Processor Pipelines and Their Properties for Static WCET Analysis

Jakob Engblom and Bengt Jonsson

Dept. of Information Technology
Uppsala University
P.O. Box 337
SE-751 05 Uppsala, Sweden
www.it.uu.se

Abstract. When developing real-time systems, the worst-case execution time (WCET) is a commonly used measure for predicting and analyzing program and system timing behavior. Such estimates should preferrably be provided by static WCET analysis tools. Their analysis is made difficult by features of common processors, such as pipelines and caches.
This paper examines the properties of single-issue in-order pipelines, based on a mathematical model of temporal constraints. The key problem addressed is to determine the distance (measured in number of subsequent instructions) over which an instruction can affect the timing behavior of other instructions, and when this effect must be considered in static WCET analysis. We characterize classes of pipelines for which static analysis can safely ignore effects longer than some arbitrary threshold. For other classes of pipelines, pipeline effects can propagate across arbitrary numbers of instructions, making it harder to design safe and precise analysis methods.
Based on our results, we discuss how to construct safe WCET analysis methods. We also prove when it is correct to use local worst-case approximations to construct an overall WCET estimate.

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. Reliable WCET estimates are necessary when designing and verifying embedded real-time systems, especially when used in safety-critical systems like vehicles and industrial plants. WCET estimates can be used to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, to check that interrupts have sufficiently short reaction times, to find performance bottlenecks, and for many other purposes.

* This work is performed within the Advanced Software Technology (ASTEC, http://www.astec.uu.se/wcet) competence center, supported by the Swedish National Innovation Systems' Administration (VINNOVA, http://www.vinnova.se) and IAR Systems (http://www.iar.com).

WCET estimates must be *safe*, i.e. guaranteed not to underestimate the execution time, and *tight*, i.e provide acceptable overestimations. The safeness of an estimate is critical when the estimate is used in the construction of a safety-critical system. The purpose of *static WCET analysis* is to generate safe and tight estimates by analyzing the source code and object code of the program without executing it.

The introduction of hardware features like caches and pipelines in the processors used for embedded real-time systems complicates static WCET analysis (and the measuring of execution times) by increasing the variability in execution time and by requiring more complex analysis methods.

A variety of concrete analysis methods for pipelines have been proposed, ranging over cycle-accurate simulators [6, 7, 22], special-purpose models using reservation tables [4, 9, 14, 21], dependence graphs [15], abstract interpretation of pipeline behavior [8, 20], and tables of instruction execution times and inter-instruction effects [2, 3].

The timing benefit (effect) of pipelines is to a large extent due to the overlapping of pairs of adjacent instructions. This has motivated techniques that use the speedup for pairs of instructions to model the effect of pipelining [2–4, 19, 21]. The WCET is then calculated by summing execution times of instructions and subtracting the speedups.

For many pipelines, there are also timing effects that only occur for sequences of three or more instructions; in this case the entire sequence has to be considered in a precise and safe timing analysis that involves the first instruction. Such *long timing effects* (LTEs) are introduced in [7].

In this paper, we investigate general properties of pipelines relevant for static WCET analysis, in particular the issue of how far away a single instruction in a program can affect the pipeline behavior of other instructions. Sometimes, it is enough to consider pairs of adjacent instructions, while other pipelines require that instructions quite far apart are handled in the same analysis unit. To our knowledge, this is the first work that explores the theoretical limitations of pipeline analysis. Previous WCET research has been more focussed on actually building concrete and useable WCET analysis methods than exploring the limits of analyzability.

The concrete contributions of this paper are the following:

- We define a mathematical model of pipeline behavior, as a basis for investigating the occurrence of LTEs.
- We give general conditions under which no LTEs occur.
- A central theorem shows that for a class of pipelines, all LTEs are time savings. This implies that a safe (but not necessarily precise) analysis can be performed by considering only time savings for short sequences of instructions (typically pairs).
- We demonstrate that in many pipeline structures, there are cases where LTEs may occur for arbitrarily long sequences, i.e., disturbances can propagate over arbitrary distances. If such LTEs may add time, then it is hard to construct a safe and precise analysis.
- We prove that in-order pipelines are not subject to the kind of timing anomalies described by Lundqvist and Stenström [16]. This indicates that local worst-case assumptions for each instruction can be used to construct the overall worst case.

We do not address the issues raised by out-of-order processors, so this should only be considered a first step in our understanding of processor pipelines.

Paper outline: Section 2 presents the model of execution times, pairwise timing effects, and long timing effects. Section 3 presents the pipeline model built on constraints. Section 4 proves and demonstrates properties of pipelines. Section 5 discusses the implications of the properties for static WCET analysis.

2 Timing Model

For the discussion in this paper, we assume that a program is represented by a set of *nodes*, each containing one or more instructions. Nodes are connected by *edges*, and *sequences* of nodes can be formed by following the edges. For simplicity, this presentation will assume that each node contains just a single instruction (however, the same model applies where each node is a basic block).

Our goal is to statically calculate the execution time for a sequence of instructions $I_1 \dots I_m$, which is denoted by $T(I_1 \dots I_m)$. We define $T(I_1 \dots I_m)$ as the time from which I_1 enters the first pipeline stage in a state where the pipeline is empty (cold pipeline) until I_m leaves its last pipeline stage. An important assumption is that the same sequence of instructions always yields the same execution time. This requires that the hardware is deterministic, and that effects outside the pipeline (like cache hits and misses, variable-length instructions, etc.) are fixed by extra information in the program graph. If such information is not known, several nodes might be used to represent the same code, with different information about its execution. Techniques for such transformations are described in, for example, [6, 8].

The execution time $T(I)$ of a single instruction I is denoted t_I. To capture the timing effect of pipelines, we introduce *timing effects* $\delta_{I_1 \dots I_m}$ for sequences $I_1 \dots I_m$, defined as follows.

$$t_I = T(I) \tag{1}$$

$$\delta_{I_1 \dots I_m} = T(I_1 \dots I_m) - T(I_2 \dots I_m) - T(I_1 \dots I_{m-1}) + T(I_2 \dots I_{m-1}) \quad m \geq 2 \tag{2}$$

We can then calculate the execution time $T(I_1 \dots I_m)$ for a sequence $I_1 \dots I_m$ in terms of times and timing effects:

$$T(I_1 \dots I_m) = \sum_{j=1}^{m} t_{I_j} + \sum_{1 \leq i < k \leq m} \delta_{I_i \dots I_k} \tag{3}$$

Equation (3) expresses that the execution time for a sequence $I_1 \dots I_m$ be obtained by adding the node times t_{I_j} for all nodes in the sequence, and the timing effects $\delta_{I_i \dots I_k}$ of all subsequences (of length ≥ 2) of the sequence.

Intuitively, the timing effects concisely capture the effect of pipelines on the timing of a sequence of instructions. For pairs of instructions, the *pairwise timing effects* correspond to the speedup obtained by the pipeline overlap between adjacent instructions, as illustrated in Figure 1. In general, pairwise effects are negative. For longer sequences of instructions, we could get *long timing effects* (LTE). Whenever $\delta_{I_1 \dots I_m} \neq 0$, this is due to instruction I_1 having some effect that disturbs the execution of instruction I_m (across the sequence $I_2 \dots I_{m-1}$). Precisely, the execution of nodes $I_2 \dots I_m$ in the sequence

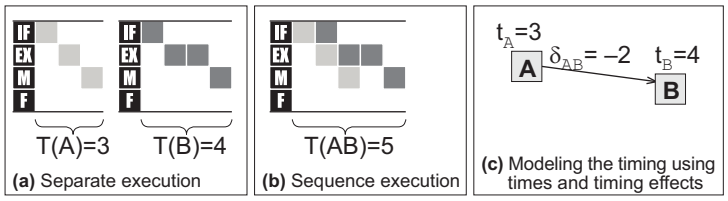


Fig. 1. Pipelining of instruction execution

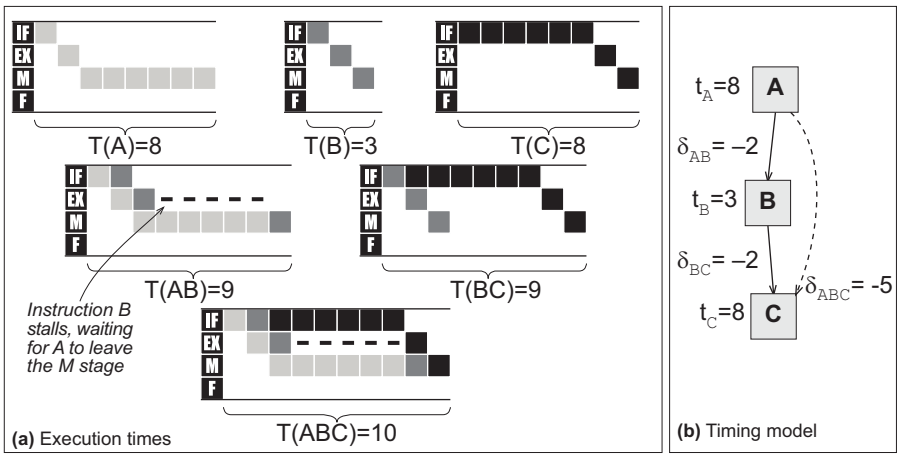


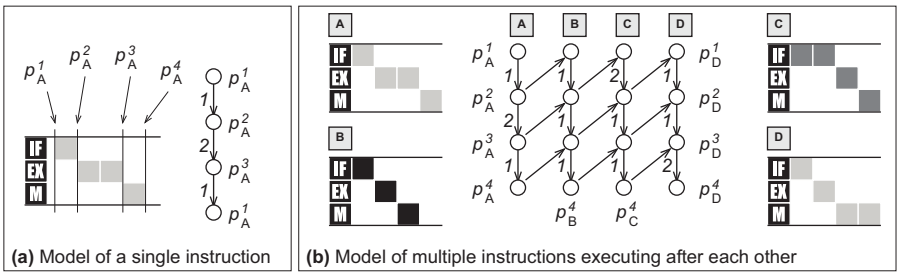
Fig. 2. Example long timing effect

$I_1 \ldots I_m$ is different from the execution of the nodes $I_2 \ldots I_m$ starting with node I_2 (see Section 4.1 for more details).

Figure 2 shows an example of a LTE, where the execution profile of `BC` is different when executed in isolation and when executed as part of the sequence `ABC`. Note that we illustrate pipeline execution using *pipeline diagrams*, similar to the *reservation tables* commonly used to describe the behavior of pipelined processors. Time runs horizontally, with each tick of time corresponding to a processor clock cycle. The pipeline stages are shown on the vertical axis. Instructions progress from upper left to lower right, and each cycle of execution is shown as a square.

LTEs can in general be both negative and positive, and must be accounted for by a static WCET analysis method that wants to be safe and tight. *Positive timing effects* add execution time to a program, and are critical to consider since otherwise an underestimate of the WCET could result. *Negative timing effects* indicate potential savings in execution time, and ignoring them only makes the WCET estimate less tight.

Positive LTEs comprise a central problem for WCET analysis, since they make it necessary to consider effects across more than just adjacent instructions assumption in order to create a safe analysis. To prove a particular WCET analysis method safe, it is necessary to address the question of whether all positive LTEs have been accounted for. A central issue in this paper is therefore to give sufficient conditions for when LTEs are guaranteed not to be positive.



(a) Model of a single instruction

(b) Model of multiple instructions executing after each other

Fig. 3. Constraint model of pipeline execution

3 Pipeline Model

We model the timing behavior of pipelined execution by simple temporal constraints. This model is only used to describe execution timing, and is not intended as a basis for building pipeline simulators. It can be used to model all in-order pipelines that do not use dynamic dispatch, including VLIW processors [6].

For in-order pipelines with a single pipeline, we consider the pipeline to consist of n *pipeline stages*. Each *instruction* I_i is considered as a sequence $r_i^1 \dots r_i^n$ of resource requirements, where r_i^j corresponds to the time the execution of the instruction requires in stage j. Only one instruction can occupy a pipeline stage at any particular point in time. Instructions are numbered from 1 to m, all instructions use all pipeline stages, and use them in the same order. Instructions proceed to the next stage as soon as possible. Time is discrete and expressed in clock cycles.

Consider the execution of a sequence $I_1 \dots I_m$ of instructions. For an instruction I_i, we let p_i^j be the point in time at which I_i enters the pipeline stage j. By convention, p_i^{n+1} is the time at which instruction I_i leaves the last stage of the pipeline. The pipelined execution of $I_1 \dots I_m$ is model by the following constraints:

$$p_i^{j+1} \geq p_i^j + r_i^j \qquad (1 \leq i \leq m, 1 \leq j \leq n) \tag{4}$$

$$p_{i+1}^j \geq p_i^{j+1} \qquad (1 \leq i < m, 1 \leq j \leq n) \tag{5}$$

Equation (4) models the fact that an instruction cannot enter its next stage before the current stage is completed, and Equation (5) models the fact that the next instruction cannot enter a certain pipeline stage before the current instruction has started its next stage.

We can graphically represent this constraint system as a weighted directed acyclic graph where the nodes correspond to the points p_i^j and the arrows correspond to the constraints between the points. Each instruction I_i is drawn as a column of points $p_i^1 \dots p_i^{n+1}$, with the constraints from Equation (4) shown as vertical arrows. The weight between p_i^j and p_i^{j+1} corresponds to r_i^j. For example, Figure 3(a) shows the points for the instruction `A` in a three-stage pipeline. $p_{\mathtt{A}}^1$ is the point where `A` enters the pipeline, and $p_{\mathtt{A}}^4$ is the point at which it leaves the pipeline. The constraints from Equation (5) are drawn as diagonal arrows with no weight, since they have weight zero. An example is shown in Figure 3(b) (we only show some of the p_i^n variables to reduce the clutter).

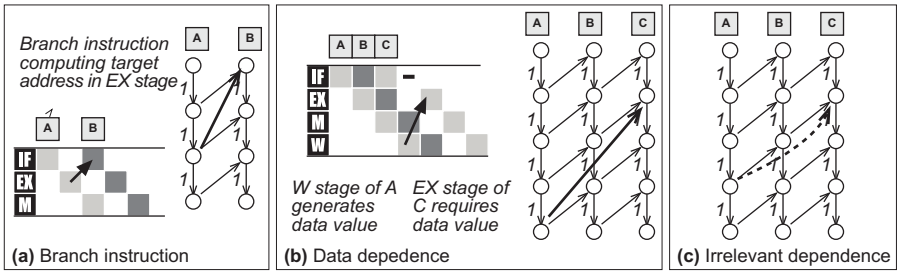


Fig. 4. Constraints for branches and data dependences

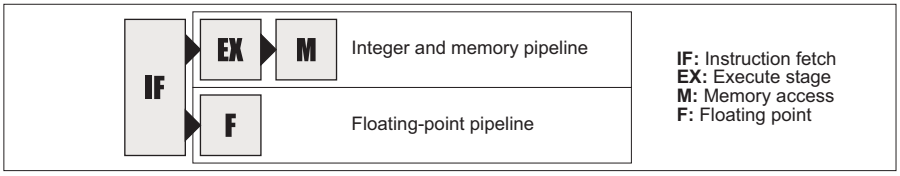


Fig. 5. Example in-order pipeline with parallel units

Additional dependences between instructions are represented by adding constraints to Equations (4) and (5).

– *Branch instructions* generate dependences between the end of the stage where the branch is decided (and the target address of the branch computed) and the fetch of the next instruction. A branch decided in stage j of instruction I_i generates the constraint

$$p_{i+1}^1 \geq p_i^{j+1} \tag{6}$$

– *Data dependences* between instructions, which imply that instruction I_i can enter stage j, only after some previous instruction I_k has completed stage l, generate the constraints

$$p_i^j \geq p_k^{l+1} \tag{7}$$

Examples of constraints for branches and data dependences are shown in Figure 4(a) and Figure 4(b). Note that these forms of constraints have effect only if they connect points that are not otherwise transitively connected via the basic constraints from Equations (4) and (5). Figure 4(c) shows some (irrelevant) data dependences that are subsumed by the basic constraints.

In a processor with multiple parallel pipelines, like the one shown in Figure 5, not all instructions will use all stages. In the constraint system, each instruction will then only have points corresponding to its entry into the pipeline (p_i^1), and points corresponding to the entry into each stage of the pipeline that it actually uses. The constraints Equations (4) and (5) are then reformulated using the functions $\texttt{previ}(i,j)$ and $\texttt{nexti}(i,j)$ which report for instruction I_i the previous and next instruction using pipeline stage j, and the functions $\texttt{prevs}(i,j)$ and $\texttt{nexts}(i,j)$ which report for a certain instruction I_i and pipeline stage j reports the previous and next pipeline stage used by I_i. The reformulated constraints are:

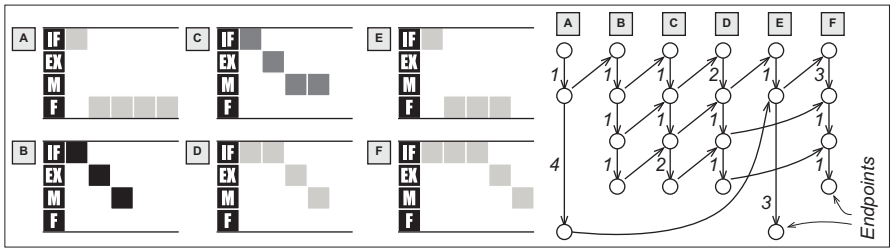


Fig. 6. Constraints for multiple pipelines

$$p_i^{\texttt{nexts}(i,j)} \geq p_i^j + r_i^j \tag{8}$$

$$p_{\texttt{nexti}(i,\texttt{prevs}(i,j))}^{\texttt{prevs}(i,j)} \geq p_i^j \tag{9}$$

Equation (8) corresponds to Equation (4) and Equation (9) corresponds to Equation (5).

Rrom these constraints we can calculate the execution time $T(I_1 \ldots I_m)$ of the sequence $I_1 \ldots I_m$ as follows. Define a *path* from p_i^j to p_k^l as a sequence of arrows from p_i^j to p_k^l in the constraint graph. The length of a path P, denoted $length(P)$, is the sum of the weights on the arrows in P. The *distance* $D(p_i^j, p_k^l)$ between two points p_i^j and p_k^l is defined as the maximal length of all possible paths from p_i^j to p_k^l (note that the distance can only be constructed if p_k^l can be reached from p_i^j):

$$D(p_i^j, p_k^l) = \max_{P \in (\text{all paths from } p_i^j \text{ to } p_k^l)} (length(P)) \tag{10}$$

We say that a path P from p_i^j to p_k^l is a *critical path* if $length(P) = D(p_i^j, p_k^l)$. In general, there may be more than one critical path between two points. The central proposition of this model is the following:

Proposition 1. *The execution time $T(I_1 \ldots I_m)$ of a sequence of instructions $I_1 \ldots I_m$ is the maximal distance from p_1^1 to some point in the constraint system.*

$$T(I_1 \ldots I_m) = \max_{1 \leq i \leq m, 1 \leq j \leq n+1} D(p_1^1, p_i^j))$$

The proof is straight-forward, see [6], and depends on the fact that the constraint graph is acyclic. Proposition 1 is analogous to a result in scheduling theory, saying that an ASAP (as-soon-as-possible) schedule always gives an optimal schedule in an acyclic graph of dependences. It is also analogous to the distance calculations used with simple temporal constraints [5].

4 Properties of Pipelines

Using the model introduced in Section 3, we can make general characterizations of when LTEs occur and whether they are positive and negative.

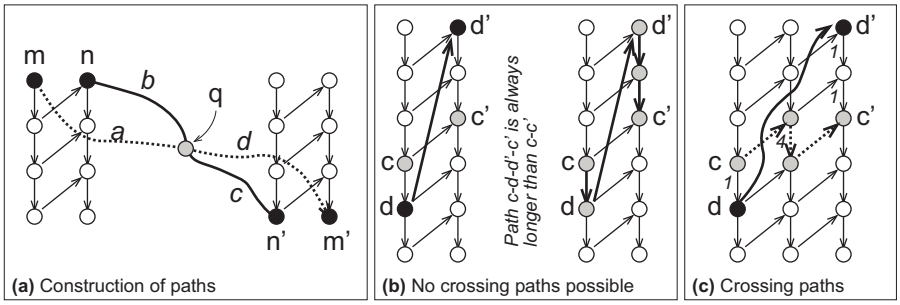


Fig. 7. Illustrating the principles of Theorem 2 and Theorem 3

4.1 Source of LTEs

Given a sequence $I_1 \dots I_m$ of instructions, we say that I_1 *stalls* I_i if $D(p_1^1, p_i^j)) > D(p_2^1, p_i^j)) + r_1^1$. Intuitively, this means that some stage j of some successor instruction I_i is delayed due to waiting for some corresponding stage of I_1 (we assume that the first stage occurs in all instructions). Note that due to data dependences and parallel pipelines, such stalls can occur between non-adjacent instructions.

Theorem 1. *For a single in-order pipeline, a timing effect $\delta_{I_1 \dots I_m} \neq 0$ can occur for a sequence of instructions $I_1 \dots I_m, m \geq 3$ only if I_1 stalls the execution of some instruction in $I_2 \dots I_m$.*

Proof. (Sketch) Intuitively, if I_1 does not disturb any of its successor instructions, the execution of $I_2 \dots I_m$ will be identical to the case when they are executed starting with I_2. The theorem can be proven by a straight-forward calculation, using the definition of stalling. A full proof is given in [6, Section 5.2.5] □

Note that for pipelines that fork, an LTE over $I_1 \dots I_m$ can occur if I_1 uses some resource that is not used by $I_2 \dots I_{m-1}$, as shown by example in Section 4.3.

4.2 Pipelines without Positive LTEs

The central result of this section is a characterization of pipeline structures that have no positive LTEs. We say that a pipeline has the *crossing critical path* (CCP) property if for any constraint system model of a sequence $I_1 \dots I_m$ of instructions with $m \geq 3$, there is a critical path P_1 between p_1^1 and p_m^{n+1}, and a critical path P_2 between p_2^1 and p_{m-1}^{n+1} such that P_1 and P_2 have a common point. We assume that there is a common last step $n + 1$ in the pipeline (thus, we do not consider forking pipelines here).

We will later give classes of pipelines that have the CCP property. The importance of CCP is that it guarantees the absence of positive LTEs.

Theorem 2. *For a sequence of instructions $I_1 \dots I_m, m \geq 2$, executing on a pipeline with the CCP property, $\delta_{I_1 \dots I_m} \leq 0$.*

Proof. (Sketch) Consider the sequence $I_1 \ldots I_m$ and its corresponding constraint system. We shall prove that $\delta_{I_1 \ldots I_m} \leq 0$. Introduce the following names for points in the constraint system (see Figure 7(a)):

- m, corresponding to the start of instruction I_1
- n, corresponding to the start of I_2
- n', corresponding to the end of I_{m-1}
- m', corresponding to the end of I_m

The distances between these points form the execution times involved in the $\delta_{I_1 \ldots I_m}$ calculation as follows:

$$\begin{aligned} T(I_1 \ldots I_m) &= D(m, m') \\ T(I_2 \ldots I_{m-1}) &= D(n, n') \\ T(I_1 \ldots I_{m-1}) &= D(m, n') \\ T(I_2 \ldots I_m) &= D(n', m) \\ \delta_{I_1 \ldots I_m} &= D(m, m') + D(n, n') - D(m, n') - D(n, m') \end{aligned}$$

To prove that $\delta_{I_1 \ldots I_m} \leq 0$, we must show that $D(m, m') + D(n, n') \leq D(m, n') + D(n, m')$. Select a critical path P_1 from m to m', shown as a dashed line in Figure 7(a), and a critical path from n to n', shown as a solid line in Figure 7(a). By the CCP property, we can choose P_1 and P_2 so that they cross at some point q, dividing each of the two paths into two parts. Label the path segments by a, b, c, and d as in Figure Figure 7(a). Using the fact that the distance between two points is at least as long as any path between the points, we get the following inequalities:

$$\begin{aligned} D(m, m') &= a + d \\ D(n, n') &= b + c \\ D(m, n') &\geq a + c \\ D(n, m') &\geq b + d \end{aligned}$$

We infer that $D(m, m') + D(n, n') \leq D(m, n') + D(n, m')$, from which we conclude $\delta_{I_1 \ldots I_m} \leq 0$. □

We can now proceed to characterize pipelines with the CCP property.

Theorem 3. *A single in-order pipeline has the CCP property if each constraint introduced by branches (Equation (6)) and data dependences (Equation (7)) either*

- *occurs between adjacent instructions, or*
- *is subsumed by the basic constraints from Equations (4) and (5), as explained in Section 3*

Proof. (Sketch) For single in-order pipelines without any branch or data dependency, the CCP property follows from the observation that the corresponding constraint graph is planar, and hence that critical paths cannot overtake each other without intersecting at some point. When adding constraints that represent branches and data dependences,

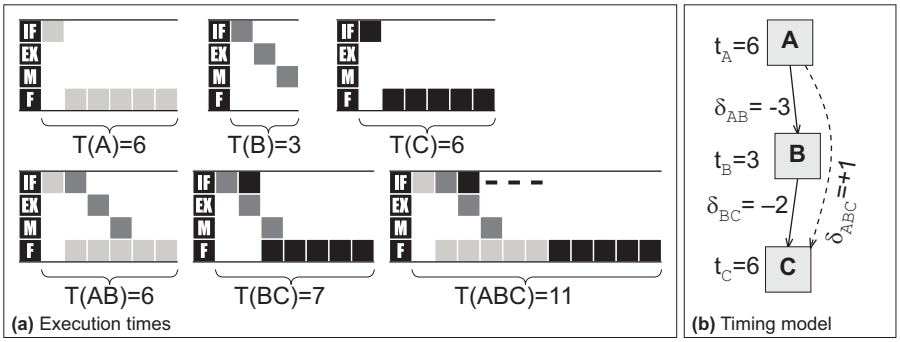


Fig. 8. Long timing effect in parallel pipelines

the critical issue is whether they may allow critical paths to overtake without intersecting. As illustrated in Figure 7(b), this is not the case for constraints between adjacent instructions, since such constraints will always be included in the critical path if they make it longer. □

Note that if we have data dependences across more than two instructions, the case in Figure 7(c) can occur, and the two paths can cross without sharing a node.

All branch dependences are between adjacent instructions, so the critical issue for the applicability of Theorem 3 is whether data dependences appear between non-adjacent instructions. No such dependences can appear if all data dependences in a pipeline only reach from a stage j to its predecessor stage $j-1$. This is thanks to the fact that since the dependence goes at most two points back in the pipeline, and all dependences beyond the non-adjacent instructions will be subsumed by the regular constraints (as illustrated in Figure 4(c)).

In practice, data dependences only between adjacent instructions is a common case, thanks to data forwarding paths [11] that avoid most data dependences. Examples of processors with pipelines exhibiting this nice behavior are the ARM9 [1], NEC V850 [17] (not the V850E), Hitachi SH7700 [12], and Infineon C167 [13].

4.3 Parallel Pipelines Cause Positive LTEs

A pipeline that forks into parallel pipelines can exhibit positive long timing effects due to interference between instructions being sent into one pipeline, if some intervening instructions are sent to some other pipeline. An example is shown in Figure 8.

There might be pipelines where such effects never actually materialize, but in most real-life forking pipelines, this type of positive timing effects do occur and have to be accounted for in WCET analysis. Examples of such processors are the NEC V850E [18], MIPS R4000 [10], MicroSPARC I [9], and basically any processor employing a separate floating point pipeline.

4.4 LTEs Across Unbounded Number of Instructions

It is not in general possible to provide a bound on the length of the sequences of instructions that can exhibit long timing effects.

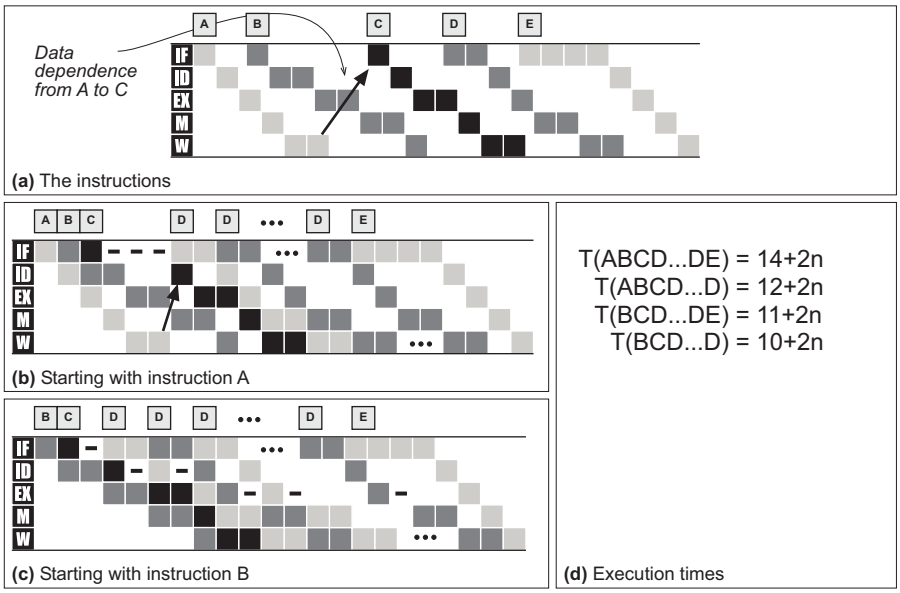


Fig. 9. Example of an unbounded positive timing effect

The example in Figure 9 demonstrates that we can get a positive LTE after an unbounded number of instructions. There is a data dependence between instructions `A` and `C`, which causes the execution of `BCD...E` to be different depending on whether `A` is executed before `B` or not. After an arbitary number of instructions `D`, we get a positive timing effect when instruction `E` is added, since:

$$\delta_{\texttt{A...E}} = 14 + 2n - 12 - 2n - 11 - 2n + 10 + 2n = +1$$

Thus, we can have LTEs across an arbitrary number of instructions. Note that there is a timing effect $\delta_{\texttt{ABC}} = -1$, but no other timing effect until the effect across `A...E` appears. One should also note that it is possible that several positive timing effects occur from the same disturbance [6].

4.5 Absence of Timing Anomalies

Using the model in Section 3, we can also establish a different but important property of in-order pipelines, the absence of *timing anomalies*, as introduced by Lundqvist and Stenström [16]. Given a sequence of instructions $I_1 \ldots I_m$, when we change the execution time of the first instruction by adding n cycles, we will get a new execution time for the sequence. If this time is less than the previous time, or the new time is more than n cycles longer than the previous time, we have a timing anomaly. We translate this property into our model by assuming that precisely one pipeline stage for I_1 is extended to take more cycles.

By example, Lundqvist and Stenström demonstrated that out-of-order processors can suffer from timing anomalies. Here, we show that for pipelines that can be modeled by our constraint systems (in-order pipelines), such effects cannot occur. Note that

branch prediction coupled with speculative cache fetches can cause timing anomalies, regardless of the simplicity of the pipeline, as demonstrated for the Coldfire 5307 [8]. To avoid timing anomalies, we require that a processor is free from all kinds of dynamic scheduling and speculative processing.

The important consequence of timing anomalies is that if anomalies occur, we cannot use the simplifying assumption that the worst-case execution time for the program can be derived by assuming worst-case execution times for each individual instruction. Otherwise, every possible pipeline and/or cache state has to be considered, which is very complex [8].

Theorem 4. *For in-order pipelines, no timing anomalies can appear when we increase the execution time of an instruction.*

Proof. The original execution time for $I_1 \dots I_m$ corresponds to the critical path in a constraint system C_1. We increase the execution time by increasing the time for I_1 to complete one of its stages, obtaining a new constraint system C_2, where some r_1^j is bigger than in C_1.

The new execution time for $I_1 \dots I_m$ corresponds to some critical path in constraint system C_2. If the arrow with r_1^j was on the critical path before, the execution time will increase by k cycles, $d = k$. If it was not, and now is included, the time will increase by at most k. If it is not on the critical path of either C_1 and C_2, $d = 0$. Thus, $0 \leq d \leq k$, and no timing anomaly can appear. □

Theorem 5. *For in-order pipelines, no timing anomalies can appear when we decrease the execution time of an instruction.*

Proof. Analogous to the proof of Theorem 4 [6]. □

5 The Safety of WCET Analysis

To perform safe WCET analysis for pipelines, we need to make sure that no positive LTEs are missed. If no positive LTEs can occur at all, as is the case for the processors with the CCP property, any timing model that considers instructions, pairs of instructions, etc. is safe, since there are no positive LTEs that can be missed. However, for many types of processors, positive LTEs can occur, and there are several ways of handling them.

For approaches only analyzing adjacent pairs of instructions, the presence of positive LTEs makes it necessary to use conservative approximations. Such an approximate model would overestimate the execution time over shorter sequences, in order to make sure that the total execution time when LTEs are taken into account is not underestimated.

One way to construct such a model is to use reservation tables and consider their pairwise concatenation, but not allow the instruction profiles to change their shape when concatenated. This means that stalls will not materialize, as each instruction will execute as it would in isolation. Thus no long effects can occur within a pipeline, according to Theorem 1. To avoid the potential of LTEs resulting from parallel pipelines, it is

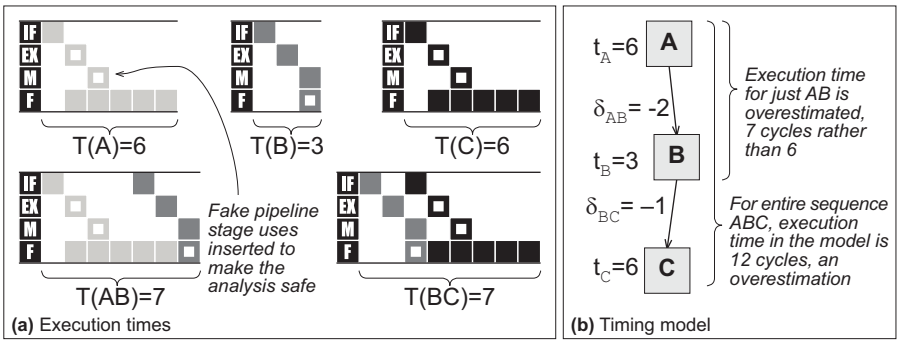


Fig. 10. Pairwise conservative model for Figure 8

necessary that each instruction (or basic block) use every pipeline stage, as proposed in [19] [1]. Also, if there are data dependences between non-adjacent instructions it is necessary to account for the worst possible delay due to data waits, in each pair of instructions. Note that just concatenating instructions pairwise without stalls [4, 21] is not necessarily safe for parallel pipelines.

Figure 10 shows how the case in Figure 8 would be modeled with this type of approximation: node `A` would not be allowed to completely overlap node `B` (as shown for the sequence `AB`), which makes the timing effect between the nodes -2 instead of -3. Thus, the positive timing effect of the interference between `A` and `C` is taken early, on the edge between `A` and `B`, which is safe but pessimistic. If a processor has instructions that can execute for quite a long time in parallel to other instructions, this type of approximation is likely to give very high overestimations.

In our WCET analysis method based on examining sequences of nodes [6, 7], it is critical to define correct criteria for the termination of the search that finds all (positive) LTEs. If those criteria are wrong, we might get an underestimated WCET. In practice, this type of analysis works very well for simple processors.

Another approach maintains multiple pipeline states for each basic block, and in each step tries to prune the states that cannot result in the longest execution time [14]. This pruning operator has to consider the potential of LTEs to be correct.

There are some analysis approaches where the pipeline behavior is modeled along paths consisting of many basic blocks [9, 22] (usually, this means that several different paths between two points in the code are analyzed). Inside each path, LTEs are absorbed with perfect precision, but where paths are concatenated, approximations that cover all possible LTEs have to be used.

A different philosophy is to maintain all possible states of the pipeline for each node in the analysis, without attempting to prune the set of pipeline states [8, 20]. In such approaches, the presence of LTEs will mean an increased analysis complexity (as each instruction will be subject to more different states). This approach is not dependent on the absence of timing anomalies and should find all LTEs. The resulting analysis is very expensive, however [8].

[1] The placement of such extra pipeline stages is qiute important to achievable precision, and has to be carefully considered for each pipeline

6 Conclusions

In this paper, we have presented a mathematical model of instruction execution on in-order single-issue pipelined processors. We have used this model to examine the timing of instructions from the perspective of static worst-case execution time (WCET) analysis, especially considering timing effects between non-adjacent instructions (long timing effects, LTEs). There are negative LTEs, which can be safely ignored, and positive LTEs that add instruction time to a sequence of instructions, and that have to be accounted for in a safe analysis.

Even simple pipelines can exhibit LTEs across arbitrary numbers of instructions, which makes it necessary for static WCET analysis to consider more than pairs of instructions. For many pipelines, all LTEs are negative, which means that they can be ignored safely, at the cost of lower precision. Measurements indicate that (we have experimented with the NEC V850E processor) ignoring all negative LTEs can give overestimations of up to 20% of the execution time of a program [6]. For processors with parallel floating point pipelines, the effect could be much greater.

Another result obtained using our pipeline model is that in-order single-issue processors are not subject to timing anomalies, which indicates that it is possible to use local worst-case assumptions to derive a global worst-case execution time. This allows for efficient static analysis, since we do not have to consider all possible execution times for each instruction in order to find the WCET.

The results in this paper indicates that certain processors are better suited for use in predictable systems. For example, the timing of a processor without timing anomalies and only negative LTEs is very easy to analyze compared to a processor with positive LTEs and timing anomalies. It would be interesting to try to design a high-performance high-predictability processor for embedded real-time systems.

In conclusion, we reached a better understanding of how to construct safe static WCET analysis methods for pipelined processors, and have identified some non-trivial problems related to the achievable safety and precision of WCET analysis. This provides static WCET analysis with a firmer theoretical background, which will help us build better analysis methods. We hope to continue this work by extending the pipeline model to more classes of pipelines and continue the investigation into pipeline properties.

References

1. ARM Ltd. *ARM 9TDMI Technical Reference Manual*, 3^{rd} edition, March 2000. Document no. DDI 0180A.
2. P. Atanassov, R. Kirner, and P. Puschner. Using Real Hardware to Create an Accurate Timing Model for Execution-Time Analysis. In *Proc. IEEE Real-Time Embedded Systems Workshop, held in conjunction with RTSS 2001*, December 2001.
3. I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proc. 7^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, pages 39–48. IEEE Computer Society Press, December 2000.

4. A. Colin and I. Puaut. A Modular and Retargetable Framework for Tree-Based WCET Analysis. In *Proc. 13th Euromicro Conference on Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, June 2001.
5. Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artifical Intelligence*, 49:61–95, 1991.
6. J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, April 2002. Acta Universitatis Upsaliensis, Dissertations from the Faculty of Science and Technology 36, `http://publications.uu.se/theses/91-554-5228-0/`.
7. J. Engblom and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, December 1999.
8. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proc. First International Workshop on Embedded Software (EMSOFT 2001), LNCS 2211*. Springer-Verlag, October 2001.
9. C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.
10. J. Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies Inc., 2nd edition, 1994.
11. J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2nd edition, 1996. ISBN 1-55860-329-8.
12. Hitachi Europe Ltd. *SH7700 Series Programming Manual*, September 1995.
13. Infineon. *Instruction Set Manual for the C166 Family*, 2nd edition, March 2001.
14. 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.
15. S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.
16. T. Lundqvist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS'99)*, December 1999.
17. NEC Corporation. *V850 Family 32/16-bit Single Chip Microcontroller User's Manual: Architecture*, 4th edition, 1995. Document no. U10243EJ4V0UM00.
18. NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3rd edition, January 1999. Document no. U12197EJ3V0UM00.
19. 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.
20. Jörn Schneider and Christian Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.
21. Friedhelm Stappert. Predicting pipelining and caching behaviour of hard real-time programs. In *Proc. of the 9th Euromicro Workshop on Real-Time Systems*. IEEE Computer Society Press, June 1997.
22. D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, and R. Ernst. Interval-Based Analysis of Software Processes. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'2001)*, June 2001.