Pipeline Timing Analysis Using a Trace-Driven Simulator

Extended Remix*

Jakob Engblom††
IAR Systems AB
Box 23051, SE-750 23 Uppsala, Sweden
email: jakob.engblom@iar.se

Andreas Ermedahl‡
Department of Computer Systems, Uppsala University
Box 325, SE-751 05 Uppsala, Sweden
email: andreas.ermedahl@docs.uu.se

Abstract

In this paper we present a technique for Worst-Case Execution Time (WCET) analysis for pipelined processors. Our technique uses a standard simulator instead of special-purpose pipeline modeling.

Our technique handle CPUs that execute multiple shorter instructions in parallel with long-running instructions. The results of other machine analyses, like cache analysis, can be used in our pipeline analysis. Also, results from high-level program flow analysis can be used to tighten the execution time predictions.

Our primary target is embedded real-time systems, and since processor simulators are standard equipment for embedded development work, our tool will be easy to port to relevant target processors.

Keywords: WCET, pipeline analysis, hard real-time, embedded systems

1. Introduction

The purpose of *Worst-Case Execution Time* (WCET) analysis is to provide a priori information about the worst possible execution time of a program before using the program in a system.

Knowing the WCET of a program or piece of a program is a necessary part of designing and verifying real-time systems. Considering that every day, more and more devices are being controlled by embedded real-time systems (from kitchen appliances through power grids to cars and other vehicles), the value of having reliable software cannot be overestimated. In many cases, a failure of an embedded real-time system will lead to a disaster, sometimes including the loss of human life.

WCET estimates are used in real-time systems development to perform scheduling and schedulability analysis [2, 3], to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times.

To be valid, a WCET estimate must be *safe* (i.e. no underestimation of the execution time), and to be useful, it must be *tight* (i.e. as little overestimation as possible).

WCET analysis is performed under the assumption that the program execution is uninterrupted (no preemptions or interrupts) and that there are no interfering background activities. Issues like interrupts and cache interference between concurrent tasks are dealt with at a higher level (e.g. in system scheduling).

The WCET depends both on program flow (like loop iterations and function calls), and on architectural factors like caches and pipelines. This paper is focused on effects of pipelines, and we assume that we have access to the results of program flow analysis and cache analysis.

Our analysis technique is based on the *Implicit Path Enumeration Technique* (IPET) [14, 21, 23], in which a program is modeled at the object code level as a set of basic blocks[1]. To each basic block, *execution times* and *execution counts* (representing the number of times the basic block is executed in the worst-case execution of the program) are attached.

Constraints on the execution counts are used to model program flow. The final execution time estimate is obtained by maximizing the sum of the execu-

*This a longer version of a paper to be presented at the 6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99)

†Jakob is an industrial PhD student at IAR Systems and Uppsala university, sharing his time between research and development work.

‡This work is performed within the Advanced Software Technology (ASTEC, http://www.docs.uu.se/astec) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK, http://www.nutek.se).

[1] A basic block is a piece of object code that is always executed in sequence, i.e. it contains no branches in or out.

tion times multiplied by the execution counts, subject to the flow constraints. There is no need to explicitly enumerate the potential execution paths of the program, making the analysis efficient.

We use a simulator to obtain execution times. By feeding the simulator with information about the memory accesses of a program, we can incorporate results from instruction and data cache analysis.

Simulators are used extensively by embedded systems developers, and we can leverage existing simulators to port our technique to a variety of hardware platforms. Furthermore, some simulators can model hardware external to the CPU, like memory speeds and I/O-units. Compared to using a special-purpose pipeline model, we gain development time, portability, and modeling power.

The specific contributions of this paper are the following:

- We show how pipeline effects that reach across several basic blocks can be exactly modeled in the IPET framework.
- We show when execution constraints should be added to a program in order to produce better execution time estimates.
- We show how WCET analysis can make use of existing simulators, rendering special-purpose pipeline models unnecessary.

Paper outline: Section 2 presents previous work in the WCET field and Section 3 gives an overview of our WCET analysis method. Section 4 presents the problem of long-running instructions, and Section 5 how we model them in the IPET framework. Section 6 ties everything together by introducing an algorithm for timing analysis based on a simulator. Section 7 discusses the real-life applicability of our approach. Finally, Sections 8 and 9 present our conclusions and plans for future work.

2. Previous Work

The IPET methodology for WCET analysis was introduced by Puschner and Schedl [23]. A similar approach was presented by Li and Malik [14].

The method was extended by Ottosson and Sjödin to include the modeling of instruction and data caches and pipeline overlap between pairs of basic blocks [21]. Unfortunately, pipeline effects across more than two basic blocks were ignored and integrating the cache modeling into IPET made the execution time of the analysis very high.

We extend and refine the approach of Ottosson and Sjödin by modeling pipeline effects across an arbitrary number of basic blocks, by demonstrating how pipeline execution times can be obtained using a simulator, and by considering the cache analysis as a separate step.

Theiling and Ferdinand [26] use a separate cache analysis combined with IPET. However, their approach does not include the modeling of pipeline effects.

Park and Shaw introduced *timing schemas* [22], in which WCET analysis is performed by traversing an abstract syntax tree for a program. This approach was extended to handle instruction caches and pipelines by Lim et al [15]. Compared to IPET, this approach has problems expressing complex flow constraints and handling unstructured code. It also requires a special-purpose pipeline model.

Another approach to timing analysis is to analyze entire explicit paths. This approach is taken in the work of Healy et al, where instruction cache analysis is followed by exploration of program paths through loops and functions [7]. Stappert and Altenbernd also explore program paths to find the longest path [25]. The IPET approach differs from these path-based approaches by not exploring complete paths, avoiding a potential explosion in the number of examined paths.

A different approach is used by Lundqvist and Stenström, who combine flow analysis and architectural analysis using symbolic execution inside a CPU simulator [16]. In contrast, we only simulate small pieces of a program, and we do not need to modify the simulator.

3. WCET Analysis Overview

The results presented in this paper should be considered in the context of our complete architecture for WCET analysis, outlined in Figure 1 (a more detailed description is given in [4]).

In order to generate a WCET estimate, a program is processed through a number of modules:

The *compiler* converts the program source code (usually C) to object code, and provides the *flow analysis* with access to the program code. The flow analysis determines the program flow such as loop bounds, infeasible paths, data dependencies, etc.

The *cache analysis* uses the flow information and the object code to perform a static analysis of the cache behavior of the program.

The cache analysis and the flow analysis define *execution scenarios* for the basic blocks in the program. An execution scenario consists of a basic block together with information on how it executes (cache hits and

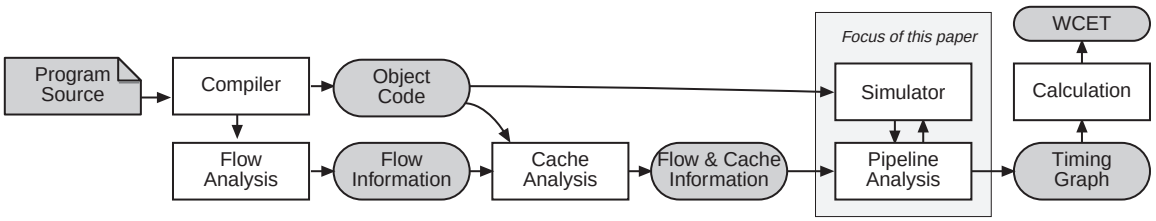

Figure 1. Overview of our WCET analysis architecture.

misses, the values of instruction operands, the speed of memory accessed, etc.).

Figure 2(a) shows the basic blocks of an example loop iterating at most 42 times, and Figure 2(b) shows the execution scenarios resulting from an instruction cache analysis in the style of Ferdinand et al [6]. We get two execution scenarios for the body of the loop, one for the first iteration, and one for the rest. The two scenarios differ in whether the loop body is present in the instruction cache or not[2].

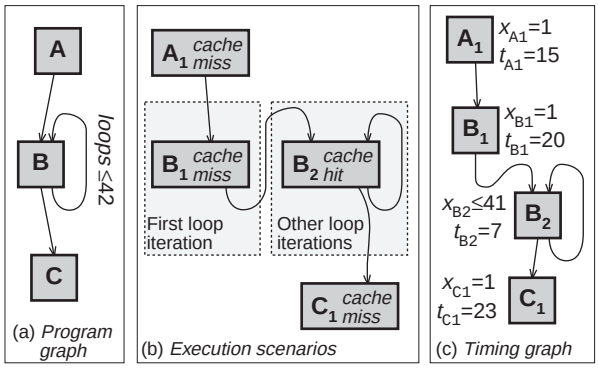

Figure 2. Example of execution scenarios for cache analysis.

The flow analysis will generate different execution scenarios for different calls to a function, depending on the values of parameters. This supports context-sensitive WCET analysis, where the same function may have several execution times, depending on the calling context.

Given the execution scenarios (e.g. Figure 2(b)), the *pipeline analysis* generates concrete execution times by using a processor simulator to run each execution scenario.

The *simulator* can be any simulator that can be run in trace-driven mode. It must have a cycle-accurate model of the CPU, but it does not need to model the semantics of the object code. The simulator must be able to accept the information contained in the execution scenario as input.

The result of the pipeline analysis is a *timing graph* (e.g. Figure 2(c)), a directed graph in which the nodes correspond to execution scenarios, and the edges represent the program flow. The edges and nodes in the timing graph are annotated with execution times (e.g. $t_{\texttt{B2}}$) and execution count variables (e.g. $x_{\texttt{B2}}$). Possible program flows are expressed as constraints over the execution count variables (e.g. $x_{\texttt{B2}} \leq 41$).

The final WCET estimate is generated by maximizing the sum of the products of the execution counts and execution times (subject to the flow constraints):

$$WCET = maximize(\sum_{\forall entity} x_{entity} \cdot t_{entity}) \qquad (1)$$

This maximization problem can be solved using either integer linear programming [23, 26] or constraint satisfaction techniques [21].

4. The Problem of Pipeline Analysis

The purpose of pipeline analysis is to model the execution time effect of the overlap between instructions in pipelined processors. In this paper, we consider pipelined CPUs that start a maximum of one single instruction each clock cycle (single dispatch) and that execute instructions in program order (in-order issue) [10]. We define the execution time of a sequence of instructions (or a single instruction) to be the time from the point that the first instruction enters the pipeline until the last instruction leaves the pipeline. For illustrations, we use a simple pipeline containing containing instruction fetch (`IF`), execute (`EX`), and memory access (`M`) stages, as well as a separate floating point (`F`) stage (which is used instead of the `EX` and `M` stages). Our approach is not limited to this pipeline structure, however.

[2] The first time through the loop, the loop body will be fetched from main memory to the cache. The second and successive iterations, however, the loop body will be in the cache and will thus execute faster.

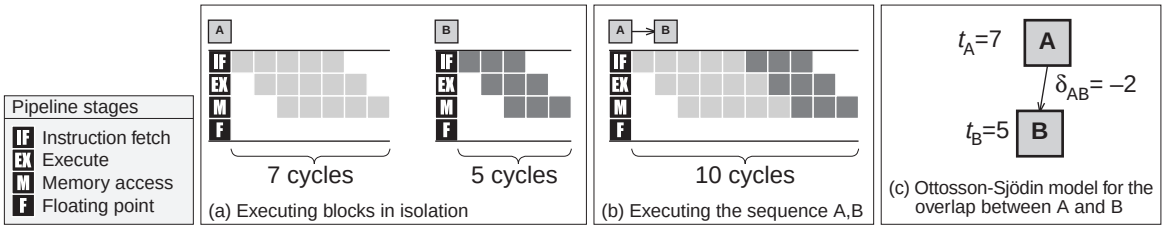

Figure 3. Example of pipeline overlap between consecutive blocks.

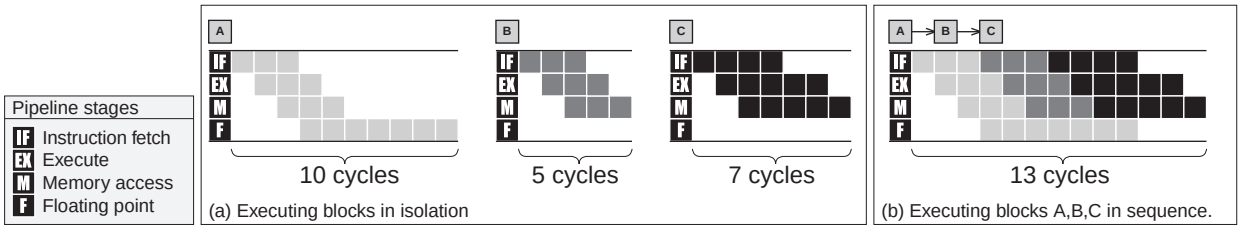

Figure 4. Example of pipeline overlap across multiple blocks.

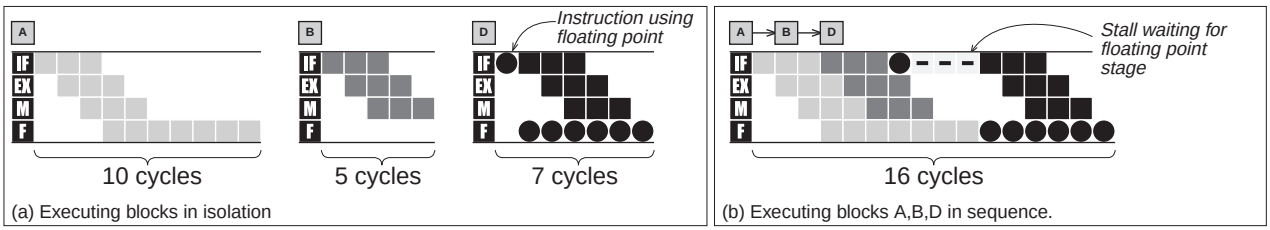

Figure 5. Example of pipeline interference across intervening blocks.

The overlap between instructions makes the execution time for a sequence of instructions (or basic blocks) less than the sum of the execution time for the individual instructions (or basic blocks). This effect is illustrated in Figure 3, with the overlap modeling according to Ottosson and Sjödin [21] illustrated in pane (c).

Modeling pipeline effects only between pairs of basic blocks might be insufficient, since in many cases there can be effects that reach across more than two basic blocks. Such effects are caused by instructions completing execution long after the other instructions in the basic block they belong to (what we call *long-running instructions*). The inability to model this is one of the big problems of previous versions of IPET.

For example, consider the MIPS R4000 [9], where the integer and floating point pipelines execute in parallel[3]. Some floating point instructions require more than 100 cycles for execution, which has the implication that if one basic block uses floating point, and its successors do not, the execution of the first block may overlap the execution of several of its successors, and maybe interfere with some later block.

A simple case of overlap with a long-running instruction is shown in Figure 4. Block `B` does not add any execution time when executed following block `A`. This effect is not captured by Ottosson and Sjödin, who make the conservative assumption that every basic block use all pipeline stages (if necessary by inserting extra stage uses into blocks that do not use certain stages), which gives a safe overestimation of the execution time.

Figure 5 shows an example of interference between a long-running instruction and a later basic block: block `D` uses floating point and gets delayed due to block `A`'s use of the floating point stage, despite the presence of block `B` between them.

Another problem is that CPU resources can be *shared* by several pipeline stages, making it possible for later instructions to delay the execution of earlier instructions (common register dependencies and pipeline

[3] All instructions pass through the same instruction fetch and decode stages before splitting into the floating point or integer pipeline.

stalls only make earlier instructions delay later instructions).

The most common case is contention over the memory interface. Most CPUs only have *one* external memory interface, and when an instruction fetch and a data access need to access the memory simultaneously, one will be forced to wait.[4]

The severity of the problem depends on the CPU architecture. On most embedded processors, for example, the program is stored in internal ROM, with a separate memory interface for RAM, making contention less common. On a cached CPU, instruction fetches and data accesses can proceed in parallel as long as they hit the cache. On simultaneous cache misses, however, contention will occur.

This paper describes how to determine the timing effects of pipeline overlap and interference with long-running instructions, even in the presence of shared memory interfaces.

5. IPET Timing Model

As described above in Section 3, we model a program as a set of nodes (representing execution scenarios for basic blocks) and edges (representing possible flows between the nodes).

The execution time (t_{node}) associated with each node represents the time it takes to execute that node in isolation. The execution count (x_{node}) corresponds to the number of times the node is executed in the worst-case execution of the program.

Furthermore, we associate *timing effects* with *sequences* of nodes. A timing effects (δ_s) represent how the execution time changes when a sequence of nodes is executed. The timing effect variables are negative to indicate a speedup, and positive to indicate a slowdown. Each timing effect variable has an associated execution count variable (x_s), representing the number of the times the sequence is executed in the worst-case program execution. Note that sequences can be of arbitrary length. The final execution time formula thus becomes (using s to denote an arbitrary sequence):

$$WCET = max(\sum_{\forall node} x_{node} \cdot t_{node} + \sum_{\forall s} x_s \cdot \delta_s) \quad (2)$$

In the following sections, we will describe how the timing effects and execution count constraints are defined and determined.

5.1. Timing Effects for Sequences

The execution time for a sequence of nodes (t_s) is the sum of the execution time for the nodes in the sequence, plus the timing effects for all subsequences of the sequence (as illustrated in Figure 6):

$$t_s = \sum_{\forall node \in s} t_{node} + \sum_{\forall s' \in subsequences(s)} \delta_{s'} \quad (3)$$

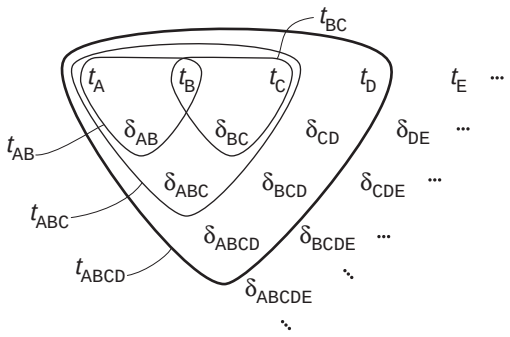

Figure 6. Graphical illustration of how times and timing effects sum up.

Using some algebraic manipulation of Equation 3, we can express the timing effect for a sequence in terms of the execution times for some subsequences. Using the notation $s[i..j]$ to indicate a subsequence ($s = s[1..l]$ if the length of s is l) we get:

$$\delta_s = t_s - t_{s[2..l]} - t_{s[1..l-1]} + t_{s[2..l-1]} \quad (4)$$

For example, $\delta_{\texttt{ABC}} = t_{\texttt{ABC}} - t_{\texttt{AB}} - t_{\texttt{BC}} + t_{\texttt{B}}$. Note that for sequences of length two, the last term becomes zero, e.g. $\delta_{\texttt{AB}} = t_{\texttt{AB}} - t_{\texttt{A}} - t_{\texttt{B}}$.

For the example in Figure 4, Equation 4 gives the following values: $\delta_{\texttt{AB}} = -5$ (reflecting that `A` completely overlaps the execution of `B`) and $\delta_{\texttt{BC}} = -2$. For the long sequence `ABC`, we get $\delta_{\texttt{ABC}} = -2$, showing that the execution of `A` overlaps *both* `B` and `C`. Taking such effects into account gives a tighter execution time estimate.

In Figure 5 we show an extreme example of interference resulting in a *positive* timing effect ($\delta_{\texttt{ABD}} = +1$). Not taking this type of effect into account may lead to an underestimation of the execution time.

Timing effects are usually negative, since interference only tends to make the overlap smaller, not remove it altogether.

Note that timing effects involving a certain node can only occur as long as some instruction from the node is still in the pipeline. This means that there is an upper bound on the length of sequences that can have timing effects. This is discussed in detail in Section 6.1 below.

[4] For a good discussion of the problem, see [12, pp. 425–427].

5.2. Execution Count Constraints

The WCET expression in Equation 2 requires that we have constraints on all execution count variables. We distinguish between four types of execution count constraints: *structural constraints*, *finiteness and start constraints*, *sequence constraints*, and *tightening constraints*.

5.2.1. Structural Constraints

Structural constraints reflect the structure of the program code. They are defined as constraints on the execution counts on the edges in the timing graph (i.e. sequences of length two), and their purpose is to preserve the flow of the program. For each node, the sum of the incoming flows is equal to the outgoing flows [21, 23, 26] (in Figure 7, the flows into and out of node `B` would give $x_{\texttt{QB}} + x_{\texttt{AB}} = x_{\texttt{B}}$ and $x_{\texttt{B}} = x_{\texttt{BR}} + x_{\texttt{BC}}$). For loops, we need to ensure that they are only executed if their header block is reached (which is a little more complex, see e.g. [21, 26]).

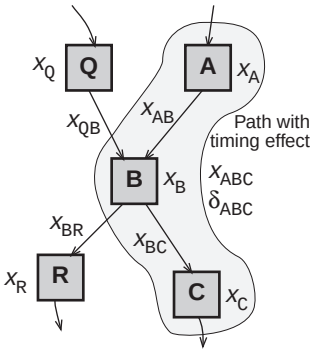

Figure 7. Example timing graph fragment.

5.2.2. Finiteness and Start Constraints

In addition to the flow constraints, we need constraints to state that the program is finite, and that it executes once.

The finiteness is ensured by constraining the header block for each loop or recursive function by a *loop bound*, stating the maximal number of iterations of the loop (see e.g. [26]). The loop bounds can be derived automatically [5, 8] or provided by manual annotation.

The program is set to execute once by constraining the execution count of the entry node of the program to one (i.e. $x_{entrynode} = 1$).

5.2.3. Sequence Constraints

We derive an upper and a lower bound for the number of executions of sequences longer than two. The bounds are expressed relative to the executions of shorter sequences, with the structural constraints representing the shortest sequence of a single basic block.

The upper bound is needed when the timing effect is positive (to bound the program execution time), and the lower bound makes sure that the execution of a negative timing effect is not ignored by the maximization function.

An upper bound on the number of executions of a sequence is derived by noting that a sequence s (of length l) is executed only if its two subsequences of length $l-1$ are executed. This is expressed using two inequalities:

$$x_s \leq x_{s[1..l-1]} \tag{5}$$
$$x_s \leq x_{s[2..l]} \tag{6}$$

For the example in Figure 7, we get the following constraints for the sequence `ABC`: $x_{\texttt{ABC}} \leq x_{\texttt{AB}}$, $x_{\texttt{ABC}} \leq x_{\texttt{BC}}$.

Note that constraints for all subsequences of a sequence have to be generated, even if those subsequences have no timing effects associated with them (for the sequence `ABCD`, we will need constraints for the sequences `ABC` and `BCD`, etc.).

The lower bound is derived by considering the stepwise construction of a sequence from prefix sequences. Intuitively, a lower bound can be derived by looking at the flow through the timing graph: a certain number of executions enter the start of the sequence, and at each node in the sequence, one edge leads the way to the next node in the sequence, but there may be other edges leading to nodes not part of the sequence, thus diminishing the total number of executions of the sequence.

The result is that a sequence can be executed as many times as its prefix of length $l-1$, minus the number of executions of all edges leaving the sequence at the second-last node.

More formally, given a prefix sequence p ending at node `A` and executing x_p times, we know that the sequence $p \circ \texttt{B}$ can be executed at most x_p times (where `B` is the next node in the sequence following `A` and $\circ$ is used to represent concatenation). Furthermore, any flows from `A` to nodes other than `B` will diminish the number of times $p \circ \texttt{B}$ must be executed. Thus, we get the following lower bound, (using $s[i]$ to pick a single node from a sequence, and $succ(n)$ to return all successor nodes to node n):

$$x_s \geq x_{s[1..l-1]} - \sum_{\forall n \in succ(s[l-1]) \neq s[l]} x_{s[l-1] \circ n} \tag{7}$$

E.g. for the sequence `ABC` in Figure 7, we generate the constraint: $x_{\texttt{ABC}} \geq x_{\texttt{AB}} - x_{\texttt{BR}}$.

5.2.4. Tightening Constraints

The constraints discussed above provide a basic frame for the calculation of a safe WCET estimate. However, the flow analysis might generate more detailed information on the possible flows of a program, information that can be used to produce a tighter WCET estimate. Such information is also expressed using constraints.

Note that IPET has no concept of "execution paths". Only the values of the execution count variables matter for the maximization of Equation 2. This means that it is usually sufficient to model infeasible paths as constraints on single nodes, unless there is a timing effect for a sequence along the infeasible path. This bounds the length of sequences of nodes that need to be considered in the flow analysis.

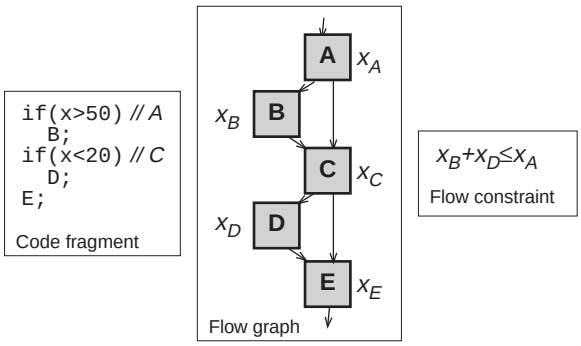

Figure 8. Example infeasible path.

For example, consider Figure 8, where the path `ABCDE` can never be taken. The effect on the execution counts can be expressed using the constraint $x_{\mathtt{B}} + x_{\mathtt{D}} \leq x_{\mathtt{A}}$. This constraint is sufficient for a tight WCET estimate, despite the fact that we do not really state that the sequence `ABCDE` is infeasible. Assuming that we pass `A` twice in the execution of the complete program, it is "allowed" to first execute the sequence `ABCDE` and then `ACE`. But we will still get a correct number of executions of `B` and `D`.

However, the infeasibility of the specific sequence `BCD` would matter if there was a timing effect for the sequence (i.e. $\delta_{\mathtt{BCD}} \neq 0$), and this infeasibility would be expressed by the constraint $x_{\mathtt{BCD}} = 0$. The constraints generated for $x_{\mathtt{BCD}}$ by Equations 5, 6, and 7 are not strong enough to alone express this fact.

6. Timing Effect Extraction

We have developed an algorithm for generating the execution time values for a timing graph using a simulator.

The basis is to use a cycle accurate simulator to obtain the execution time for a node and sequence of nodes. The results are then used to calculate timing effects (or simply stored as times for individual nodes). This process is illustrated in Figure 9. In the example, $t_{\mathtt{QR}}$ is smaller than the sum of $t_{\mathtt{Q}}$ and $t_{\mathtt{R}}$, and thus the timing effect $\delta_{\mathtt{QR}}$ is negative (according to Equation 4). We run sequences of length one, then of length two, then of length three, etc., until all timing effects have been captured.

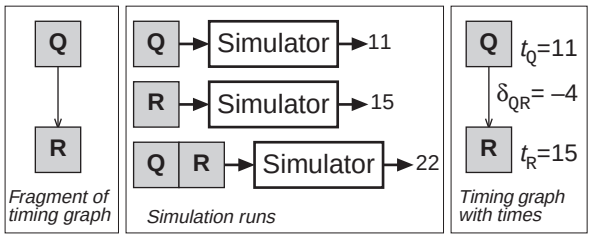

Figure 9. Timing effects using a simulator.

The algorithm for extracting pipeline timing effects is given in Figure 10 where *TGraph* is the timing graph, and *CspSet* is the set of flow constraints, execution count variables, and timing variables.

Our algorithm uses an auxiliary "database", *TimeDB*, to hold the execution times for sequences. If the time for a sequence is not present in the *TimeDB*, it will get the simulator to execute the corresponding sequence and store the execution time. Thus, each sequence only has to be executed once.

The set *SeqSet* holds the sequences that are to be executed. We start the main loop of the algorithm with a list containing all pairs in the timing graph.

The algorithm takes a sequence from the *SeqSet*, executes it, and determines whether a timing effect is present. If a timing effect is found, the corresponding timing effect variable are added to the *CspSet*. Finally, we check whether the sequence needs to be extended to capture more timing effects. If so, the sequence is extended, and all extensions to the sequence are inserted into *SeqSet* (there might be several extensions to each sequence – one for each successor to the last node in the sequence).

6.1. Extension Condition

A central problem in the algorithm is to determine when to generate extensions to a sequence. Sequences that do not cause timing effects should not be executed. Since our algorithm works by successively extending sequences, we call this the *extension condition*. The solution to this problem depends on the properties of the processor pipeline.

```
// First, get timing for each node in TGraph
for each node n in TGraph do
  s = create sequence with only n in
  t_n = get time for sequence s from TimeDB
  add time t_n for node n to CspSet
end do

// Second, generate initial sequences of length two
for each node n in TGraph do
  for each node m ∈ succ(n) in TGraph do
    s = sequence n ∘ m
    add sequence s to SeqSet
  end do
end do

// Third, generate all timing effects and constraints
until SeqSet is empty do
  s = remove sequence from SeqSet
  // calculate time for edge
  l = get length of s
  get times t_s, t_s[2..l], t_s[1..l-1] and t_s[2..l-1]
    from TimeDB
  calculate δ_s = t_s − t_s[2..l] − t_s[1..l-1] + t_s[2..l-1]
  // add timing effects and constraints
  add δ_s for sequence s to CspSet
  add constraints for sequence s to CspSet
  // check if longer sequences are needed
  for each node n ∈ succ(s[l]) in TGraph do
    if (s[1] can affect n) then
      s' = create sequence s ∘ n
      add sequence s' to SeqSet
    end if
  end do
end do
```

Figure 10. Algorithm

We have designed extension conditions that works for ordinary in-order single-issue pipelines. We assume that the CPU pipeline fulfills the following conditions:

1. At most one instruction is started each cycle (single issue).

2. Instructions are processed in-order.

3. No instruction can affect the execution time of another instructions after it exited from the pipeline: all delay effects are taken immediately, and do not propagate to later instructions.

4. The only resource shared between pipeline stages is the memory interface, and it is only shared between the instruction fetch stage (the entry point to the pipeline) and the (data) memory access stage.

5. Access to the memory interface is arbitrated in such a way that the memory access stage can be stalled at most once by an instruction fetch before progressing.[5]

6. Besides the effects of assumptions 4 and 5, no instruction can affect the execution time of an instruction entering the pipeline before it.

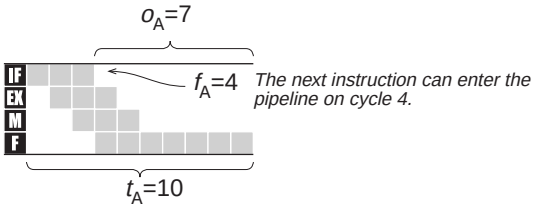

Figure 11. Illustration of the definition of o_A.

These conditions hold for all single-issue CPUs we have studied, and they allow us to avoid the timing anomalies for dynamically-scheduled CPUs presented by Lundqvist and Stenström [17].

According to assumptions 2 and 3, we can terminate the process of examining longer sequences when all instructions from the first node in the sequence have left the pipeline. However, since we are using a simulator, we cannot directly inspect the pipeline state. Instead, we need to find a (safe) approximation using the information made available to us by the simulator:

- The execution time for a certain stream of instructions.
- The point in time when a certain instruction enters the pipeline (available since we feed the simulator with instructions).

Our approximation is to stop the generation of longer sequences starting with a node `A` when o_A instructions have been fetched into the pipeline after the last instruction of `A` was fetched.

The value of o_A depends on the properties of node `A` and is defined as the number of clock cycles that `A` executes after the last instruction of `A` has been fetched into the pipeline when `A` executes on its own. Note that o_A is determined by using clock cycles, but that it is used for instruction counting. It can be calculated as $o_A = t_A - f_A + 1$, where f_A is the time for the first

[5] It is not necessary that the memory access stage does not starve the instruction fetch stage, since there are only a finite number of instructions in the pipeline, thus making the delay on the instruction fetch bounded.

instruction fetch after all instructions from A have been fetched. Figure 11 illustrates the relation between o_A, f_A and t_A.

According to assumption 1, at most o_A instructions can be executed in o_A clock cycles. Thus, unless the execution of node A gets extended due to pipeline interference from later instructions, no more than o_A successive instructions can be affected by instructions belonging to node A.

According to assumptions 4 and 6, the only way to extend the execution time of A is when there is a conflict between a memory access by any instruction in A and the fetch of a later instruction. In this case, the execution of A might[6] require more time.

However, according to assumption 5, the blocked instruction will grab the memory immediately after the end of the instruction fetch. Thus, no more instructions will be fetched until after the memory access has completed, and the instruction in A can continue executing. This means that each delay to an instruction in A will lead to a corresponding delay in the instruction fetching, allowing no more instructions to be fetched than if there had been no conflict.

Note that there is an upper bound on the longest sequence that can have a pipeline effect, since every instruction and thus every node will complete its execution sooner or later (after a node has finished, it can no longer affect other nodes and thus there will be no timing effects). Also, for pipelined CPUs, timing effects are always present between pairs of nodes.

6.2. Algorithm Complexity

An upper bound on the number of sequences that we need to execute using our algorithm is $O(n \cdot k^{l-1})$ where n is the number of nodes in the timing graph, k is the maximum number of successors a node can have, and l is the maximum distance (in nodes) between two nodes where we can get a pipeline effect.

Proof: first consider the number of sequence simulations that are made with a specific node as the start node. We will get k^0 paths of length 1, k^1 paths of length 2, ..., k^{l-1} paths of length l. This means that the total number of simulations runs needed for one start node is never more than $\sum_{i=0}^{l-1} k^i$, which can be simplified to [24]:

$$\frac{k^l - 1}{k - 1}$$

Since the pipeline timing effect for a sequence is formulated using execution times for shorter sequences, and each sequence of nodes only need to be executed once, we conclude that the maximum number of simulation runs needed for n nodes is bounded by:

$$n \cdot \frac{k^l - 1}{k - 1} \Rightarrow O(n \cdot k^{l-1})$$

Note that on average, k is likely to be less than two (most basic blocks in a program have only one or two successors).

[6] For example, on the Hitachi SH3 [12] the memory access in A will take precedence, leading to zero delay for A, but if the instruction fetch takes precedence, A will be delayed.

7. Applicability and Evaluation

The WCET analysis approach presented in this paper is applicable to all pipelined CPUs featuring in-order single-dispatch. This includes a large set of CPU cores: all simple 8- and 16-bit processors (most of which have no pipeline at all), and most of the embedded 32-bit CPUs.

For all CPUs where all instructions flow through all stages in the pipeline, the analysis of pipeline effects across more than two nodes is not necessary. Examples of such CPUs are the NEC V850E [20], the ARM7 [1], and the integer part of the MIPS R4000 [9].

The modeling of pipeline effects across more than two basic blocks is needed when instructions start execution in a common pipeline, but then branch out to parallel functional units. A typical example is the floating point pipeline of the MIPS R4000 discussed earlier.

There are some cases where integer pipelines exhibit this kind of partially parallel behavior. For example, on the Mitsubishi M32R [18], loads can execute in parallel to the instructions following them. On the Hitachi SH7700 [11], multiplications execute for up to three cycles in a special pipeline stage (after the finish stage for non-multiplication instructions).

It is difficult to experimentally evaluate our approach, since the applicability of the full model depends heavily on the processor and the instructions used in program code. However, it is clear that there are many processors on the market today requiring an advanced pipeline model like the one we have presented. We have broadened the applicability of the IPET method to include more CPUs than was previously possible to model.

8. Future work

Our long-term aim is to integrate a WCET analysis tool within the framework of a the commercial IAR Systems C/C++ development environment for embedded systems [13], which will make WCET analysis a

viable and accessible technique for practioners in the real-time and embedded systems field.

In addition, there are several interesting machine-level timing analysis issues that need to be addressed in order to allow timing analysis of advanced CPUs.

We need to extend our pipeline analysis to multiple-issue (especially important for DSP applications) and out-of-order pipelines.

Data cache and cache hierarchy modeling is still an open problem, even though some work has been performed [27, 19]. We consider data caches to be more important, since they are likely to appear in embedded applications before multi-level cache hierarchies.

There are many other performance-enhancing features used in modern processors that need to be modeled. A primary research target is advanced branch processing features like branch target buffers, branch predictors, and branch folding [10].

An interesting project would be to use real hardware instead of a simulator. This is, however, rather difficult because of the difficulties involved in controlling and obtaining time measurements from real pipelined hardware[7].

9. Conclusions

This paper demonstrates solutions to several problems involved in the low-level timing analysis for pipelined CPUs using the Implicit Path Enumeration Technique (IPET) [23]:

- How to handle pipeline interference across multiple basic blocks, without using conservative assumptions.
- How to identify relevant paths across multiple basic blocks in the program path modeling, without resorting to modeling all possible paths.
- How to obtain the execution times for basic blocks from existing simulators, thus removing the problem of defining a special-purpose pipeline analysis technique.
- How to incorporate results from cache analysis (and analyses) into the pipeline model, while keeping the analyses separate.

Using these techniques, we can handle arbitrary in-order single-issue pipelines, in a tool that is easy to port and where it is easy to incorporate new analyses (e.g. for branch prediction or data caches).

[7] Breakpoints often have a "skid" where the CPU may run for several cycles before stopping, which makes exact timing very difficult.

We show how a real-life processor (mis)feature can be modeled, enabling the use of automatic analysis for wide range of commercial CPU cores presently being used for embedded real-time systems.

Acknowledgements

The authors would like to thank Jan Gustafsson, Hans Hansson, Bengt Jonsson and Mikael Sjödin for their fruitful comments on drafts of this article. The responses from the anonymous reviewers were very helpful in the editing of the conference version of the text.

References

[1] Advanced Risc Machines Ltd. *ARM7 Data Sheet*, December 1994.

[2] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority pre-emptive scheduling: an historical perspective. *Real-Time Systems*, 8(2/3):129–154, 1995.

[3] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – a revolution in on-board communications. *Volvo Technology Report*, 1:9–19, 1998.

[4] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards industry-strength worst case execution time analysis. Technical Report ASTEC 99/02, Advanced Software Technology Center (ASTEC), April 1999. Submitted to the Kluwer Journal of Real-Time Systems.

[5] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proc. Euro-Par'97 Parallel Processing, Lecture Notes in Computer Science (LNCS) 1300*, pages 1298–1307. Springer Verlag, August 1997.

[6] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

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

[8] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In

Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98), June 1998. URL: `http://www.docs.uu.se/~mic/papers.html`.

[9] J. Heinrich. *MIPS R4000 Microprocessor User's Manual.* MIPS Technologies Inc., 2nd edition, 1994.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2nd edition, 1996. ISBN 1-55860-329-8.

[11] Hitachi Europe Ltd. *SH7700 Series Programming Manual*, September 1995.

[12] Hitachi Ltd. *Hitachi SuperH RISC engine SH-3/SH-3E/SH3-DSP Programming Manual*, 2nd edition, 1999.

[13] IAR Systems WWW homepage. URL: `http://www.iar.com`.

[14] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[15] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An accurate worst-case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995. URL: `http://archi.snu.ac.kr/symin/ets.ps`.

[16] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, November 1999.

[17] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. Technical Report 99-5, Chalmers University of Technology, 1999.

[18] Mitsubishi Electric. *M32R family Software Manual*, July 1998.

[19] F. Müller. Timing predictions for multi-level caches. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, pages 29–36, Jun 1997. URL: `http://www.cs.fsu.edu/~mueller/publications.html`.

[20] NEC. *V850 Family 32/16-bit Single Chip Microcontroller User's Manual: Architecture*, 4th edition, 1995. Document no. U10243EJ4V0UM00.

[21] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.

[22] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. In *Proc. 11th IEEE Real-Time Systems Symposium (RTSS'90)*, pages 72–81, December 1990.

[23] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[24] L. Råde and B. Westergren. *Beta, Methematical Handbook.* Studentlitteratur, 2nd edition, 1990. Page 174. ISBN 91-44-25052-5.

[25] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. Technical Report 27-97, C-LAB, Paderborn, 1997. URL: `http://www.c-lab.de/~peter/paper.html`.

[26] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[27] R. White, F. Müller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, June 1997.