# 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 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 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. Such timing analysis of software has been studied extensively [4, 14, 15, 21, 24, 26] due to its inherent importance in schedulability analysis. Usually it 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 last decade, researchers have studied the effects of pipeline, cache and their interaction on program execution time. Most of 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, 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 [18]. The implication of this problem is that the overall WCET of a program can exceed the estimate obtained by maximizing latencies of individual instructions. Consequently, all possible schedules of instructions with variable latencies need to be considered for estimating the WCET of even a single basic block. Recently, Heckman et al. [9] have modeled PowerPC 755 (an out-of-order processor) for timing analysis. We discuss the relationship of their work with ours in the next section.

Our aim in this paper is 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 [28], which estimates the execution time of tasks with data dependences and resource contentions. For estimating the WCET of a basic block, we exploit and augment their technique by treating individual instructions as tasks. Clearly, there are data 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 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 effect of instruction cache and branch prediction.

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: WCET estimation for a basic block (Section 4) followed by the estimation of a complete program (Section 5). Integration of instruction cache and branch prediction with our pipeline analysis is 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 [21, 24]. 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 microarchitectural modeling. One of them is a two-phased approach; it uses *abstract interpretation* [26] 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 [15]. 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 [27].

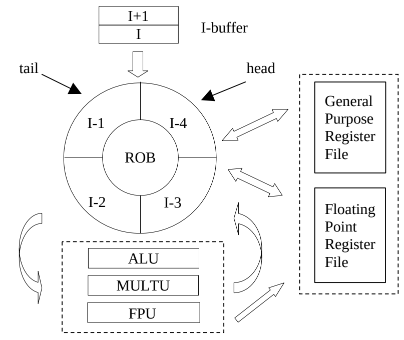
Pipelining is the core technique universally employed in modern processors and has been studied extensively for WCET analysis. Prior works in this area have successfully modeled in-order pipelines. Zhang et al. [29] model a simple pipeline structure with only two stages. Lim et al. [16] compute the WCET for RISC processors with pipelines and caches through an extension of Shaw's timing schema [24]. A case study with this approach for MIPS R3000/R3010 processors has been made by Hur et al. [11]. Their work has been extended in [17] to model multiple-issue machines. Healy et al. [7] 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 [18] combine instruction level simulation with path analysis by allowing symbolic execution of instructions (whose operands are unknown). Schneider and Ferdinand [23] have applied abstract interpretation to model superscalar processors (SUNSPARC 1).

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

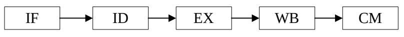
Our motivation towards studying pipeline modeling has come from a different angle. The problem of timing anomaly [19] makes it difficult to perform pipeline WCET analysis without an exhaustive enumeration of pipeline schedules. Thus, we are interested in the following question: can we achieve pipeline modeling where the time 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). To answer this question, we have considered a limited pipeline model where the timing anomaly arises from variable latency instructions and instruction cache. Note that we have also studied how features like branch prediction can be integrated into our interval-based pipeline analysis framework; however, this feature has *not* been implemented. Whether or not our integrated interval based modeling of all the micro-architectural features will scale up in timing/precision remains a topic of future investigation.

## 3. Difficulty in modeling out-of-order execution

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. The out-of-order execution improves processor's performance significantly as it replaces pipeline stalls (due to dependences and/or resource contentions) with useful computations. However,



(a) Processor model



(b) Pipeline stages

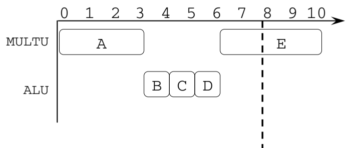
Figure 1: Out-of-order processor pipeline model

| # | Instruction |
|---|---|
| A | mult r3 r1 r2 |
| B | add r3 r3 8 |
| C | and r3 r3 0xff |
| D | addu r5 r4 8 |
| E | mult r5 r5 r6 |

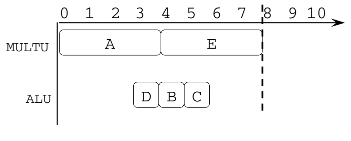
(a) Instruction sequence

| | |
|---|---|
| MULTU | 1 ~ 4 cycles |
| ALU | 1 cycle |

(b) Latencies



(c) Instruction A executes 3 cycles



(d) Instruction A executes 4 cycles

Figure 2: Timing anomaly due to variable latency instructions

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 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 [25]. 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 discussion, forms the core of the pipeline. Instructions are stored in this buffer from the time they are dispatched to the time they are committed (see `CM` stage).
3. **Instruction Execute (`EX`)**. An instruction in the ROB is issued to its corresponding functional unit for execution when all its operands are ready and the functional unit is available. If more than one instruction corresponding to a function unit are ready for execution, the earliest instruction is issued for execution. We assume that the functional units are not pipelined, that is, an instruction can be issued to a functional unit F only after the previous instruction occupying F has completed execution. We also assume that the number of instructions issued in a clock cycle is only bounded by the number of functional units. The `EX` stage exhibits true *out-of-order* behavior as an instruction can start execution irrespective of whether earlier instructions have started execution or not.
4. **Write Back (`WB`)**. In this stage, instructions that have finished execution forward their results to awaiting instructions, if any, in the ROB. If all the operands of an awaiting instruction become ready, the instruction will be among the candidates scheduled for execution in the next cycle. We assume that there is no contention in the `WB` stage, that is, an instruction that has finished execution can always write back its results immediately. Clearly, instructions can write back results 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.

## 3.2. Timing anomaly

The out-of-order execution has a serious impact on WCET analysis in the form of *timing anomaly* observed by Lundqvist and Stenström [19]. 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 cycles $3$ and $4$, respectively. Instruction `D` is ready for execution in cycle $3$ itself, but it can only be scheduled for execution in cycle $5$ after `B` and `C` (which appear earlier in program order). The overall execution time in this case is $10$ cycles. In the second scenario as illustrated in Figure 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 perform WCET estimation for $K^N$ schedules. To address this problem, Lundqvist and Stenström[19] propose 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.

In contrast, Engblom's framework [4] to model processor pipeline is based on the concept of pipeline *timing effects*, which reflects the impact of an earlier instruction's execution on later instructions. The execution time of a sequence of instructions is obtained by only considering its own execution time and the timing effects from nearby instructions. To make sure that there is no under-estimation, long timing effects (timing effects from remote instructions) should never be positive. The author discuss the conditions for the absence of positive long timing effects. The conclusions drawn are in favor of simpler in-order pipelines with no positive long timing effects.

## 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. First, we formulate the problem in terms of an execution graph where the edges represent the data dependences as well as dependences among different pipeline stages. Secondly, we maintain 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.

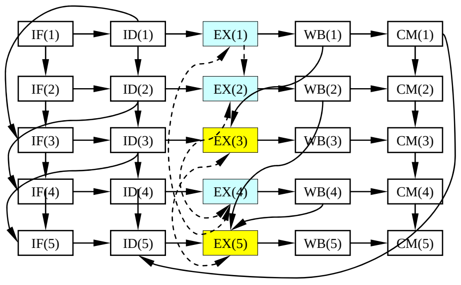
### 4.1. Problem formulation

*Execution graph* Given a basic block, each node in the corresponding execution graph represents a tuple: an instruction identifier and a pipeline stage, denoted as *stage(instruction identifier)*. For example, `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 an 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)`.

```
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 basic block and the corresponding execution graph

- 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)`.
- 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` (*i.e.* 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 contentions 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)` is 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 never be ready after 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. [6]. In their work, the dependence graph is obtained from a concrete simulation run, that is, a trace of dynamic instructions. Therefore, the actual resource contentions exercised in that particular run are known and the nodes are annotated with the execution latency as well as the wait time for a functional unit. They study how much each instruction can be delayed (the slack) without increasing the execution time of the run. Our execution graph is static and all possible resource contentions between instructions are represented for purposes of static analysis.

*Problem definition* Given a basic block 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 $\texttt{CM}(\mathrm{I_N})$ assuming $\texttt{IF}(\mathrm{I_1})$ starts at time zero. Note that this problem is *not* equivalent to finding the longest path from $\texttt{IF}(\mathrm{I_1})$ to $\texttt{CM}(\mathrm{I_N})$ 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 `ID(I)` to `EX(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 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 [28].

This technique by Yen and Wolf analyzes a system consisting of several periodic tasks represented by task graphs

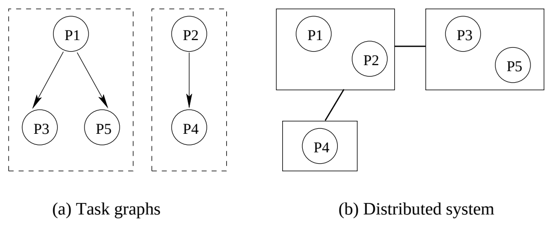


Figure 4: The distributed system (by Yen and Wolf)

as in Figure 4(a). Each task consists of a partially ordered set of processes, and each process has lower and upper bounds on its computation time. The hardware architecture consists of a set of Processing Elements (PE) connected via communication edges 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 execution. The algorithm estimates the worst case completion time of all the tasks.

The problem addressed by Yen and Wolf's algorithm is similar to our analysis problem in some key aspects. 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. In $\mathcal{P}$, one can use the well-known result by Lehoczky et al. [13] to calculate response time, whereas it is not possible to do so in $\mathcal{P}'$.

## 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* is 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)) iff 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)`) iff `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.

*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 [28]. However, as mentioned in Section 4.2, computation of latest times and earliest times (Figures 6 and 7) are different in our case. The top level algorithm iteratively performs two operations: timing bounds computation and separations analysis. The first operation is done by *LatestTimes* and *EarliestTimes*, which compute the upper and lower timing bounds of the nodes. The second operation is done by *MaxSeparations*, which identifies pairs of nodes whose lifetimes are separated. The tighter the bounds 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 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 time of node $i$ can be delayed by at most one late contender.

Apart from the delay due to late contenders of node $i$, 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 lines 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 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 [28], which is a modification of [20], 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. At the same time, experimental results indicate that the resultant loss of precision is negligible for 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 estimation technique for basic blocks. First, extensions to the technique are made by taking into account contexts before and after a basic block. This is because our estimation technique developed in the last section is based on assumption that the pipeline is clean at the beginning of a basic block. Secondly, after WCETs of basic blocks in the program are obtained, integer linear programming (ILP) 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. The program's WCET estimate is then provided by an ILP solver.

### 5.1. Modeling the impact of contexts

The execution context of a basic block $B$ is defined in terms of instructions which are preceding or succeeding $B$. Their effects on $B$ are two fold: (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 question here 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

```
1 $separated[., .] = false; step = 0;$
2 foreach node $i$ in $G$ do
3     $earliest[t_i^{start}] := 0; latest[t_i^{finish}] := \infty; latest[t_i^{start}] := \infty; earliest[t_i^{finish}] := MIN\_LATENCY_i;$
4 repeat
5     LatestTimes($G$); EarliestTimes($G$);
6     MaxSeparations($G$);
7     $step := step + 1$ ;
  until $separated[., .]$ is unchanged or $step > limit$;
8 $WCET = latest[t_{sink}^{finish}]$;  /* $sink$ 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     $S_{late} := late\_contenders(i) \bigcap \{j \mid \neg separated(i,j) \bigwedge earliest[t_j^{start}] < latest[t_i^{ready}]\}$;
5     if $S_{late} \neq \phi$ then
6         $latest[t_i^{start}] := \min\left(\max_{j \in S_{late}}\left(latest[t_j^{finish}]\right),\ latest[t_i^{ready}] + MAX\_LATENCY_i - 1\right)$;
7     $S_{early} := early\_contenders(i) \bigcap \{j \mid \neg separated(i,j)\}$ ;
8     if $S_{early} \neq \phi$ then
9         $tmp := \min\left(\max_{j \in S_{early}}\left(latest[t_j^{finish}]\right),\ latest[t_i^{start}] + \left|S_{early}\right| \times MAX\_LATENCY_i\right)$;
10        $latest[t_i^{start}] := \max\left(tmp,\ latest[t_i^{start}]\right)$;
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)

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 $B$ (we only consider contentions and not dependences because data dependences between $B$ and instructions after $B$ cannot affect the execution time of $B$). These instructions directly affect the execution time of $B$, and are captured as $B$'s context. 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 (called the *body*) and the epilogue.

We conservatively model the contention in the EX stage faced by instructions in a basic block $B$ due to instructions in the prologue and epilogue of $B$. Thus, we assume that (1) all contentions are true while estimating latest times, and (2) there is no contention in earliest times estimation. To model the effect of data dependencies between the prologue of $B$ and the instructions in $B$, we need to establish an upper bound on the times at which the instructions in prologue complete their WB stage; these can be directly obtained from upper bounds on their completion times for the EX stage. Due to the size restriction of the re-order buffer/instruction buffer and the structure of the execution graph, we can bound the latest commit time (and hence the latest time for completing EX) of the first instruction in $B$'s prologue w.r.t. the fetch of the first instruction in $B$. The latest commit time of the first instruction of $B$'s prologue in turn bounds the latest time for starting EX of the second instruction in $B$'s prologue and so on. Details are omitted for space considerations.

After modeling the contentions/data dependencies of a basic block $B$ with its prologue/epilogue, we run the *EarliestTimes* and *LatestTimes* algorithm (see Section 4) on $B$. These two algorithms iteratively tighten the bounds on the nodes of $B$, always using the contention/dependency information about the prologue and the epilogue. This allows us to safely estimate $t'_B$: *the time from the fetch of first instruction of $B$ to the commit of its last instruction*.

Finally, we note that the execution time of two consecutive basic blocks $B1$ and $B$ is not equal to the sum of their execution times in a pipelined processor due to overlapped execution. 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. The only exception is the first basic block 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 $t_B$ (the time from the commit of the last instruction of $B$'s predecessor (say $B1$) to the commit of $B$'s last instruction), we need to estimate the minimum overlap between $B1$'s and $B$'s execution. We omit the details of the overlap calculation here. The minimum overlap between $B1$ and $B$ is deducted from $t'_B$ (refer last paragraph) to obtain $t_B$, the execution time estimate of $B$.

```
1  $earliest[t^{ready}_{src}] := 0;$  /* $src$ is the root node of $G$ */
2  foreach node $i$ in $G$ in topologically sorted order do
3      $earliest[t^{start}_i] := earliest[t^{ready}_i];$
4      $S_{late} := late\_contenders(i) \bigcap \{j \mid \neg separated(i,j) \bigwedge latest[t^{start}_j] < earliest[t^{ready}_i] < earliest[t^{finish}_j]\};$
5      $S_{early} := early\_contenders(i) \bigcap \{j \mid \neg separated(i,j) \bigwedge latest[t^{start}_j] \leq earliest[t^{ready}_i] < earliest[t^{finish}_j]\};$
6      $S := S_{late} \bigcup S_{early};$
7      if $S \neq \phi$ then
8          $earliest[t^{start}_i] := \max\left(\max_{j \in S}\left(earliest[t^{finish}_j]\right),\ earliest[t^{ready}_i]\right);$
9      $earliest[t^{finish}_i] := earliest[t^{start}_i] + MIN\_LATENCY_i\ ;$
10     foreach immediate successor $k$ of $i$ do
11         $earliest[t^{ready}_k] = max(earliest[t^{ready}_k],\ earliest[t^{finish}_i]);$
```

Figure 7: EarliestTimes(G)

### 5.2. Using ILP to model control flow

For each basic block $B_i$, we can estimate $B_i$'s execution time with all possible contexts. The maximum of these estimates is taken as $B_i$'s WCET, denoted $t_{B_i}$. After the WCETs of basic blocks are estimated, the WCET of the overall program can be conveniently computed by using the integer linear programming technique. The WCET is formulated as an objective function to be maximized:

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

where $t_{B_i}$ is a constant (as mentioned in the preceding) and $n_i$ is an ILP variable denoting the execution counts of basic block $B_i$. The objective function is maximized subject to a set of constraints formulated from the control flow graph. These constraints correspond to the flow constraints, that is, the execution count of any basic block is equal to (a) the sum of its inflows, and (b) the sum of its outflows. In addition, the ILP solver is provided bounds on the maximum number of iterations for loops and maximum depth of recursive invocations for recursive procedures. These bounds can be user provided, or can be computed offline for certain programs (see [8]).

## 6. Integrating other features

In this section, we discuss how to model and integrate other microarchitectural features in our pipeline analysis framework. We select two such features for discussion: instruction cache and branch prediction.

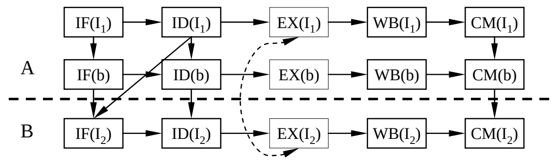
### 6.1. Instruction cache

Instruction cache is employed to bridge the gap between a fast processor and a slow memory system [10]. Our method of integrating instruction cache and pipeline is a separated approach where instruction cache modeling is followed by pipeline modeling. First, we adopt the method proposed by Ferdinand and Wilhelm [5] for instruction cache modeling. The instructions are categorized into four classes: *always hit*, *always miss*, *persistent* (an instruction is persistent if it is guaranteed to stay in the instruction cache once it is loaded) and *not classified*. Next, we integrate this categorization information into our pipeline analysis framework.

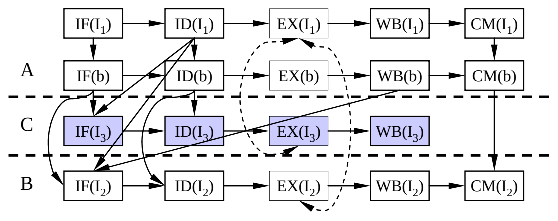
Without instruction cache modeling we assume that any `IF` node in the execution graph needs only one cycle to execute (perfect instruction cache). With instruction cache modeling, let us assume that cache hit latency is one cycle and cache miss latency is N cycles. Now, the execution time of an `IF` node depends on the categorization of the corresponding instruction as follows.

- If an instruction I is classified as *always hit*, then `IF(I)` node needs one cycle to execute as before.
- If an instruction I is classified as *always miss*, then `IF(I)` node needs N cycles to execute.
- If an instruction I is classified as *persistent*, then the the latency of the corresponding `IF(I)` node is N cycles for the first execution and one cycle for the subsequent executions.
- If an instruction I is *not classified*, then `IF(I)` node needs either one cycle or N cycles to execute. We represent its latency by an interval [1, N].

This change in the latency of `IF` nodes causes minimal modifications to our WCET estimation algorithm. Note that as the `IF` nodes execute in order, there is no contention among them. Therefore, lines 4–10 in algorithm *LatestTimes* (Figure 6) and lines 4–8 in algorithm *EarliestTimes* (Figure 7) remain unchanged. The only change is that the $MIN\_LATENCY$ and $MAX\_LATENCY$ used for execution latency of a node `IF(I)` should be modified in line 11 and line 9 in Figures 6 and 7 respectively; this modification should be done according to the categorization of instruction `I` as discussed above.



(a) Original execution graph



(b) Modifications due to branch misprediction

Figure 8: Execution graph with branch prediction

## 6.2. Branch prediction

Branch prediction is widely employed in modern processors to reduce pipeline stalls due to control hazards [10]. In this section, we discuss the integration of branch prediction modeling with our pipeline analysis. We first categorize each conditional branch as *always correctly predicted*, *always mispredicted*, or *not classified*. More refined categorizations of branches are possible as discussed in [2].

Let us now consider the example given in Figure 8(a) where $A$ is a basic block at the end of which a branch instruction $b$ appears. $B$ is the basic block that is executed if $b$ is correctly predicted. For clarity, we have shown only two instructions corresponding to A and one corresponding to B in Figure 8. Obviously, if $b$ is categorized as *always correctly predicted*, then our algorithm requires no changes. However, if $b$ is categorized as *always mispredicted*, then the instructions along the wrong path may affect the execution time. We make the following changes to capture its effect. We note here that the branch instruction $b$ can appear at the end of a basic block we are currently estimating or it may appear in the prologue/epilogue. In all the cases, the modifications to the corresponding portion of the execution graph remain the same. First, we insert the instructions along the wrong path after $b$, which are labeled $C$, in the execution graph as shown in Figure 8(b) (again for clarity, we have shown only one instruction of $C$ in the figure). Note that the length of $C$ is bounded by $R$, the size of the reorder buffer. However, as we do not know exactly when branch $b$ will be resolved, we conservatively include $R-1$ instructions along the wrong path in $C$ (i.e., $b$ might be resolved as late as possible). The instructions in $C$ cannot commit; therefore commit nodes, and their successors are excluded from the execution graph. The edges between nodes in $A$ and nodes in $C$ are built as usual. However, there is no edge between $C$ and $B$ because their lifetimes cannot overlap. Edges between $A$ and $B$ are kept unchanged; they are not affected by insertion of $C$. Again, this is because we do not know when $b$ will be resolved and so we conservatively assume that it might be resolved as early as possible. Finally, there is an extra edge connecting the write-back node of $b$ with the fetch node of the first instruction in $B$ to reflect the fact that instructions in the correct path are fetched after the resolution of the mispredicted branch.

If $b$ is categorized as *not classified*, safe estimation of WCET bound requires that both the correct and incorrect prediction of $b$ are considered. This can be done by constructing the execution graph as in Figure 8(b) with a simple modification: all contention edges involving a node from $C$ are now defined as *conditional edges*. These conditional edges are only considered for computing latest times and are ignored for earliest times calculation of the nodes in $A$. The extra edge from $b$'s write-back node to the first node in $B$ is also defined as a conditional edge and is only considered for latest times calculation of nodes in $B$. The intuition behind this approach is to take both possibilities of the prediction of branch $b$ into account so as to compute safe upper and lower bounds.

| Program | Description |
|---|---|
| matsum | Summation of two 100x100 matrices |
| fdct | Fast Discrete Cosine Transform |
| fft | 1024-point Fast Fourier Transformation |
| whet | Whetstone benchmark |
| fir | FIR filter with Gaussian function |
| ludcmp | LU decomposition algorithm |
| minver | Inversion of a floating point matrix |

Table 1: The benchmark programs

## 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. [15] for WCET analysis and the other three benchmarks are from the real-time research group at Seoul National University [22]. Among them `matsum` has no variable-latency instructions, `fdct` has a few, and the rest of the benchmarks have many such instructions. *In our experiments, we have integrated instruction cache modeling with pipeline analysis; however the modeling of branch prediction has not yet been integrated into our estimation tool.*

### 7.1. Methodology

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 of the program. It then performs instruction cache analysis and feeds the categorization information to the pipeline analysis module, which produces WCET for basic blocks. Finally, the tool generates the ILP constraints and the objective function for the program's WCET. The ILP problem is solved by CPLEX [3], a commercial ILP solver to generate the *Estimated WCET*.

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. Note that we assume single-cycle latency for load/store unit because we have not modeled data cache in our tool. We model a 4KB instruction cache with 4-way associativity, 32 byte line size and LRU replacement policy. Cache hit latency is one cycle and cache miss latency is 10 cycles. We also assume perfect branch prediction as we have not yet integrated branch prediction modeling in our tool. 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 up to $0.1$ second.

The detection of actual WCET for comparison purposes is quite difficult for out-of-order pipelines. We note that even for a program whose execution path is *independent* of data inputs has many possible execution times due to variable latency instructions. Since a program usually has a large set of possible data inputs, an exhaustive simulation is infeasible to detect the actual WCET. Instead, we use several data inputs that are likely to produce longer execution times for simulation. We call the maximum execution time produced through simulation as *Observed WCET* and it is a lower bound on the actual WCET.

### 7.2. Results

Table 2 presents the observed WCET (column *Obs. WCET*) and the estimated WCET (column *Est. WCET*), as well as the ratio of the estimated WCET to the observed WCET. The estimated WCET is not far from the observed WCET for most benchmarks specially considering the fact that the difference between actual and observed WCET is unknown. The inaccuracy in our estimation comes from three sources. First, the bounds on execution counts of basic blocks in the estimation are often higher than the actual execution counts during simulation. Secondly, the algorithm for estimating the WCET of a basic block and the impact of execution context introduces some amount of pessimism. Finally, some instructions that may hit in or miss in the cache are categorized as *not classified*. Therefore their execution times are considered as an interval from a single cycle (hit) to the cache miss penalty (10 cycles in our experiments). This gives rise to additional overestimation.

| 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 2: Accuracy of estimation

## 8. Discussion

Timing anomaly in out-of-order processors complicates WCET analysis by invalidating the assumption that local worst cases always lead to global worst case. On the other hand, an exhaustive enumeration of all possible local cases is anticipated to be quite inefficient. In this paper, we have modeled an out-of-order processor pipeline and combined it with instruction cache modeling for WCET analysis. Our approach avoids exhaustive enumeration of all cases. We have implemented our technique and experimentally validated its estimation accuracy using several standard benchmark programs. In future, we plan to implement and integrate the modeling of other microarchitectural features such as branch prediction in our analysis tool.

## Acknowledgments

We thank the anonymous reviewers for their comments. This work has been partially supported by National University of Singapore research project R252-000-171-112.

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

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

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

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

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

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

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

[9] 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.

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

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

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

[23] 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.

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

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

[26] 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.

[27] 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.

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

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