A Framework for Calculating WCET Based on Execution Decision Diagrams

ZHENYU BAI, HUGUES CASSÉ, MARIANNE DE MICHIEL, THOMAS CARLE, and CHRISTINE ROCHANGE, CNRS - IRIT - University of Toulouse, France

Due to the dynamic behaviour of acceleration mechanisms such as caches and branch predictors, static Worst-Case Execution Time (WCET) analysis methods tend to scale poorly to modern hardware architectures. As a result, a trade-off must be found between the duration and the precision of the analysis, leading to an overestimation of the WCET bounds. In turn, this reduces the schedulability and resource usage of the system. In this paper we present a new data structure to speed up the analysis: the *eXecution Decision Diagram* (XDD), which is an ad-hoc extension of *Binary Decision Diagrams* tailored for WCET analysis problems. We show how XDDs can be used to represent efficiently execution states in a modern hardware platform. Moreover, we propose a new process to build the *Integer Linear Programming* system of the *Implicit Path Enumeration Technique* using XDD. We use benchmark applications to demonstrate how the use of an XDD substantially increases the scalability of WCET analysis and the precision of the obtained WCET.

CCS Concepts: • **Computer systems organization → Real-time systems**.

Additional Key Words and Phrases: static WCET analysis, pipeline analysis, variable latencies, timing anomalies

1 INTRODUCTION

In order to ensure the timely execution of hard real-time applications, scheduling analysis techniques must consider safe upper bounds on the possible execution durations of tasks or runnables, which are referred to as *Worst-Case Execution Times* (WCET). Various approaches have been developed to derive such bounds [26]. Those based on static analysis techniques aim at computing safe upper bounds, provided their knowledge of the underlying hardware platform is accurate enough. However, it is also desirable to have as-tight-as-possible WCET bounds since over-estimations may lead to over-dimensioning the system, with consequent waste of precious processor resources. Besides, the complexity and the scalability of analysis are negatively correlated. The difficulty of computing precise execution patterns is sometimes a concern, so a trade-off between precision and analysis time must be found.

The complexity of the control flow of nowadays programs is far too high to enable the analysis to consider the explicit execution paths. Therefore, static WCET analyses split the code of a task into *basic blocks* (BB) – short instruction sequences with no branches in except at the entry and no branches out except at the exit. Then, the global WCET is derived from the individual WCETs of BBs. The most common approach to static WCET analysis is the *Implicit Path Enumeration Technique* (IPET) which consists of four steps in the latest implementation [18]:

(1) *path analysis* – scans the application code to isolate BBs and to derive some execution properties such as loop bounds or infeasible paths,

Authors' address: Zhenyu Bai, zhenyu.bai@irit.fr; Hugues Cassé, hugues.casse@irit.fr; Marianne De Michiel, marianne.de-michiel@irit.fr; Thomas Carle, thomas.carle@irit.fr; Christine Rochange, christine.rochange@irit.fr, CNRS - IRIT - University of Toulouse, 118 route de Narbonne, Toulouse, France, 31062.

1539-9087/2022/8-ART111 $15.00

https://doi.org/https://doi.org/10.1145/3476879

(2) *history-based hardware analysis* – captures the behavior of mechanisms such as caches, and branch predictors,
(3) *local timing analysis* – computes the individual WCET of BBs
(4) *global timing analysis* – determines the WCET for the whole task using an Integer Linear Program (ILP) that maximises the execution time over all the possible execution paths.

Several methods have been proposed to estimate the WCET of a sequence of assembly-level instructions taking into account the processor pipeline and the hardware accelerator components: some are based on abstract interpretation techniques [25] while others use *Execution Graphs* (XG) that capture the timing semantics of a sequence of instructions as they go through the processor pipeline [17, 23]. The issues that we address in this paper are inherent to the WCET computation problem, regardless of the chosen method. We demonstrate our solution on the XG based approach developed in [23] but we believe that our ideas are generic enough to be applied to the other approaches.

Motivation. One difficulty when determining the WCET of a BB (step 3 in the process described above) is that the latency of an operation can have several values. For example, the time needed to fetch an instruction depends on whether it causes a hit or a miss in the instruction cache. Similarly, the execution latency of an instruction may be variable: a memory load may hit or miss in the data cache, the calculation time of a multiplication may depend on the operand values, etc. As for branches, the delay to start fetching the target instruction depends on the branch prediction. These latencies are obtained from preliminary analyses performed in step 2. Whenever several latency values have been found possible (e.g. when the cache analysis was not able to classify an access as *AlwaysHit* or *AlwaysMiss*), one might be tempted to consider the largest value as the worst case. However, it has been shown that processors, in particular those that implement advanced mechanisms to enhance average performance, often exhibit so-called *timing anomalies*: a local worst-case latency does not necessarily lead to a global WCET [20, 22]. In other words, considering all unclassified accesses to the cache as cache misses might underestimate the WCET. As a consequence, the only safe approach is to consider all possible combinations of Hit/Miss over the set of accesses to the cache. Another important issue is that the execution time of instructions may vary by several orders of magnitude in different execution contexts: considering the worst case in all contexts may lead to large over-estimation of the global WCET. This problem is sometimes called *context dependent execution time* [26]. Unfortunately, considering all combinations might lead to consider an exponential number of cases. This would significantly lengthen the analysis while, as mentioned above, the analysis time can be a concern. Besides, we have found out that, in practice, many combinations of instruction latencies eventually lead to the same result. This is mainly due to the pipeline *hiding* some local delays. In fact, the pipeline is designed to *hide* local delays, which leads to improve general performance. This observation guided us to a new approach that reduces the number of times by exploiting the fact that several different combinations produce the same or similar execution patterns.

Contribution. In this paper, we present an efficient solution to estimate the execution times of BBs in the presence of variable latencies that we call *events*, and to use the set of execution times for different *configurations*[1] of events to tighten the overall WCET. The main idea is to factorize the computation performed on an XG to obtain the execution time of a BB. By embedding the configurations of events and their effects on the BB time inside our representation, we are able to benefit from the latency hiding properties of the pipeline and to speed up the analysis. Our approach is based on a refinement of *Binary Decision Diagrams* (BDD) [1, 21] that we call *eXecution Decision Diagram* (XDD). We prove that using XDD is functionally equivalent to the existing XG evaluation method that analyzes exhaustively all configuration of events. We also propose a new process to build the ILP system so

[1] We call configuration a particular combination of events happening at run-time. The term configuration does not imply that the corresponding combination of events is decided by the programmer.

as to leverage the precise information provided by our XDD analysis on basic blocks. We report experimental results showing that using XDD and this new ILP building method provides a significant improvement on the scalability and the precision of the WCET analysis.

Outline. Section 2 provides background information on the WCET analysis of sequences of instructions. Section 3 introduces an initial implementation of XDDs. Section 4 presents how to use XDDs to generate the ILP system. Experimental results are reported and discussed in Section 5. Section 6 reviews related work and Section 7 concludes the paper and discusses plans for future work.

2 BACKGROUND

This section presents the fundamental concepts used to compute the worst-case duration of instruction sequences. Computing the worst-case duration of BBs encompasses the program representation, the method to compute the execution time and the modeling of the underlying hardware behaviour.

2.1 Control Flow Graph

The set of machine instructions is denoted $\mathcal{I}$ and the set of sequences of instructions $\mathcal{I}^*$. A program is represented using *Control Flow Graphs* (CFG) $G = \langle V_{\text{CFG}}, E_{\text{CFG}}, \epsilon \rangle$, where:

- $V_{\text{CFG}} \in \mathcal{I}^*$ is the set of *basic blocks* (BB). A BB is a sequence of instructions from $\mathcal{I}$ such that the control flow can (a) enter the BB only through its first instruction and (b) leave the BB only through its last instruction,
- $E_{\text{CFG}} \subset V_{\text{CFG}} \times V_{\text{CFG}}$ is the set of edges representing the execution flow between BBs,
- $\epsilon \in V_{\text{CFG}}$ is a unique BB without predecessor which represents the entry point of the program.

We consider that G is connected: there exists a path from ϵ to each vertex of V_{CFG}. An edge of E_{CFG}, from a BB a to a BB b, is denoted $a \rightarrow b$.

2.2 Execution Graphs

Pipeline Let us consider an m-stage pipelined processor. The set of its pipeline stages is denoted by $S = [S_1, S_2, ..., S_m]$. The execution of BB $a \in V_{\text{CFG}}$ that consists in the sequence of instructions $I = [I_1, I_2, ..., I_n]$, $I_i \in \mathcal{I}$ on that processor can be represented by an Execution Graph (XG) [23].

An Execution Graph (XG) is a graph $(V_{\text{XG}}, E_{\text{XG}})$ whose vertices $V_{\text{XG}} \subseteq I \times S$ are pairs $[I_i/S_j]$ representing the processing of instruction I_i in stage S_j.

Each vertex v is assigned a latency $\lambda_v \in \mathbb{N}$ that represents the time spent by the instruction in the pipeline stage. We denote by $\alpha \in V_{\text{XG}}$ the first vertex of the first instruction, $[I_1/S_1]$, and by ω the last vertex of the last instruction of the BB, $[I_n/S_m]$.

Edges $E_{\text{XG}} \subset V_{\text{XG}} \times V_{\text{XG}}$ represent timing dependencies: instructions must traverse pipeline stages in a fixed order according to the hardware architecture, instructions are fetched in the program order, some instruction pairs exhibit data dependencies, instructions must wait for a free slot before being inserted into buffers, etc. An edge $v \rightarrow w \in E_{\text{XG}}$ can be solid or dotted: a solid edge means that w can only start after the end of v while a dotted edge means that w can start at the same time as v but not earlier. Dotted edges can express superscalarity, e.g. two instructions being decoded at the same cycle but not out of order. The nature of an edge is represented by $\delta_{v \rightarrow w} = 0$ if $v \rightarrow w$ is dotted, $\delta_{v \rightarrow w} = 1$ otherwise. Note that an XG cannot contain any cycle.

The *ready time* of a vertex $w \in V_{\text{XG}}$ is denoted by ρ_w and is computed as follows:

$$\begin{aligned} \rho_\alpha &= 0 \\ \forall w \neq \alpha, \rho_w &= \max_{v \rightarrow w \in E_{\text{XG}}} \rho_v + \delta_{v \rightarrow w} \times \lambda_v \end{aligned} \tag{1}$$

This computation is repeated for each vertex following a topological ordering of the graph. At the end of computation, the time spent by the BB in the pipeline may be computed by:

$$t = \rho_\omega + \lambda_\omega$$

Note that this calculus assumes that the pipeline is empty when the block starts its execution, so that instructions cannot be delayed by earlier instructions that might occupy hardware resources (pipeline stages, functional units, buffer slots, etc.) or create further dependencies. In [23], a node has several ready times related to the time at which each resource is released by earlier instructions. Given that these additional times are computed exactly the same way as in Equation 1, we omit them in this paper for the sake of simplicity.

Furthermore, the computation of a BB's execution time as presented above would be pessimistic since it does not account for the overlapping execution of successive basic blocks in the pipeline. To enhance accuracy, it is recommended to build an execution graph for each edge $a \rightarrow b \in E_{\text{CFG}}$, including the instructions of both BBs a and b in sequence. It is then possible to derive an execution time $t_{a\rightarrow b}$ for each predecessor of b in the CFG. This time is computed as the delay between the processing of the last instruction of a in the last pipeline stage (denoted by $\widetilde{\omega}$) and the processing of the last instruction of b in the last pipeline stage (ω):

$$t_{a\rightarrow b} = \rho_\omega + \lambda_\omega - (\rho_{\widetilde{\omega}} + \lambda_{\widetilde{\omega}})$$

2.3 Events

An *event* represents any occurrence of a variable XG node processing time. This encompasses the effect of hardware accelerators such as caches or branch predictors but also variable-latency instructions such as multiplications, the execution time of which often depends on the operand values.

An event $e \in \mathcal{E}$ is a tuple $\langle I_i,\ S_j,\ t_e,\ x_e \rangle$ where:

- $I_i \in \mathcal{I}$ is the instruction impacted by the event,
- $S_j \in S$ is the pipeline stage in which the event occurs,
- $t_e \in \mathbb{N}$ is the cost of the event (in cycles) that is applied to individual XG node $[I_i/S_j]$ if the event is active.
- x_e is an expression that represents an upper bound on the number of occurrences of the event. This bound can be found through history-based hardware analyses: for example, the cache persistence analysis [10] provides bounds of occurrences of cache access events. However, bound may be unknown ($x_e = \infty$). If this bound is not ∞, it can leverage the ILP formulation to tighten the WCET [18].

Note that when an XG node may have several (more than two) different latency values, as many events as required are attached to it.

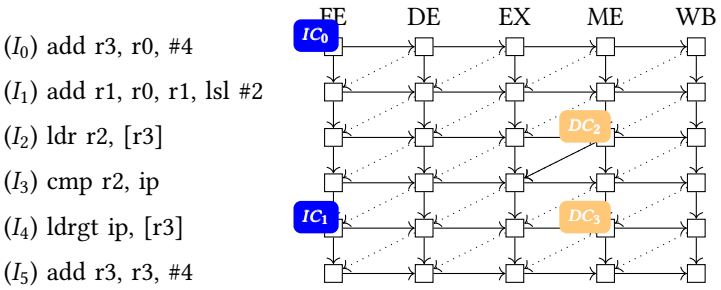
Figure 1 shows an example XG that represents the execution of a sequence of two short BBs, a and b, in a 5-stage (FE – fetch, DE – decode, EX – execute, ME – memory, WB – write-back) in-order pipeline. Basic block a spans from I_0 to I_4 and b only contains instruction I_5. Instructions are shown on the left of the XG, each facing the vertices that represent its traversal of the pipeline. Pipeline stages are shown on the upper row. Events related to the instruction (resp. data) cache behaviour are labeled by IC_x (resp. DC_x). They are attached to vertices that stand for an instruction fetch or a memory data access when a cache miss is possible (as found by cache analysis).

2.4 WCET of a BB in the presence of events

As explained in the introduction, computing the WCET of a basic block by assuming that all the events actually occur (and then systematically accounting for their cost) would be unsafe because of potential timing anomalies [20, 22]. As a consequence, we must compute the BB execution time for every possible combination of events.

We denote by $\mathcal{E}$ the set of events potentially occurring during the execution of a sequence of two BBs and by $|\mathcal{E}|$ its cardinality. A *configuration* of events, $\gamma \in \Gamma : \mathcal{E} \rightarrow \{0, 1\}$, is a function indicating whether an event $e \in \mathcal{E}$

Fig. 1. An xg decorated with events



is active ($\gamma(e) = 1$) or not ($\gamma(e) = 0$). For each configuration $\gamma \in \Gamma$, the latencies of xg vertices must be adjusted to reflect the additional delays due to active events. Assuming that all events are independent, the xg has to be recomputed as many as $|\Gamma| = 2^{|\mathcal{E}|}$ times.

We denote by $T : \Gamma \rightarrow \mathbb{N}$, the domain representing a time duration for each configuration of events. Now, the execution time can be expressed as a function $t^* \in T$ that returns a different value for each configuration in Γ.

Let $\lambda_v(\gamma)$ denote the latency of xg vertex v in configuration γ. $\lambda_v(\gamma)$ is the sum of costs of the events attached to v that are active in γ. The computation of the xg can now be reformulated as:

$$\begin{aligned} \rho^*_\alpha(\gamma) &= 0 \\ \forall w \neq \alpha, \rho^*_w(\gamma) &= \max_{v \rightarrow w \in E_{\text{xg}}} \rho^*_v(\gamma) + \delta_{v \rightarrow w} \times \lambda_v(\gamma) \end{aligned} \tag{2}$$

2.5 Example

Let us consider the xg in Figure 1 and assume that the cost of every event is 10 cycles (latency to access the main memory through the cache) while the default latency of every xg node is 1 cycle. If $\mathcal{E}_v$ represents the set of events of $\mathcal{E}$ that are attached to an xg node v, we have:

$$\lambda_v(\gamma) = \max(1, \sum_{e \in \mathcal{E}_v} 10.\gamma(e))$$

The analysis starts with:

- $\rho^*_{[I_0/FE]}(\gamma) = 0$
- $\lambda_{[I_0/FE]}(\gamma) = \max(1, 10.\gamma(IC_0))$
- $\rho^*_{[I_0/DE]}(\gamma) = \rho^*_{[I_0/FE]}(\gamma) + \lambda_{[I_0/FE]}(\gamma) = \max(1, 10.\gamma(IC_0))$
- ...
- $\rho^*_{[I_2/EX]}(\gamma) = 3 + \max(1, 10.\gamma(IC_0))$
- $\rho^*_{[I_2/ME]}(\gamma) = 4 + \max(1, 10.\gamma(IC_0))$
- ...
- $\rho^*_{[I_3/DE]}(\gamma) = 3 + \max(1, 10.\gamma(IC_0))$

When a vertex, such as $[I_3/EX]$, has multiple predecessors, Equation 2 introduces a *max* that can be simplified:

$$\begin{aligned}\rho^*_{[I_3/EX]}(\gamma) &= max(4 + max(1, 10\gamma(IC_0)),\ 3 + max(1, 10\gamma(IC_0)),\\ &\qquad 4 + max(1, 10\gamma(IC_0)) + max(1, 10\gamma(DC_2)))\\ &= 4 + max(1, 10\gamma(IC_0)) + max(1, 10\gamma(DC_2))\\ &= 6 + 9\gamma(IC_0) + 9\gamma(DC_2)\end{aligned}$$

However this is not always possible. For example:

$$\begin{aligned}\rho^*_{[I_4/EX]}(\gamma) &= max(4 + max(1, 10\gamma(IC_0)) + max(1, 10\gamma(DC_2)),\\ &\qquad 3 + max(1, 10\gamma(IC_0)) + max(1, 10\gamma(IC_1)),\\ &\qquad 4 + max(1, 10\gamma(IC_0)) + max(1, 10\gamma(DC_2)))\\ &= 3 + max(1, 10\gamma(IC_0)) + max(1 + max(1, 10\gamma(DC_2)), max(1, 10\gamma(IC_1)))\\ &= 6 + 9\gamma(IC_0) + max(8\gamma(IC_1), 9\gamma(DC_2))\end{aligned}$$

In practice, computing the block's execution time for the 2^4 possible event configurations leads to less than 2^4 different results. This is due to the structure of the pipeline that enables a partial absorption of vertex latencies: the timing variability induced by an event in one part of the pipeline can be compensated by another event in another part of the pipeline, resulting in the same overall execution time regardless of the occurrence or not of the first event. In our model this absorption is expressed by the *max* function in Equation 1.

The computation of $\rho^*_{[I_3/EX]}(\gamma)$ was simplified using integer arithmetic properties. Implementing it as-is would require the use of symbolic calculus that (a) is time-costly and (b) does not guarantee minimal representation. As an alternative, we introduce a data structure, named *Execution Decision Diagram* (XDD), that:

- is equivalent to the representation of time as symbolic expressions;
- takes advantage of possible simplifications due to the pipeline structure;
- can be easily imported into the ILP formulation of the global WCET computation, i.e. that can be used as T.

3 EXECUTION DECISION DIAGRAMS

An eXecution Decision Diagram (XDD) is a data structure that represents a set of times caused by different configurations of events. In other terms, XDDs are a compact representation of the T domain, enabling simplification of expressions derived in the analysis of an XG. Unlike symbolic calculus, XDDs are specialized to perform efficiently the operations that we need: the maximum and the addition.

3.1 Definitions

An XDD can be seen as a Binary Decision Diagram [1] in which Boolean variables are replaced by events and Boolean leaves by possible times.

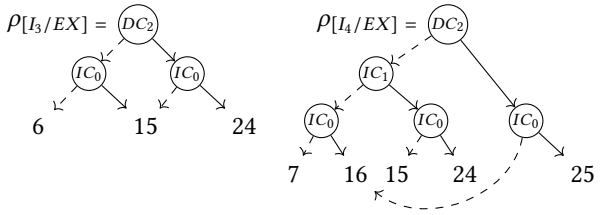
Definition 3.1. An XDD is defined recursively by:

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

with $k \in \mathbb{Z}$, $e \in \mathcal{E}$ and $\overline{f}, f \in \text{XDD}$.

A $\text{NODE}(e, \overline{f}, f)$ represents alternative times depending on the occurrence of event e: f if event e is active and $\overline{f}$ otherwise. A $\text{LEAF}(k)$ represents a constant time $k \in \mathbb{Z}$. The path from the top node to a $\text{LEAF}(k)$ determines the configuration of events that results in the leaf time.

Example. The following XDDs represent $\rho^*_{[I_3/EX]}$ and $\rho^*_{[I_4/EX]}$ from the example in Section 2.5. Events are represented in circles and solid (resp. dashed) edges correspond to the activation (resp. inactivation) of events.



While $\rho[I_3/EX]$ is straight-forwardly built by evaluating all combinations of DC_2 and IC_0, $\rho_{[I_4/EX]}$ benefits from factorizations in the XDD building: IC_1 node on the right side of DC_2 has been removed. Indeed, in $\rho^*_{[I_4/EX]} = 6 + 9\gamma(IC_0) + max(8\gamma(IC_1), 9\gamma(DC_2))$, the term $8\gamma(IC_1)$ is useless when $\gamma(DC_2) = 1$ as $8\gamma(IC_1) < 9\gamma(DC_2)$ whatever the value of $\gamma(IC_1)$. Unlike XDDs, this property cannot be used to simplify the symbolic expression of $\rho^*_{[I_4/EX]}$.

Instantiation. The basic use of an XDD is to evaluate a time, for a given particular configuration of events. Based on the structure of the XDD, we define the *instantiation* of XDD for a configuration $\gamma \in \Gamma$ as:

Definition 3.2. $\forall f \in \text{XDD}, \gamma \in \Gamma$,

$$f^{[\gamma]} = \begin{cases} k & if\ f = \text{LEAF}(k) \\ \overline{g}^{[\gamma]} & if\ f = \text{NODE}(e, \overline{g}, g) \wedge \neg\, \gamma(e) \\ g^{[\gamma]} & if\ f = \text{NODE}(e, \overline{g}, g) \wedge \gamma(e) \end{cases}$$

The instantiation determines the leaf that corresponds to a particular configuration. When a leaf is reached, the result is the leaf value. For any other node, the alternative (branch) that matches the configuration is selected and the search continues downward in the XDD. Note that the XDD node for an event is replaced by one of its sub-XDDs when both alternatives are equal.

3.2 XDD Canonicity

We now present the properties that ensure the efficiency of XDDs.

Order. As for BDDs, an order on the events is necessary to ensure a canonical representation. This order can also have a significant impact on the performance of XDD analysis. For now, we consider that there is a total order on $\mathcal{E}$ denoted by $\preceq$:

$$\forall e_1, e_2 \in \mathcal{E}, e_1 \preceq e_2 \vee e_2 \preceq e_1.$$
$$\forall e_1, e_2 \in \mathcal{E}, e_1 \preceq e_2 \wedge e_2 \preceq e_1 \Rightarrow e_1 = e_2.$$
$$\forall e_1, e_2, e_3 \in \mathcal{E}, e_1 \preceq e_2 \wedge e_2 \preceq e_3 \Rightarrow e_1 \preceq e_3$$

This order is used in the XDD to structure the chain of nodes. $\forall e_1 \neq e_2 \in \mathcal{E}$ with $e_1 \preceq e_2$, the nodes built on e_1 in the XDD must be deeper than the nodes built on e_2. To enforce that the leaves are the deepest, we define $e_\perp$, satisfying $\forall e \in \mathcal{E} \backslash \{e_\perp\}$, $e_\perp \preceq e$. To simplify the notation, we define the function evt : XDD $\rightarrow \mathcal{E}$ s.t. $evt(\text{NODE}(e, \overline{g}, g)) = e$ and $evt(\text{LEAF}(k)) = e_\perp$. We used an order based on the address of the instructions hosting the event. A precise discussion about this order can be found in [2].

To enforce this order among XDD nodes, we define an invariant $Order(f)$ with $f \in$ XDD :

Definition 3.3. $\forall f \in \text{XDD},\ Order(f) =$

$$\begin{cases} \top & if\ f = \text{LEAF}(k) \\ (evt(\overline{g}) \preceq e) \wedge (evt(g) \preceq e) & \\ \wedge\ Order(\overline{g}) \wedge Order(g) & if\ f = \text{NODE}(e, \overline{g}, g) \end{cases}$$

Compactness. Similarly we impose an invariant property $Comp(f)$ with $f \in$ XDD to ensure the compactness of XDDs: no node with the same sub-XDD on each side should exist.

Definition 3.4. $\forall f \in \text{XDD},\ Comp(f) =$

$$\begin{cases} \top & if\ f = \text{LEAF}(k) \\ (\overline{g} \neq g) \wedge Comp(\overline{g}) \wedge Comp(g) & if\ f = \text{NODE}(e, \overline{g}, g) \end{cases}$$

Canonicity. By combining the invariants for compactness and event ordering, the canonicity invariant $Can(f)$ with $f \in$ XDD is defined by:

Definition 3.5. $\forall f \in \text{XDD},\ Can(f) = Comp(f) \wedge Order(f)$

3.3 XDD operators

Based on the algorithms proposed in [1] for BDDs, we define two XDD operators that are required for the computation of XGs: $\otimes$ and $\oplus$ to implement respectively the addition and the maximum in the XG calculation. In fact, both operators can be derived from the XG operations in T using a common method described below.

The extension $\odot$ of an operation $\boxdot$ consists in combining XDDs according to their nature. If two leaves are added, the result is a leaf whose value is the application of $\boxdot$ on both leaf value **(a)**. If two nodes with the same event are combined, the operation $\boxdot$ is propagated equally on each side of the node **(b)** and **(e)**. If the events are different, the $\boxdot$ is propagated according to the order of events **(c)**, **(d)**, **(f)** and **(g)**. Particularly, applying $\odot$ to an XDD leaf and a node propagates the operation along the children of the node.

It is also worth noting that properties **(b)**, **(c)** and **(d)** guarantee that the compactness invariant *Comp* is respected by $\odot$, and properties **(e)**, **(f)**, **(e)** and **(g)** guarantee that the events ordering invariant *Order* is also respected by $\odot$, meaning that applying $\odot$ to two canonical XDDs produces a canonical XDD.

Using Definition 3.6, we define operator $\otimes$ by replacing $\boxdot$ by the addition and operator $\oplus$ by replacing $\boxdot$ by the maximum operation. As we just noted, it means that both $\otimes$ and $\oplus$ preserve the canonicity of XDDs.

Definition 3.6. Any binary operation on $\mathbb{Z}$, $\boxdot : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, can be extended to an XDD binary operation $\odot : \text{XDD} \times \text{XDD} \to \text{XDD}$ with the following definition:

$$\forall f_1, f_2 \in \text{XDD}, f_1 \odot f_2 = \begin{cases} \text{LEAF}(k_1 \boxdot k_2) & \text{if } f_1 = \text{LEAF}(k_1) \wedge f_2 = \text{LEAF}(k_2) \textbf{(a)} \\ g_1 \odot g_2 & \text{if } f_1 = \text{NODE}(e, \overline{g_1}, g_1) \\ & \wedge f_2 = \text{NODE}(e, \overline{g_2}, g_2) \\ & \wedge \overline{g_1} \odot \overline{g_2} = g_1 \odot g_2 \quad \textbf{(b)} \\ f_1 \odot \overline{g_2} & \text{if } f_2 = \text{NODE}(e_2, \overline{g_2}, g_2) \\ & \wedge (evt(f_1) \preceq e_2) \\ & \wedge ((f_1 \odot \overline{g_2}) = (f_1 \odot g_2)) \quad \textbf{(c)} \\ \overline{g_1} \odot f_2 & \text{if } f_1 = \text{NODE}(e_1, \overline{g_1}, g_1) \\ & \wedge (evt(f_2) \preceq e_1) \\ & \wedge ((\overline{g_1} \odot f_2) = (g_1 \odot f_2)) \quad \textbf{(d)} \\ \text{NODE}(e, \overline{g_1} \odot \overline{g_2}, g_1 \odot g_2) & \text{if } f_1 = \text{NODE}(e, \overline{g_1}, g_1) \\ & \wedge f_2 = \text{NODE}(e, \overline{g_2}, g_2) \quad \textbf{(e)} \\ \text{NODE}(e_2, f_1 \odot \overline{g_2}, f_1 \odot g_2) & \text{if } f_2 = \text{NODE}(e_2, \overline{g_2}, g_2) \\ & \wedge (evt(f_1) \prec e_2) \quad \textbf{(f)} \\ \text{NODE}(e_1, f_2 \odot \overline{g_1}, f_2 \odot g_1) & \text{if } f_1 = \text{NODE}(e_1, \overline{g_1}, g_1) \\ & \wedge (evt(f_2) \prec e_1) \quad \textbf{(g)} \end{cases}$$

3.4 Using an XDD in XG analysis

Equation 2 can be transported in the XDD framework with a straight effect: the computation of the XG for all configurations only requires one pass over the XG. $\oplus$ and $\otimes$ are naturally used but we also need to define the XDD equivalent of λ_v, $v \in V_{\text{XG}}$.

Definition 3.7. We first define $\lambda_e^\# : \mathcal{E} \to \text{XDD}$, converting to an XDD an event e that has a cost of k_e when active and 0 when inactive.

$$\lambda_e^\# = \text{NODE}(e, \text{LEAF}(0), \text{LEAF}(k_e))$$

λ_v, the time spent in an XG node for a particular configuration, can be now represented by $\lambda_v^\#$.

Definition 3.8. If node v undergoes a set of events $\mathcal{E}_v$, $\lambda_v^\#$ is expressed by:

$$\lambda_v^\# = \text{LEAF}(\lambda_v) \otimes \bigotimes_{e \in \mathcal{E}_v} \lambda_e^\#$$

The time spent in a stage is the default time spent in the stage plus the sum of all possible event costs.

Equation 2 is rewritten as:

$$\begin{aligned} \rho_\alpha^\# &= \text{LEAF}(0) \\ \forall w \in V_{\text{XG}}, \rho_w^\# &= \bigoplus_{v \to w \in E_{\text{XG}}} \rho_v^\# \otimes (\delta_{v \to w} \times \lambda_v^\#) \end{aligned} \tag{3}$$

with $\rho_w^\# \in \text{XDD}$. A $\rho_v^\#$ is associated to each XG node, representing the ready time for *all possible event configurations* for this node.

The multiplication by $\delta_{v \to w}$ is in fact a selection operation simply implemented as:

- $f \times \delta_{v \to w} = f$ if $\delta_{v \to w} = 1$

- $f \times \delta_{v \to w} = \text{LEAF}(0)$ if $\delta_{v \to w} = 0$

Finally, we compute the execution time of BB b preceded by a: $t^{\#}_{a \to b} = \rho^{\#}_{\omega} \oslash \rho^{\#}_{\widetilde{\omega}}$. Operator $\oslash$ is defined according to Definition 3.6 with the minus (-) operator as $\boxdot$.

The procedure to apply XDD in XG is similar to the procedure to obtain the symbolic representation shown in Figure 1, by replacing + and *max* by $\otimes$ and $\oplus$. The benefit of XDD over T is that it allows treating events in a compact manner with the compactness defined as above.

In [2], we prove that XDD calculation is equivalent to symbolic representation. Moreover we show how the XDD implementation can be improved (a) by equipping each node with *max* and *min* values of their leaves to cut some calculations, (b) by using memoization of intermediate calculation results and (c) by selecting an adapted event order. Combining these techniques, the XDD computation is very effective. Moreover, the next section presents how the XDDs can be involved in the WCET computation by IPET approach.

4 EVENTS AND IPET METHOD

The *Implicit Path Enumeration Technique* (IPET) [19] is currently the most fruitful WCET calculation approach: the WCET is expressed as the objective function of a maximized *Integer Linear Programming* (ILP) system. The variables are the BB's counters of occurrences and their weights in the objective function represent their worst-case execution times. In addition, the variables are bounded by the flow constraint of the program CFG.

The introduction of events, presented in Section 4.1, changes the IPET formulation and has a noteworthy consequence: the execution time of a BB is a set of possible values instead of a single worst-case execution time. To limit the number of variables required to consider all the possible event configurations, a solution that groups configurations is proposed in Section 4.2 and Section 4.3 shows how precision can be preserved. Finally, Section 4.4 shows how XDDs can be used to implement this approach efficiently.

4.1 Impact of events on the ILP maximization

The basic ILP objective function used to determine a WCET with IPET [19] is the following:

$$C = max \sum_{a \in V_{\text{CFG}}} t_a x_a \tag{4}$$

where C is the WCET, t_a is the execution time in cycles of BB a and x_a is the execution count of a on the WCET path. The variables x_a are constrained by the execution paths (BB chaining in the CFG, loop bounds, infeasible paths, etc).

As seen in Section 2.2, an XG estimates the execution time of a BB after one of its predecessors. Eq. 4 becomes:

$$C = max \sum_{a \to b \in E_{\text{CFG}}} t_{a \to b} x_{a \to b} \tag{5}$$

with $x_{a \to b}$ the number of occurrences of edge $a \to b$ in the WCET path(s)[2] and $t_{a \to b}$ the execution time of b after a resulting from the resolution of the corresponding XG.

In Eq. 5, $t_{a \to b}$ is the unique execution time of BB b along edge $a \to b$. But in the presence of events, this time is not unique anymore and depends on the occurrence of events along $a \to b$. If we consider that all the events on $a \to b$, $\mathcal{E}_{a \to b}$, are independent from each other, BB b can exhibit at most $2^{|\mathcal{E}_{a \to b}|}$ different times along $a \to b$, that is, as many configurations as can be found in $\Gamma_{a \to b}$. Let $x^{\gamma}_{a \to b}$ denote the execution count of each event

[2]The IPET approach does not determine an explicit longest path but instead, BB execution counts that may correspond to several longest execution paths.

configuration $\gamma \in \Gamma_{a\to b}$ and let $t^{\gamma}_{a\to b}$ denote the execution time of b along $a \to b$ under the event configuration γ. Equation 5 becomes:

$$C = max \sum_{a\to b \in E_{\text{CFG}}} \left(\sum_{\gamma \in \Gamma_{a\to b}} t^{\gamma}_{a\to b} x^{\gamma}_{a\to b} \right) \tag{6}$$

To maintain the consistency of the ILP constraints that use variable $x_{a\to b}$, we add a constraint that stipulates that the total number of event configuration occurrences is equal to the execution count of edge $a \to b$ on the WCET path :

$$\forall a \to b \in E_{\text{CFG}}, x_{a\to b} = \sum_{\gamma \in \Gamma_{a\to b}} x^{\gamma}_{a\to b} \tag{7}$$

Now, to tighten the WCET bound even further, we can use the bounds on the events introduced in Section 2.3. For each event e, the following constraint is added to the ILP system:

$$x_e \geq \sum_{\gamma \in \Gamma_{a\to b} \wedge \gamma(e)} x^{\gamma}_{a\to b} \tag{8}$$

where x_e is a bound on event e (possibly an ILP formula referring to other variables of the system).

Now, we can foresee the main challenge of the approach proposed in this paper, as illustrated by the following example: from our experience in WCET calculation for industrial applications, a medium-sized task may be composed of 50,000 BBs and 75,000 edges. Each edge, on average for a middle-range embedded micro-controller, is concerned by 8 events, that is, 256 configurations. For such a medium-sized task, we need $75,000 \times 256$ ILP variables to account for each configuration, that is more than 19 million variables. The size of the ILP system is too large for usual ILP solvers[3] that will either fail, or take too much time. Fortunately, as observed in Section 2, experiments show that the actual number of different execution times for a BB is much below its theoretical maximum. This property is leveraged in the next section to reduce the number of event-related variables.

4.2 Mitigating the explosion of the number of variables

This section presents a method to reduce the number of variables used in the ILP system by taking benefit from (a) the relatively low number of different execution times for a BB, and (b) from the possibility to overestimate the WCET. For the sake of simplicity, we focus on a particular edge $a \to b$ and on the corresponding term in the objective function:

$$X_{a\to b} = \sum_{\gamma \in \Gamma_{a\to b}} t^{\gamma}_{a\to b} x^{\gamma}_{a\to b} \tag{9}$$

The evaluation of this term is related to the rest of the system through the constraints on the edge count variable (Eq. 7) and the bounds x_{e_i} on the events e_i that may occur along $a \to b$ (Eq. 8).

To reduce the number of variables, originally one for each configuration, the set of configurations $\Gamma_{a\to b}$ is partitioned into $P = \{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$. P must meet the following conditions:

- $\Gamma_{a\to b} = \cup_{1\leq i\leq n} \Gamma_i$
- $\forall 1 \leq i \neq j \leq n, \Gamma_i \cap \Gamma_j = \emptyset$
- $\forall 1 \leq i \leq n, \Gamma_i \neq \emptyset$

A variable x_i $(1 \leq i \leq n)$ is assigned to each part Γ_i and represents the sum of the configuration counter variables in Γ_i:

$$\forall 1 \leq i \leq n, x_i = \sum_{\gamma \in \Gamma_i} x^{\gamma}_{a\to b} \tag{10}$$

[3]One can argue that, in the future, ILP solvers will be able to support that amount of variables but in the meantime real-time applications will also evolve and become more complex.

Without any loss of precision, x_i variables can replace $x^{\gamma}_{a\to b}$ in Eq. 7:

$$x_{a\to b} = \sum_{\gamma\in\Gamma_{a\to b}} x^{\gamma}_{a\to b} = \sum_{1\le i\le n} x_i \tag{11}$$

In the same way, the x_i variables can replace $x^{\gamma}_{a\to b}$ in the objective function, that is in the term $X_{a\to b}$ from Eq. 9, that can be re-written as $\widehat{X}_{a\to b}$:

$$\widehat{X}_{a\to b} = \sum_{1\le i\le n} t_i\ x_i \tag{12}$$

with $t_i \in \mathbb{N}$ the time associated to part Γ_i. To ensure the soundness of the objective function, t_i is formulated as:

$$\forall i \in \mathbb{N}, 1 \le i \le n, t_i = \max_{\gamma\in\Gamma_i} t^{\gamma}_{a\to b} \tag{13}$$

Eq. 13 ensures to get an overestimation $\widehat{X}_{a\to b}$ of the actual time contribution of $X_{a\to b}$ to the WCET and therefore to obtain a sound WCET. This is demonstrated below:

Proof.

$$\begin{aligned}
X_{a\to b} &= \sum_{\gamma\in\Gamma_{a\to b}} t^{\gamma}_{a\to b} x^{\gamma}_{a\to b} \\
&= \sum_{1\le i\le n}\ \sum_{\gamma\in\Gamma_i} t^{\gamma}_{a\to b} x^{\gamma}_{a\to b} \\
&\le \sum_{1\le i\le n}\ \sum_{\gamma\in\Gamma_i} t_i\ x^{\gamma}_{a\to b} && (using\ Eq.\ 13) \\
&= \sum_{1\le i\le n} t_i \left(\sum_{\gamma\in\Gamma_i} x^{\gamma}_{a\to b}\right) \\
&= \sum_{1\le i\le n} t_i\ x_i && (using\ Eq.\ 10) \\
&= \widehat{X}_{a\to b}
\end{aligned}$$

□

The demonstration is straightforward. The only tricky part of it is the injection of $\le$, which holds because t_i is the maximum of $t^{\gamma}_{a\to b}$ with $\gamma \in \Gamma_i$ (Eq. 13).

4.3 Bounding the part counters

4.3.1 Maximization effect. Let us consider a partition strictly ordered by $t_i : P = [\Gamma_1, ...\Gamma_n]$ and $\forall 1 \le i < n,\ t_i \le t_{i+1}$, and the corresponding term in the objective function $\sum t_i x_i$. The only constraint on counters x_i is $\sum x_i = x_{a\to b}$. As is, the maximization process by the ILP solver (based on a simplex resolution) would push the counter x_n corresponding to the partition part having the maximum t_i, that is t_n, to the value of $x_{a\to b}$, letting all other $x_{1\le i<n}$ to 0. Partition P would be equivalent to a partition with a single class: $[\Gamma^* = \cup_{1\le i\le n}\Gamma_i = \Gamma_{a\to b}]$, which would come down to the traditional approach.

More generally, let $P = [\Gamma_1, ..., \Gamma_n]$ be a strictly ordered partition with Γ_j unbounded and parts $\Gamma_{j<i\le n}$ bounded. The $x_{j<i\le n}$ will take as much as possible from $x_{a\to b}$ and x_j will get the remainder: $x_j = x_{a\to b} - \sum_{j<i\le n} x_i$. On

the contrary, the variables $x_{i<j}$ will be set to 0 and an equivalent partition would be:

$$\left[\left(\Gamma^* = \bigcup_{1\leq i\leq j} \Gamma_i\right), \Gamma_{j+1}, ..., \Gamma_n\right]$$

where Γ^* is called the *nullified part*. The new partition with the *nullified part* does not cause any precision loss in the ILP system as the $x_{1\leq i<j}$ are set to 0.

Considering this property of the ILP system, it becomes desirable to bound part counters x_i to prevent the *maximization effect* as much as possible. The event constraints introduced in Section 2.3, bounding the configuration counters, can be used to bound part counters. However, all events are not bounded according to the results of history-based hardware analyses. A configuration that has only inactive or active but unbounded events ($x_e = \infty$) is considered as unbounded because no event constraint can be applied to it.

$$unbounded(\gamma) \Leftrightarrow \forall e \in \mathcal{E}, \neg\gamma(e) \vee x_e = \infty$$

As a result, a part containing such a configuration γ is also unbounded since the counter variable of this configuration is unbounded, so the counter variable of the part cannot be bounded in any way. In such a case, the only constraint which applies to the counter of an unbounded part is that the sum of all part counters equals the occurrence count of edge $a \rightarrow b$ (Eq. 11).

Generally, if an unbounded configuration γ^* exists, the t^* of the *nullified part* Γ^* is at least t^{γ^*}. Therefore, keeping in mind the *maximization effect*, the *nullified part* Γ^* of a set of configurations $\Gamma_{a\rightarrow b}$ can be identified with the largest time t^* of all unbounded configurations.

$$\Gamma^* = \{\gamma \in \Gamma_{a\rightarrow b} \mid t^{\gamma}_{a\rightarrow b} \leqslant t^*\} \text{ with } t^* = \max_{\gamma\in\Gamma_{a\rightarrow b} \wedge\ unbounded(\gamma)} t^{\gamma}_{a\rightarrow b}$$

4.3.2 Bounding with events. Eq. 11 shows how the variables x_i generated for the parts Γ_i replace the configuration variables $x^{\gamma}_{a\rightarrow b}$ and are constrained by the edge counter $x_{a\rightarrow b}$. The configuration variables have a second source of constraints: the bounded events in Eq 8. To benefit from these constraints, we have to replace configuration variables $x^{\gamma}_{a\rightarrow b}$ with part variables x_i but there is not always an exact match between configuration variables involved in an event constraint and the part variables.

In the simplest case, a part Γ_i may be *entirely bounded* by an event $e \in \mathcal{E}$, meaning that e is active for all configurations of Γ_i. Eq. 8 can be rewritten as:

$$\begin{aligned} & x_e \geq \sum_{\gamma\in\Gamma_{a\rightarrow b} \wedge\ \gamma(e)} x^{\gamma}_{a\rightarrow b} \\ \Longleftrightarrow\ & x_e \geq \left(\sum_{\gamma\in\Gamma_i \wedge\ \gamma(e)} x^{\gamma}_{a\rightarrow b}\right) + \left(\sum_{\gamma\in(\Gamma_{a\rightarrow b}\setminus\Gamma_i) \wedge\ \gamma(e)} x^{\gamma}_{a\rightarrow b}\right) \\ \Longleftrightarrow\ & x_e \geq x_i + \left(\sum_{\gamma\in(\Gamma_{a\rightarrow b}\setminus\Gamma_i) \wedge\ \gamma(e)} x^{\gamma}_{a\rightarrow b}\right) \end{aligned} \tag{14}$$

This process can be iterated for each part *entirely bounded* by the event e resulting in:

$$x_e \geq \left(\sum_{j<i\leq n \wedge\ \forall\gamma\in\Gamma_i, \gamma(e)} x_i\right) + \left(\sum_{\gamma\in(\Gamma_{a\rightarrow b}\setminus B_i) \wedge\ \gamma(e)} x^{\gamma}_{a\rightarrow b}\right) \tag{15}$$

with B_i the set of configurations included in parts entirely bounded by e, $B_i = \cup_{j<i\leq n \wedge\ \forall\gamma\in\Gamma_i, \gamma(e)}\Gamma_i$.

However, there may be parts Γ_i that contain some configurations where e is active and others where e is inactive. This means that x_i does not exactly match a subset of configurations involved in Equation 8 and cannot be inserted as is in the constraint. Such a part is said to be *partially bounded*. These unbounded configuration variables are exposed in the sum of $x^{\gamma}_{a\to b}$ of Eq. 15. Unfortunately, these variables cannot be separately processed because they are already accounted in their own part counter variable x_i but we identified two strategies to get rid of them:

Removing a variable from a constraint. In the maximized ILP system used for WCET calculation, removing a variable with positive coefficient from a constraint as Eq. 14 leads to an enlargement of the original convex hull represented by the set of constraints. This hull exposes more solutions to the maximization and ensures the solution to be bigger or equal to the original one. In the general case, removing a variable could lead to an unbounded ILP system but not when removing variables from event constraints. Applying this property to remove the *nullified part*, Eq. 14 becomes:

$$x_e \geq \sum_{j<i\leq n \ \wedge \ \forall \gamma \in \Gamma_i, \gamma(e)} x_i \tag{16}$$

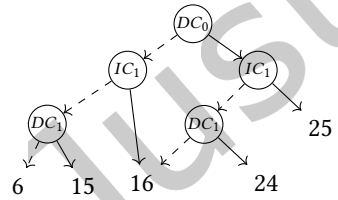
Splitting a part. Sometimes, there is no event fully bounding a part Γ_i but the part can be split into a set of sub-parts $\Gamma_{i.e_1}$, ..., $\Gamma_{i.e_m}$, each one fully bounded by a different event e_1, e_2, ..., e_m. In fact, this comes to change the partition into:

$$[\Gamma^*, \Gamma_{j+1}, ..., \Gamma_{i.e_1}, ..., \Gamma_{i.e_m}, \Gamma_{i+1}, ..., \Gamma_n]$$

The split parts counter variables $x_{i.e_k}$ can then be inserted in Equation 16 as they are fully bounded by the splitting events e_k. However, this technique should be used moderately: it may introduce a lot of new variables and the *maximization effect* with the relaxed constraints between the split variables with the same time is prone to generate overestimation.

The next section uses the properties exposed here to generate an effective partition and the objective function using XDD.

4.4 Partitioning and Generating the ILP System using XDDs



DC_1 is unbounded.

$$\begin{aligned}
P &= \{\Gamma_6, \Gamma_{15}, \Gamma_{16}, \Gamma_{24}, \Gamma_{25}\} \\
\Gamma_6 &= \{\overline{DC_0}.\overline{IC_1}.\overline{DC_1}\} \\
\Gamma_{15} &= \{\overline{DC_0}.\overline{IC_1}.DC_1\} \\
\Gamma_{16} &= \{\overline{DC_0}.IC_1, DC_0.\overline{IC_1}.\overline{DC_1}\} \\
\Gamma_{24} &= \{DC_0.\overline{IC_1}.DC_1\} \\
\Gamma_{25} &= \{DC_0.IC_1\}
\end{aligned}$$

Fig. 2. Example of XDD representation of times

As a result of XG analysis, the XDDs record the relationship between configurations and execution times, which is necessary to build the ILP system according to Eq. 6, Eq. 7 and Eq. 8. As a LEAF(k) of an XDD and all the paths from the root to it represent the set of configurations leading to the execution time k, grouping configurations by time comes naturally to mind. This is illustrated in Figure 2 where the block time XDD is shown on the left. The

right of the figure exposes the partition according to the times (the leaves of the XDD) and the contents of the parts: sets of event paths in the XDD that may be overlined when an inactive edge is taken, or not for active edges.

We have shown in 4.3.1 that the *nullified part* groups some parts and therefore reduces the number of variables. Grouping by time also makes it easier to find the *nullified part*. Recall that inactive events on a path are represented by the inactive branches (left branch, or $\overline{f}$) of nodes $\text{NODE}(e, \overline{f}, f)$; active events are represented by the active branches (right branch, or f). If a path leading to $\text{LEAF}(k)$ contains only inactive or unbounded active events, at least one configuration leading to $\text{LEAF}(k)$ is unbounded. Such a path is called an *unbounded path*. Recall that the *nullified part* is identified by the largest time t^* of all unbounded configurations. We denote by γ^* the unbounded configuration corresponding to time t^*. Finding γ^* is equivalent to finding the greatest k of $\text{LEAF}(k)$ linked to an *unbounded path* which is easy thanks to the XDD structure. In the example of Figure 2, assuming that only DC_1 is unbounded, the nullified part is made of $\Gamma_6 \cup \Gamma_{15}$: other parts are bounded by DC_0 or IC_1.

After removing the *nullified part*, the remaining parts need to be bounded as precisely as possible. In Section 4.4.2, we propose an efficient graph traversal algorithm on XDDs which provides the bounding information for these parts. The bounding information allows (a) to find directly the *nullified part*, (b) to categorize the parts, which will help to build their constraints, and (c) to generate the constraints on counter variables of parts from the constraints on events. The generation process of constraints and objective function is presented in Section 4.4.3.

4.4.1 Grouping boundable configurations. Previously, we have observed that $\text{LEAF}(k)$ and all paths from the root to it represent the set of configurations leading to execution time k. From the point of view of grouping, a partition by time is naturally formed:

$$P = [\Gamma_t]_{t \in \mathbb{N}} \text{ where } \forall \Gamma_t \in P, \forall \gamma \in \Gamma_t, t^{\gamma}_{a \to b} = t \tag{17}$$

To simplify the re-writing of constraints, we associate two sets to each partition $\Gamma_t, t > t^*$: $\mathcal{E}^{Entire}_{\Gamma_t}$ and $\mathcal{E}^{Partial}_{\Gamma_t}$. $\mathcal{E}^{Entire}_{\Gamma_t} \subset \mathcal{E}_{a \to b}$ records the set of events that are active and bounded for each configuration of Γ_t:

$$\mathcal{E}^{Entire}_{\Gamma_t} = \{e \in \mathcal{E}_{a \to b} \mid x_e \neq \infty \wedge (\forall \gamma \in \Gamma_t, \gamma(e))\} \tag{18}$$

This set corresponds to the event constraints in which the counter variable x_t can simply replace the old counter variables of configurations according to Eq.14.

$\mathcal{E}^{Partial}_{\Gamma_t} \subset \mathcal{E}_{a \to b}$ records the set of events that are active and bounded for at least one configuration of Γ_t:

$$\mathcal{E}^{Partial}_{\Gamma_t} = \{e \in \mathcal{E}_{a \to b} \mid x_e \neq \infty \wedge (\exists \gamma \in \Gamma_t, \gamma(e) \wedge \neg unbounded(\gamma))\} \tag{19}$$

This set corresponds to the event constraints in which the counter variable x_t cannot replace directly old counter variables of configurations. However, relaxed constraints can be built using techniques described in 4.3.2.

4.4.2 Finding bounding information. After grouping configurations by time, the bounding information of each part needs to be computed in the XDD structure. To do so, our heuristic determines:

- the bounding category of each part: *entirely bounded*, *partially bounded*, or *unbounded*,
- if bounded, which events bound the part.

Since exploiting and recording all possible paths in XDDs to answer these two questions is time-consuming, we propose an abstraction of XDD paths by a set of active and bounding events on the path. We define the function Ω_t that collects the set of abstract paths from root to an XDD leaf for part Γ_t:

Definition 4.1. $\forall f \in \text{XDD}, t \in \mathbb{N}, \Omega_t : \wp(\mathcal{E}) \times \text{XDD} \rightarrow \wp(\wp(\mathcal{E}))$

$$\Omega_t(\pi, f) = \begin{cases} \{\pi\} & if\ f = \text{LEAF}(t) \\ \{\emptyset\} & if\ f = \text{LEAF}(k) \wedge k \neq t \\ \Omega_t(\pi \cup \{e\}, g) \cup \Omega_t(\pi, \overline{g}) & if\ f = \text{NODE}(e, \overline{g}, g) \wedge e \neq \infty \\ \Omega_t(\pi, g) \cup \Omega_t(\pi, \overline{g}) & if\ f = \text{NODE}(e, \overline{g}, g) \wedge e = \infty \end{cases}$$

Function Ω_t traverses an XDD downwards, recording the active and bounding events on each path, until a $\text{LEAF}(t)$ is reached. We note that the definition of this function is recursive, while it can be efficiently implemented as a *breadth-first search*-like traversing algorithm.

As we initially do not know the *nullified part*, any part may be unbounded. To find the *nullified part* during the computation of the bounding information, one has to compute the bounding information from the highest leaf to the lowest leaf. Once an unbounded part is found, itself and all lower parts form together the *nullified part*.

The bounding category can be determined by the computation of $\Omega_t(\emptyset, root)$ for each t:

$$entirely\_bounded(\Gamma_t) \Leftrightarrow \mathcal{E}_{\Gamma_t}^{Entire} \neq \emptyset \Leftrightarrow \left(\bigcap_{\pi \in \Omega_t(\emptyset, root)} \pi \right) \neq \emptyset$$

$$unbounded(\Gamma_t) \Leftrightarrow \emptyset \in \Omega_t(\emptyset, root)$$

$$partially\_bounded(\Gamma_t) \Leftrightarrow \neg unbounded(\Gamma_t) \wedge \neg entirely\_bounded(\Gamma_t)$$

Once the bounding category has been determined, $\mathcal{E}_{\Gamma_t}^{Entire}$ or $\mathcal{E}_{\Gamma_t}^{Partial}$ can be computed:

$$\mathcal{E}_{\Gamma_t}^{Entire} = \bigcap_{\pi \in \Omega_t(\emptyset, root)} \pi$$

and:

$$\mathcal{E}_{\Gamma_t}^{Partial} = \bigcup_{\pi \in \Omega_t(\emptyset, root)} \pi$$

4.4.3 Bounding parts with events. This section discusses how the events can be used to bound the parts, depending on their category. Recall that before grouping, the initial constraints are given by Eq. 8.:

$$\sum_{\gamma \in \Gamma_{a \rightarrow b} \wedge \gamma(e)} x_{a \rightarrow b}^{\gamma} = x_e$$

Case 1 – entirely bounded part: As Γ_t is entirely bounded, it must be a subset of all configurations that have $e \in \mathcal{E}_{\Gamma_t}^{Entire}$ active:

$$\forall e \in \mathcal{E}_{\Gamma_t}^{Entire}, \Gamma_t \subset \{\gamma \in \Gamma_{a \rightarrow b} | \gamma(e)\} \tag{20}$$

However, for an event e, there may exist multiple parts that are entirely bounded by e. According to Eq.14, all of them can fit into the event constraint. To generate the event constraint according to Eq.15, we have to record all these parts for each e:

$$\forall e \in \mathcal{E}_{a \rightarrow b}, P_e^{Entire} = \{\Gamma_t \in P \mid e \in \mathcal{E}_{\Gamma_t}^{Entire}\}$$

Note that, P_e^{Entire} is initialized as an empty set for each e and is updated as the bounding information is determined for each part. Finally, with P_e^{Entire}, Eq.15 can be re-written as:

$$\forall e \in \mathcal{E}_{a\to b}, \sum_{\Gamma_t \in P_e^{Entire}} x_t \leq x_e \tag{21}$$

This operation is safe because we are removing some positive terms (second part) from Eq. 15. An overestimation may be introduced because we are not sure that P_e^{Entire} covers all configurations that have e active. i.e. the term that we removed from $Eq.15$. In Figure 2, as DC_0 entirely bounds Γ_{24} and Γ_{25}, the constraint $x_{24} + x_{25} \leq x_{DC_0}$ is added to the ILP system.

Case 2 – partially bounded part: This situation is more complicated: an x_t of a partially bounded Γ_t groups counter variables of configurations in different event constraints. This means that the x_t cannot replace counter variables in Eq. 15. As all events that partially bound the part are recorded in $\mathcal{E}_{\Gamma_t}^{Partial}$, the part is split according to the technique illustrated in 4.3.2, into $\{\Gamma_{t.e} \mid e \in \mathcal{E}_{\Gamma_t}^{Partial}\}$. Since these sub-parts are bounded, their counter variables are counted into P_e^{Entire}, following the same procedure as for entirely bounded parts. In the example of Figure 2, DC_0 entirely bounds Γ_{24} and Γ_{25} but partially Γ_{16}. Γ_{16} is split into parts $\Gamma_{16.DC_0}$ and $\Gamma_{16.IC_1}$ which produces the following constraints:

$$x_{16.DC_0} + x_{24} + x_{25} \leq x_{DC_0}$$
$$x_{16.IC_1} + x_{25} \leq x_{IC_1}$$

Case 3 – unbounded part: As described in 4.4.2, the bounding information is determined for the greatest leaves first. Once the highest unbounded Γ_j is detected, itself and all lowest parts should be sorted into *nullified part*. The counter variable of this *nullified part* x^* is created and no constraint is created so that the *nullified part* is only bounded in the ILP system by the constraint on total execution count of edge $a \to b$. The overall result for the example of Figure 2 is the following term added to the WCET expression in ILP:

$$X_{a\to b} = 15x^* + 16x_{16.DC_0} + 16x_{16.IC_1} + 24x_{24} + 25x_{25}$$

With the constraint:

$$x_{a\to b} = x^* + x_{16.DC_0} + x_{16.IC_1} + x_{24} + x_{25}$$

The most important advantage of our grouping heuristic is that it naturally fits with the structure of XDDs, which makes it easy to implement. Moreover, it provides the bounding information of parts and automatically covers the detection of the *nullified part* so the grouping and the detection of the *nullified part* can be simultaneously performed.

5 EVALUATION

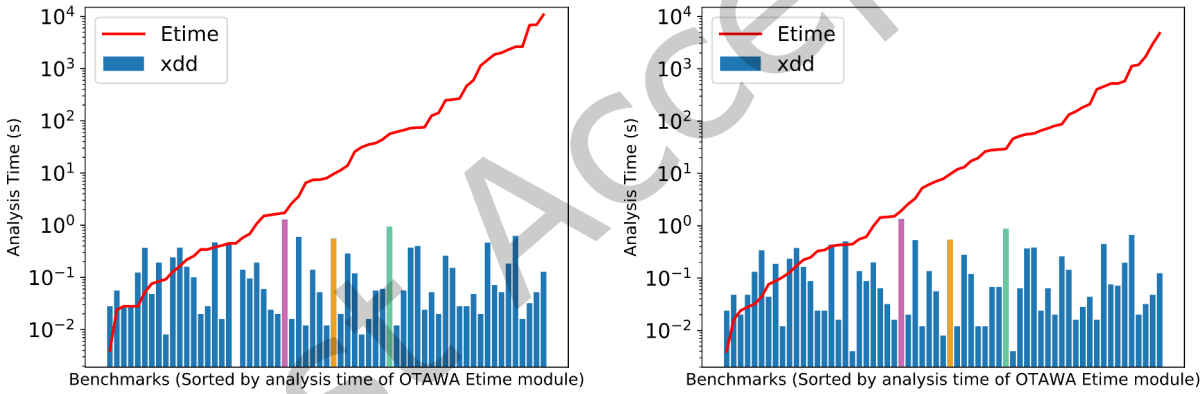
We now present the experiments we performed to evaluate the efficiency of XDDs. We used *OTAWA*, an open toolbox dedicated to static WCET analysis [3], that includes analysis engines able to identify events of instruction cache and data cache. We have implemented the XDD approach and compared it to the approach currently used in *OTAWA*, referred to as *Etime*, which consists in analysing each XG once for each possible event configuration. We first compare the two approaches; then we evaluate the number of nodes and leaves in XDDs as a function of the number of events attached to the basic blocks.

Simple	Complex
5 stages: FE, DE, EX, MEM, WB	4 stages: FE, DE, EX, WB
no fetch queue	fetch queue size = 3
1 instruction/cycle	3 instructions/cycle (3-ways super-scalar)
1 Execution Stage	1 ALU, 1 FPU, 1 MU (including address computation)
Multiplication = 6 cycles, Division = 12 cycles	Multiplication = 2 cycles, Division = 7 cycles
Floating point addition = 10 cycle	Floating point addition = 1 cycles
Floating point multiplication = 12 cycles	Floating point multiplication = 2 cycles
Floating point division = 37 cycles	Floating point division = 7 cycles
2-way 16KB LRU L1 instruction cache, Hit = 1 cycle, Miss = 8 cycles	
2-way 8KB LRU L1 write-back data cache, Hit = 1 cycle, Miss = 8 cycles	

Table 1. Target hardware architecture details.

5.1 Experimental framework

We considered 81% of the TACLe benchmark suite [9]. The binaries are compiled for the ARMv7 instruction set[4]. The remaining 19% had to be discarded due to shortcomings in the current version of the *OTAWA* tool box.



(a) Simple architecture

(b) Complex architecture

Fig. 3. Analysis time.

We modeled two in-order pipelined architectures: one representative of simple embedded processors and a more complex Tricore Aurix-like one. They cover both ends of processor families typically used in embedded systems and will provide an insight into the influence of the pipeline complexity on the XDD computation performance. Table 1 provides more details on both architectures. The simple architecture is composed of a classic 5-stage in-order scalar pipeline able to fetch at most 1 instruction per cycle, and whose execution stage is able to process one instruction at a time. The complex architecture is 3-way superscalar: it can fetch and process at most 3 independent instructions per cycle, thanks to a larger fetch queue, and to the presence of three separate

[4]For further compilation options, please check the Makefile at https://github.com/tcarle/tacle-bench/blob/master/Makefile.cfg.

functional units composing the execution stage. On both architectures, we have L1 caches and independent instruction and data memories buses.

All the experiments were performed on a server composed of 8 Intel Xeon E25 cores (@2.4GHz) sharing 32GB of RAM. The analysis time is measured using the software timer of Linux of 1 ms precision. Our implementation of XDD is single-threaded but multiple experiments were executed in parallel.

The source code and further details on experiments in 5.2 and 5.3 are available on Zenodo[5]. A second version for the experiments in 5.4 is also available on Zenodo[6].

Split threshold. The exhaustive *Etime* algorithm computation capacity is limited by its exponential complexity. We have observed that it generally performs a BB analysis in a reasonable time (total analysis time of each benchmark is less than 3 hours) if the number of events in the BB is less than 15 (thereafter called *split threshold*). To reduce the analysis time when a BB contains too many events, the BB is split according to the split threshold. A direct consequence of this technique is an increase of the overestimation since it does not consider the overlap of BBs inside the pipeline at the split boundaries. Yet, as modelling pipeline operations is never straightforward, we suspect that other overestimating effects may occur but this falls out of the scope of this paper.

A first benefit of XDDs is that the limit on the number of events is pushed significantly further: they are able to support up to 136 events on most of the TACLe benchmarks, covering 99% of their BBs without split. Only `rijndael_enc` and `gsm_dec`, which contain BBs with more than 300 events, require the split threshold to be set to 120 events for the simple architecture and to 100 for the complex architecture. This suggests that we could, in the future, use an adaptive splitting policy instead of a fixed threshold.

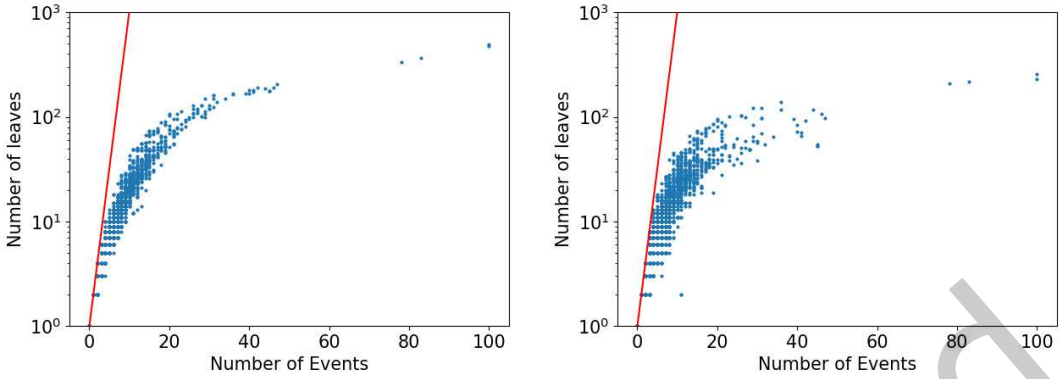
5.2 XG Analysis Time

Figures 3a and 3b plot the XG analysis time of the *Etime* and XDD approaches. The x-axis represents the benchmarks ordered by their analysis time using *Etime*, which provides a raw experimental estimation of their complexity (*Etime* being an exhaustive computation). The y-axis shows the analysis time in logarithmic scale. For both analyses, the split threshold is set to 15 to fit the limitations of *Etime*. The red line plots the increasing analysis time of *Etime* across the set of benchmarks and the vertical bars show the corresponding XDD analysis time.

The *Etime* analysis duration follows an exponential pattern over the set of benchmarks and reaches 7 minutes in the worst cases. In the meantime, the analysis time using XDD remains lower than one second in almost all cases. Yet, as the *Etime* is exhaustive, its execution time is exponential with respect to the number of events in the BB but the split threshold set to 15 prevents exponential explosion. In any case, the computation time mainly depends on the total number of events of the benchmark and on the number of blocks containing more than 15 events. The most time consuming benchmarks, for example, *mpeg2* (pink bars in the Figure 3), *rijndael_enc* (orange bars) and *statemate* (green bars), are also the ones that have the most events in total, and that have blocks containing the largest number of events. This observation applies well to most benchmarks but more details can be found in experimental data published on Zenodo.

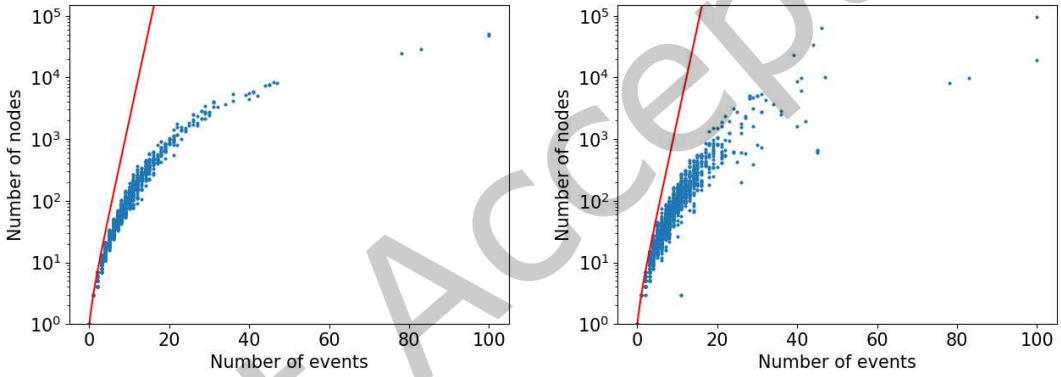
While the analysis time for *Etime* grows steadily, no pattern emerges for the analysis time using XDD. This is particularly striking with one benchmark in Figure 3a that has a very low analysis time (< 1ms): it is reported as 0 ms because of the precision of the measurement service of the host operating system. A small set of benchmarks (9 for the simple architecture and 10 for the complex one) exhibit a slightly worse analysis time with XDDs than with *Etime*, but this overhead is too small to be representative, in particular because it falls within the precision margin of the experimental platform.

[5]https://zenodo.org/record/3756621/files/LCTES.tar?download=1
[6]https://zenodo.org/record/4095452/files/TECS.tar?download=1



(a) Simple architecture (b) Complex architecture

Fig. 4. Number of leaves of resulting XDD with respect to the number of events.



(a) Simple architecture (b) Complex architecture

Fig. 5. Number of nodes of resulting XDD with respect to the number of events.

5.3 XDD Compactness

The idea of compactness comes from the observation that the amount of possible execution times of a BB is generally much smaller than the theoretical upper bound $2^{|\mathcal{E}|}$ with $|\mathcal{E}|$ events involved in the XG. To confirm that this phenomenon frequently occurs, we measure the number of leaves with respect to $|\mathcal{E}|$. In order to allow large numbers of events, the split threshold is set to 100. Figures 4a and 4b show the results for both the simple and complex architectures. Each dot represents the number of leaves (vertical axis with logarithmic scale) as a function of its number of events (horizontal axis). We also plot the $2^{|\mathcal{E}|}$ line (red line) as reference. When the number of events grows, the gap between the theoretical upper bound and the actual number of leaves widens, as the number of leaves clearly does not follow an exponential growth. This validates our initial assumption.

The benefits of the XDD approach stem from the absorption effect of the processor pipeline. However, the impact of this phenomenon on XDD depends on the benchmark and on the target architecture and is therefore

difficult to estimate without a complete computation of the XG. Hence, we statistically quantify the impact of absorption on the size of the final XDD, which is strongly correlated to the analysis time of XDDs. We consider a split threshold of 100, which allows the analysis to finish in a few minutes. Figures 5a and 5b show the number of nodes and leaves (vertical axis) of the final XDDs with respect to the number of events (horizontal axis). The final XDD is obtained at the end of an XG analysis to represent all the possible execution times of the BB. The theoretical upper bound on the amount of nodes in the XDD is $2^{|\mathcal{E}|+1} - 1$, and is plotted as reference (red line). This bound is reached whenever there is no absorption of events in the pipeline. Experimental results confirm that the number of nodes is much less than the theoretical upper bound, which means that the simplifications often occur.

The two previous experiments show similar results for the analysis of both architectures. Yet, the cloud of dots is thicker for the complex architecture meaning there is more variability for the size of XDDs. This reflects that the increase of *Instruction Level Parallelism* induced by superscalar architectures allows more variable patterns of instruction execution inside the pipeline.

5.4 Configuration Grouping and ILP Construction

The grouping algorithm aims to reduce the number of variables, while also preserving precision. To evaluate the analysis time and the precision of the partitioning algorithm, the *XDD grouping* heuristic exposed in 4.4 is compared with the original algorithm of OTAWA and with two extreme algorithms, *exhaustive grouping* and *max grouping*:

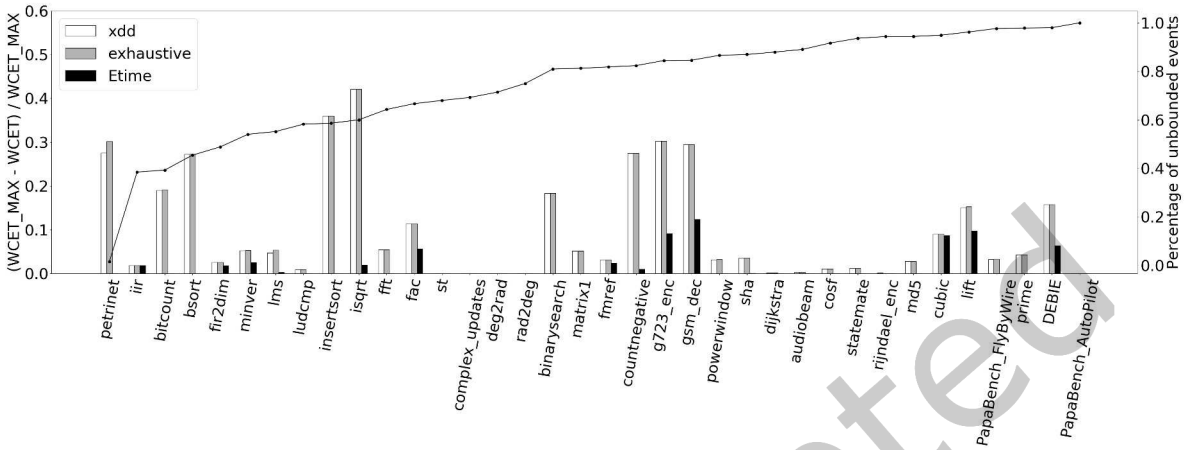
- *Etime* – The original algorithm of OTAWA aims to make a partition of two parts, *High Time Set* and *Low Time Set*. It splits the parts sorted by time in the two sets such that a heuristic time estimation function is minimized. It is not able to split the parts to accommodate event bounding.
- *exhaustive grouping* – It creates one variable per configuration i.e. $2^{|\mathcal{E}|}$ variables. This is the most expensive calculation but it keeps 100% precision. However, this approach may produce ILP systems too large to be solved in practice.
- *max grouping* – This method uses one time per BB, the maximal WCET. It is equivalent to create only the *nullified part*. Hence, only one variable with the maximal WCET as coefficient is added into the objective function. It is clearly the most imprecise but the fastest calculation. In particular, the time details provided by the XG analysis are ignored.

In fact, both *exhaustive grouping* and *max grouping* use XDD structures to represent the mapping of configurations to times that results from the XG analysis. For *max grouping*, only the biggest $\textsc{leaf}(k)$ is taken as the WCET of BB. For the *exhaustive grouping* approach, a XDD tree traversal algorithm is applied to find out all configurations then create the corresponding variables and constraints.

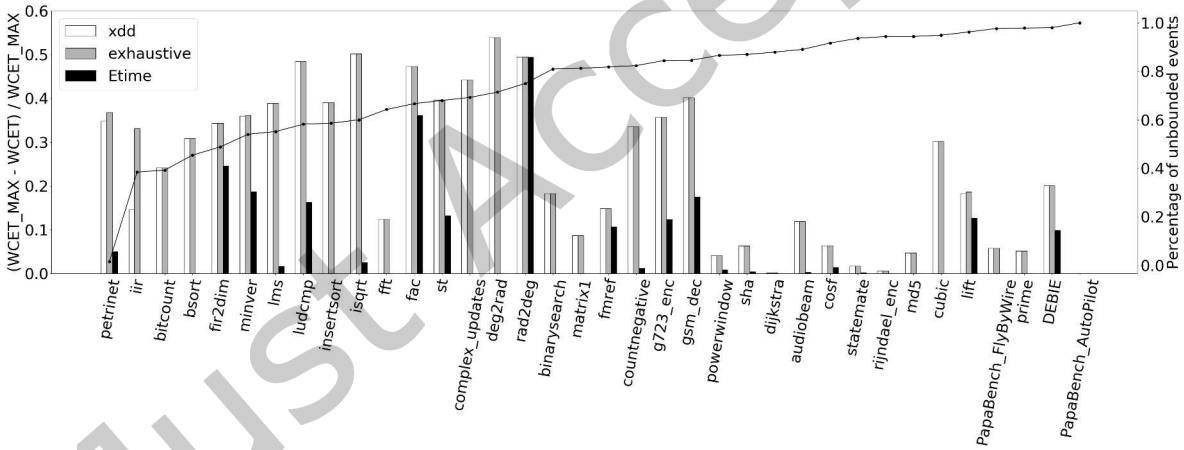
The performance of these algorithms in terms of WCET precision is shown in Figure 6. The *split threshold* is set to 15 due to the performance limitation of the *exhaustive grouping* and Etime approaches. *XDD grouping*, *Etime* and *exhaustive grouping* are compared to the *max grouping*. The precision improvement (higher is better) is calculated as $(w_{max} - w_g)/w_{max}$ where w_{max} is the WCET of *max grouping* and w_g the WCET of the other grouping methods.

The Figure shows that *XDD grouping* and *exhaustive grouping* provide similar improvement for about 80% of the benchmarks. They both produce tighter WCET than *max grouping* and *Etime*. Compared to *max grouping*, the maximal improvement of 42.1% is observed on *isqrt* for the simple architecture and 53.9% on *deg2rad* for the complex architecture. The original algorithm of OTAWA, *Etime* performs slightly better than *max grouping* on around 50% of benchmarks for the simple architecture. However, *XDD grouping* provides significant improvement for the complex architecture.

Note that this metric using WCET precision depends strongly on the result of history-based hardware analyses that compute bounds for events. If no bound can be found for an event, no grouping strategy can help to reduce the induced non-determinism: either the pipeline is able to assimilate event latencies, or the event duration is



(a) Simple architecture



(b) Complex architecture

Fig. 6. WCET improvement.

accounted for in the WCET overestimation as is. As more and more events are unbounded, the WCET produced by *XDD grouping* and *exhaustive grouping* gets closer to *max grouping*. To reveal the impact of this factor, we also plot the ratio of unbounded events as the black, dotted line. The ratio value is represented in the right y-axis. On the complex architecture, it is clear that this ratio has an important impact on the WCET: the benchmarks that have more unbounded events have less WCET improvement regardless of the applied grouping algorithm. In fact, the imprecision comes from the history-based hardware analyses and can not be alleviated by the grouping strategy. It must be observed that this metric is not absolute: it depends a lot on the difference between the

estimated WCET and the real WCET, that can not be computed in practice. If the estimated WCET is already tight, only very few improvements can be expected. Yet, it remains a good indicator of the quality of the grouping.

Another important characteristic of the grouping algorithms is their analysis time. As the global analysis is the same for different grouping algorithms, we compare directly the total analysis time of each entire program, from reading the binary file to resolving the ILP system to obtain the final WCET. The split threshold is set to 15 as in the previous experiment, due to the limitation of the *exhaustive grouping* and *Etime* algorithms. Despite this threshold, the analysis of program `rijndael_enc` using exhaustive grouping failed to finish in 12 hours that is reflected by a missing bar in Figure 7a.

Figure 7 shows the analysis time (in seconds) by benchmark (lower is better). The y-axis is in logarithmic scale. On both architectures, except for *petrinet*, *XDD grouping* shows similar analysis times as *max grouping*, but is much faster than *exhaustive grouping* and *Etime*. For small benchmarks whose analysis time is below 1 second, the difference is not significant due to the possible errors in the measurement process on the experimentation platform. Considering the time of history-based hardware analysis, *XDD grouping* is almost as fast as *max grouping* while additional time is spent to visit the XDD tree to build bounding information for events.

Combining the statistics on performance and the statistics on precision, we can conclude that *XDD grouping* is a very promising approach for WCET calculation: it is mostly as fast as *max grouping* and mostly as precise as *exhaustive grouping*. We have observed in the experiments that some history-based hardware analyses, especially data cache analyses, provide imprecise results and produce a large number of unbounded events for certain benchmarks. However, it is difficult to quantify the impact of such analyses on the ILP algorithm by simply enabling/disabling the corresponding events since their latency may be hidden by the absorbing effect of the pipeline. The study on the impact of different analyses and how they may be related to the latency hiding effect remains interesting, but needs deeper investigation.

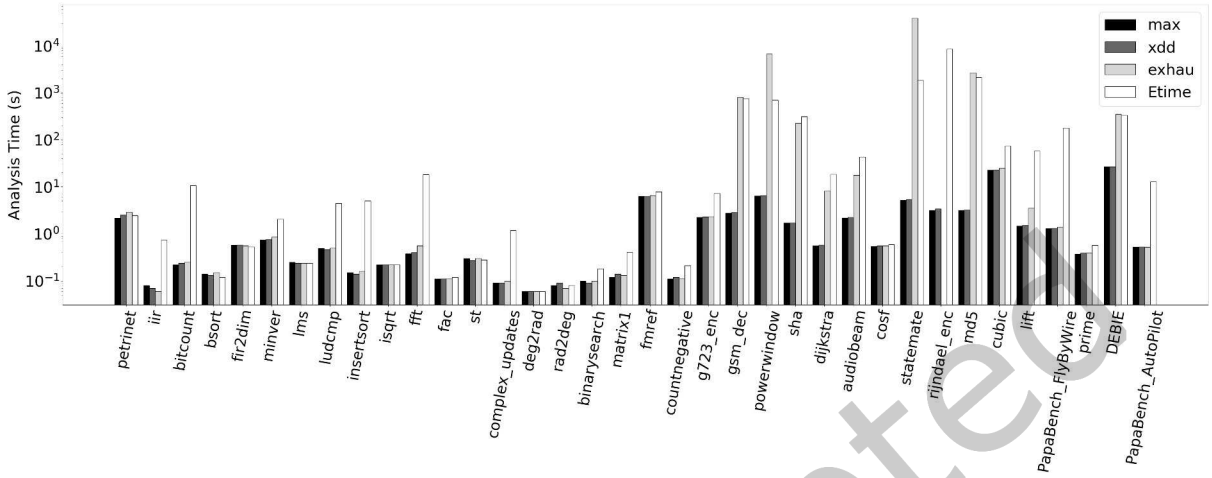
6 RELATED WORK

The precise estimation of the execution time of basic blocks is crucial in the static analysis of a task's WCET. The two main challenges come from pipelined execution and variable instruction latencies.

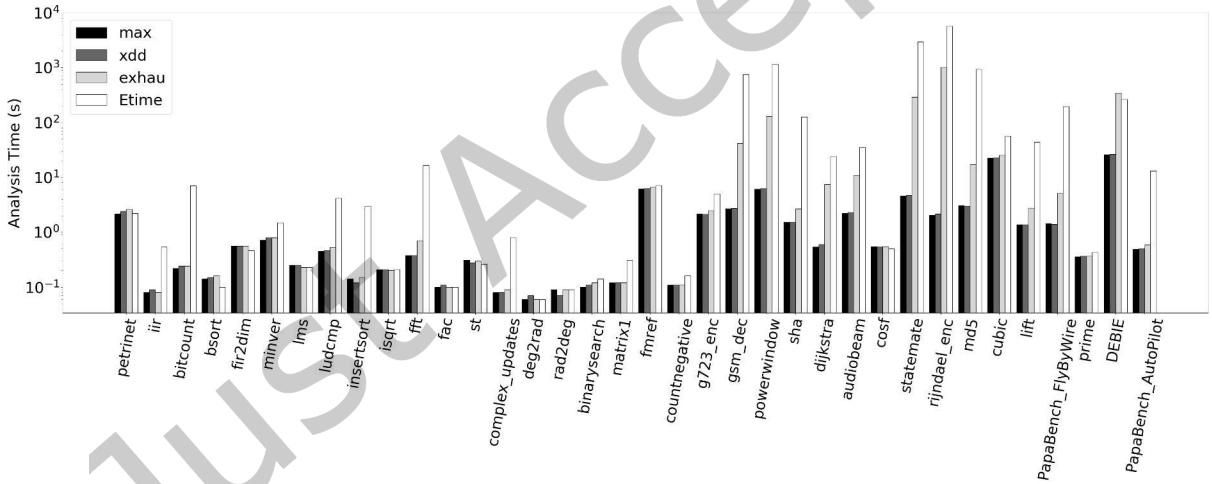
Beside ad-hoc algorithms dedicated to specific pipelines [12, 13, 15], a simulator-based approach was proposed by Engblom et al. [7, 8]. Although it takes the overlapping of blocks in the pipeline into account, time events are added as-is to the final time (a) incurring an overestimation and (b) preventing the support for timing anomalies. Healy et al. [11] compute the WCET by deriving a *pipeline diagram* for each block representing the traversal of the stages by each instruction, and by composing these diagrams. The *time events* are taken into account by modifying the content of the diagram, that in turn produces an impact on the next block diagrams. Unfortunately, this method can only apply to very simple microprocessors.

A first kind of generic method to compute basic block execution times was proposed by Kassem et al. [14]. It uses automata to represent the different states of the pipeline. Transitions between states are triggered by a mix of events recording the instruction execution phases and other hardware effects. Cassez et al. [4, 6] use a similar approach but hardware analysis and WCET calculation are integrated into a timed model checker, which prevents the exhaustive building of all pipeline states. Yet, although these methods speed up the traversal of states, they often result in huge automata.

Another successful approach makes use of *Abstract Interpretation* to compute the reachable pipeline states and to bound the inherent state blowup by abstraction. This approach developed by Thesing et al. in [16, 25] is implemented in the *aiT* toolsuite and has been succesfully used on real micro-architectures and applications. Timing events are managed by duplicating the code blocks in so-called *Abstract Pipeline State Graphs* [24] to track the multiple event latencies. To our knowledge, there is no published report on the impact of the event-related latency variability on the (empirical) complexity and duration of the analysis.



(a) Simple architecture



(b) Complex architecture

Fig. 7. Total analysis time.

Basically used to optimize Boolean functions, BDDs have been successfully used to avoid explicit computation in symbolic model checking [5]. In a different context, Wilhelm et al. proposed to use BDDs to compact the pipeline state representation to perform abstract interpretation for pipeline analysis [27].

7 CONCLUSION

This paper introduces the XDD data structure, which is an adaptation of the BDD structure to the particular problem of WCET computation in the presence of variable latencies. It shows that the use of XDDs to compute and to represent execution times speeds up the analysis. The increase in performance stems from leveraging the latency-hiding properties of microprocessor pipelines. Moreover, we prove that this improvement comes at no cost with respect to the precision of the analysis. We also show that using XDDs significantly reduces the empirical complexity of the analysis compared to the existing *Etime* method, which allows performing WCET analysis on larger and more complex applications. In experiments, the analysis time was reduced to less than 1 second for all the analyzed benchmarks taken from the *TACLe* suite (for a *split threshold* of 15), while the *Etime* method can take up to 7 minutes. Moreover, observing the number of nodes and leaves in the XDDs confirmed our initial assumption that the pipeline hides some execution latencies and that this can efficiently be accounted for by the XDD structure. Our results show the efficiency of factoring nodes that yield the same execution time, compared to an exhaustive computation which becomes intractable as soon as a BB is subject to more than 15 events.

As more precise pipeline analyses provide richer results, the building process of the ILP system within the IPET technique has been adapted. Concretely, all possible execution times resulting from XG analysis are employed to build the ILP system. Moreover, additional event-related constraints have been introduced to prevent the *maximization effect* induced by the ILP solving method. We propose a new ILP system building process that (a) takes advantage from the new results of the pipeline analysis: it creates more variables that express more precisely the execution patterns; (b) prevents the blow-up of useless variables in the ILP system; (c) exploits event bounds, that are issued from history-based hardware analysis (for example cache persistence analysis [10]), and (d) is easy and efficient to implement with XDDs. The experimental results show that this approach is almost as precise as the exhaustive algorithm and as fast as the simplest algorithm while the simplest algorithm is almost equivalent to the traditional way to compute WCET: taking only the maximal execution time of each BB.

As future work, we plan to extend the applicability of XDDs to other models of architectures, such as out-of-order pipelines, and to further increase the performance on the involved analyses. Another research perspective is to introduce relationships between events, to model more complex behaviors of the architectures. For example, a memory access resulting in a Miss in the L1 cache could also cause a Miss in the L2 cache. In our current model both Misses would be represented as separate events, even though the Miss in L2 cannot occur if there is no Miss in L1. Taking into account the existing correlation between the two events in this example could reduce the size of the corresponding XDD, thus easing the analysis of more complex architectures.

REFERENCES

[1] Henrik Reif Andersen. 1997. An introduction to binary decision diagrams. *Lecture notes, IT University of Copenhagen* (1997).

[2] Zhenyu Bai, Hugues Cassé, Marianne De Michiel, Thomas Carle, and Chistine Rochange. 2020. Improving the Performance of WCET Analysis in the Presence of Variable Latencies *(LCTES '20)*. 119–130.

[3] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems (SEUS)*. 35–46.

[4] Jean-Luc Béchennec and Franck Cassez. 2011. Computation of WCET using Program Slicing and Real-Time Model-Checking. *CoRR* abs/1105.1633 (2011). arXiv:1105.1633

[5] Jerry R. Burch, Edmund Melson Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1992. Symbolic model checking: 1020 States and beyond. *Information and Computation* 98 (June 1992), 142–170.

[6] Franck Cassez and Pablo González de Aledo Marugán. 2015. Timed Automata for Modelling Caches and Pipelines. *Proceedings Workshop on Models for Formal Analysis of Real Systems, MARS* 196 (2015), 37–45.

[7] Jakob Engblom. 2002. *Processor Pipelines and and Static Worst-Case Execution Time Analysis.* Ph.D. Dissertation. University of Uppsala.

[8] Jakob Engblom and Andreas Ermedahl. 1999. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. of 6th International Conference on Real-Time Computing Systems and Applications (RTCSA)*.

[9] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis*.
[10] Christian Ferdinand. 2005. *A Fast and Efficient Cache Persistence Analysis*. Technical Report. Saarländische Universitëts- und Landesbibliothek / Naturwissenschaftlich-Technische Fakultët I.
[11] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. 1999. Bounding Pipeline and Instruction Cache Performance. *IEEE Trans. Comput.* 48, 1 (Jan. 1999), 53–70.
[12] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. 2000. Worst-case execution time analysis for digital signal processors. In *10th European Signal Processing Conference*. 1–4.
[13] Niklas Holsti and Sami Saarinen. 2002. Status of the Bound-T WCET tool. *Space Systems Finland Ltd* (2002).
[14] Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Yvon Trinquet, and Guillaume Savaton. 2008. Simulator generation using an automaton based pipeline model for timing analysis. In *International Multiconference on Computer Science and Information Technology*. IEEE, 657–664.
[15] Raimund Kirner. 2012. The WCET analysis tool CalcWcet167. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 158–172.
[16] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. 2002. Pipeline modeling for timing analysis. In *International Static Analysis Symposium*. Springer, 294–309.
[17] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. 2006. Modeling out-of-order processors for WCET analysis. *Real-Time Systems* 34, 3 (2006), 195–227.
[18] Yau-Tsun Steven Li and Sharad Malik. 1995. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, Vol. 30.11. 88–98.
[19] Yau.-Tsun S. Li and Sharad Malik. 1995. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*. 88–98.
[20] Thomas Lundqvist and Per Stenstrom. 1999. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium*. 12–21.
[21] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. 1990. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *27th ACM/IEEE Design Automation Conference*. 52–57.
[22] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. 2006. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*.
[23] Christine Rochange and Pascal Sainrat. 2009. A context-parameterized model for static analysis of execution times. *Transactions on High-Performance Embedded Architectures and Compilers II* (2009), 222–241.
[24] Ingmar Jendrik Stein. 2010. *ILP-based path analysis on abstract pipeline state graphs*. Ph.D. Dissertation. Saarland University.
[25] Stephan Thesing. 2004. *Safe and precise WCET determination by abstract interpretation of pipeline models*. Ph.D. Dissertation. Saarland University.
[26] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 1–53.
[27] Stephan Wilhelm. 2007. Efficient Analysis of Pipeline Models for WCET Computation. In *Workshop on Worst-Case Execution Time (WCET'07)*.