i only if the contention can be guaranteed.

Separation analysis identifies separated pairs of nodes. Given a pair of nodes i and j, if $earliest[t_i^{ready}] \geq latest[t_j^{finish}]$, i can never be ready before j completes execution; therefore, i and j cannot overlap. The algorithm *MaxSeparations* given by Yen and Wolf [25], which is a modification of [17], can identify more cases of separated nodes than the obvious ones satisfying the above constraint. In our problem, given two nodes i and j in the execution graph, we have simply set $separated(i,j)$ to true if $earliest[t_i^{ready}] \geq latest[t_j^{finish}]$ or $earliest[t_j^{ready}] \geq latest[t_i^{finish}]$. This simplified definition of separated nodes substantially increases the efficiency of our analysis; in the experiments we found that the resultant loss of precision is negligible in our problem.

5. Estimating WCET of a complete program

In this section, we discuss how the WCET of a complete program is computed based on our technique for basic blocks. Firstly, extentions to the technique are made by taking into account contexts before and after a basic block, as previous algorithms are based on assumptions that there are no instructions outside the basic block interacting with it and the pipeline is clean at the begining of a basic block. Secondly, after WCETs of basic blocks in the program are obtained, the integer linear programing technique is used to formulate the WCET of the overall program as an objective function to be maximized under constraints derived from the control flow graph. Extra constraints might be provided by the user to further exclude infeasible paths. Finally, the ILP solver will be used to search the worst case path and produce the WCET of the program.

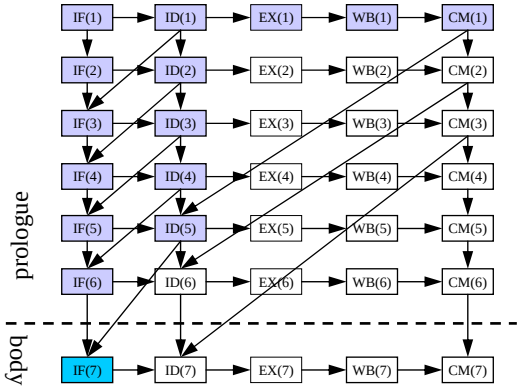


Figure 8: Conservative estimation of upper bounds of the prologue

5.1. Modeling impact from contexts

The execution context of a basic block B is defined in terms of instructions which are immediately preceding or succeeding B. Their effects to B are two folds: (1) pipeline stalls due to data dependences (only instructions preceding B can stall B in this regard); (2) pipeline stalls due to resource contentions. A problem is: how many context instructions need to be considered? Assuming a 2-entry instruction buffer and a 4-entry re-order buffer, at most 5 instructions remain in the pipeline when B enters the pipeline. Similarly due to the 4-entry reorder buffer, at most 3 instructions after B can contend with instructions in B (no data dependences from B on them, so only consider contentions). We will only consider these instructions which might affect B directly as contexts. Conservative assumptions are made in case of any uncertainties. For ease of discussion, we call context instructions before (after) a basic block B *prologue* (*epilogue*) of B. Now an execution graph consisting of three parts is constructed: the prologue, the basic block and the epilogue. The basic block in the middle is called the *body*.

Prologue To consider the effect from the prologue to the execution of the body, we first need to derive the timing

intervals of the prologue instructions, such that their impact on other instructions can be evaluated. As the execution context of the prologue itself is not clear, the process of deriving their intervals is conducted in the following way.

The latest times of the prologue nodes are computed according to several rules applied to the execution graph. The first rule says that a node x's latest finish time is bounded by its successor y's latest ready time because only after x has finished can y be ready. Therefore if y's latest ready time is known, it can be used as x's latest finish time. In case more than one successor's latest ready time is known, take the smallest one as y's latest finish time. The second rule says that a node x's latest start and ready times are bounded by its latest finish time. Therefore if x's latest finish time is known, its latest start time can be computed as: $t_x^{start} = t_x^{finish} - MIN\_LATENCY_x$, where $MIN\_LATENCY_x$ is the minimum latency of x. For x's ready time, as it cannot be later than its start time, it is safe to use x's latest start time as its latest ready time. Combining the two rules, latest times on any node in the prologue which has a path to the first node $IF(7)$ (whose ready time is assumed to be 0) of the body in Figure 8 can be obtained. Those nodes are shaded in Figure 8. The third rule sets the x's latest ready time with the maximum finish time of its predecessors. It will be used in combination with the latest time algorithm to compute the upper bounds for the rest nodes whose timing bounds cannot be simply derived from rule 1 and rule 2.

This process starts from $\texttt{EX}(2)$. It may have data dependences or resource contentions with earlier instructions which are not drawn in Figure 8. However, all the earlier instructions must have finished their executions at the ready time of $CM(1)$. Therefore, we take the maximum of $CM(1)$'s latest ready and $ID(2)$'s latest finish time as $EX(2)$'s latest ready time. Its latest start and finish times are computed by algorithm in Figure 4.3. Note at this moment, timing intervals of late contenders are unkown, therefore we let the late contentions be as much as possible (but they are still constrained by latencies of functional units). After $EX(2)$'s latest times are computed, $WB(2)$ and $CM(2)$ can be processed with rule 3. After that, other prologue instructions can be processed similarly as instruction 2. It proceeds in instruction order until all prologue nodes are processed.

For earliest times, as they have less significance on affecting the worst case execution time. We assume safe earliest times, $-\infty$, for prologue nodes.

Epilogue To compute timing intervals of epilogue instructions, contentions from their late contenders (which are unknown) have to be maximized for latest times, therefore line 6 in Figure 4.3 becomes $latest[t_i^{start}] = latest[t_i^{ready}] + MAX\_LANTENCY_i - 1$. If i is a single-cycle instruction, then there is no pessimism introduced. Otherwise its lastest start time could be overestimated for a few cycles. A longer epilogue, where instuctions directly contending with body instructions are followed by more instructions, could reduce overesitmation as their late contenders become available now. In practice, we make the epilogue size being twice the size of the reorder buffer. The computations of the earliest times of the epilogue instructions are the same as that of the body instructions.

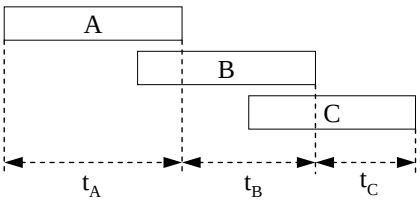


Figure 9: Overlaped execution

With timing intervals of prologue and epilogue nodes initialzed, body nodes can be processed similarly as in Section 4.3.

Overlap We note that the execution time of two consecutive basic blocks is not equal to the sum of their execution times in a pipelined processor due to overlapped execution, as shown in Figure 9. It is obvious that the overlapped part should be considered only once. Therefore the execution time of a basic block is redefined as the interval between the commit of the last instruction preceding it to the commit of the last instruction of its own (i.e., block B in figure 9). The only exception is the first basic block (i.e., block A in Figure 9) in the program, for which we use the old definition; this is because that block's execution time is not counted as part of any other block's execution time. To conservatively estimate the redefined execution time t_B of a basic block B (now the old definition is refered as t'_B), we need to estimate the minimum overlap, denoted as δ, between A and B and deduct it from t'_B. By the new definition, the overlap is the interval between the last commit of A to the first fetch of B. A simple method is used to estimate δ as below.

For convenience, let us give indices to instructions in the two consecutive blocks A and B. The first instruction of B is 0; other instructions are indexed by their distances to this instruction, e.g., the last two instructions in A have indices -1 and -2 respectively. Now δ is the least possible time elapsed from $IF(0)$ to $CM(-1)$. , This can be done by checking the paths from $IF(0)$'s two predecessors, $IF(-1)$ and $ID(-2)$, to $CM(-1)$. The time taken from the finish of $IF(-1)$ to the finish of $CM(-1)$ cannot be less than the length of the longest path connecting them (here a path's length is the sum of minimum latencies of nodes on the path). Let the length be l_1, and the length of the longest path from $ID(-2)$ to $CM(-1)$ is l_2. Then δ is the minimum of l_1 and l_2.

So far we have presented a method for taking into account the impact of execution context and a method for deducting minimum overlap. Although the methods are safe, they may not produce tight results. Let us discuss the problem with a larger reorder buffer (8 entries) and a larger instruction fetch buffer (4 entries), which are the ones used in our experimentation. In this case, we have a larger prologue with 12 entries. $IF(0)$ (the first node in the body) has a predecessor $ID(-4)$ (in the prologue); $ID(-4)$ in turn has a predecessor $CM(-12)$. With above method, the latest finish times of $ID(-4)$ and $CM(-12)$ are 0 and -1 respectively. However, given the long distance between $CM(-12)$ and $IF(0)$, $CM(-12)$ normally finishes long before $IF(0)$ is ready. For example, the interval could be 7 cycles if instrutions go through the pipeline smoothly. Such a pessimistic value of $CM(-12)$ will make $CM(0)$'s latest finish time be at cycle 11 or later, which means it takes 11 cycles to go through a five-stage pipeline (from $IF(0)$ to $CM(0)$). Although this is possible, but it rarely happens in reality.

To get tight estimation, We present an alternative way for dealing with the context. This method assumes time 0 at the completion of $CM(-1)$ instead of at $IF(0)$. This way, latest times of all nodes in the prologue can be derived by a simple backward traversal from $CM(0)$. For example, the latest finish time of $CM(-12)$ computed is -11. Now $IF(0)$ is no longer ready at 0. Its latest ready time will be available after its predecessors ($IF(-1)$ and $ID(-4)$) have been traversed. By assuming minimum latencies of all instructions being 1 cycle, $IF(0)$'s latest ready time is at -4, and the interval bewteen $CM(-12)$ and $IF(0)$ is 7 cycles – much better than the 1 cycle interval obtained from the earlier method! One may ask: Why not abandon the earlier method and just use this one? This is because we cannot derive lower bounds for any nodes, either in the prologue or in the body, with this method. The lack of meaningful lower bounds prevents us from getting tight execution time with our algorithm – no pair of contenders will be identified as *separated*. Therefore we use the earlier method to identify impossible contentions, which will be explored by an extra pass using this method. The extra pass will produce an improved estimation. Note he extra pass has implicitly considered the overlap since we assume the finish of $CM(-1)$ be time 0. Therefore no need to derive a minimum overlap.

5.2. Using ILP to model control flow

After the WCETs of basic blocks are estimated, the WCET of the overall program can be conveniently computed by using the integer linear programing technique. The WCET is formulated as an objective function to be maximized:

$$T_{prog} = \sum_{i=1}^{N} t_i * n_i$$

where t_i is the estimated WCET of basic block i and n_i is a variable denoting the execution counts of basic block i. The objective function is maximized subjected to a set of constraints formulated from the control flow graph. Extra constraints are provided by the user to further exclude infeasible paths therefore improve the quality of the produced results. Note loop bounds and recursion depths of procedures which have not been derived automatically must be provided by the user.

Another option is using program paths instead of basic blocks as analysis units. A program path typically spans several basic blocks. The advantage of using program paths is that the overestimation introduced by execution contexts is dilated. In our work, paths are formed with following constraints.

1. A path always starts/ends at boundaries of basic blocks.
2. A path stops at a loop back, procedure call or return.
3. A path spans no more than 8 basic blocks.
4. A path has 16 or more instructions, unless limited by above two constraints.

After program paths are formed, a *path flow graph* (PFG) is constructed, with nodes being program paths and edges between paths. An edge from path P_1 to P_2 indicates an edge from P_1's last basic block to P_2's first basic block in the orginal CFG. When ILP problem is formulated, flow constraints will be based on paths instead of basic blocks. For user constraints which are given in terms of basic blocks, they will to be translated into constraints in terms of paths. This can be done by replacing a basic block variable with paths spanning it. PFGs are usually more complex that CFGs due to several reasons. Firstly, nodes in PFGs are more strongly connected. Secondly, it takes more time to analyze a unit with a longer sequence of instructions. Lastly, the transformed user constraints are more complex thus will take the ILP solver more time. We will evaluate the increased complexity in Section 7.

6. Integrate modelings of other microarchitectures

Microarchitectures, like branch prediction, instruction cache and pipeline, affect the instruction timing dynamicly. Modeling them together has been proved to be more difficult than modeling them in isolation, mainly due to their interactions.

In this section, we discuss how modelings of other microarchitectures can be integrated with our pipeline analysis. We select two microarchitectures: instruction cache and branch prediction, which widely employed in modern processors, for discussion.

6.1. Instruction cache

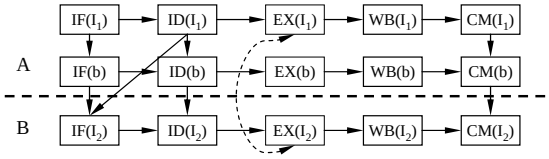
Instruction cache is employed to bridge the gap between a faster processor and a slower memory system. Our method of integrating cache modeling with pipeline analysis is a separated approach: instruction cache modeling is followed by pipeline modeling. For instruction cache modeling, we adopt the method proposed by Ferdinand and Wilhelm [4], in which instructions are categorized into four classes: *always hit, always miss persistent* (A piece of code is persistent if it is guaranteed to stay in the instruction cache once it is loaded.) and *not classified*. The cache categorizations will be incorporated into the pipeline analysis in a way as described next.

In pure pipeline modeling, `IF(i)` always finishes in one cycle. By assuming a cache miss taking N cycles, an `IF(i)` classified as *always miss* takes N cycles; an *not classified* `IF(i)` takes either one cycle or N cycles, in this case, we can safely say it happens in an interval $[1, N]$; an `IF(i)` classified as *always hit* takes one cycle; finally for an `IF(i)` classified as *persistent*, its first execution is analized with an N cycle execution time and its other excutions are analized with an 1 cycle execution. Now consider the algorithm for computing latest times in Figure 4.3 for a node `IF(i)`. As there is no contention in the instruction fetch stage, there is nothing to do from line 4 to line 10. On line 11 $MAX\_LATENCY_i$ is now N instead of 1 in previous pure pipeline modeling when `IF(i)` is either an *always miss* or *not classified*. It can be seen that essentially there is no change the algorithm for latest times, similarly for the algorithm for earliest times.

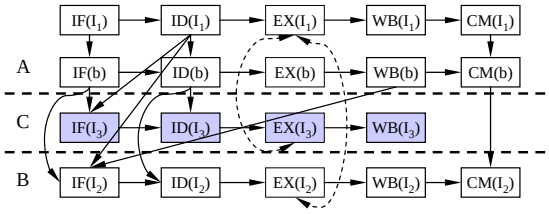
6.2. Branch prediction

Branch predictions are widely employed in modern processors for reducing pipeline stalls due to control hazards [8]. In this part, we discuss the integration of branch prediction modeling with out-of-order pipeline modeling. We assume there is a technique that can categorize each conditional branch as: (a) *always correct*, (b) *always mispredicted*, and (c) *unclear*.

Let us now consider the example given in Figure 10(a). There is a branch instruction b. Instructions down to b are called part A; b's succeeding instructions are called B. For simplicity, we only show one more instruction I_1 in A and one instruction I_2 in B. Obviously, if b is categorized as *always correctly predicted*, then no changes are needed. If b



(a) Original execution graph



(b) Modifications due to branch misprediction

Figure 10: Changes of execution graph with branch prediction

is categorized as *always mispredicted*, instructions on the wrong path will be partially executed and they affect other instructions. We make following changes to capture their effect. First, we insert the instructions on the wrong path. The number of such instructions are bounded by the size of reorder buffer. In Figure 10(b) they are labeled C and we only show one instruction, I_3. Since I_3 cannot commit, its commit node is removed. Next we construct edges. Edges connecting instructions in A and those in C are constructed as usual, e.g., an edge $EX(I_1) \leftrightarrow EX(I_3)$ is drawn if we assume I_1 and I_3 are contenders. Edges connecting instructions in A and those in B are not affected by the insertion of C, e.g., $ID(I_1) \rightarrow IF(I_2)$. There are no edges between C and B because their lifetime do not overlap, e.g., even if I_3 and I_2 use the same functional unit and no data dependence, they cannot contend with each other. Additionaly, an edge $WB(b) \rightarrow IF(I_2)$ is drawn to reflect the fact that instructions will be fetched upon the resolving of the branch outcome. Finally, since instructions in C are flushed when b is resolved, their contentions to instructions in A cannot be beyond the finish time of $WB(b)$. We can use this fact to further tighten the estimation. This can be done by changing $latest[t_j^{finish}]$ into $min(latest[t_j^{finish}, latest[t_{WB(b)}^{finish}])$ on Line 6 in the algorithm in Figure 6 in case j is an instruction from C.

If b is categorized as *unclear*, safe estimation of WCET bound requires that both the correct and incorrect prediction of b are considered. For this purpose, we make following modifications to Figure 10(b). All contention edges involving a node from C are now defined as *conditional edges*, i.e., $EX(I_1) \rightarrow EX(I_3)$. These conditional edges are only considered for estimating latest times and are ignored for estimating earliest times for nodes in A. The extra edge $WB(b) \rightarrow IF(I_2)$ is also a conditional edge, which

is only considered for estimating latest times for nodes in B. The intuition behind this approach is that both possibilities of the prediction of b are taken into account therefore safe upper bounds and lower bounds are guaranteed.

6.3. Branch prediction + Instruction cache

We have discussed how instruction cache and branch prediction can be integrated individually with pipeline analysis. They serve as the basis for the task in this part – integrate the three parts together. On the other hand, our experience of working on them gives us such observation, which we think should be a general guideline for modeling microarchitectures for WCET analysis.

Our observation is: Since effects of other microarchitecures will eventually be reflected by the pipeline behavior (An instruction cache miss stalls pipeline fetch for some cycles, a branch misprediction results in pipeline flush, etc. This is natural becauase the pipeline is where input data is fed in, processed and output data is produced. Other microarchitectures are employed just to make the pipeline running as smoothly as possible), pipeline analysis should be the core of a timing analysis framework. How a pipeline analysis depends on analyses of other components is crutial for the entire framework. For example, if a pipeline analysis technique is more capable of proceeding on with less clear behaviors of other microarchitectures (cache hit , cache miss, or both possible?) than another technique which asks for deterministic information (cache hit or cache miss?), meanwhile it does not sacrifice much accuracy, then it is more applicable to processors with complex architectures.

Fortunately, our interval based pipeline analysis is not very demanding on information of other microarchitectures. For example, for an instruction i which might be hit or miss, we only need to adjust the latency of node $IF(i)$ to an interval between 1 and the cache miss penalty. For a branch instruction b which might be correctely predicted or mispredicted, a slight modification to the execution graph is sufficient, as shown in Section 6.2.

Now we present a framework to integrate branch prediction, instruction cache and out-of-order pipline together for WCET analysis. We make following assumptions: (1) No support for Multi-level branch predictions, that is, a branch prediction is not allowed inside a pending branch and the processor stalls subsequent instructions. (2) The branch predictor is only updated by the outcome of a branch, and no other components can update it in any other situation. (3) Instructions on predicted path are fetched from memory if they are missed from the instruction cache. However, the processor stops a pending fetch of a mispredicted instruction immediately at the resolving of the branch outcome. In such cases, the cache state is not changed by that instruction.

With the second assumption, instruction cache and pipeline cannot change the predicted direction of a branch, although they might change the duration of a branch prediction. But our technique does not depend on such details. Therefore we can first categorize branches and modify execution graphs similarly as in Section 6.2.

We also need to know cache behaviors of instructions for estimating execution times of basic blocks. Originaly, this was done by traversing the CFG iteratively. At each basic block, its cache state is updated and populated to its successors. The traversal will eventualy stop when no more change is made to the old cache state of any basic block.

Note in our problem, the cache state at a basic block B is no longer updated merely by instructions in B – those on the mispredicted (or possibly mispredicted) path should also be considered. The question is: how many such instructions should be considered and how many of them will definitely/possibly/never be brought into the instruction cache by a misprediction? As it depends on the pipeline behavior, it should be studied together with pipeline analysis. This can be done by pluging in some extra code into algorithms in Figure 4.3 and 4.3 for this purpose. The added code is given in Figure 11.

```
   /* This part is added between line 4 and 5 in Figure 4.3 */
1  tmp_state ← cache_state[bb];

   /* This part is added between line 11 and 12 in Figure 4.3 */
2  if i is an instruction fetch node then
3      if i is in normal execution then
4          FullUpdate(tmp_state, i);
5      else
           /* i is in misprediction of branch b */
6          if always_mispredicted(b) and latest[t_i^finish] ≤
           earliest[t_WB(b)^finish] then
7              FullUpdate(tmp_state, i);
8          else if earliest[t_i^finish] ≤ latest[t_WB(b)^finish] then
9              PartialUpdate(tmp_state, i);
10         else
11             /* no update to tmp_state */

   /* This part is added after line 9 in Figure 4.3 */
12 /* Update cache states of succeeding basic blocks with tmp_state; */
```

Figure 11: Modifictions to Figure 4.3 and Figure 4.3 for instruction cache categorization

In Figure 11, we first copy the cache state at the entry of the basic block under analysis into a temporary cache state (line 1). All updatings to the cache state are performed on the temporary cache state and it will be used to update cache states of succeeding basic blocks. Note the cache state remembers age inforamtion of instructions in terms of inter-

vals $[youngest, oldest]$. The age of an instruction i represents the number of newer *conflicting* instructions (in the same cache set of i but with different tags) in the cache. The values $youngest$/$oldest$ represent the least/most possible number of such conflicting instructions. Therefore the interval covers all possible situations. Line 6 to 11 update the cache state if node i is an instruction fetch. A normal fetch always updates the cache state. But the updating by a fetch in a misprediction depends on its timing as well as the timing of the mispredicted branch b'. If i always finishes before b's outcome is resolved, and b is *always mispredicted*, then i *definitely* updates the cache state (line 6 and 7). The procedure $FullUpdate(tmp\_state, i)$ on line 7 updates both youngest and oldest ages of instructions in the same set of i (the article [4] gives details on how ages of instructions are updated). If i can never finish before b's outcome is resolved, then i has no chance to update the cache state (line 10 and 11). In other situations, the updating could possibly happen, and the procedure $PartialUpdate(tmp\_state, i)$ only updates oldest ages of eligible conflicting instructions (to reflect the cache state in case the updating is true), but does not update their youngest ages (to reflect the cache state in case the updating is false). For the updating of i itself, only its $youngest$ age is updated to be 0, which means i could be the most recent instruction in the cache if the updating is true. Lastly, when the estimation in Figure 4.3 terminates, the temporary cache state is used to update cache states of succeeding basic blocks (line 11).

Now we have presented the algorithm for updating cache state at a basic block. The next problem is to estimate execution times of basic blocks. Recall in pure pipeline modeling, a basic block's exection time is esimated only once. Here it is not feasible because a stable cache state at a basic block takes multiple passes of traversing basic blocks in the CFG. Each time a basic block is reached, the algorithm (in Figure 4.3, 4.3, 7 with changes in Figure 12) is invoked. Before the final pass the block is reached, the cache information at its entry is coarse, and the estimated time will not be accurate. The algorithm for obtaining an accurate estimation is shown in Figure 12. performed.

```
1  Initialize empty cache states at each basic block;
2  worklist ← root;  /* root is the start block of CFG */
3  repeat
4      bb ← worklist;
5      Visit(bb);
6      foreach successor bb′ of bb do
7          if bb′'s cache state is changed by bb then
8              worklist ← bb′;
   until worklist is empty;
   foreach basic block bb ∈ CFG do
9      Estimate(bb);
```

Figure 12: Estimating execution times of basic blocks

In Figure 12, the CFG is travesed iteratively. Each time a basic block bb is visited, the procedure $Visit$ is invoked to estimate its execution time and to update the cache state. Its successors will be put into the $worklist$ if bb introduced changes to their cache states. The iterative process will terminate when the worklist is empty, which means all cache states of all basic blocks have been stablized. The rest work is easy. We estimate execution times of basic blocks with the stable cache information, then use them to formulate the ILP problem. Finally the WCET of the whole program will be estimated by an ILP solver.

Program	Description	Size
matsum	Summation of two 100x100 matrices	192
fdct	Fast Discrete Cosine Transform	5744
fft	1024-point Fast Fourier Transformation	2472
whet	whetstone benchmark	2496
fir	FIR filter with Gaussian function	3848
ludcmp	LU decomposition algorithm	5152
minver	Inversion of a floating point matrix	6664

Table 1. The benchmark programs

6.4. Conlcusion

Based on the disussions in Section 6.2 and 6.3, we believe integrating our pipeline analysis engine with other microachitectures is promising. However, it only provides a framework. WCET analysis on a processor also depends on whether we have a solution for modeling its other microachitectures. For example, we need a good quality technique for branch prediction modeling in Section 6.3. This is part of our future work.

7. Experimental evaluation

In this section, we evaluate the accuracy of our estimation technique for seven benchmarks listed in Table 1. These benchmarks have been widely used for WCET analysis. The first four benchmarks have been used by Li et al. [11] for WCET analysis and the other three benchmarks are from the real-time research group at Seoul National University [19]. Among them `matsum` has no variable-latency instructions, fdct has a few and the rest of the benchmarks have many variable-latency instructions.

7.1. Methodology

We have integrated instruction cache modeling with pipeline analysis, but our implement has not include branch prediction because so far we have not developed a technique that categorizes branches in good precision. Therefore we assume perfect branch prediction. We also assume load instructions always complete in a single cycle because data cache has not been modeled by us so far. Note

that even with these simplications, a program whose execution path is independent of data inputs may still have many possible execution times due to variable lateny instructions. Since a program usually has a huge set of data inputs, an exhaustive simulation is infeasible. What we do is to select several data inputs, which are likely to produce longer execution times than the rest data inputs, for simulation. Therefore the actual WCET is unknown and the observed WCET is an underestimation to it.

We use the SimpleScalar architectural simulation toolset [1] 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 estimation tool that accepts the SimpleScalar executable annotated with user-provided constraints such as loop bounds. It is parameterized with respect to the cache configurations, the latencies of the functional units as well as the number of entries in the I-buffer and the ROB. The estimation tool first disassembles the code, identifies the basic blocks, and constructs the the control flow graph (CFG) of the program. It then performs instruction cache analysis and feeds the cache information to the pipeline analysis, which produces WCETs for basic blocks. Finally, the tool generates ILP constraints and the objective function for the program's WCET. The ILP problem is solved by CPLEX [2], a commercial ILP solver.

Our processor model has a 4-entry I-buffer and an 8-entry ROB. It contains the following variable latency functional unit types: (a) an integer multiplication unit with $1 \sim 4$ cycle latency, (b) a floating point add unit with $1 \sim 2$ cycle latency, and (c) a floating point multiplication unit with $1 \sim 12$ cycle latency. In addition, the processor has an integer ALU unit and a load/store unit, each with one cycle latency. The instruction cache has a size of 4k bytes and its configuration is: 32 sets, 4-way associativity, 32 bytes per cache line, 1 cycle for a hit and 10 cycles for a miss. The replacement policy employed is LRU. We run all the experiments on a 1.3 GHz Pentium IV machine with 1 GB memory. The estimation tool takes less than 1 second for every benchmark. This includes the time taken by CPLEX solver, which is upto 0.1 second.

7.2. Results

We first justify the need to model out-of-order execution. A question that may arise is why not use in-order processors to avoid the complexity of WCET analysis. In particular, our modeling is not useful if the inaccuracy in estimation of out-of-order pipeline outweights the performance improvement it offers. In Figure 13, We compare the increase of WCETs observed on an in-order pipeline over WCETs observed on the out-of-order pipeline (the first set of bars) to WCET overestimation on the out-of-order pipeline (the second set of bars). The in-order execution is performed on an in-order pipeline model very similar to the out-of-order pipeline in this paper, execept that instructions are issued to functional units in program order. It can be seen that the increased time of in-order execution is significant, sometimes also much more than the corresponding overestimation. This fact justifies that modeling out-of-order pipelines for WCET analysis is necessary.

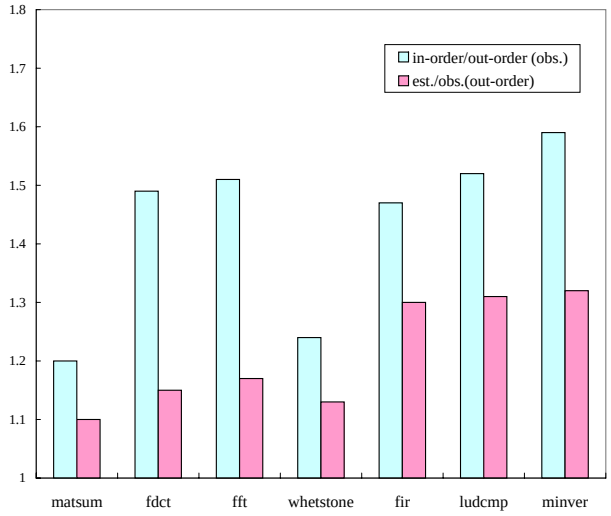


Figure 13: Comparison between increase of in-order/out-of-order observed WCETs and overestimation on out-of-order pipeline

Program	Obs. WCET	Est. WCET	Ratio
matsum	100813	111111	1.10
fdct	9131	10503	1.15
fft	1082847	1268466	1.17
whet	909033	1028778	1.13
fir	43358	56150	1.30
ludcmp	10681	14013	1.31
minver	6454	8546	1.32

Table 2. Accuracy of estimation

Table 2 gives details of the observed WCETs and estimated WCETs. Column *Obs. WCET* in Figure 2 gives the observed WCETS of the benchmarks, and column *Est. WCET* gives the estimated WCETS. Their ratios are listed in column *Ratio*. The estimated WCETs are not far from the observed WCETs considering high complexities of out-of-order execution. The inaccuracy in our estimation comes from two sources. Firstly, the bounds on execution counts of basic blocks in the estimation are often higer than what are actually executed in the simulation. Secondly, the algo-

Program	Obs. WCET	Est. WCET	Ratio
matsum	100813	106160	1.05
fdct	9131	10325	1.13
fft	1082847	1238706	1.14
whet	909033	999650	1.10
fir	43358	52397	1.21
ludcmp	10681	13388	1.25
minver	6454	8070	1.25

Table 3. Accuracy of analysis with paths

Program	block based		path based	
	Analysis	ILP	Analysis	ILP
matsum	0.01	0.01	0.03	0.01
fdct	0.06	0.01	0.08	0.01
fft	0.11	0.01	0.50	0.03
whet	0.08	0.01	1.80	0.10
fir	0.63	0.02	2.05	0.07
ludcmp	0.25	0.02	0.53	0.02
minver	0.04	0.04	3.05	0.16

Table 4. Time(sec.) of analysis/ILP solving

rithms for estimating timing intervals of a basic block without considering contexts could introduce some amount of overestimation. When execution contexts of a basic block are considered, a simple method is used to conservatively take into account their impact on the basic block. This would contribute some amount of overestimation as well.

As has been discussed section 5.2, we can transform the control flow graph into a path flow graph and use paths as analysis units. The advantage is that the impact of execution contexts and underestimation of overlap are dilated with a longer sequence of instructions and less frequency of such overestimations being introduced.

The results of program path based analysis are given in Table 3. The observed WCETs are not affected by analysis methods so the column *Obs. WCET* is kept unchanged. Compare to the basic block based analysis, the estimation produces tigher results as shown by column *Est. WCET* as well as column *Ratio*.

One concern is how much the complexity is increased by using program paths as analysis units. Table 4 compares the time of path based analysis to the time of basic block based analysis. Columns *Analysis* are the execution time of our program. The work of our program involves reading objective code of a benchmark, constructing control flow graph, performing some transformations on the control flow graph (such as loop unrolling, inter-procedural analysis), predicting WCET of analysis units, and formulating the ILP problem at last. Columns *ILP* are the time spent by

Program	block based		path based	
	#units	len/unit	#units	len/unit
matsum	10	5	14	15
fdct	36	32	27	51
fft	97	18	122	32
whet	64	16	158	26
fir	435	18	330	34
ludcmp	199	14	130	29
minver	333	13	415	34

Table 5. Complexity of analysis

Program	Obs. WCET	Est. WCET	Ratio
matsum	100867	111163	1.10
fdct	10658	12240	1.15
fft	1083416	1270386	1.17
whet	909531	1029380	1.13
fir	44120	59426	1.35
ludcmp	11948	16283	1.36
minver	8235	11053	1.34

Table 6. Accuracy of estimation

the ILP solver, which works on the formulated ILP problem and produces WCET. It can be seen that the analysis time and the ILP solving time are very short on all benchmarks for both basic block based analysis and program path based analysis. Although there are some amount of increase with the path based analysis, it is still very efficient.

Table 5 gives some explanation for what accounted for the increase of complexity in a path based analysis. Columns *#units* are the numbers of analysis units (basic blocks in CFG or program paths in PFG). Most benchmarks have more analysis units by using program paths. Columns *len/unit* give the average number of instructions in an analysis unit (epilogue instructions are counted as they are iterativly processed like body instructions). Because paths have more instructions than basic blocks, they take the algorithm more time to run.

Finally, we present results of integrated pipeline and instruction cache analysis. With the instruction cache configuration ($4k$ bytes) in our experimentation, only *fdct*, *ludcmp* and *minver* have conflicting cache misses, the other programs are smaller than the cache in size, therefore only cold misses arised. In Table 6, we observed similar relationship between estimation and observation. The slight increase of overestimation is mainly from instruction cache analysis, as what happened in other WCET analysis techniques which modeled instruction cache.

8. Discussion

Timing anomalies appearing in out-of-order processors complicate Worst Case Execution Time (WCET) analysis by invalidating the assumption that local worst case always lead to global worst case. On the other hand, an exhausive enumeration of all possible local cases is anticipated to be quite inefficient. In this paper, we have modeled an out-of-order processor pipeline. We also discussed how modelings of other microarchitectures. Two important microarchitectures: instruction cache and branch prediction are used as examples (the integratio with instruction cache has been implemented and results are fairly accurate). From these examples, we believe integrating our pipeline analysis engine with modelings of other processor components is promising. However, it only provides a framework. WCET analysis on a processor also depends on whether we have a solution for modeling its other microachitectures. For example, we need a good quality technique for branch prediction modeling in Section 6.2 and Section 6.3. This is part of our future work.

References

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

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

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

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

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

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

[7] 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), July 2003.

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

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

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

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

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

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

[14] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Intl. Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1998.

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

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

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

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

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

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

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

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

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

[24] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937*, 2004.

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

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