Computing Execution Times with eXecution Decision Diagrams in the Presence of Out-Of-Order Resources

Zhenyu Bai, Hugues Cassé, Thomas Carle, Christine Rochange

IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3

118 route de Narbonne 31062 Toulouse, France

Email: *firstname.surname*@irit.fr

Abstract—Worst-Case Execution Time (WCET) is a key component for the verification of critical real-time applications. Yet, even the simplest microprocessors implement pipelines with concurrently-accessed resources, such as the memory bus shared by fetch and memory stages. Although their *in-order* pipelines are, by nature, very deterministic, the bus can cause out-of-order accesses to the memory and, therefore, *timing anomalies*: local timing effects that can have global effects but that cannot be easily composed to estimate the global WCET. To cope with this situation, WCET analyses have to generate important over-estimations in order to preserve safety of the computed times or have to explicitly track all possible executions. In the latter case, the presence of out-of-order behavior leads to a combinatorial blowup of the number of pipeline states for which efficient state abstractions are difficult to design. This paper proposes instead a compact and exact representation of the timings in the pipeline, using eXecution Decision Diagram (XDD) [1]. We show how XDD can be used to model pipeline states all along the execution paths by leveraging the algebraic properties of XDD. This computational model allows to compute the exact temporal behavior at *control flow graph* level and is amenable to efficiently and precisely support WCET calculation in presence of out-of-order bus accesses. This model is finally experimented on the TACLe benchmark suite and we observe good performance making this approach appropriate for industrial applications.

Index Terms—real-time, WCET, static analysis, pipeline

I. INTRODUCTION

The correct behavior of hard real-time systems depends not only on its functional behavior but also on its temporal behavior. The latter is guaranteed by the scheduling analysis of the tasks composing the system, which relies on the estimation of their Worst-Case Execution Time (WCET).

With modern processors, the execution time of a code snippet is difficult to determine. For instance, on a processor equipped with cache memories, the latency of memory accesses is variable: it depends on whether the access results in a *Cache Miss* or a *Cache Hit*. Although the latency of memory access it-self is statically known (i.e. the latency in case of a Miss), it cannot be easily accounted when computing the execution time. The presence, in modern micro-architectures, of pipelined and superscalar execution and other mechanisms to favor instruction-level parallelism achieving high performance causes a large variability in the execution times and makes the WCET analysis to suffer from *timing anomalies* [2]–[5]. Briefly, timing anomalies state that the

WCET analysis cannot assert a local worst case with a constant worst case temporal effect. Illustrated on the case of cache accesses, this means that Cache Miss (longer access) cannot be asserted to be the global worst case and no constant worst time contribution to the global WCET can be determined [6], [7].

Unless the target processor is proved to be timing-anomalies free, a safe and precise WCET analysis has to capture them by precisely tracking the execution states of the micro-architecture. Hence the WCET computation is generally decomposed into two parts [8]. Firstly, *global analyses*, independent from pipeline's structure, are performed: they typically encompass cache and branch-prediction analyses, which determine the behavior of these mechanisms at instruction level. Secondly, the *pipeline analysis* uses the information provided by *global analyses* to determine how instructions are executed through the pipeline, and to determine the (worst-case) execution times of instruction sequences.

In [1], Bai et al. show that the eXecution Decision Diagram (XDD) is good data structure to record times in pipeline analysis. An XDD can be deemed as a lossless compression of the relationship between the execution time of an instruction sequence and the combination of the occurrence of timing variations. By implanting XDD into the pipeline model based on the Execution Graph (XG), they have achieved exact and efficient pipeline analysis on sequentially executed instructions in in-order pipelines. In this [1], the pipeline analysis is designed to consider *Basic Blocks* (BB) of program independently by calculating their worst-case execution context. However, with the presence of out-of-order resources like shared buses, the conservative use of a worst-case context does not hold any more. We need to precisely track the possible execution contexts in order to evaluate how the concurrent accesses to the bus are interlaced. The pipeline analysis has to analyze the micro-architecture states on the whole of program i.e. at Control-Flow Graph (CFG) level.

a) Contribution: This paper shows (a) how we adapt the original graph based pipeline model proposed in [1] into a data-flow analysis applied at CFG level which computes exactly all possible temporal pipeline states; (b) how, by leveraging the algebraic properties of XDD, we construct an efficient computational model of our analysis; and (c) how we exploit the precise pipeline states produced by this computa-

tional model to support a typical out-of-order resource: the shared memory bus between instruction cache and data cache to access the main memory.

b) Outline: Section II presents the background knowledge about the XG model and XDD. In section III, we extend the original model of XG with XDD to a resource-based model which is able to express the state of pipeline with a vector. Later in this section, we show how to leverage the algebraic properties of XDD in order to ameliorate the performance of the analysis. In section IV, we show how to build the complete analysis at CFG level. Section V extends our model to support the shared memory bus. Experimentation in section VI demonstrates the efficiency of our analysis on realistic benchmarks. Several metrics are examined and discussed. Related works are presented in section VII and we conclude in Section VIII.

II. BACKGROUND

As the analyzed program has several execution paths and possibly loops, it is impossible to track all the possible execution traces. The static WCET analysis approach we use in this paper models the whole program as a CFG, then computes the WCET of each BB and determines the WCET using the *Implicit Path Enumeration Techniques* (IPET) [9]. Thus, the pipeline analysis aims to determine the execution time of each BB, for example, using *Execution Graphs*.

A. Execution Graphs

An eXecution Graph (XG) [10], [11] models the temporal behavior of an instruction sequence (like a BB) executed in the pipeline. The key idea of XG is to model the temporal behavior by considering the dependencies arising between instructions during their execution in the pipeline stages. For example, an instruction have to exit a pipeline stage to start its execution in the next stage, an instruction have to read a register after another instruction has written this register, etc. This results in a dependency graph: a vertex represents the progress of an instruction in a pipeline stage; the edges represent the precedence relationships between these vertices. Formally, let $\mathcal{I}$ be the set of machines instructions, and let XG be a *Directed Acyclic Graph* (DAG) $G_{XG} = \langle V_{XG}, E_{XG} \rangle$ built for an instruction sequence $Seq \in \mathcal{I}^*$ s.t.

- V_{XG} is the set of vertices defined by $V_{XG} = \{[I_i/s] \mid I_i \in Seq \land s \in \mathcal{P}\}$, with $\mathcal{P}$ the set of pipeline stages.
- $E_{XG} \subset V_{XG} \times V_{XG}$, the set of edges, is built according to the dependencies in the considered pipeline.

In addition, an XG is decorated with temporal information:
- $\lambda_v \in \mathbb{N}$ is the latency of the XG vertex v, that is, the time spent in this vertex.
- $\delta_{v \to w} \in \{0, 1\}$ represents the effect of dependencies of the edge $v \to w \in E_{XG}$. If $\delta_{v \to w} = 1$ (solid), w starts after the end of v; if $\delta_{v \to w} = 0$ (dotted), w can start at the same time or after the start of v.

Examples in this paper consider a 5-stages (FE – fetch, DE – decode, EX – execute, ME – memory, WB – write-back), in-order, 2-scalar pipeline but the presented algorithms
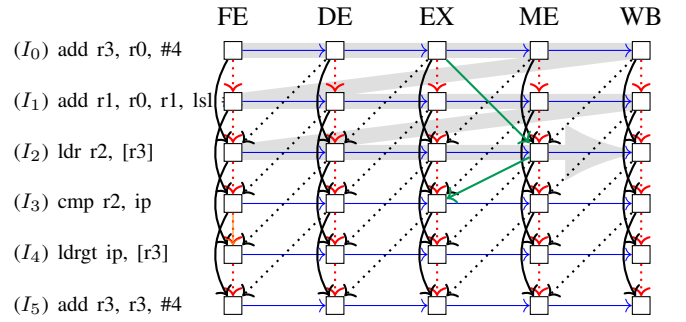

Fig. 1: XG model of an instruction sequence.

are not limited to this configuration. Figure 1 shows the XG for this pipeline and for the sequence of instructions on the left. The vertices correspond to the use of a stage (column headers) by an instruction (row headers) through the pipeline. The edges are generated according to the following dependencies:

- The horizontal solid edges model the *Pipeline Order*: an instruction goes through the pipeline in the order of stages.
- The vertical dotted edges model the parallel execution of instructions in the super-scalar stages (*Program Order*).
- The vertical solid bent edges model the capacity limit of the stages – 2 instructions per cycle (*Capacity Order*).
- The slanted dotted edges model the capacity of FIFO queues (2 instructions) between stages (*Queue Capacity*).
- The slanted solid edges model the *Data Dependencies* between instructions when an instruction reads a register written by a prior instruction.

The set of dependency edges shown above are typical for in-order pipelines. Depending on a particular pipeline design, rules to build the edges may be added or removed to account for specific features.

Using an XG, the start time of an instruction in a stage ρ_w is computed as the earliest time at which all incoming dependencies are satisfied and the end time ρ_w^* as ρ_w increased by the time passed by the instruction in the stage:

$$\rho_w = \max_{v \to w \in E_{XG}} \rho_v + \delta_{v \to w} \times \lambda_v \qquad (1)$$

$$\rho_w^* = \rho_w + \lambda_v \qquad (2)$$

The execution time of the instruction sequence is obtained by calculating the start time of each vertex following a topological order in the XG. Since the pipeline is in-order (all resources are allocated in program order), the instruction timing only depends on prior instructions meaning that, at least, one topological order exists. In in-order processors, this order is implied by the combination of the *Pipeline Order* (horizontal edges) and of the *Capacity Order* (vertical edges). It is highlighted in the example XG of Figure 1 by the light gray arrow in the background.

The computation of time in XG is fast and efficient but, as soon as the pipeline produces variable times (like Cache Hits/Misses), to be precise, the computation has to be performed for each combination of these times, causing a computation complexity blowup. Yet, the data structure presented in the next section alleviates this issue.

B. Execution Decision Diagram

The eXecution Decision Diagram (XDD) is inspired from the Binary Decision Diagram (BDD) [12], [13] and its Multi-Terminal BDD (MTBDD) variant [14]. An XDD is a DAG that is recursively defined as:

Definition II.1.

$$\text{XDD} = \text{LEAF}(k) \mid \text{NODE}(e, \overline{f}, f) \qquad (3)$$

The Boolean variables $e \in \mathcal{E}$ in the nodes are called *events*: they model the uncertainty in the analysis regarding the micro-architecture state and its impact on the time e.g. whether a particular cache access results in a Hit or in a Miss. The sub-trees $f, \overline{f} \in \text{XDD}$ represent, respectively, the situations where the event e happens or not. The leaves of XDDs stores the execution times $k \in \mathbb{Z}^{\#} = \mathbb{Z} \cup \{+\infty, -\infty\}$.

As in OBDDs [13], XDDs deploy hash consing techniques to guarantee the unicity of the sub-trees instances and to speed up the calculations. Thus, identical sub-trees share the same instance in memory. This compression allows the XDDs to represent efficiently the relationship between the combinations of events (called *configurations*) and the corresponding execution times. A configuration $\gamma \in \Gamma$ is the combination of activation or inactivation of events – $\Gamma = \wp(\mathcal{E})$, and corresponds to a path from the root node to a leaf in the XDD DAG.
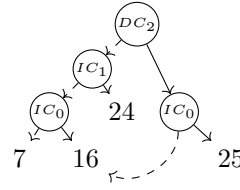
When the events are taken into account, the actual time for each vertex is represented by a map between the event configurations and the corresponding times (i.e. in the domain $\Gamma \to \mathbb{Z}^{\#}$) which size is combinatorial with respect to the number of events and thus is costly in computation time. XDDs efficiently solve this problem by factorizing identical sub-trees. In [1], it has been shown that the subtree factorization frequently occurs in realistic benchmarks which largely speeds up the analysis. Yet, an isomorphism between XDD and $\Gamma \to \mathbb{Z}^{\#}$ exists: $\text{XDD} \xrightarrow[\beta]{\alpha} (\Gamma \to \mathbb{Z}^{\#})$. This means that XDD can be deemed as a lossless compression of the map $(\Gamma \to \mathbb{Z}^{\#})$. In the remainder of the paper, we note $f[\gamma]$ (for $f \in \text{XDD}$ or $f \in \Gamma \to \mathbb{Z}^{\#}$) the time corresponding to the configuration γ in f.

Figure 2a shows an example of XDD with 8 possible configurations: right edges of a node e represent the activation of event e and the left ones the non-activation. The original map between configuration and the execution time is shown in the figure 2b on the right (IC and DC represent, respectively, the instruction and data cache events, over-line bar denotes inactive events).

[1] has shown that any binary operation $\odot$ on $\mathbb{Z}^{\#}$ used in $(\Gamma \to \mathbb{Z}^{\#})$ can be transferred in the XDD domain in an equivalent operation $\boxdot$ such that performing the operation in XDD domain is lossless:

$$\forall s_1, s_2 \in (\Gamma \to \mathbb{Z}^{\#})^2, \forall \gamma \in \Gamma,$$
$$s_1[\gamma] \odot s_2[\gamma] = (\alpha(s_1) \boxdot \alpha(s_2))[\gamma] \qquad (4)$$

With α the morphism from $(\Gamma \to \mathbb{Z}^{\#})$ to XDD.

(a) Example XDD

Configuration	time
IC_0, IC_1, DC_2	25
$IC_0, \overline{IC_1}, DC_2$	25
$\overline{IC_0}, IC_1, DC_2$	16
$\overline{IC_0}, \overline{IC_1}, DC_2$	16
$IC_0, IC_1, \overline{DC_2}$	24
$IC_0, IC_1, \overline{DC_2}$	24
$IC_0, \overline{IC_1}, \overline{DC_2}$	16
$\overline{IC_0}, \overline{IC_1}, \overline{DC_2}$	7

(b) Explicit representation

(c) Example of $\oplus$

Fig. 2: Example of XDDs

The implementation of this operation is detailed in [1]. Shortly, $\boxdot$ combines the XDD operands along the sub-trees and applies $\odot$ when leaves need to be combined. As the operations used in XG analysis are max and $+$ (Eq. 1), the equivalent operations on XDDs are, respectively $\oplus$ and $\otimes$:

$$\forall s_1, s_2 \in (\Gamma \to \mathbb{Z}^{\#})^2, \forall \gamma \in \Gamma,$$
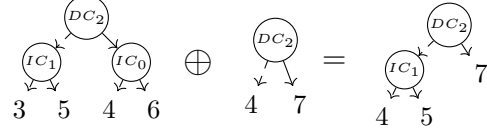$$s_1[\gamma] + s_2[\gamma] = (\alpha(s_1) \otimes \alpha(s_2))[\gamma] \qquad (5)$$
$$max(s_1[\gamma], s_2[\gamma]) = (\alpha(s_1) \oplus \alpha(s_2))[\gamma]$$

The work of $\oplus$ and its ability to reduce the size of XDDs is illustrated in the example of Figure 2c.

The isomorphism guarantees that using XDD to perform the XG analysis is *precise* by following the dependency resolution rule (Equation 1) and following the proposed topological order. By exploiting the properties of XDDs, the next section proposes a new paradigm of BB time calculation to support the pipeline analysis at the CFG level and of out-of-order accesses to the bus.

III. RESOURCE BASED MODEL

The usual approach – consisting in building and solving an XG for each BB on its own, is no more sustainable when out-of-order bus accesses have to be supported. Indeed, the bus access interactions can span over BB bounds while XGs for whole execution paths are inconvenient to build and the number of execution paths is intractable.

This section proposes to solve this issue by turning the original XG model into a state machine model where the pipeline analysis is performed by applying transitions on the pipeline states. Moreover, by leveraging the algebraic property of XDD, we improve the computational model by implementing the transitions as matrices multiplications. The matrices can be pre-computed before the pipeline analysis.

A. Temporal State

The dependencies $d \in \mathcal{D}$ in the XG model the uses and releases of *resources* e.g. stages, queues etc. For instance, in our 5-stage in-order pipeline, determining the start time of an instruction $I_i \in \mathcal{I}$ in stage DE requires:

Program Order	Capacity Order		Pipeline Order	Queue Capacity	
$\rho_{[I_{i-1}/DE]}$	$\rho^*_{[I_{i-2}/DE]}$	$\rho^*_{[I_{i-1}/DE]}$	$\rho^*_{[I_i/FE]}$	$\rho_{[I_{i-2}/EX]}$	$\rho_{[I_{i-1}/EX]}$

TABLE I: Necessary information determining the start time of any I_i in the DE stage.

- the start time of the previous instruction in the DE stage: $\rho_{[I_{i-1}/DE]}$ (Program Order),
- the end time of the second last instruction in the DE stage: $\rho^*_{[I_{i-2}/DE]}$ (Capacity Order),
- the end time of I_i in the FE stage: $\rho^*_{[I_i/FE]}$ (Pipeline Order),
- the start time of the second last instruction in the EX stage: $\rho_{[I_{i-2}/EX]}$ (Queue Capacity).

Where I_{i-n} represents the nth previous instruction. $\rho_{[I_i/s]}$ and $\rho^*_{[I_i/s]}$ respectively stand for the start and the end time of the $[I_i/s]$ vertex. The actual start time of I_i in DE is the earliest date at which all dependencies are satisfied:

$$\rho_{[I_i/DE]} = \rho_{[I_{i-1}/DE]} \oplus \rho^*_{[I_{i-2}/DE]}$$
$$\oplus \rho^*_{[I_i/FE]} \oplus \rho_{[I_{i-2}/EX]}$$
$$\rho^*_{[I_{i-2}/DE]} = \rho_{[I_{i-2}/DE]} \otimes \lambda_{[I_{i-2}/DE]}$$
$$\rho^*_{[I_i/FE]} = \rho_{[I_i/FE]} \otimes \lambda_{[I_i/FE]}$$

This corresponds to the computation of Equation 1 extended to the XDD domain: we use $\oplus$ instead of max and $\otimes$ instead of $+$. As in the integer case, each computation requires the results of the computations of previous instructions (e.g. $\rho_{[I_i/DE]}$ requires $\rho_{[I_{i-1}/DE]}$ and $\rho^*_{[I_{i-2}/DE]}$) which correspond to the release time of the concerned resources.

Table I sums up the dependency information required to compute the start time of any instruction I_i in the DE stage. The necessary information may differ depending on the pipeline architecture, but an important point is that any architecture that can be described in the XG model can also be expressed as a vector of XDDs as Table I.

In the same fashion, such a vector can be built for each instruction and for each stage of the pipeline. Table II shows the complete dependency information to be maintained for all stages of the example pipeline: each line represents a stage and each column represents a dependency on a resource to be satisfied to start the stage execution. The symbol $-\infty$ is used when no dependency is required[1]. I_{fetch}, I_{load}, I_{store} and I_{R_i} are, respectively, the last instructions that fetched an instruction block from memory, performed a load, a store and wrote to register R_i (in stage s_{Ri}).

Finally, Table II sums up the set of dependencies an instruction has to satisfy considering all possible pipeline stage, i.e. $\mathcal{D}$. Grouped in an XDD *vector*, defined as $\mathcal{S} = \mathrm{XDD}^{|\mathcal{D}|}$, they precisely represent the *temporal state* of the pipeline. For a given stage s, a *temporal state* $\vec{S} \in \mathcal{S}$, and $\mathcal{D}_{[I_i/s]} \subset \mathcal{D}$ the set of dependencies applicable to XG vertex $[I_i/s]$, start and end times can now be rewritten as:

$$\rho_{[I_i/s]} = \bigoplus_{d \in \mathcal{D}_{[I_i/s]}} \vec{S}[i_d] \quad (6)$$

$$\rho^*_{[I_i/s]} = \rho_{[I_i/s]} \otimes \lambda_{[I_i/s]} \quad (7)$$

[1] $-\infty$ is convenient as it is neutral for the max operation.

Notice that the function $\delta_{v \to w}$ is useful as its effect is supported by the dependency on start or end time in $\vec{S}$.

B. Pipeline Analysis with Temporal States

We now present how *temporal states* are updated during the analysis to account for the execution of instructions in the pipeline. To simplify the computations, we add a slot ρ at index i_ρ into the state vector that records the *current time* all along the analysis, which we call the *time pointer*.

Definition III.1. Following the principle of XG analysis, the behavior of an instruction in a stage can be divided into four steps.

- Step 1. Before being executed in the stage, the instruction waits until all dependencies are satisfied (Eq. 6). To model this behavior with the *temporal state*, the time pointer is reset to $\mathbb{0} = LEAF(-\infty)$ (Step 1.1). Then, each dependency time is accumulated with $\oplus$ into the time pointer (Step 1.2). At the end of Step 1, the time pointer records the maximum release time of all dependencies which is the actual start time for the analyzed XG vertex. The transitions for the *temporal state* are defined with the functions τ_{reset} and τ_{wait}:

$$\tau_{reset} : \mathcal{S} \to \mathcal{S},$$
$$\tau_{reset}(\vec{S}) = \vec{S}' \mid \begin{cases} \vec{S}'[i] = \vec{S}[i] \otimes \mathbb{0} \text{ if } i = i_\rho \\ \vec{S}'[i] = \vec{S}[i] \text{ otherwise} \end{cases} \quad (8)$$

$$\tau_{wait} : \mathbb{N} \times \mathcal{S} \to \mathcal{S},$$
$$\tau_{wait}(x, \vec{S}) = \vec{S}' \mid \begin{cases} \vec{S}'[i] = \vec{S}[i] \oplus \vec{S}[x] \text{ if } i = i_\rho \\ \vec{S}'[i] = \vec{S}[i] \text{ otherwise} \end{cases}$$
$$(9)$$

τ_{wait} has to be called for each dependency (with index x in the *temporal state* vector) of the current vertex.

- Step 2. Some resources are released at the start of an XG vertex. The corresponding dependencies (e.g. Program Order and Queue Capacity) have to be updated with the start time ρ. This is done with τ_{move}:

$$\tau_{move} : \mathbb{N} \times \mathbb{N} \times \mathcal{S} \to \mathcal{S},$$
$$\tau_{move}(i_{dest}, i_{src}, \vec{S}) = \vec{S}' \mid \begin{cases} \vec{S}'[i] = \vec{S}[i_{src}] \text{ if } i = i_{dest} \\ \vec{S}'[i] = \vec{S}[i] \text{ otherwise} \end{cases}$$
$$(10)$$

τ_{move} copies a vector element into another element and overwrite the destination value. Updating the dependency of single resource turns out to copy ρ into the slot of the dependency: for example updating the start time of a stage $s \in Stages$ with index i_s consists in $\tau_{move}(i_s, i_\rho, \vec{S})$. The state of FIFO resources (like queues) requires to update several *temporal state* slots (i to $i + n - 1$ with n the FIFO capacity) to express the

	Prog. Order	Capacity Order		Pipeline Order	Queue Capacity	
FE	$\rho_{[I_{i-1}/FE]}$	$\rho^*_{[I_{i-1}/FE]}$	$\rho^*_{[I_{i-2}/FE]}$	$-\infty$	$\rho_{[I_{i-1}/DE]}$	$\rho_{[I_{i-2}/DE]}$
DE	$\rho_{[I_{i-1}/DE]}$	$\rho^*_{[I_{i-1}/DE]}$	$\rho^*_{[I_{i-2}/DE]}$	$\rho^*_{[I_i/FE]}$	$\rho_{[I_{i-1}/EX]}$	$\rho_{[I_{i-2}/EX]}$
EX	$\rho_{[I_{i-1}/EX]}$	$\rho^*_{[I_{i-1}/EX]}$	$\rho^*_{[I_{i-2}/EX]}$	$\rho^*_{[I_i/DE]}$	$\rho_{[I_{i-1}/ME]}$	$\rho_{[I_{i-2}/ME]}$
ME	$\rho_{[I_{i-1}/CM]}$	$\rho^*_{[I_{i-1}/ME]}$	$\rho^*_{[I_{i-2}/ME]}$	$\rho^*_{[I_i/EX]}$	$\rho_{[I_{i-1}/CM]}$	$\rho_{[I_{i-2}/CM]}$
CM	$\rho_{[I_{i-1}/ME]}$	$\rho^*_{[I_{i-1}/CM]}$	$\rho^*_{[I_{i-2}/CM]}$	$\rho^*_{[I_i/ME]}$	$-\infty$	$-\infty$

	Fetch Order	Memory Order		Data Dependencies		
FE	$\rho^*_{[I_{fetch}/FE]}$	$-\infty$		$-\infty$		
DE	$-\infty$	$-\infty$		$-\infty$		
EX	$-\infty$	$-\infty$		$\rho^*_{[I_{R0}/s_{R0}]}$	$\rho^*_{[I_{R0}/s_{R1}]}$	$\dots$
ME	$-\infty$	$\rho^*_{[I_{load}/ME]}$	$\rho^*_{[I_{store}/ME]}$	$\rho^*_{[I_{R0}/s_{R0}]}$	$\rho^*_{[I_{R1}/s_{R1}]}$	$\dots$
CM	$-\infty$	$-\infty$		$-\infty$		

TABLE II: The *temporal state*.

shift of the n last FIFO uses. Hence FIFO resources are updated by a series of τ_{move} on the n FIFO slots in the *temporal state* and by setting the first slot to ρ, the use time for the first FIFO element:

$$\forall j \in [i, i+n-2], \tau_{move}(j+1, j, \vec{S});$$
$$\tau_{move}(i, i_\rho, \vec{S});$$

- Step 3. The started instruction spends $\lambda_{[I_i/s]}$ cycles in the stage.

$$\tau_{consume} : \mathbb{N} \times \mathcal{S} \to \mathcal{S},$$
$$\tau_{consume}(\lambda_{[I_i/s]}, \vec{S}) = \vec{S}' \left| \begin{cases} \vec{S}'[i] = \vec{S}[i] \otimes \lambda_{[I_i/s]} & \text{if } i = i_\rho \\ \vec{S}'[i] = \vec{S}[i] & \text{otherwise} \end{cases} \right. \tag{11}$$

- Step 4. The instruction finishes its execution and the dependencies recording the end time of the current vertex are updated. The τ_{move} operation of Step 2 is used.

As in the original XG resolution model, the computational model with *temporal states* has to follow the topological order so that the times recorded in the XDD vector refer to the correct timing of resources. In other words, if the state is correctly updated according to the rules stated above, the resource-based model is equivalent to the original XG analysis but expressed in state machine fashion. The implementation using XDDs extends the model to consider all possible cases according to the timing variations without any loss. The BB analysis is consequently *exact* with respect to the XG pipeline model.

C. The computational model

An important property of XDD domain is that, equipped with $\oplus$ and $\otimes$, it forms the semiring $\langle \text{XDD}, \oplus, \otimes, \mathbb{0}, \mathbb{1} \rangle$ with $\mathbb{0} = LEAF(-\infty)$ and $\mathbb{1} = LEAF(0)$. As the functions τ are affine in this domain, their application can be expressed as matrix multiplications. By combining and pre-computing these matrices, they will help to speed up the pipeline analysis at CFG level as some BBs need to be recomputed several times in different execution contexts.

Scalar and matrix multiplication on XDD semiring is similar to the linear algebra over $\mathbb{R}$ by replacing $+$ by $\oplus$, $\times$ by $\otimes$:

Definition III.2. The scalar multiplication is defined by:

$$\cdot : \text{XDD}^N \times \text{XDD}^N \to \text{XDD},$$
$$[f_0, f_1, ..., f_{N-1}] \cdot [f'_0, f'_1, ..., f'_{N-1}] = \bigoplus_{0 \le i \le N-1} f_i \otimes f'_i$$

Definition III.3. The matrix multiplication is defined by:

$$\cdot : \text{XDD}^{N \times M} \times \text{XDD}^{M \times L} \to \text{XDD}^{N \times L},$$
$$B \cdot C = \left[\quad A_{i,j} \quad \right] | A_{i,j} = \bigoplus_{1 \le k \le M} B_{i,k} \otimes C_{k,j}$$

Definition III.4. The identity matrix Id on XDD semiring is defined by:

$$Id = \left[\quad A_{i,j} \quad \right] | A_{i,j} = \begin{cases} \mathbb{1} & \text{if } i = j \\ \mathbb{0} & \text{otherwise} \end{cases}$$

Notice that, by definition, $\vec{S} \cdot Id = \vec{S}$: any matrix column at index i composed of $\mathbb{0}$ except with a $\mathbb{1}$ in the row i maintains unchanged the value of $\vec{S}[i]$ in the resulting vector. To implement the transitions functions τ as matrix multiplications, the matrix Id is taken as a basis and only the cells that have an effect on the vector have to be changed

1) A $\mathbb{0}$ on the diagonal of the Id matrix at the timer pointer position resets it: $\vec{S}[i_\rho] \otimes \mathbb{0} = \mathbb{0}$:

$$\tau_{reset}(\vec{S}) = \vec{S} \cdot M_{reset}$$
$$= \vec{S} \cdot \left[\quad A_{i,j} \quad \right] | A_{i,j} = \begin{cases} \mathbb{0} & \text{if } i = j = i_\rho \\ Id_{i,j} & \text{otherwise} \end{cases} \tag{12}$$

2) For a given slot at index x in $\vec{S}$, $\tau_{wait}(x, \vec{S})$ is represented by a matrix $M_{wait(x)}$ with a $\mathbb{1}$ at position (x, i_ρ) resulting in the operation $\rho \oplus (\mathbb{1} \otimes \vec{S}[x])$:

$$\tau_{wait}(x, \vec{S}) = \vec{S} \cdot M_{wait(x)}$$

$$= \vec{S} \cdot \begin{bmatrix} & A_{ij} & \end{bmatrix} \Big| A_{ij} = \begin{cases} \mathbb{1} \text{ if } i = i_\rho \wedge j = x \\ Id_{ij} \text{ otherwise} \end{cases} \quad (13)$$

3) $\tau_{move}(i_{src}, i_{dest}, \vec{S})$ is represented by a matrix $M_{move(i_{src}, i_{dest})}$ where the element at (i_{dest}, i_{dest}) is set to $\mathbb{0}$ and the element (i_{dest}, i_{src}) to $\mathbb{1}$ s.t. element i_{dest} in the result becomes $(\mathbb{0} \otimes \vec{S}[i_{dest}]) \oplus (\mathbb{1} \otimes \vec{S}[i_{src}]) = \vec{S}[i_{src}]$.

$$\tau_{move}(i_{src}, i_{dest}, \vec{S}) = \vec{S} \cdot M_{move(i_{src}, i_{dest})}$$

$$= \vec{S} \cdot \begin{bmatrix} & A_{ij} & \end{bmatrix} \Big| A_{ij} = \begin{cases} \mathbb{0} \text{ if } i = j = i_{dest} \\ \mathbb{1} \text{ if } i = i_{src} \wedge j = i_{dest} \\ Id_{i,j} \text{ otherwise} \end{cases}$$
$$(14)$$

4) For a given latency λ, $\tau_{consume}(\lambda, \vec{S})$ can be represented by a matrix $M_{consume(\lambda)}$, obtained from Id by putting λ at position (i_ρ, i_ρ).

$$\tau_{consume}(\lambda, \vec{S}) = \vec{S} \cdot M_{consume}^\lambda$$

$$= \vec{S} \cdot \begin{bmatrix} & A_{ij} & \end{bmatrix} \Big| A_{ij} = \begin{cases} \lambda \text{ if } i = j = i_\rho \\ Id_{i,j} \text{ otherwise} \end{cases} \quad (15)$$

Theorem III.1. *Each applied transition function τ to the timing vector is a linear map from $\mathcal{S}$ to $\mathcal{S}$*

Proof. Direct since we have already given the matrix representation of each transition in Definition III.4. $\qquad \square$

Consequently, the operation performed at each step is also linear because they are combination of τ functions. Their matrix representation is simply the multiplication of each invoked τ. For example,

$$M_{Step_1[I_i/s]} = M_{reset} \cdot \prod_{d \in \mathcal{D}_{[I_i/s]}} M_{wait(i_d)}$$

With $\mathcal{D}_{[I_i/s]}$ the set of dependencies required by $[I_i/s]$ and i_d the index of resource d in the state vector.

Similarly, we can express $M_{Step_2[I_i/s]}, M_{Step_3[I_i/s]}$ and $M_{Step_4[I_i/s]}$ by invoking the corresponding τ functions. As each step is linear, the operation when analyzing one instruction on a stage is also linear because it is the combination of the 4 steps.

$$M_{[I_i/s]} = M_{Step_1[I_i/s]} \cdot M_{Step_2[I_i/s]} \cdot M_{Step_3[I_i/s]} \cdot M_{Step_4[I_i/s]}$$

Finally, the whole analysis of a BB $a \in V$ is composed by the analysis of each instruction on each stage:

$$M_a = \prod_{I_i \in a} \prod_{s \in \mathcal{P}} M_{[I_i/s]}$$

With a matrix as M_a, it is easy and fast to compute the output *temporal state* $\vec{S}' \in \mathcal{S}$ corresponding to an input *temporal state* $\vec{S} \in \mathcal{S}$ for a BB a:

$$\vec{S}' = \vec{S} \cdot M_a \quad (16)$$

IV. PIPELINE ANALYSIS ON THE CFG

This section extends the *temporal state* computational model, presented in the previous section, to the complete analysis of the CFG. It consists, mainly, in tracking the explicit set of possible *temporal states* for each BB all over the CFG execution paths.

A. Computing the context with Rebasing operation

So far, the *temporal state* contains times relative to the start of a BB. As the analysis on CFG starts from the entry point of the program, the recorded times are execution times relative to the start of the program and the *temporal states* have to be tracked for all possible execution paths. This is generally infeasible because of the number of execution paths and especially because of the presence loops. In fact, the main reason to compute exact *temporal states* at CFG level is to determine bus accesses timings but these timings does not need to be absolute with respect to the start of the program. Instead, the times can be relative to different time bases arbitrarily chosen, while, to preserve the soudness of the computation, XDDs with different bases are not mixed. We call this operation *rebasing*.

Rebasing a state is changing the origin of the timeline of the times it contains. For now, the *temporal state* at the end of a BB a represents the delay induced by the execution of a to the start of following BB b. Considering that a new time base $T \in$ XDD is the start of b, we can get a new *temporal state* relative to T by subtracting T from the times in the *temporal state* in the base of a. The outcome is a *temporal state* containing XDDs with positive or negative times relative to T. The relationship between times and events in the *temporal state* is preserved. The subtraction in XDDs $\oslash$ is built in the usual way from $-$ operator (Eq. 4).

Rebasing a *temporal state* is lossless simply because $\oslash$ is reversible. By adding T (with $\oplus$), one can find back the state before rebasing. Rebasing is very helpful to reduce the size of XDDs in the *temporal state*: an event removed by rebasing has no effect on the following BBs but it does not mean it has no effect. In fact, its contribution to the overall WCET is simply linear with respect to the number of occurrences of the BB. Intuitively, the execution of an instruction depends on the execution of nearby instructions and thus, the effect of events is rather short term and it is often eliminated by rebasing.

B. Events Generation within loops

The events calculated by global analyses are linked to a particular instruction. The pipeline analysis of a BB presented so far deems the occurrence of events unique. This is not true when an event arises in a BB contained in a loop as it may occur or not in different iterations. We would get unsound timings if we denotes these different event occurrence with the same event node in the XDD. To fix this, a *generation number* is associated with each event. To prevent *temporal state* blowup, this *generation number* is relative to the current iteration and is incremented in the current *temporal state* each time the analysis restarts the loop. The *generation number* thus

distinguishes the events in different iterations. However, this method does not result in an endless increase of generation because (a) the effect of events is often bound in the time and (b) the WCET calculation requires to bound the loop iterations.

C. The CFG pipeline analysis

Finally, the complete pipeline analysis is designed like a classical data-flow analysis with a work list. Each BB is associated with a set of input *temporal states* and a set of output *temporal states* (initially empty). The analysis starts with an initial *temporal state* at the entry of the CFG and propagates the new states all along the CFG paths. For each entry edge of a BB, the input state set is the union of the output states of the preceding BBs. Each input state is updated by multiplying it with the pre-computed matrix and is rebased to make a new output state. If the set of the output states differs from the original set, the successors of the current BB are pushed into the work list. The process is repeated until finding a fix-point on all sets of input/output states.

This process may be subject to state explosion blow-up caused either by the control flow or by the timing variations i.e. the events. Using XDD, the variability caused by events is efficiently recorded without any loss thanks to its *compaction* property. Besides, the analysis at CFG level collects the set of all possible pipeline states meaning it is also lossless according to the the variability caused by the control-flow. In turn, this means that the resulting set of vectors of XDDs contains sufficient information to determine the exact temporal behavior of each BB in all possible situations.

V. MODELING THE SHARED MEMORY BUS

A frequent design in embedded microprocessors is to have the instruction and data caches sharing a common bus to the memory (or to a shared L2 cache). So far, our pipeline analysis required the target processor to be in-order to ensure a correct evaluation order but a shared memory bus introduces an out-of-order behavior that raises a new difficulty: the variability created by events in the start times of FE and ME stages may change the access order to the shared bus. As XG dependencies are not expressive enough to model out-of-order bus allocations, this section proposes an extension to the pipeline analysis to manage efficiently the shared bus accesses according to the different configurations of the *temporal states*. It supports the usual bus arbitration policy: first-come-first-served, with the priority given to the ME stage in case of synchronous bus accesses.

A. Bus scheduling topology

Since we consider an in-order pipeline, the number of possible contention scenarios on the shared bus is limited. For instance, an instruction using the bus in the ME stage cannot contend with any subsequent instructions in the ME stage (load/store memory order is preserved). In the same fashion, the bus accesses by FE stage are performed following the *Program Order*. Moreover, the *Pipeline Order* ensures that a request emitted by an instruction in the FE stage acquires the

bus before a request emitted by the same instruction in the ME stage. This means that the bus allocation in an in-order pipeline is almost completely in-order, with only one exception: the bus usage in the ME stage by an instruction denoted ME_0 may be delayed by a bus request in the FE stage by a subsequent instruction denoted $FE_{i|i>0}$. To simplify the notation in this section, ME_0 and $FE_{i|i>0}$ denotes as well the instructions as the XG vertices in their respective stage. The instructions in-between are disregarded but are still accounted for in the update matrices for the *temporal states*.

To sum up, FE_i can delay ME_0 only if FE_i is ready to enter FE stage before ME_0 is ready. In the XG model, this situation can only happen when FE_i does not depend on ME_0, that is, when there is no path from ME_0 to FE_i [2].

In the example of Table III, we consider that ME_0 can only be delayed by FE_1, FE_2 and FE_3. For a particular configuration of events, there are four possible schedules that are shown in the first column of the table. These four schedules correspond to the four possible ways to interleave ME_0 with FE_i accesses. The actual schedule is determined by comparing the ready time of ME_0 (ρ_{ME_0}) with the ones of FE_1, FE_2 and FE_3 (resp. ρ_{FE_1}, ρ_{FE_2} and ρ_{FE_3}): the center column shows the condition corresponding to each schedule. The third column gives the actual time at which ME_0 gets the bus with λ_{BUS} denoting the latency to access to the bus (including the memory transaction): if ME_0 is the first to be ready, then it gets the bus at time ρ_{ME_0}. Otherwise, ME_0 gets the bus at the maximum time between its ready time and the release time of the FE_i contender that get the bus before.

B. Batch bus scheduling with XDD

Table III shows the schedule of ME_0 for a fixed configuration. Yet, the times are recorded with XDDs and a particular XDD may support configurations with different schedules. Figure 3 shows how the bus contention scheduling presented in the previous paragraph is extended to XDDs. Let us consider a lightly simpler scenario: ME_0 may be delayed by FE_1 and FE_2. The instruction memory access at FE_1 may experiment instruction cache Hits or Misses represented by event ic_1. The access at FE_2 is classified as Always Miss, meaning that it always requests the bus. The latency of bus access is 9 cycles.

XDD (a) shows the ready time of ME_0 and (b) the initial value of $\hat{\rho}_{ME_0}$, the scheduling time of ME_0 on the bus ($+\infty$ means that no access is yet scheduled). (c) shows the initial value of ρ_{rel}, recording the release time of the bus by FE_i ($-\infty$ denotes that the bus is not used by any FE_i for now).

The ready time of FE_1 (d) is computed from the initial state $\vec{S}_0$ and the matrix between ME_0 and FE_1. The event ic_1 indicates with $-\infty$ the configuration where FE_1 does not use the bus (hence it is not concerned by the contention).

ρ_{ME_0} (a) and ρ_{FE_1} (d) are compared using $\blacktriangleleft_{ME}$ to get the configurations and the time $-\rho_{schedME_0}$ (e) where ME_0 takes the bus, i.e. is scheduled, before FE_1 ($\blacktriangleleft_{ME}$ is formally defined in Eq. 17). Other configurations are assigned $+\infty$ denoting they are not processed yet. Notice that $-\infty$

---

[2]The occurrence of such situations is limited by the size of the inter-stage queues in the pipeline.

Schedule	Condition	Scheduling time of ME_0
$ME_0, FE1, FE2, FE3$	$\rho_{ME_0} \leq \rho_{FE_1}$	ρ_{ME_0}
$FE1, ME_0, FE2, FE3$	$\rho_{FE_1} < \rho_{ME_0} \leq \rho_{FE_2}$	$max(\rho_{FE_1} + \lambda_{BUS}, \rho_{ME_0})$
$FE1, FE2, ME_0, FE3$	$\rho_{FE_2} < \rho_{ME_0} \leq \rho_{FE_3}$	$max(\rho_{FE_2} + \lambda_{BUS}, \rho_{ME_0})$
$FE1, FE2, FE3, ME_0$	$\rho_{FE_3} < \rho_{ME_0}$	$max(\rho_{FE_3} + \lambda_{BUS}, \rho_{ME_0})$

TABLE III: Possible schedules of ME_0 with subsequent FEs.

Fig. 3: Batch bus scheduling with XDDs.

configurations in ρ_{FE_i} does not allow ME_0 to be scheduled as subsequent $FE_{j>i}$ might allocate the bus before ME_0. $\rho_{schedME_0}$ is then used to update $\hat{\rho}_{ME_0}$ using the minimum operator $\ominus$ (f).

$\rho_{schedFE_1}$ (g), the configurations where FE_1 gets the bus is computed in a similar way as $\rho_{schedME_0}$ but with operator $\blacktriangleleft_{FE}$ that selects the configurations of FE with the strict $<$ comparison instead of $\leq$ because ME stage has priority over FE stage. By adding the latency of the bus (λ_{BUS}) to $\rho_{schedFE_1}$, we are able to update, using $\oplus$, the release time of the bus after $FE_1 - \rho_{rel}$ (h). Finally, we compute the actual schedule of $FE_1 - \hat{\rho}_{FE_1}$ (i) which is the time of $\rho_{schedFE_1}$ if FE_1 is scheduled, otherwise the release time of the bus by ME_0 ($\hat{\rho}_{ME_0} \otimes \lambda_{BUS}$). Now, as the actual schedule of FE_1 is known, the release time of the bus at FE_1 is computed and is used to adjust the *temporal state*. By multiplying the state $\vec{S}_{FE_1}$ by the matrix $M_{FE_1-FE_2}$, we get the ready time of FE_2 (j). In the second iteration, first, ρ_{ME_0} (a) is compared with ρ_{FE_2} (j) with the operator $\blacktriangleleft_{ME}$. The actual scheduling time $\rho_{schedME_0}$ (k) is computed by considering the maximum between $\rho_{schedME_0}$ and the release time of the bus by FE_1 (ρ_{rel}) according to the third column of Table III. Then, $\hat{\rho}_{ME_0}$ is updated (l). The schedule of $FE_2 - \rho_{schedFE_2}$ (m) is computed with the operator $\blacktriangleleft_{FE}$ applied to ρ_{FE_2} and ρ_{ME_0} which is then used to update the release time of the bus ρ_{rel} (n). The actual schedule of FE_2 is computed with respect to the use of the bus by ME_0 (o).

When the end of the sequence is reached, there are no further subsequent instructions that may contend with ME_0

and the remaining $+\infty$ in $\hat{\rho}_{ME_0}$ represents configurations accessing the bus after FE_1 and FE_2. They are replaced by the maximum between the ready time of ME_0 and the release times of the bus by FE_1 and FE_2, ρ_{rel}.

Operators $\blacktriangleleft_{ME}$ and $\blacktriangleleft_{FE}$ have a straight-forward definitions setting to $+\infty$ the configurations where ME, respectively FE, does not get the bus:

$$\forall f_{ME}, f_{FE} \in \text{XDD}^2, \forall \gamma \in \Gamma,$$

$$(f_{ME} \blacktriangleleft_{ME} f_{FE})[\gamma] = \begin{cases} f_{ME}[\gamma] & \text{if } f_{ME}[\gamma] \leq f_{FE}[\gamma], \\ +\infty & \text{otherwise} \end{cases}$$

$$(f_{FE} \blacktriangleleft_{FE} f_{ME})[\gamma] = \begin{cases} f_{FE}[\gamma] & \text{if } f_{FE}[\gamma] < f_{ME}[\gamma], \\ +\infty & \text{otherwise} \end{cases}$$

(17)

All these calculations seems a bit complex but it must be kept in mind that real XDDs are much more complex with much more configurations and relying on the XDD operators allows to benefit from the XDDs optimizations.

C. Contention Analysis

The contention analysis depicted in the preceding example is described more formally in this paragraph. Basically, the pipeline analysis is extended by splitting a BB at *contention points*, the XG node where a bus access may occur i.e. ME or FE stages causing cache misses. Then they are grouped in a sequence of one ME access followed by zero or several FE accesses, $(ME_0, FE_{0<i\leq n})$. The instructions between *contention points* are summarized by a pre-computed matrix.

Algorithm 1 is then applied to compute the possible inter-leaving of bus accesses for all configurations of the sequence $(ME_0, FE_{0 < i \leq n})$. Additionally, it takes as input the *temporal state* $\vec{S}_0$. The result is the definitive schedule of $ME_0 - \hat{\rho}_{ME_0}$ and of $FE_i - \hat{\rho}_{FE_i}$.

Algorithm 1: Contention computation.

Input: $\vec{S}_0 \in \mathcal{S}$, $(ME_0, FE_{1 \leq i \leq n})$
Output: $(\hat{\rho}_{ME_0}, \hat{\rho}_{FE_{1 \leq i \leq n}})$

1   $\hat{\rho}_{ME_0} = \text{LEAF}(+\infty)$
2   $\rho_{rel} := \text{LEAF}(-\infty)$
3   $\vec{S}_{FE_1} := \vec{S}_{ME_0} \cdot M_{ME_0 - FE1}$
4   $i := 1;$
5   $\rho_{ME_0} := \vec{S}_0[i_\rho]$
6   **while** $i \leq n \wedge (\exists \gamma \in \Gamma \wedge \hat{\rho}_{ME_0}[\gamma] = +\infty)$ **do**
7   **if** $FE_i.mustUseBus()$ **then**
8         $\rho_{FE_i} := \vec{S}_{FE_i}[i_{FE}]$
9   **else**
10        $\rho_{FE_i} := \vec{S}_{FE_i}[i_{FE}] \otimes \text{NODE}(ic_i, -\infty, 0)$
11      $\rho_{schedME_0} := (\rho_{ME_0} \blacktriangleleft_{ME} \rho_{FE_i}) \oplus \rho_{rel}$
12      $\hat{\rho}_{ME_0} := \hat{\rho}_{ME_0} \ominus \rho_{sched}$
13      $\rho_{schedFE_i} := \rho_{FE_i} \blacktriangleleft_{FE} \rho_{ME_0}$
14      $\rho_{rel} := \rho_{rel} \oplus (\rho_{schedFE_i} \otimes \lambda_{BUS})$
15      $\hat{\rho}_{FE_i} := \rho_{schedFE_i} \ominus (\hat{\rho}_{ME_0} \otimes \lambda_{ME_0})$
16      $\vec{S}_{FE_{i+1}} :=$
        $(\vec{S}_{FE_i} \oplus [\mathbb{0}, ..., \mathbb{0}, \hat{\rho}_{FE_i} \otimes \lambda_{BUS}]) \cdot M_{FE_i - FE_{i+1}}$
17      $i = i + 1$
18   $\hat{\rho}_{ME_0} := \hat{\rho}_{ME_0} \ominus (\rho_{rel} \oplus \rho_{ME_0})$

Initially, ME_0 is considered as not scheduled whatever the considered configuration and $\hat{\rho}_{ME_0}$ is set to $LEAF(+\infty)$ (line 1). It will then be updated after considering the contention with each subsequent FE_i. When ME_0 does not contain $+\infty$ anymore or when all FE_i has been processed, ME_0 schedule is complete (condition at line 6). Line 2 initializes ρ_{rel} that records the release time of the bus by FE_i to $-\infty$ as no FE_i has been processed yet.

In line 3, the *temporal state* just before FE_1 is computed by applying the matrix $M_{ME_0 - FE_1}$ to the initial state $\vec{S}_0$; i is initialized in line 4 and will range over the *Contention Points*, 1 to n. The ready time of ME_0 is recorded into ρ_{ME_0} at line 5. Lines 7-10 compute the ready time of FE_i if the access results always or sometimes in a Miss (according to $mustUseBus()$). The latter case is expressed by the event ic_i and by adding the $\text{NODE}(ic_i, -\infty, 0)$ to ρ_{FE_i}: $-\infty$ denotes the case where ic_i does not arise and there is no bus access.

$\rho_{schedME_0}$, ME_0 configurations getting the bus before FE_i, is computed with $\blacktriangleleft_{ME}$ at line 11 by comparing the ready time of ME_0 with the ones of FE_i. According to the last column of Table III, these configurations are fixed by taking the maximum between the ready time of ME_0 and the release time of the bus ρ_{rel}. The schedule of ME_0 at this iteration is accumulated in the definitive schedule of ME_0 at line 12. At line 13, the schedule of FE_i is computed.

Notice that as the ready time of FE_i contains $-\infty$ to denote the case where it does not use the bus, these $-\infty$ are kept in $\rho_{shcedFE_i}$. By adding the bus latency λ_{BUS} to $\rho_{schedFE_i}$ and then $\oplus$ with ρ_{rel}, the release time of the bus is only updated for configurations γ where FE_i uses and gets the bus $- \rho_{schedFE_i}[\gamma] \neq +\infty$ (line 14). Notice that the $+\infty$ in ρ_{rel} cannot overwrite the release time of the bus by FE_i because FE_i cannot get the bus if any prior $FE_{j < i}$ does not get the bus. At line 15, the actual schedule of FE_i is computed by replacing the $+\infty$ in $\rho_{schedFE_i}$ (where FE_i loses contention in favor of ME_0) by the release time of the bus by ME_0. Configurations where time is $+\infty$ in $\rho_{schedFE_i}$ must not be $+\infty$ in $\hat{\rho}_{ME_0}$ because only one of both FE_i or ME_0 is scheduled. However, as $\rho_{schedFE_i}$ configurations different from $+\infty$ are lower than $\hat{\rho}_{ME_0}$ (otherwise it is considered as non-scheduled), $\ominus$ can be used to implement the replacement.

At line 16, the *temporal state* is updated regarding the schedules of FE_i, by applying $\oplus$ between the time pointer of the state vector and the release time of the bus by FE_i. The updated state is then multiplied by matrix $M_{FE_i - FE_{i+1}}$ to obtain the ready time of FE_{i+1}. Line 18 takes into account the $+\infty$ configurations remaining in $\hat{\rho}_{ME_0}$ that are not already scheduled by the loop. The times assigned to these configurations are the maximum between the ready time of ME_0 and the bus release time by FE_i. Notice that $+\infty$ in the $\hat{\rho}_{ME_0}$ may also be caused by the fact that none of FE_i have used the bus: this time is recorded as $-\infty$ in ρ_{rel} and is hence automatically overwritten by the ready time of ME_0.

VI. EXPERIMENTS

The performance of the analysis strongly depends on the size of the XDDs in the pipeline states and the number of pipeline states. Both characteristics are related to some inherent properties of the analyzed program and of the micro-architecture whose impact is difficult to estimate. Therefore, we experiment our analysis on realistic benchmarks that empirically provides a better understanding of the performances.

A. Experiment Setup

The pipeline used in the examples of the previous sections was chosen to improve the readability of the article. For the experimentation, we prefer a more powerful micro-architecture with more parallelism leading to more complex *temporal states*. In addition, this new pipeline allows to demonstrate the scalability of our approach.

The experimentation pipeline has 4 stages able to process 4 instructions per cycle: FE, DE, EX, CM. It fetches instructions from the main memory in the FE stage via a single level instruction cache. The FE stage is able to fetch simultaneously 4 instructions of the same memory block with a latency of 7 cycles for miss (without considering contention). The DE stage decodes the instructions and the EX stage handles all arithmetic, floating point and memory related operations in several Functional Units (FU). 4 ALUs (Arithmetic and Logic Units) are available and can be simultaneously used if no data dependencies are present. The latency of arithmetic operations is 1 cycle for addition and subtraction; 2 cycles

for multiplication and 7 cycles for division. 1 FPU (Floating Point Unit) is available with latencies of 3 cycles for addition and substraction, 5 cycles for multiplication and 12 cycles for division. One MU (Memory Unit) is also available to handle memory related operations (load and store). In case of a multiple load/store operations, the memory accesses are performed in order, and if one load/store needs to use the bus, it occupies the bus until all loads/stores are completed. The latency of memory accesses is the same as for FE stage. An issue buffer at EX stage distributes the instruction to EX FUs with respect to the operations realized by the instruction. Instructions using the same FU are executed in-order in EX stage; instructions using different FUs are executed out-of-order (if no data dependencies exist).

The instruction cache is a 16 KBits 2-way set associative LRU (Least Recent Used) cache. The data cache is a 8 KBits 2-way set associative LRU cache. Both caches have only one level and share the same bus to access the main memory. We think that this architecture is representative of mid-range processors used in real-time embedded systems.

The whole CFG analysis is implemented using the OTAWA toolbox [15]. Global analyses, including instruction and data cache analyses, control flow analyses etc. are provided by OTAWA. The benchmarks are taken from the TACLe suite [16] compiled for armv7 instruction set with hard floating point unit. Among 83 tasks to be analyzed, 7 of them[3] failed due to limitations in OTAWA.

B. Number of Temporal States

The first experiment explores the number of *temporal states* along the edges of BBs over all benchmarks (representing the output of the source BBs and the input of sink BBs). The experimental results are shown in Figure 4. The x-axis is the number of pipeline states, the y-axis is in logarithm scale and shows the number of edges for each quantity of states. The displayed statistics accumulate data from all TACLe's benchmarks. The risk, with our approach, is to face to a blowup in the number of states. Fortunately, the experimentation shows that most of the edges have less than 20 output states except in some rare cases where the number of states is much higher. This generally means that most of timing variations due to events are efficiently represented in the XDDs of the *temporal states*. As expected, the XDDs successfully prevent the state explosion and keep the pipeline analysis tractable at CFG level. The presence of some rare cases that have a lot of states is not blocking as the analysis time is reasonable in most of cases (confer Section VI-D).

C. Events Lifetime

The second experiment measures the lifetime of events during the analysis. The longer the lifetime of events, the larger the complexity of the analysis in terms of state number and XDD size. In our micro-architecture, an event is created by a cache access and may disappear from the XDD during the analysis, for two reasons. (a) It is absorbed by the pipeline: for example, when an instruction stalls at EX stage due

[3]$pm, recursion, quicksort, huff\_enc, mpeg2, gsm\_enc, ammunition$

Fig. 4: The distribution of number of pipeline states.

to a data cache miss, next instructions may go on in the pipeline completely hiding the stalling time. This event will only stay alive in a short time window during the analysis of other instructions executed in parallel. (b) The events are *stabilized* and disappear thanks to the rebasing operation. Intuitively, we assume that in most situations the events raised by an instruction only impact nearby instructions. This is demonstrated by tracking the liveness of events in the analysis.

However, the pipeline analysis is only able to provide this information at the granularity of *Contention Points* because the instruction execution effect between *Contention Points* is summed up by matrices. As collecting these statistics at a finer granularity would have an important adverse effect on the analysis time, we survey the liveness of events on this basis. Events are deemed as dead at *Contention Points* whose *temporal states* does not contain the event regarding all XDDs contained in the vector. Thus, the lifetime statistics are over-estimated by the number of instructions between *Contention Points*. Besides, the pipeline states are only rebased at the end of BBs so the lifetime of events in the middle of BBs does not consider the potential death due to rebasing. In the end, the measured lifetime in this experimentation is an over-estimation of the actual lifetime of events.

Figure 5 shows the experimental results. The x-axis is the lifetime of events (in instructions with limitations described above) and y-axis, in logarithm scale, shows the number of events having this lifetime. These are also accumulated from the whole set of TACLe's benchmarks. The statistics show that most of events have short lifetime (below 50 instructions). We have observed a unique lifetime of 602 instructions that is not represented to keep the figure readable. It turns out that in most of situation where the lifetime is greater than 50, the events are in a BB of a big number of instructions. In the extreme case with 602-instruction event lifetime, the involved BB is made of 617 instructions (in benchmark *md5*) and the reported lifetime is an effect of the granularity level. Despite this very infrequent case, as most of events have short lifetime, the *temporal state* size (sum of XDDs sizes of the vector) stays reasonable and the analysis remains efficient.

D. Analysis time

The analysis time includes the time to pre-compute the matrices and the time of the pipeline analysis on the CFGs. The measurement is performed on a virtual machine running

Fig. 5: The distribution of events lifetime.

on a cloud server with 8GB RAM and 4-core Intel Broadwell processors. Only 2 cores are occupied simultaneously to run the benchmarks. We have also measured the analysis time without pre-computing the matrices (with a timeout of 1 hour) in order to clarify the advantage of that optimization. The results are shown in Figure 6. The x-axis shows the benchmarks and the y-axis provides the analysis time in seconds with a logarithmic scale. The analysis time with matrices is recorded as green bars. At worst, the analysis with matrices finishes in 553s (9m13s). In most cases, the analysis finishes in about $1 - 20s$. In contrast, the analysis without matrix has both memory usage and speed issues as shown by the red (crashed because out of 8G RAM) and yellow bars (timeout after 1 hour). For those finishing within 1 hour (blue bars), matrix optimization brings 217% speed-up in average[4]. The rare cases where the analysis without matrix is faster are simple benchmarks where the cost of computing the matrices is not compensated by the speed-up. This result reveals that the pre-computation of matrices effectively reduces intermediate redundant computations, which enhances the analysis performance in terms of speed and memory usage. Considering the presence of some exceptional cases, large number of *temporal states*, long lifetime events, we think that the analysis is still able to handle them and finishes in a reasonable time. Moreover, we observed that the industry-like applications encompassed in TACLe benchmarks – Rosace, Debbie and Papabench, are analyzed in short times – respectively 15s, 10mn, 15mn summing the times of all tasks composing them. Therefore, our approach could be used in industrial real-time applications (for example, Airbus requires at most 48H between the detection of a bug and the distribution of the fix, including temporal verification).

VII. RELATED WORKS

The pipeline model used in the aiT WCET analyzer is maybe the most successful and the most used pipeline analysis [17], [18]. They define the state by the time left in each resource of the pipeline and update it at the granularity of the processor cycle. Based on Abstract Interpretation framework [19], according to our knowledge, they use power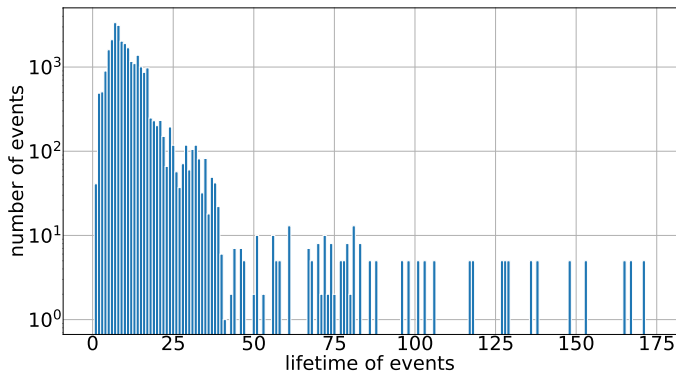 set domain to keep the set of possible pipeline states. Therefore, this model also suffers from combinatorial complexity caused by

---

[4]average speed up = sum of the time of all benchmarks without matrix divided by sum of time with matrix.

the presence of timing anomalies as it has to keep all possibles states. They have proposed several approaches to reduce the complexity. (a) Although the literature provides very few details about that, close states seem to be joined to form abstract states at the cost of loss of precision, (b) in [20], they show how to use Binary Decision Diagram to compress the state machine representation of their analysis system. Reineke et al. in [21] defines sufficient condition to drop *not-worst* cases in order to reduce the number of states. This work has been extended later in [22], [23] that provides a theoretical basis to design strict-in-order pipeline where timing anomalies are proven to not occur [6], [24], thus allowing to more easily drop non-worst case states. However, for now, the price to pay for the design of timing-anomaly-free pipelines is still a significant loss of performance.

Model checking can also be used for WCET analyses [25], [26]. With the help of mature theories and tools of model checking, the solving procedure itself is well optimized and is independent of the timed model of the target program and the micro-architecture which eases the design of modular and flexible analyses [27], [28]. However, in general, model checking-based analyses have to completely explore the domain of traces of the program and the domain of program inputs. On the one hand, this provides tight WCETs without over-estimation, and precise information about the worst execution pattern. On the other hand, the large search domain combined with the micro-architecture model complexity questions the scalability of these analyses for complex program and architectures.

Another approach to pipeline analysis is the *Execution Graph* proposed by Li et al [10], close to what is presented in Section II-A. They analyze the WCET at the scope of BBs and calculate the worst execution context. To support timing variation, the XG computational model uses intervals to representing minimum and maximum times. The contention between instructions is considered by checking the intersection of time intervals. If a contention occurs, the interval is extended accordingly. The XG solving algorithm repeats the computation until a fix-point is reached. However, in the presence of lots of events the interval representation tends to trigger a chain reaction: the imprecision due to the interval representation create contentions that are actually impossible which extends the interval and involves more impossible contentions. Moreover, with respect to the micro-architecture, making precise assumption on the worst execution context is not always trivial. Another XG based approach is proposed by Rochange et al. in [11] that computes the execution time of BBs for each combination of events what makes the algorithm to tend toward combinatorial complexity. In addition, the contention analysis requires to examine all cases leading to an exponential complexity.

VIII. CONCLUSION

In this paper, we have formally defined a state representation useful for the pipeline time analysis. It is derived from the XG model but the times are replaced by XDDs that efficiently represent time variation caused by the raise of events. An XDD is a data structure working as a lossless compression of a map between the event configurations and the execution time.

Fig. 6: The analysis time.

XDD's advantage is its ability to compact the time variation compared to the use of power set map. Moreover, we represent the pipeline state as a vector of XDDs. This simplifies the design of pipeline analysis at CFG level and allows to leverage the algebraic properties of XDDs to represent the analysis of instruction sequences as matrices multiplications. These matrices can be statically determined before the analysis which significantly speeds up the analysis. Together with rebasing and generation number, the presented analysis enables the tracking of exact timing behavior over the CFG of program.

Secondly, we extend this analysis to support the shared bus between fetch and memory pipeline stages. The shared bus is dynamically allocated and may experiment out-of-order access. *Temporal states* obtained so far are used to track precisely the bus accesses schedule. Based on the survey of the topology of the bus usage by instructions, we designed a contention analysis to support bus access times in the *temporal states*. The contention analysis needs only to be invoked upon *contention points* while instructions in-between are summarized by the aforementioned matrices.

The experimentation has been conducted on TACLe's benchmarks. The measurement of the number of pipeline states per edge in the CFG showed that our approach is able to efficiently represent the timing variations. Then, we produce a rough (but conservative) evaluation of the event lifetime that shows that the effect of events are generally short term in the CFG. Some exceptions are observed (edges with a lot of states or events with long lifetime), but they are not problematic as they are very infrequent. The analysis time shows that our analysis is very efficient and suggests that it could be used for industrial applications.

As future works, we could benefit from the exact tracking of the *temporal states* to more precisely qualify the effects of timing variations in different micro-architectures. This could be used to eventually qualify good or bad micro-architecture design in terms of predictability. This may help to find better compromises in the design of predictable pipelines, which could also alleviate over-stringent constraints on the pipeline such as strict-in-order execution, that often limits the performance of the processor. We plan to extend our approach to all out-of-order resources. Although our operators and matrices calculation are correct whatever the out-of-order resource, the modeling of the interleaving of resource acquisitions might not scale.

REFERENCES

[1] Z. Bai, H. Cassé, M. De Michiel, T. Carle, and C. Rochange, "Improving the Performance of WCET Analysis in the Presence of Variable Latencies," in *ACM SIGPLAN/SIGBED LCTES*, 2020.

[2] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *IEEE RTSS*, 1999.

[3] J. Eisinger, I. Polian, B. Becker, S. Thesing, R. Wilhelm, and A. Metzner, "Automatic Identification of Timing Anomalies for Cycle-Accurate Worst-Case Execution Time Analysis," in *IEEE DDECS*, 2006.

[4] G. Gebhard, "Timing anomalies reloaded," in *WCET 2010*.

[5] F. Cassez, R. R. Hansen, and M. C. Olesen, "What is a timing anomaly?" in *WCET*, 2012.

[6] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder, "Principles of timing anomalies in superscalar processors," in *QSIC'05*.

[7] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A Definition and Classification of Timing Anomalies," in *WCET'06*.

[8] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm, "Static Timing Analysis for Hard Real-Time Systems," in *VMCAI*, 2010.

[9] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *ACM SIGPLAN LCTES*, 1995.

[10] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for software timing analysis," in *IEEE RTSS*, 2004.

[11] C. Rochange and P. Sainrat, "A Context-Parameterized Model for Static Analysis of Execution Times," in *HIPEAC II*, ser. LNCS, 2009.

[12] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on computers*, vol. 27, no. 06, pp. 509–516, 1978.

[13] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, 1992.

[14] M. Fujita, P. C. McGeer, and J.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *Formal methods in system design*, vol. 10, no. 2, pp. 149–169, 1997.

[15] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: An open toolbox for adaptive wcet analysis," in *IFIP SEUS*, 2010.

[16] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "Taclebench: A benchmark collection to support worst-case execution time research," in *WCET*, 2016.

[17] J. Schneider and C. Ferdinand, "Pipeline behavior prediction for super-scalar processors by abstract interpretation," *ACM SIGPLAN Notices*, vol. 34, no. 7, pp. 35–44, May 1999.

[18] S. Thesing, "Safe and precise wcet determination by abstract interpretation of pipeline models," Ph.D. dissertation, Univ. Saarland, 2004.

[19] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *4th ACM SIGACT-SIGPLAN symposium on PLDI*, 1977.

[20] S. Wilhelm, "Symbolic representations in WCET analysis," Ph.D. dissertation, Saarland University, 2012.

[21] J. Reineke and R. Sen, "Sound and Efficient WCET Analysis in the Presence of Timing Anomalies," in *WCET'09*, 2009.

[22] S. Hahn, J. Reineke, and R. Wilhelm, "Toward Compact Abstractions for Processor Pipelines," ser. LNCS, 2015, vol. 9360, pp. 205–220.

[23] S. Hahn and J. Reineke, "Design and analysis of SIC: a provably timing-predictable pipelined processor core," *Real-Time Systems*, vol. 56, 2020.

[24] J. Engblom and B. Jonsson, "Processor pipelines and their properties for static WCET analysis," in *EMSOFT*, ser. LNCS, 2002.

[25] F. Cassez, "Timed games for computing wcet for pipelined processors with caches," in *IEEE ACSD*, 2011.

[26] R. Metta, M. Becker, P. Bokil, S. Chakraborty, and R. Venkatesh, "Tic: a scalable model checking based approach to wcet estimation," *ACM SIGPLAN Notices*, vol. 51, no. 5, pp. 72–81, 2016.

[27] F. Cassez and J.-L. Béchennec, "Timing analysis of binary programs with uppaal," in *IEEE ACSD*, 2013.

[28] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen, "Metamoc: Modular execution time analysis using model checking," in *WCET*, 2010.