Computing Execution Times With Execution Decision Diagrams in the Presence of Out-of-Order Resources

Zhenyu Bai, Hugues Cassé, Thomas Carle, and Christine Rochange

Abstract—We propose a precise and efficient pipeline analysis to tackle the problem of out-of-order resources in modern embedded microprocessors for the computation of the worst-case execution time (WCET). Such resources are prone to timing anomalies (Reineke et al., 2006). To remain sound, the timing analysis must either rely on huge timing over-estimations or consider all possible pipeline states which usually leads to a combinatorial blowup. To cope with this situation, we build an efficient computational model by leveraging the algebraic properties of the execution decision diagram (Bai et al., 2020) which is able to track precisely all pipeline states all along the execution paths of the analyzed program while keeping the analysis time within acceptable range. We show how to apply this analysis at the control flow graph (CFG) level, and how to account for a typical out-of-order resource: the shared memory bus between the instruction and data caches. We observe a gain in precision of the WCET ranging from 20% to 80% compared to the state-of-the-art pipeline analysis of the OTAWA WCET toolset. The analysis time shows that our approach scales to realistic benchmarks, making it appropriate for industrial applications.

Index Terms—Pipeline, real time, static analysis, worst-case execution time (WCET).

I. INTRODUCTION

The correctness of hard real-time systems depends not only on their functional behavior but also on their 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. The presence, in modern microarchitectures, of pipelined and superscalar execution and other mechanisms to favor instruction-level parallelism and microarchitectures, of pipelined and superscalar execution and whether or not it contributes to the global WCET is not determined [1], [7]. Unless the target processor is proven to be free of timing anomalies, a safe and precise WCET analysis has to capture them by precisely tracking the execution states of the microarchitecture.

The WCET computation is generally broken down into three parts [8]. First, global analyses, independent of the pipeline’s structure, are performed: they typically encompass cache and branch-prediction analyses, which determine the behavior of these mechanisms at instruction level. Second, the pipeline analysis uses the information provided by global analyses to determine how instructions are executed through the pipeline, and to compute the (worst-case) execution times of basic blocks (BBs). Finally, an ILP system is built to maximize the execution time considering the individual WCETs of BBs computed in the previous step, with constraints expressing the structure of the control flow graph (CFG) and flow facts, such as loop bounds. The solution of the ILP system is the WCET of the whole program. This method is called implicit path enumeration technique (IPET) [9].

The most important bottleneck in the classical structure of WCET analysis is the lack of efficient abstraction of pipeline. Existing approaches either tend to be not scalable or imprecise. However, our previous approach [2] shows that the execution decision diagram (XDD) is a good data structure to record times within the pipeline analysis that improves its performance. An XDD can be deemed as a lossless compression of the relationship between the execution time and the combination of timing variations. By implanting XDD into the execution graph (XG) pipeline model, we achieved exact and efficient pipeline analysis on BBs through in-order pipelines. In [2], the pipeline analysis is designed to consider BBs of a program independently by calculating their worst-case execution context. However, with out-of-order accesses to resources, such as shared buses, the conservative use of a worst-case context does not hold anymore. We need to precisely track the possible execution contexts in order to evaluate how the concurrent accesses to the bus are interleaved. The pipeline analysis has to analyze the microarchitecture states on the whole program, that is, at CFG level.

1) Contributions: This article presents: a) how we adapt the original graph-based pipeline model proposed in [2] into a data-flow analysis applied at CFG level that computes exactly all the possible temporal pipeline states; b) how we construct an efficient computational model of our analysis by leveraging the algebraic properties of XDD,
which turns the state transition into matrix multiplication on a semi-ring; c) how we exploit the precise pipeline states produced by this computational model to support a typical out-of-order resource: the memory bus that is shared by the instruction and data caches to access the main memory; and d) the evaluation of our model on realistic applications (TACLe’s benchmarks [10]) that shows its scalability. Several parameters are measured during the experiment to give some explanation of its good performance. The precision gain of our approach is compared with the state-of-the-art pipeline analysis of OTAWA [11] and shows improvement of the final WCET ranging from 20% to 80%.

2) Outline: First, the related works are discussed in Section II. Section III provides background information on the XG model and the XDD structure. In Section IV, we extend the original model of XG with XDD to a resource-based model which is able to express the state of the pipeline with a vector. Later in this section, we show how to leverage the algebraic properties of XDD in order to improve the performance of the analysis by precomputing the calculus as matrices. In Section V, we show how to build the complete analysis at CFG level. Section VI extends our model to support the shared memory bus. Experiments reported in Section VII demonstrate the efficiency and the precision of our analysis on realistic benchmarks. Several metrics are considered and discussed, and we conclude in Section VIII.

II. RELATED WORK

The most complex part of the classical structure of WCET analysis (global analyses + pipeline analysis + IPET) is the pipeline analysis because of the presence of timing anomalies and the complexity of modern microarchitectures. We may classify the state-of-the-art pipeline analyses into two classes: those at CFG level and those at BB level.

The aiT analyzer is one of the most successful WCET analysis tools and it performs pipeline analysis at CFG level [12], [13]. Concretely, it builds an abstract model of the pipeline: this model describes the transition of pipeline states at the granularity of processor cycles. Its pipeline analysis, based on the abstract interpretation framework [14], computes an overset of the possible pipeline states over the CFG of the program. To the best of our knowledge, it uses power set domain to keep the set of possible pipeline states. Therefore, this model suffers from combinatorial complexity caused by the presence of timing anomalies as it has to keep all possible states. Several approaches were proposed to reduce the complexity: 1) although the literature provides very few details about that, close states seem to be joined to form abstract states at the cost of a loss of precision; 2) Wilhelm [15] showed how to use a binary decision diagram (BDD) to compress the state machine representation of their analysis system; and 3) Reineke and Sen [16] defined a sufficient condition to drop not-worst cases in order to reduce the number of states. This work has been extended later in [17] and [18], providing a theoretical basis to design strictly in-order pipeline where timing anomalies are proven to not occur [7], [19], thus, allowing to more easily drop nonworst case states.

Another approach to pipeline analysis is to compute locally the WCET per BB. The XG proposed by Li et al. [20] (chronos analyzer) is a very representative example. Instead of computing the pipeline state over the CFG, they make assumptions about the worst execution context of each BB. To avoid using powerset domain to keep the pipeline states, they use interval domain to keep the minimal and maximal execution times of instructions. 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 creates contentions that are actually impossible, which extends the interval and involves more impossible contentions—hence, introducing considerable overestimation. Moreover, with respect to the microarchitecture, making precise assumptions on the worst execution context is not always simple. Another XG-based approach is proposed by Rochange and Sainrat [21] that computes the execution time of BBs for each combination of events, which makes the algorithm tending toward combinatorial complexity. In addition, the contention analysis requires examining all cases, leading to an exponential complexity.

Another different direction of static WCET analysis uses model checking approaches. The principle of these approaches is to express the analyzed program and the complete microarchitecture (including cache, pipeline, and all timing related components) by a timed automaton without abstraction. The WCET is computed by exploring all possible states of the automaton. Since all states are reached, it is compatible with timing anomalies by nature. These approaches usually provide exact WCETs and are able to give the longest path, which may help the developers to further optimize the WCET. On the other hand, they generally have scalability issues when the microarchitecture becomes complex (this results in a larger pipeline state domain) or when the program under analysis becomes complex (this results in an intractable amount of execution traces to explore). One of the most well known tools in this category is UPPAAL [22]. To the best of our knowledge, the best experiment using UPPAAL, reported in [23], is not able to analyze all the benchmarks of the Mälardalen suite, but only the simplest ones. Some hypotheses are also made to turn some programs into single-path programs, which greatly reduces the complexity of the problem.

Lim et al. [24] computed the WCET of BBs with reservation tables that simulate the reservation of resources in the pipeline at each cycle. The worst reservation, hence, gives the worst execution time. In [25], the worst behavior of the pipeline, caches and branch predictors are obtained by simulations at BB level. In [26], the worst timing contributions of different components (cache, pipeline, and others) are computed separately and then safely composed using rules defined in this article. Engblom and Jonsson [19] bound the distance of temporal effects between instructions, so as to compute the WCET per instruction with a safe over-estimation.

Our Analysis: The pipeline model proposed in this article is inspired from a key idea of the XG approach: it uses dependencies to describe the temporal behavior of the pipeline, which allows the analysis to work at pipeline stage level
(unlike aiT that works at the granularity of processor cycles). Our strategy improves the efficiency of the analysis because one dependency often spans over several processor cycles, which leads to a reduced number of applied transitions and a more compact dependency representation.

Compared to the original XG approach, this new model considers the effect of instructions of other BBs on the timing of the analyzed BB (instead of considering each BB in isolation), and it uses XDDs (instead of the interval domain in the original XG, or the powerset domain in the case of aiT) to efficiently represent all possible pipeline states without loss of precision. Theoretically, this approach is, thus, more precise than the original XG approach, and more efficient than the approach used by aiT when tracking explicitly all possible pipeline states.

III. BACKGROUND

As a program under analysis usually 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 article models the whole program as a CFG, then computes the WCET of each BB and determines the WCET using the IPET method. Thus, the pipeline analysis aims at determining the execution time of each BB, for example, using XGs.

A. Execution Graphs

An XG [20], [21] models the temporal behavior of an instruction sequence (such as a BB) executed in the pipeline. The key idea of the 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 has to leave a pipeline stage before starting its execution in the next stage, an instruction has to read a register after a prior 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 machine instructions, and let $G_{XG}$ be a directed acyclic graph (DAG) $G_{XG} = \langle V_{XG}, E_{XG} \rangle$ built for an instruction sequence $I^* \cup \exists$ such that:

1) $V_{XG}$ is the set of vertices defined by $V_{XG} = \{I_i/s \mid I_i \in I^* \land s \in \mathcal{I}\}$, with $\mathcal{I}$ the set of pipeline stages;

2) $E_{XG} \subseteq V_{XG} \times V_{XG}$, the set of edges, is built according to the dependencies in the considered pipeline.

In addition, an XG is decorated with temporal information.

1) $\lambda_v \in \mathbb{N}$ is the latency of vertex $v$, that is, the time spent in this vertex.

2) $\delta_{v \rightarrow w} \in \{0, 1\}$ represents the kind of dependency expressed by edge $v \rightarrow w \in E_{XG}$. If $\delta_{v \rightarrow w} = 1$ (solid), $w$ starts after the end of $v$; if $\delta_{v \rightarrow w} = 0$ (dotted), $w$ can start at the same time as or after the start of $v$.

Examples in this article consider a five stage (FE—fetch, DE—decode, EX—execute, ME—memory, WB—write back) in-order 2-way superscalar pipeline but the presented algorithms are not limited to this configuration. Fig. 1 shows the XG for this pipeline and for the sequence of instructions listed on the left. The vertices correspond to the use of a pipeline stage (column headers) by an instruction (row headers). The edges reflect the following dependencies.

1) The horizontal solid edges model the Pipeline Order: an instruction goes through pipeline stages in a given order.

2) The vertical dotted edges model the parallel execution of instructions in the superscalar stages (Program Order).

3) The vertical solid bent edges model the width of the stages—two instructions per cycle (Capacity Order).

4) The slanted dotted edges model the capacity of FIFO queues (two instructions) between stages (Queue Capacity).

5) 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 is typical for inorder 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 $\rho_w$ is computed as the earliest time at which all incoming dependencies are satisfied and the end time $\rho_w^*$ as $\rho_w$ increased by the time passed by the instruction in the stage

$$
\rho_w = \max_{v \rightarrow w \in E_{XG}} \rho_v + \delta_{v \rightarrow w} \times \lambda_v
$$

$$
\rho_w^* = \rho_w + \lambda_w.
$$

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 the program order), the timing of an instruction only depends on prior instructions, meaning that, at least, one topological order exists. For 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 Fig. 1 by the light gray arrow in the background.

The computation of an XG is fast and efficient. However, as soon as the pipeline produces variable latencies (e.g., to reflect a possible cache hit or miss), precision is achieved at the cost of computing the XG for each combination of these latencies. This may engender a computation complexity explosion. The data structure presented in the next section alleviates this issue.

B. Execution Decision Diagrams

The XDD is inspired from the BDD [27], [28] and its multiterminal BDD (MTBDD) variant [29]. An XDD is a DAG that is recursively defined as Definition 1.

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
Definition 1:

\[
\text{XDD} = \text{LEAF}(k) \cup \text{NODE}(e, f, \gamma).
\]  

The Boolean variables \( e \in \mathcal{E} \) in nodes are called events: they model the uncertainty in the analysis regarding the microarchitecture state and its impact on the time, e.g., whether a particular cache access results in a hit or in a miss. The subtrees \( f, \gamma \in \text{XDD} \) represent, respectively, the situations where the event \( e \) happens or not. The leaves of an XDD store the execution times \( k \in \mathbb{Z}^\# = \mathbb{Z} \cup \{+\infty, -\infty\} \).

As in ordered BDDs [28], XDDs deploy hash consing techniques to guarantee the unicity of the subtree instances and speed up the calculation: identical subtrees share the same instance in memory. This compression allows the XDDs to represent efficiently the relationships between combinations of events (called configurations) and the corresponding execution times. A configuration \( \gamma \in \Gamma \) is the combination of activation or inactivation of events (\( \Gamma = \phi(\mathcal{E}) \)) and corresponds to a path from the root node to a leaf in the XDD DAG.

When events are taken into account, the actual time for each configuration in the resulting XDD is exactly the time corresponding to configuration \( \gamma \) with \( \alpha(\gamma) \) the morphism from \( \Gamma \rightarrow \mathbb{Z}^\# \) to \( \mathbb{Z}^\# \) such that performing the operation in \( \text{XDD} \) and \( \Gamma \rightarrow \mathbb{Z}^\# \) exists: \( \text{XDD} \stackrel{\beta}{\cong} (\Gamma \rightarrow \mathbb{Z}^\#) \). This means that XDD can be deemed as a lossless compression of the map \( (\Gamma \rightarrow \mathbb{Z}^\#) \). In the remainder of this article, we note \( f(\gamma) \) the time corresponding to configuration \( \gamma \) in \( f \) (for \( f \in \text{XDD} \) or \( f \in \Gamma \rightarrow \mathbb{Z}^\# \) since both are equivalent from a functional point of view).

In [2], it was shown that any binary operation \( \odot \) on \( \mathbb{Z}^\# \) used in \( (\Gamma \rightarrow \mathbb{Z}^\#) \) can be transferred in the XDD domain in an equivalent operation \( \boxplus \) such that performing the operation in the XDD domain is lossless

\[
\forall s_1, s_2 \in (\Gamma \rightarrow \mathbb{Z}^\#)^2 \forall \gamma \in \Gamma
\]
\[
s_1[\gamma] \odot s_2[\gamma] = (\alpha(s_1) \boxplus \alpha(s_2))[\gamma]
\]
with \( \alpha \) the morphism from \( (\Gamma \rightarrow \mathbb{Z}^\#) \) to \( \text{XDD} \).

The operations used in XG are max and \( + \) (1). These two operations are transferred into the domain of XDD, respectively, \( \odot \) and \( \boxplus \),\(^1\) such that the time corresponding to each configuration in the resulting XDD is exactly the time computed for each configuration without XDD

\[
\forall s_1, s_2 \in (\Gamma \rightarrow \mathbb{Z}^\#)^2 \forall \gamma \in \Gamma
\]
\[
s_1[\gamma] + s_2[\gamma] = (\alpha(s_1) \boxplus \alpha(s_2))[\gamma]
\]
\[
\max(s_1[\gamma], s_2[\gamma]) = (\alpha(s_1) \odot \alpha(s_2))[\gamma].
\]

The implementation of this operation is detailed in [2]. Shortly, \( \boxplus \) combines XDD operands along subtrees and applies \( \odot \) when leaves need to be combined.

\(^1\)It must be emphasized that the name of the transferred operators is not related to the original operator: \( \odot \) does not correspond to \( + \) but max, and it is \( \boxplus \) that corresponds to \( + \), this is due to the algebraic property of XDD semi-ring that we will mention later in Section IV-C.

IV. RESOURCE-BASED MODEL

The usual approach, that consists 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. The bus access interactions can indeed span over BB bounds.

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 pipeline states. Moreover, by leveraging the algebraic property of XDDs, we improve the computational model by implementing transitions as products of matrices. The matrices can be precomputed before the pipeline analysis.

![Fig. 2. Example of XDDs. (a) Example of \( \oplus \). (b) Explicit representation. (c) Example of \( \otimes \).](image-url)
As in the integer case, each computation requires the results on the structure of the pipeline, but an important point is that elements, each element being an XDD. The vector depends on the structure of the pipeline, 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.

Such a vector can be similarly built for each instruction and each pipeline stage. Table II shows the complete dependency information to be maintained for each stage of our example pipeline. The symbol \(-\infty\) denotes the absence of dependency.2 \(\rho_{\text{fetch}}\), \(\rho_{\text{load}}\), \(\rho_{\text{store}}\), and \(\rho_{\text{reg}}\) are, respectively, the last instructions that have fetched an instruction block from memory, performed a load, performed a store, and wrote to register \(R_i\) (in stage \(sR_i\)).

Let \(D\) be the set of dependencies listed in Table II. The 

| Table I Information Needed to Determine the Start Time of Any \(I_i\) in Stage DE |
|-------------------|-------------------|-------------------|-------------------|-------------------|
| **Program Order** | **Capacity Order** | **Pipeline Order** | **Queue Capacity** |
| \(\rho_{[I_{i-1}/DE]}\) | \(\rho^*_{[I_{i-3}/DE]}\) | \(\rho_{[I_{i-1}/FE]}\) | \(\rho_{[I_{i-2}/EX]}\) |
| \(\rho_{[I_{i-1}/DE]}\) | \(\rho^*_{[I_{i-3}/DE]}\) | \(\rho^*_{[I_{i-1}/FE]}\) | \(\rho^*_{[I_{i-2}/EX]}\) |
| \(\rho_{[I_{i-1}/EX]}\) | \(\rho^*_{[I_{i-1}/EX]}\) | \(\rho_{[I_{i-2}/EX]}\) | \(\rho_{[I_{i-2}/EX]}\) |
| \(\rho_{[I_{i-1}/CM]}\) | \(\rho^*_{[I_{i-1}/CM]}\) | \(\rho_{[I_{i-1}/CM]}\) | \(\rho^*_{[I_{i-2}/CM]}\) |

**A. Temporal State**

Each dependency \(d \in D\) in the XG model the use of resources, such as pipeline 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.

1) The start time of the previous instruction in stage DE:
\[ \rho_{[I_{i-1}/DE]} \] —the program order dependency due to the use of DE stage in program order.

2) The end time of the second last instruction in stage DE:
\[ \rho^*_{[I_{i-2}/DE]} \] —the capacity order dependency due to the use of DE stage with limited capacity.

3) The end time of \(I_i\) in stage FE: \(\rho^*_{[I_{i}/FE]}\) —the pipeline order dependency due to the use of the pipeline.

4) The start time of the second last instruction in stage EX: \(\rho_{[I_{i-2}/EX]}\) —the Queue Capacity dependency due to the use of the queue between DE and EX with limited capacity

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

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

This corresponds to the computation of (1) extended to the XDD domain: we use \(\oplus\) instead of max and \(\odot\) instead of +. As in the integer case, each computation requires the use of the queue between DE and EX with limited capacity

\[
\rho^*_{[I_{i-2}/DE]} = \rho_{[I_{i-2}/DE]} \odot \lambda_{[I_{i-2}/DE]}
\]

\[
\rho^*_{[I_{i}/FE]} = \rho_{[I_{i}/FE]} \odot \lambda_{[I_{i}/FE]}
\]

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 computation, we add an element \(\rho\) at index \(i_d\) to the state vector. It records the current time along the analysis: we refer to \(\rho\) as the time pointer.

Definition 2: Following the principle of XG analysis, the behavior of an instruction in a pipeline stage can be broken down into four steps.

1) Step 1: The instruction cannot be executed in the stage before all its dependencies are satisfied (5). This is modeled as follows. First, the time pointer is reset to \(0 = \text{LEAF}(\infty)\) (step 1.1). Then, each dependency time

\[
\rho_{[I_{i}/s]} = \bigoplus_{d \in D_{[I_{i}/s]}} \tilde{s}_{[i_d]}
\]

\[
\rho^*_{[I_{i}/s]} = \rho_{[I_{i}/s]} \odot \lambda_{[I_{i}/s]}
\]

where \(i_d\) denotes the index of dependency \(d\) within vector \(\tilde{s}\).
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 $\tau_{\text{reset}}$ and $\tau_{\text{wait}}$

\[
\tau_{\text{reset}} : \mathcal{S} \to \mathcal{S}
\]

\[
\tau_{\text{reset}}(\tilde{S}) = \tilde{S}
\]

\[
\tau_{\text{wait}} : \mathbb{N} \times \mathcal{S} \to \mathcal{S}
\]

\[
\tau_{\text{wait}}(x, \tilde{S}) = \tilde{S}^i = \begin{cases} 
\tilde{S}^i[i] = \tilde{S}[i] \oplus \tilde{S}[x], & \text{if } i = i_0 \\
\tilde{S}^i[i] = \tilde{S}[i], & \text{otherwise}
\end{cases}
\]

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

2) Step 2: Some resources (e.g., the queues between stages) are released at the start of an XG vertex. The corresponding dependencies (Queue Capacity) have to be updated with the start time $\rho$ recorded by the time pointer $\rho$. Generally, the update of the vector is done with $\tau_{\text{move}}$

\[
\tau_{\text{move}} : \mathbb{N} \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}
\]

\[
\tau_{\text{move}}(i_{\text{dest}}, i_{\text{src}, i}, \tilde{S}) = \tilde{S}^i = \begin{cases} 
\tilde{S}^i[i] = \tilde{S}[i_{\text{src}, i}], & \text{if } i = i_{\text{dest}} \\
\tilde{S}^i[i] = \tilde{S}[i], & \text{otherwise}
\end{cases}
\]

3) Step 3: The started instruction spends $\lambda_{i, \ell / s}$ cycles in the stage. After this step, the value of the time pointer is the end time of $[i, \ell / s]$

\[
\tau_{\text{consume}} : \mathbb{N} \times \mathcal{S} \to \mathcal{S}
\]

\[
\tau_{\text{consume}}(\lambda_{i, \ell / s}, \tilde{S}) = \tilde{S}^i = \begin{cases} 
\tilde{S}^i[i] = \tilde{S}[i] \oplus \lambda_{i, \ell / s}, & \text{if } i = i_0 \\
\tilde{S}^i[i] = \tilde{S}[i], & \text{otherwise}
\end{cases}
\]

4) Step 4: The instruction finishes its execution and the dependencies recording the end time of the current vertex are updated. The $\tau_{\text{move}}$ operation is used in the same way as in the step 2.

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. Computational Model

An important property of the XDD domain is that, equipped with $\oplus$ and $\otimes$, it forms the semi-ring $(\text{XDD}, \oplus, \otimes, 0, 1)$ with $0 = \text{LEAF}(-\infty)$ and $1 = \text{LEAF}(0)$. As the transition functions $\tau$ are affine in this domain, their application can be expressed as matrix multiplications. Combining and precomputing these matrices will help to speed up the pipeline analysis at CFG level since some BBs need to be recomputed several times in different execution contexts.

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

Definition 3: The scalar multiplication is defined by

\[
\cdot : \text{XDD}^N \times \text{XDD}^N \to \text{XDD}
\]

\[
[f_0, f_1, \ldots, f_{n-1}] \cdot [f_0', f_1', \ldots, f_{n-1}'] = \bigoplus_{0 \leq i < N-1} f_i \otimes f_i'.
\]

Definition 4: The matrix multiplication is defined by

\[
B \cdot C = \begin{bmatrix} A_{i,j} \end{bmatrix} \cdot \begin{bmatrix} C_{i,j} \end{bmatrix} = \bigoplus_{1 \leq k \leq M} B_{i,k} \otimes C_{k,j}.
\]

Definition 5: The identity matrix $\text{Id}$ on the XDD semi-ring is defined by

\[
\text{Id} = \begin{bmatrix} A_{i,j} \end{bmatrix} = \begin{cases} 1, & \text{if } i = j \\
0, & \text{otherwise}
\end{cases}
\]

Note that, by definition, $\tilde{S} \cdot \text{Id} = \tilde{S}$: any matrix column at index $i$ contains only 0 except for the row $i$ that contains a 1, which maintains unchanged the value of $\tilde{S}[i]$ in the resulting vector. To implement the transition functions $\tau$ as matrix multiplications, the matrix $\text{Id}$ is taken as a basis and only the cells that have an effect on the vector have to be changed.

1) A 0 on the diagonal of the $\text{Id}$ matrix at the timer pointer position resets it: $\tilde{S}[i_0] \oplus 0 = 0$

\[
\tau_{\text{reset}}(\tilde{S}) = \tilde{S} \cdot \text{M}_{\text{reset}}
\]

\[
\tau_{\text{reset}}(\tilde{S}) = \tilde{S} \cdot \begin{bmatrix} A_{i,j} \end{bmatrix} = \begin{cases} 0, & \text{if } i = j = i_0 \\
\text{Id}_{i,j}, & \text{otherwise}
\end{cases}
\]

2) For a given slot at index $x$ in $\tilde{S}$, $\tau_{\text{wait}}(x, \tilde{S})$ is represented by a matrix $\text{M}_{\text{wait}(x)}$ with a 1 at position $(x, i_0)$ resulting in the operation $\tilde{S} \oplus (\text{Id} \otimes \tilde{S}[x])$

\[
\tau_{\text{wait}}(x, \tilde{S}) = \tilde{S} \cdot \text{M}_{\text{wait}(x)}
\]
\[ S \cdot A_{ij} |A_{ij} = \begin{cases} 1, & \text{if } i = i_0 \land j = x \\ I_{id_{ij}}, & \text{otherwise.} \end{cases} \]

3) \( \tau_{\text{move}(i_{src}, i_{dest}, \bar{S})} \) is represented by a matrix \( M_{\text{move}(i_{src}, i_{dest})} \) where the element at \((i_{dest}, i_{dest})\) is set to 0 and the element \((i_{dest}, \bar{i}_{src})\) to 1. s.t. element \( i_{dest} \) in the result becomes \((0 \odot \bar{S}[i_{dest}]) + (1 \odot \bar{S}[i_{src}]) = \bar{S}[i_{src}] \)

\[ \tau_{\text{move}(i_{src}, i_{dest}, \bar{S})} = \bar{S} \cdot M_{\text{move}(i_{src}, i_{dest})} \]

4) For a given latency \( \lambda \), \( \tau_{\text{consume}(\lambda, \bar{S})} \) can be represented by a matrix \( M_{\text{consume}(\lambda)} \), obtained from \( Id \) by putting \( \lambda \) at position \( (i_{0}, i_{0}) \)

\[ \tau_{\text{consume}(\lambda, \bar{S})} = \bar{S} \cdot M_{\text{consume}} \]

\[ = \bar{S} \cdot A_{ij} |A_{ij} = \begin{cases} \lambda, & \text{if } i = j = i_0 \\ I_{id_{ij}}, & \text{otherwise.} \end{cases} \]

Theorem 1: Each transition function \( \tau \) applied to the timing vector is a linear map from \( S \) to \( S \).

Proof: The proof is straightforward using the matrix representation of the transitions provided in Definition 5.

Consequently, the operations performed at each stage are also linear because they are combination of \( \tau \) functions. Their matrix representation is simply the multiplication of each invoked \( \tau \) function. For example

\[ M_{\text{Step1}[i/s]} = M_{\text{Reset}} \cdot \prod_{\alpha \in \mathcal{D}[i/s]} M_{\text{Wait}(i_d)} \]

with \( \mathcal{D}[i/s] \) the set of dependencies required by \([i/s]\) and \( i_d \) the index of resource \( \alpha \) in the state vector.

Similarly, we can express \( M_{\text{Step2}[i/s]} \), \( M_{\text{Step3}[i/s]} \), and \( M_{\text{Step4}[i/s]} \) by invoking the corresponding \( \tau \) 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 four steps

\[ M_{[i/s]} = M_{\text{Step1}[i/s]} \cdot M_{\text{Step2}[i/s]} \cdot M_{\text{Step3}[i/s]} \cdot M_{\text{Step4}[i/s]} \]

Finally, the whole analysis of a BB \( a \in V \) is composed by the analysis of each instruction in each pipeline stage

\[ M_a = \prod_{i \in a \in S \in \mathcal{P}} M_{[i/s]} \]

With a matrix, such as \( M_a \), it is easy and fast to compute the output temporal state \( \bar{S} \in S \) corresponding to an input temporal state \( \bar{S} \in S \) for a BB \( a \)

\[ \bar{S} = \bar{S} \cdot M_a \]

V. 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 mainly consists in tracking the explicit set of possible temporal states for each BB all over the CFG execution paths.

A. Computing the Context With Rebasin Operation

In an analysis at BB level, a temporal state contains times relative to the start of a BB. In an analysis at CFG-level, a temporal state should contain times relative to the start of the program and the temporal states should be tracked for all possible execution paths. This is generally infeasible because of the huge number of execution paths, especially, in the presence of loops.

Now the main reason for which we want to compute exact temporal states at CFG level is to determine the exact timings of bus accesses. But these timings do not need to be absolute with respect to the start of the program. Instead, they can be relative to arbitrarily chosen time bases, as long as XDDs with different bases are not mixed.

We call rebasing the operation that consists in changing the origin of the timeline of a temporal state. The temporal state at the end of BB \( a \) represents the delay induced by the execution of \( a \) to the start of the following BB \( b \). Considering the start of \( b \) as a new time base \( T \in \text{XDD} \), we can get a new temporal state relative to \( T \) by subtracting \( T \) from the times in the temporal state in the base of BB \( a \). 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 \( \odot \) is built in the usual way from the operator (3).

Rebasing a temporal state is lossless because \( \odot \) is reversible. By adding \( T \) (with \( \odot \)), 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 at all. 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. Thus, the effect of events is generally short term and is, in practice, often eliminated by rebasing.

B. Generation of Events Within Loops

Each event computed by global analyses is 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 of a loop body, since it may occur or not in different iterations. We would get unsound timings if we denoted these different event occurrences with the same event node in the XDD. To fix this, a generation number is associated with each event. To prevent a blowup of temporal states, this generation number is relative to the current iteration and is incremented in the current temporal state each time the analysis iterates in the loop. The generation number, thus, distinguishes occurrences of an event in different iterations. Fortunately, this method does not result in an endless increase of generations because: 1) the effect of events is often bounded in time and 2) the WCET calculation requires to bound the number of loop iterations.

C. CFG Pipeline Analysis

Finally, the complete pipeline analysis is designed like a classical data-flow analysis with a work list. Each BB is
TABLE III
POSSIBLE SCHEDULES OF ME0 WITH SUBSEQUENT FES

| Schedule       | Condition                                      | Scheduling time of ME0 |
|----------------|-----------------------------------------------|------------------------|
| ME0, FE1, FE2, FE3 | \( \rho_{ME0} < \rho_{FE1} \)                 | max(\( \rho_{FE1} + \lambda_{BUS} \rho_{ME0} \)) |
| FE1, ME0, FE2, FE3 | \( \rho_{FE1} < \rho_{ME0} \leq \rho_{FE2} \) | max(\( \rho_{FE2} + \lambda_{BUS} \rho_{ME0} \)) |
| FE1, FE2, ME0, FE3 | \( \rho_{FE1} < \rho_{ME0} \leq \rho_{FE2} \) | max(\( \rho_{FE2} + \lambda_{BUS} \rho_{ME0} \)) |
| FE1, FE2, FE3, ME0 | \( \rho_{FE2} < \rho_{ME0} \)                 | max(\( \rho_{FE2} + \lambda_{BUS} \rho_{ME0} \)) |

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 (i.e., multiplied by the precomputed matrix) and rebased to make a new output state. If the set of its output states differs from the original set, the successors of the BB are pushed into the work list. The process is repeated until finding a fix point on all sets of state. If the set of its output states differs from the original set, the precomputed matrix) and rebased to make a new output state. For readers familiar with abstract interpretation, this is actually the collecting semantics of our pipeline semantics.

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 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. They can then be used to precisely analyze the temporal behavior in the presence of out-of-order scheduled resources.

VI. MODELING THE SHARED MEMORY BUS

In previous work, our analysis was valid only when all the resources were accessed in the program order. However, in embedded microprocessors, the instruction and data caches often share a common bus to the memory (or to a unified L2 cache). This introduces an out-of-order behavior: an instruction cache miss might access the shared bus before a data cache miss generated by an older instruction. In other words, the variability due to events in the start times of the FE and ME nodes may change the access order to the shared bus. Since XG dependencies are not expressive enough to model out-of-order bus allocations, we propose, in this section, an extension to the pipeline analysis to efficiently manage the shared bus accesses according to the different configurations of the temporal states. It supports the usual bus arbitration policy: first-come–first-served (FCFS), with priority given to the ME stage in case of simultaneous bus accesses.

A. Bus Scheduling Properties

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). Similarly, bus accesses by the FE stage follow the program order. Moreover, the pipeline order ensures that a request emitted by an instruction in the FE stage is granted 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: a bus request in the ME stage by an instruction denoted ME0 may be delayed by a bus request in the FE stage by a younger instruction denoted FE\(_{i|j>0}\). To simplify the notation in this section, ME0 and FE\(_{i|j>0}\) denote the instructions as well as the XG vertex (e.g., ME0 also denotes vertex [ME0/ME]; FE\(_1\) also denotes [FE\(_1/FE\)]. In-between instructions are disregarded but are still accounted for in the matrices used to update the temporal states.

To sum up, FE\(_i\) can delay ME0 only if FE\(_i\) is ready to enter FE stage before ME0 is ready to enter ME. In the XG model, this situation can only happen when FE\(_i\) does not depend on ME0, that is, when there is no path from ME0 to FE\(_i\).\(^3\)

3The occurrence of such situations is limited by the size of the interstage queues in the pipeline.
cycles. The events $e_0$ and $e_1$ are events related to the instructions before ME$_0$ that are assumed to impact the start time of ME$_0$ and FE$_2$.

XDD (a) shows the ready time of ME$_0$ and (b) the initial value of $\tilde{\rho}_{ME_0}$—the scheduling time of ME$_0$ on the bus (+\(\infty\) means that no access is yet scheduled). (c) shows the initial value of $\rho_{rel}$, recording the release time of the bus by FE$_i$ ($-\infty$ denotes that the bus is not used by any FE for now).

The ready time of FE$_1$ (d) is computed from the initial state $S_0$ 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).

$\rho_{ME_0}$ (a) and $\rho_{FE_1}$ (d) are compared using $\triangleright_{ME}$ to get the configurations and the time $[\rho_{schedME_0}(e)]$ when ME$_0$ takes the bus, i.e., is scheduled, before FE$_1$ ([$\triangleright_{ME}$ is formally defined in (16)]. Other configurations are assigned $+\infty$, denoting that they are not processed yet. Notice that a $-\infty$ configuration in $\rho_{FE_1}$ does not allow ME$_0$ to be scheduled as subsequent FE$_{2\ldots i}$ 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), i.e., the configurations where FE$_1$ gets the bus, is computed in a similar way as $\rho_{schedME_0}$, but with operator $\triangleright_{FE}$ that selects the configurations in which FE gets the bus, using the strict $<$ comparison instead of $\leq$ because the ME stage has priority over the FE stage in case of simultaneous requests. By adding the latency of the bus ($\lambda_{BUS}$) to $\rho_{schedFE_1}$, we are able to update, using $\odot$, the release time of the bus after FE$_1$ ($\rho_{rel}$ (h)). Finally, we compute the actual schedule of FE$_1$ $[\rho_{FE_1}(i)]$ that is the time of $\rho_{schedFE_1}$ if FE$_1$ is scheduled, or otherwise the release time of the bus by ME$_0$ ($\hat{\rho}_{ME_0} \odot \lambda_{BUS}$). 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 $S_{FE_1}$ by the matrix $M_{FE_1 \rightarrow FE_2}$, we get the ready time of FE$_2$ (i).

In the second iteration, first, $\rho_{ME_0}$ (a) is compared with $\rho_{FE_2}$ (j) with the operator $\triangleright_{ME}$. 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$_1$ ($\rho_{rel}$) 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 $\triangleright_{FE}$ applied to $\rho_{FE_1}$ and $\rho_{ME_0}$ which is then used to update the release time of the bus $\rho_{rel}$ (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$_2$, $\rho_{rel}$.

Operator $\triangleright_{ME}$ (resp. $\triangleright_{FE}$) has a straightforward definition, setting to $+\infty$ the configurations where ME (resp. FE) does not get the bus

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

$$\langle ME \triangleright_{ME} FE \rangle[\gamma] = \begin{cases} f_{ME}[\gamma], & \text{if } f_{ME}[\gamma] \leq f_{FE}[\gamma] \\ +\infty, & \text{otherwise} \end{cases}$$

$$\langle FE \triangleright_{FE} ME \rangle[\gamma] = \begin{cases} f_{FE}[\gamma], & \text{if } f_{FE}[\gamma] < f_{ME}[\gamma] \\ +\infty, & \text{otherwise} \end{cases}$$

All these calculations seem 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 previous example is described more formally in this paragraph. Basically, the pipeline analysis is extended by splitting the BBs at contention points, i.e., at the XG nodes where a bus access may occur (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$_{0\ldots i\ldots n}$). The instructions between two successive contention points are summarized by a precomputed matrix.
Algorithm 1: Contention Computation

Input: $S_0 \in S$, $(ME_0, FE_{i \leq n})$
Output: $(\hat{\rho}_{ME_0}, \hat{\rho}_{FE_{i \leq n}})$

1. $\hat{\rho}_{ME_0} = LEAF(+\infty)$
2. $\hat{\rho}_{rel} = LEAF(-\infty)$
3. $\hat{S}_{FE_i} := \hat{S}_{ME_0} \cdot M_{ME_0} - FE_1$
4. $i = 1$
5. $\hat{\rho}_{ME_0} := \hat{S}_{FE_i}[iFE]$
6. while $i \leq n \land (\exists \gamma \in \Gamma \land \hat{\rho}_{ME_0}[\gamma] = +\infty)$ do
7. if $FE_i,\mustUseBus()$ then
8. $\hat{\rho}_{FE_i} := \hat{S}_{FE_i}[iFE]$
9. else
10. $\hat{\rho}_{FE_i} := \hat{S}_{FE_i}[iFE] \oplus NODE(\hat{\rho}_{rel}, -\infty, 0)$
11. $\hat{\rho}_{schedME_0} := (\hat{\rho}_{ME_0} \otimes ME \cdot PFE) \oplus \hat{\rho}_{rel}$
12. $\hat{\rho}_{ME_0} := \hat{\rho}_{ME_0} \oplus \hat{\rho}_{schedME_0}$
13. $\hat{\rho}_{schedFE_i} := \hat{\rho}_{FE_i} \oplus FE \cdot \hat{\rho}_{ME_0}$
14. $\hat{\rho}_{rel} := \hat{\rho}_{rel} \oplus (\hat{\rho}_{schedFE_i} \otimes \lambda_{BUS})$
15. $\hat{S}_{FE_{i+1}} := (\hat{S}_{FE_i} \oplus [0, \ldots, 0, \hat{\rho}_{FE_i} \otimes \lambda_{BUS}]) 
\cdot M_{FE_i} - FE_{i+1}$
16. $i = i + 1$
17. $\hat{\rho}_{ME_0} := \hat{\rho}_{ME_0} \oplus (\hat{\rho}_{rel} \oplus \hat{\rho}_{ME_0})$

Algorithm 1 is then applied to compute the possible interleaving of bus accesses for all configurations of the sequence $(ME_0, FE_{i \leq n})$. Additionally, it takes as input the temporal state $S_0$. The result is the definitive schedule of $ME_0$ (i.e., $\hat{\rho}_{ME_0}$) and of $FE_i$ (i.e., $\hat{\rho}_{FE_i}$).

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 $FE_i$ have been processed, the schedule of $ME_0$ is complete (condition at line 6). Line 2 initializes $\hat{\rho}_{rel}$ 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 $S_0$: $i$ is initialized in line 4 and will range over the Contention Points, 1 to $n$. The ready time of $ME_0$ is recorded into $\hat{\rho}_{ME_0}$ at line 5. Lines 7–10 compute the ready time of $FE_i$ if the access always or sometimes results in a miss [according to $\mustUseBus()$]. The latter case is expressed by the event $ic_i$ and by adding the NODE$(\hat{\rho}_{rel}, -\infty, 0)$ to $\hat{\rho}_{FE_i}$: $-\infty$ denotes the case where $ic_i$ does not arise and there is no bus access.

$\rho_{schedME_0}$ configurations getting the bus before $FE_i$, is computed with $\otimes$ at line 11 by comparing the ready times 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 $\hat{\rho}_{rel}$. 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 $\rho_{schedFE_i}$. By adding the bus latency $\lambda_{BUS}$ to $\rho_{schedFE_i}$ and then $\oplus$ with $\hat{\rho}_{rel}$, the release time of the bus is only updated for configurations $\gamma$ where $FE_i$ uses and gets the bus ($\rho_{schedFE_i}[\gamma] \neq +\infty$) (line 14). Notice that the $+\infty$ in $\hat{\rho}_{rel}$ 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 $FE_i$ or $ME_0$ is scheduled. However, as $\rho_{schedFE_i}$ configurations not leading to $+\infty$ are lower than $\hat{\rho}_{ME_0}$ (otherwise, it is considered as nonscheduled), $\ominus$ can be used to implement the replacement.

At line 16, the temporal state is updated regarding the schedules of $FE_i$, by applying $\ominus$ 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 remaining $+\infty$ configurations 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 $\hat{\rho}_{ME_0}$ may also be caused by the fact that none of the $FE_i$ have used the bus: this time is recorded as $-\infty$ in $\hat{\rho}_{rel}$ and is, hence, automatically overwritten by the ready time of $ME_0$.

VII. Evaluation

The performance of the analysis strongly depends on the size of the XDDs in the pipeline states and on the number of pipeline states. Both metrics are related to some inherent properties of the analyzed program and of the microarchitecture, the impact of which is difficult to theoretically estimate. Therefore, we experiment our analysis on realistic benchmarks that empirically provide a better understanding of the performance.

A. Experimental Setup

The pipeline used in the examples of the previous sections was chosen to enhance the readability of the article. For the experimentation, we consider a more powerful microarchitecture with more parallelism leading to more complex temporal states. In addition, this new pipeline allows demonstrating the scalability of our approach.

Our pipeline has four stages (FE, DE, EX, and CM), each able to process four instructions per cycle. In the FE stage, it fetches instructions from a single-level instruction cache. The FE stage is able to fetch four instructions of the same memory block simultaneously, with a latency of seven cycles in case of a cache miss (ignoring possible contention). The DE stage decodes the instructions and the EX stage handles all arithmetic, floating point, and memory-related operations in several functional units (FUs). Four arithmetic and logic units (ALUs) are available and can be simultaneously used by independent instructions. The latency of arithmetic operations is one cycle for additions and subtractions, two cycles for multiplications and seven cycles for divisions. One floating point unit (FPU) is available with a latency of three cycles for additions and subtractions, five cycles for multiplications, and 12 cycles for divisions. A single memory unit (MU) handles memory-related operations (loads and stores). In case of a multiple load/store operation, the memory accesses are performed in order, and if one multiple load/store needs to use the memory bus (in
case of data cache misses), it occupies it until all loads/stores are completed. The latency of data accesses is the same as for instruction fetches. In the EX stage, an issue buffer distributes the instructions to the appropriate FUs. Instructions using the same FU are executed in order; instructions using different FUs can be executed out-of-order (if the absence of data dependencies).

The instruction cache is a 16-KB 2-way set associative least recent used (LRU) cache. The data cache is a 8-KB 2-way set associative LRU cache. There is a single level of caches and they share the same bus to the main memory. We believe that this architecture is representative of many mid-range processors used in real-time embedded systems.

The whole CFG analysis is implemented using the OTAWA toolbox [11]. Global analyses, including instruction and data cache analyses as well as control flow analyses, are provided by OTAWA. The benchmarks are taken from the TACLe suite [10] compiled for the armv7 instruction set with a hard FPU. Among 79 tasks to be analyzed, five fail due to limitations in OTAWA (unsupported irregular control flow, e.g., unable to solve dynamic branch address) and another five have been dropped due to the lack of annotations on the bounds of recursive function calls, for which the WCET computation impossible.

B. Number of Temporal States

The first experiment explores the number of temporal states along the edges of BBs (representing the output of the source BBs and the input of sink BBs). The experimental results are shown in Fig. 4. The x-axis shows the number of pipeline states and the y-axis shows the number of edges for each amount of states (in a logarithmic scale). The bars accumulate data from all the benchmarks.

The risk, with our approach at CFG level, is to face an exploding number of states which is a common concern of collecting semantic-based analyses. Fortunately, the experimentation shows that most of the edges have less than 20 output states. This means that most of the 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 where the number of states is much higher is not blocking as the analysis time remains reasonable in most cases (cf. Section VII-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 number of states and size of XDDs. In our microarchitecture, an event is created by a cache access and may disappear from the XDD during the analysis, for two reasons: 1) it is absorbed by the pipeline: for example, when an instruction stalls in the EX stage due to a data cache miss, the next instructions may still be processed in the pipeline, completely hiding the stalling time—this event will stay alive only in a short time window during the analysis of other instructions executed in parallel and 2) 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. Therefore, the lifetime of events provides a fine estimation of both the size of XDDs and the number of pipeline states of our analysis at CFG level. As complement, in [2], we provide detailed discussion and experiments, on different microarchitecture setups, of the size of XDDs for BB-level analysis where the pipeline absorption effect is the primary factor.

However, the pipeline analysis is only able to provide the lifetime information at the granularity of contention points because the effect of executing instructions between contention points is expressed 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 if the temporal state of which does not contain the event in any XDD 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.

Fig. 5 shows the experimental results. The x-axis is the lifetime of events (in instructions with limitations described above) and the y-axis, in logarithm scale, shows the number of events having this lifetime. These are also accumulated from the whole set of TACLe benchmarks. The statistics show that most events have a 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 situations where the lifetime is greater than 50, the events are in a very long BB. In the extreme case of the 602-instruction event lifetime, the involved BB contains 617 instructions (in benchmark md5) and the reported lifetime is an effect of the granularity level. Despite this very infrequent case, most events have short lifetimes and the size of temporal states (sum of XDDs sizes of the vector) is reasonable and the analysis remains efficient.

D. Analysis Time

The analysis time includes the time to precompute the matrices and the duration of the pipeline analysis on the CFG.
We run the analysis on a virtual machine on a cloud server with 8 GB RAM and 4-core Intel Broadwell processors. Only 2 cores are occupied simultaneously to process the benchmarks. We also measure the analysis time without leveraging the algebraic property of XDD to precompute the matrices (with a timeout of 1 h) in order to clarify the benefit of this optimization. The results are shown in Fig. 6. The x-axis shows the benchmarks and the y-axis plots the analysis time in seconds in a logarithmic scale. The analysis time with matrices is shown as green bars. At worst, the analysis with matrices finishes in 553 s (9 m 13 s). In most cases, it finishes in about 1–20 s. In contrast, the analysis without matrices has both memory usage and speed issues as shown by the red (crashed because out of 8G RAM) and yellow bars (timeout after 1 h). For those finishing within 1 h (blue bars), the matrix optimization brings 217% speed-up in average. The rare cases where the analysis without matrix is faster are simple benchmarks where the cost of computing the matrices is not balanced by the speed-up. This result reveals that the precomputation of matrices effectively reduces intermediate redundant computations, which enhances the analysis performance in terms of speed and memory usage. The analysis is still able to handle problematic cases (those that have a large number of temporal states or long lifetime events) in a reasonable time. Moreover, we observe that the industry-like applications encompassed in the TACLeBench collection (Debbie and Papabench) are analyzed in short times (respectively, 10 and 15 min, summing the analysis times of all their tasks). Therefore, we believe that our approach could be used in industrial real-time applications (e.g., air bus requires at most 48 h between the detection of a bug and the distribution of the fix, including temporal verification).

**E. Precision**

The presence of out-of-order resources enables potential timing anomalies [7]. This means that the pipeline analysis cannot assume the local worst case to be the global worst case upon timing variations. In the presence of possible timing anomalies, a tradeoff between precision and performance is more difficult to achieve: the analysis either has to track exactly all possible timing states or to cover the timing variations with a safe but largely over-estimated cost. In the example of the shared bus, the cost for not tracking precise pipeline states is to cover the latency of memory access with an upper bound by considering the maximal potential contention with prior and subsequent instructions. To evaluate this over-estimation, we implemented this strategy in etime, the state-of-the-art pipeline analysis of OTAWA. We compare the WCET estimated with etime to the one estimated with the new approach.

Concretely, the etime analysis implements the same pipeline model (i.e., XG based) except for it does not use XDDs and considers the WCET of BBs locally instead of at CFG level. In order to tighten the WCET, it considers all the combinations of events for each BB. But since it disregards the execution context, it may choose to drop the context at any time to reduce the exponential number of pipeline states—which is not safe with timing anomalies. Moreover, etime generates only one execution time for each BB (the worst one), while the new pipeline analysis provides all the possible execution times to the IPET stage. The way IPET can handle these multiple execution times has been detailed in [30] and is out of the scope of this article, but it clearly benefits the new approach. Finally, as etime originally did not support out-of-order accesses, we created a new type of event with a latency of one memory access that is associated to each memory access not classified as Always Hit or Always Miss in order to cover the waiting time of the shared bus.

As both etime and the approach presented in this article are implemented with OTAWA, we are able to ensure that the micro-architectural model and global analyses are the same. Disregarding the safety issue due to ignoring timing anomalies, we believe that the WCET obtained by etime can be used as a reference of a classical static WCET approach that considers out-of-order resources without tracking exact pipeline states.

The experimental results are shown in Fig. 7, in which the x-axis shows the benchmarks and the y-axis shows the gain in precision computed by $\frac{t_{etime} - t_{new}}{t_{etime}}$, where $t_{etime}$ is the WCET obtained by etime and $t_{new}$ is the WCET obtained by our new approach. Due to the limited performance of etime, the epic benchmark does not finish within 6 h. The susan benchmark also fails due to a bug in OTAWA. This is why the bars for these two benchmarks miss in the figure.

The experimental results show that, for most of the benchmarks, the WCET obtained by using the new approach is improved by 40%–80%. In some rare cases (fif, two tasks of papabench, rad2deg, rijndael_enc, rijndael_dec, and statefate), the gain is smaller but still around 20%–40%: these benchmarks are the least memory intensive and get the worst results from global analyses, which explains why the over-estimation on memory latency and the improved IPET (see [30]) have a lower impact.

This impressive improvement on the final WCET is explained by the precise estimation of the bus latency, the benefits of which are amplified by the improved IPET formulation. Concerning the latency of memory accesses, the new approach computes the exact waiting time. If we consider that the actual waiting time is low (which is the case in the example microarchitecture that is almost in-order), the worst memory latency accounted in etime is nearly doubled. The improved IPET applied to the new approach also contributes to the improvement: according to our experiments in [30] it appears that it helps to improve the IPET formulation.
achieves itself 0%–50% of the gain precision depending on the benchmark. Moreover, the over-estimation on the pipeline analysis is even amplified by standard IPET (as opposed to our improved version [30]) because the WCET of each BB is multiplied by the BB’s execution count. This explains the impressive improvement up to 80% observed in Fig. 7.

VIII. CONCLUSION

In this article, we formally define an efficient state representation for pipeline timing analysis using vectors of XDDs. It enables the analysis of the pipeline at CFG level. By leveraging the algebraic properties of XDDs, we perform the analysis as a sequence of matrix multiplications, which significantly reduces the analysis time. This analysis also improves the precision of the estimated WCET by 20%–80% on the TACLe benchmarks compared to previous analysis implemented in the OTAWA toolset. We extend our new analysis to precisely take into account the contention on the memory bus shared by the instruction and data caches. This makes the analysis sound for timing-anomaly prone processors. The measured analysis time for the TACLe benchmarks shows that the approach is scalable and should be practical for industrial applications.

As future work, we could benefit from the exact tracking of the temporal states to qualify more precisely the effects of timing variations in different microarchitectures. This could be used to eventually qualify good or bad microarchitecture designs in terms of predictability. This may also help to find better tradeoffs in the design of time-predictable pipelines and alleviate over-stringent constraints on the pipeline, such as strict-in-order execution, that often limits the performance of the processor. We also plan to extend our approach to all kinds
of out-of-order scheduled resources. Although our operators and matrices calculation are correct whatever the out-of-order resource, the generalization of the contention model will likely raise performance issues as one can no longer leverage the constraints on the contention upon concurrent accesses. That said, we must find a scalable model of the interleaving of resource acquisitions when multiple out-of-order resources are involved.

REFERENCES

[1] J. Reineke et al., “A definition and classification of timing anomalies,” in Proc. 6th Int. Workshop Worst-Case Execution Time Anal. (WCET), Dagshtuhl, Germany, vol. 4, 2006, pp. 23–28. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2006/671

[2] 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 Proc. ACM SIGPLAN/SIGBED LCTES, 2020, pp. 119–130.

[3] T. Lundqvist and P. Stenström, “Timing anomalies in dynamically scheduled microprocessors,” in Proc. IEEE RTSS, 1999, pp. 12–21.

[4] J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm, “Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis,” in Proc. IEEE DDECS, 2006, pp. 15–20.

[5] G. Gebhard, “Timing anomalies reloaded,” in Proc. WCET, 2010, pp. 1–10.

[6] F. Cassez, R. R. Hansen, and M. C. Olesen, “What is a timing anomaly?” in Proc. WCET, 2012, pp. 1–12.

[7] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder, “Principles of timing anomalies in superscalar processors,” in Proc. QSC, 2005, pp. 295–306.

[8] R. Wilhelm et al., “Static timing analysis for hard real-time systems,” in Proc. VMCAI, 2010, pp. 3–22.

[9] Y.-T. S. Li and S. Malik, “Performance analysis of embedded software using implicit path enumeration,” in Proc. ACM SIGPLAN LCTES, 1995, pp. 88–98.

[10] H. Falk et al., “FACLeBench: A benchmark collection to support worst-case execution time research,” in Proc. WCET, 2016, p. 2.

[11] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, “OTAWA: An open toolbox for adaptive WCET analysis,” in Proc. IFIP SEUS, 2010, pp. 35–46.

[12] J. Schneider and C. Ferdinand, “Pipeline behavior prediction for superscalar processors by abstract interpretation,” ACM SIGPLAN Notices, vol. 34, no. 7, pp. 35–44, May 1999.

[13] S. Thesing, “Safe and precise WCET determination by abstract interpretation of pipeline models,” Ph.D. dissertation, Naturwissenschaftlich-Technischen Fakultäten, Univ. Saarland, Saarbrücken, Germany, 2004.

[14] P. Cousot and R. Cousot, “Abstract interpretation: A uniﬁed lattice model for static analysis of programs by construction or approximation of ﬁxpoints,” in Proc. 4th ACM SIGACT-SIGPLAN Symp. PLDI, 1977, pp. 238–252.

[15] S. Wilhelm, “Symbolic representations in WCET analysis,” Ph.D. dissertation, Naturwissenschaftlich-Technischen Fakultäten, Univ. Saarland, Saarbrücken, Germany, 2012.

[16] J. Reineke and R. Sen, “Sound and efficient WCET analysis in the presence of timing anomalies,” in Proc. 9th Int. Workshop Worst-Case Execution Time Anal. (WCET), Dagshtuhl, Germany, vol. 10, 2009. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2009/2289

[17] S. Hahn, J. Reineke, and R. Wilhelm, “Toward compact abstractions for processor pipelines,” in Correct System Design (Lecture Notes in Computer Science 9360), R. Meyer, A. Platzer, H. Wehrheim, Eds. Cham, Switzerland: Springer Int., 2015, pp. 205–220, doi: 10.1007/978-3-319-23506-6_14.

[18] S. Hahn and J. Reineke, “Design and analysis of SIC: A provably timing-predictable pipelined processor core,” Real-Time Syst., vol. 56, no. 2, pp. 207–245, 2020.

[19] J. Engblom and B. Jonsson, “Processor pipelines and their properties for static WCET analysis,” in Proc. EMSOFT, 2002, pp. 334–348.

[20] X. Li, A. Roychoudhury, and T. Mitra, “Modeling out-of-order processors for software timing analysis,” in Proc. IEEE RTSS, 2004, pp. 92–103.

[21] C. Rochange and P. Sainrat, “A context-parameterized model for static analysis of execution times,” in Transactions on High-Performance Embedded Architectures and Compilers II, P. Stenström, Ed. Berlin, Germany: Springer, 2009, pp. 222–241, doi: 10.1007/978-3-642-00904-4_12.

[22] G. Behrmann et al., “Uppaal 4.0,” in Proc. QEST, 2006, pp. 125–126.

[23] F. Cassez, “Timed games for computing WCET for pipelined processors with caches,” in Proc. IEEE ACSD, 2011, pp. 195–204.

[24] S.-S. Lim et al., “An accurate worst case timing analysis for RISC processors,” IEEE Trans. Softw. Eng., vol. 21, no. 7, pp. 593–604, Jul. 1995.

[25] A. Colin and I. Puaut, “A modular and retargetable framework for tree-based WCET analysis,” in Proc. 13th Euromicro Conf. Real-Time Syst., 2001, pp. 37–44.

[26] R. Kirner, A. Kadlec, and P. Puschner, “Precise worst-case execution time analysis for processors with timing anomalies,” in Proc. 21st Euromicro Conf. Real-Time Syst., 2009, pp. 119–128.

[27] S. B. Akers, “Binary decision diagrams,” IEEE Trans. Comput., vol. C-27, no. 6, pp. 509–516, Jun. 1978.

[28] R. E. Bryant, “Symbolic Boolean manipulation with ordered binary-decision diagrams,” ACM Comput. Surveys, vol. 24, no. 3, pp. 293–318, 1992.

[29] M. Fujita, P. C. McGeer, and J.-Y. Yang, “Multi-terminal binary decision diagrams: An efficient data structure for matrix representation,” Formal Methods Syst. Design, vol. 10, no. 2, pp. 149–169, 1997.

[30] Z. Bai, H. Cassé, M. De Michiel, T. Carle, and C. Rochange, “A framework for calculating WCET based on execution decision diagrams,” ACM Trans. Embedded Comput. Syst., vol. 21, no. 3, pp. 1–26, 2022.

Zhenyu Bai is currently pursuing the Ph.D. degree with IRIT-Université Toulouse 3, Toulouse, France. His main research interest concerns static program analysis and worst-case execution time calculation.

Hugues Cassé received the Ph.D. degree in computer science from the University of Toulouse, Toulouse, France, in 2001. He is an Associate Professor with IRIT-Université Toulouse 3, Toulouse, where he teaches machine architecture, compilation, and embedded systems. His main research interest concerns static program analysis and worst-case execution time calculation.

Thomas Carle received the engineering degree from École nationale des mines de Nancy, Nancy, France, in 2011, and the Ph.D. degree in computer science from Université Pierre et Marie Curie Paris 6, Paris, France, in 2014. He is a Lecturer with IRIT-Université Toulouse 3, Toulouse, France. His research focus is on the timing predictability of embedded real-time systems, especially in parallel processors (multi/many-cores, GPUs), through the use of compilation, static analysis, and static scheduling techniques.

Christine Rochange received the Ph.D. degree in computer science from the University of Toulouse, Toulouse, France, in 1993. She is a Professor with IRIT Université Toulouse 3, Toulouse. Her research interests are on static WCET analysis, more specifically on the modeling of processor and GPU architecture.