Improving the Performance of WCET Analysis in the Presence of Variable Latencies

Zhenyu Bai, Hugues Cassé, Marianne de Michiel, Thomas Carle, Christine Rochange

Improving the Performance of WCET Analysis in the Presence of Variable Latencies

Zhenyu Bai
CNRS - IRIT - University of Toulouse
zhenyu.bai@irit.fr

Hugues Cassé
CNRS - IRIT - University of Toulouse
hugues.casse@irit.fr

Marianne De Michiel
CNRS - IRIT - University of Toulouse
marianne.de-michiel@irit.fr

Thomas Carle
CNRS - IRIT - University of Toulouse
thomas.carle@irit.fr

Christine Rochange
CNRS - IRIT - University of Toulouse
christine.rochange@irit.fr

Abstract

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 tradeoff must be made between the duration and the precision of the analysis, leading to an overestimation of the WCET bounds. This in turn 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 and durations of instruction sequencesn a modern hardware platform. We demonstrate on realistic applications how the use of an XDD substantially increases the scalability of WCET analysis.

Keywords: static WCET analysis, pipeline analysis, variable latencies, timing anomalies

ACM Reference Format:
Zhenyu Bai, Hugues Cassé, Marianne De Michiel, Thomas Carle, and Christine Rochange. 2020. Improving the Performance of WCET Analysis in the Presence of Variable Latencies. In *Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '20), June 16, 2020, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/http://dx.doi.org/10.1145/3372799.3394371

LCTES'20, June 16, 2020, London, United Kingdom

ACM ISBN ISBN 978-1-4503-7094-3/20/06…$15.00
https://doi.org/10.1145/3372799.3394371

1 Introduction

In order to guarantee the correct execution of hard real-time applications, both scheduling and schedulability 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 [24]. Those based on static analysis techniques aim at computing guaranteed upper bounds, provided their knowledge of the underlying hardware platform is correct. However, it is also desirable to have as-tight-as-possible WCET bounds since overestimations may lead to over-dimensioning the system. Besides, the duration of the analysis is sometimes a concern, so a tradeoff between precision and analysis time must be found.

Instead of considering end-to-end execution paths, which would be far too complex, static WCET analysis splits the code of a task (or runnable) into short instruction sequences and derives its global WCET from the individual WCETs of the sequences. The most common approach is the *Implicit Path Enumeration Technique* (IPET) [16] which consists in four steps:

1. *path analysis* – scans the application code to isolate instruction sequences and to derive some execution properties such as loop bounds or infeasible paths
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 instruction sequences
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 instruction sequences taking into account the processor pipeline and the hardware accelerator components: some are based on abstract interpretation techniques [23] while others use *Execution Graphs* (XG) that capture the timing semantics of a sequence of instructions as they go through the processor pipeline [21][15]. In this

paper, we focus on the approach developed in [21] but we believe that our ideas can be applied to the other approaches.

Motivation. One difficulty when determining the WCET of an instruction sequence (step 3 in the process described above) arises when the latency of an operation can have several values. For example, the time needed to fetch an instruction depends on whether it hits or misses 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 result 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* nor *AlwaysMiss*), one might be tempted to consider the largest value as the worst case. However, it has been shown that processors, in particular modern processors that implement advanced mechanisms to enhance the average performance, often exhibit so-called *timing anomalies*: a local worst-case latency does not necessarily lead to a global WCET [20]. 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 the possible latencies for each XG node and edge, and then all the possible values and all their possible combinations. Unfortunately, this might lead to consider many cases: for example, an instruction sequence with 10 variable-latency instructions would have to to be analysed 1024 times, assuming that each instruction has only two possible latency values. This would sensibly lengthen the analysis, while, as mentioned before, the analysis time can be a concern. Note that in some cases, we have even observed that the combinatorial complexity of the analysis made it intractable. Besides, we have found out that, in practice, many combinations finally lead to the same value. This is mainly due to the pipeline *hiding* some local delays. This observation guided us to a new approach that reduces the analysis time by exploiting the fact that several latency combinations produce the same WCET.

Contribution. In this paper we present an efficient solution to estimate the WCET of instruction sequences in the presence of variable latencies. The main idea is to factorize the evaluation of an XG for various node latencies. By embedding inside our time representation the occurrences of events and their effects on the sequence latency, we are able to benefit from the latency absorbing properties of the pipeline and to speed up the analysis. Our approach is based on a refinement of *Binary Decision Diagrams* (BDD) [1, 19] 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, and we report experimental results showing that this new approach significantly improves the scalability 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, which is enhanced in Section 4. 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. This encompasses the program representation, the method to compute the execution time and the support to take the behaviour of the underlying hardware into account.

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[1],
- $E_{\text{CFG}} \subset V_{\text{CFG}} \times V_{\text{CFG}}$ is the set of edges representing the execution flow between BBs,
- $\epsilon \in V_{\text{CFG}}$ is a unique BB without predecessor that 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 BB a to BB b, is denoted $a \rightarrow b$.

2.2 Execution Graphs

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) [21].

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, $[I_n/S_m]$.

Edges $E_{\text{XG}} \subset V_{\text{XG}} \times V_{\text{XG}}$ represent timing dependencies: pipeline stages must be traversed in the architectural order,

[1] It is not mandatory to have maximal BBs although this is likely to improve the precision of the results.

instructions are fetched in the program order , some instruction pairs exhibit data dependencies, instructions must wait for a free slot before being inserted into a buffer, 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, the time spent by the BB in the pipeline could 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 [21], 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 like caches or branch predictors but also variable-latency instructions such as multiplications and divisions, 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 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 in the ILP formulation used to derive the WCET [16].

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

Figure 1 shows an example XG that represents the execution of two short BBs in sequence, 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 is only made of 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 [17, 20]. 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 $e \in \mathcal{E}$ 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 for each configuration of events. This time may have several possible values and can be expressed as a function $t^* \in T$ that returns a different value for each configuration in Γ.

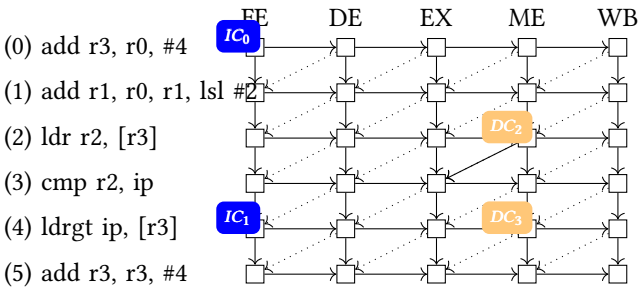
Let $\lambda_v(\gamma)$ denote the latency of XG vertex v in configuration γ. It is the sum of costs of the events attached to v that are active in γ. The computation of the XG can now be reformulated as:

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

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

Figure 1. An XG decorated with events



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 sometimes 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:

$$\begin{aligned}\rho^*_{[I_4/EX]}(\gamma) &= max(4 + max(1, 10\gamma(IC_0) + max(1, 10\gamma(DC_2)),\\ &\qquad 3 + max(1, 10\gamma(IC_0)) + max(1, 10\gamma(IC_1))\\ &\qquad 4 + max(1, 10\gamma(IC_0)) + max(1, 10\gamma(DC_2))\\ &= 4 + max(1, 10\gamma(IC_0))\\ &\quad + max(max(1, 10\gamma(IC_1)), 1 + max(1, 10\gamma(DC_2)))\\ &= 7 + 9\gamma(IC_0) + max(8\gamma(IC_1), 9\gamma(DC_2))\end{aligned}$$

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 shortened 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 symbolic representation;
- 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 induced by different possible 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 variables are replaced by events and Boolean leaves by possible times.

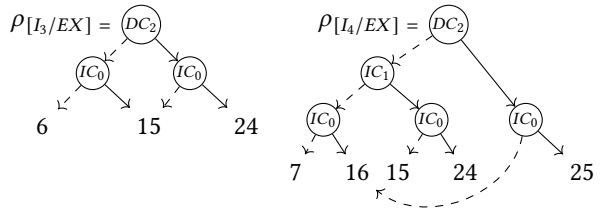
Definition 3.1. An XDD is defined recursively by:

$$\text{XDD} = \textsc{leaf}(k) \mid \textsc{node}(e, \overline{f}, f)$$

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

A $\textsc{node}(e, \overline{f}, f)$ represents alternative times depending on the occurrence of event e: f if event e is active and $\overline{f}$ otherwise. A $\textsc{leaf}(k)$ represents a constant time $k \in \mathbb{Z}$. The path from the top node to a $\textsc{leaf}(k)$ 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. It is worth noting that IC_1 is dominated by DC_2 when the latter is active: when $\gamma(DC_2) = 1$ in $\rho^*_{[I_4/EX]}$, $1 + max(1, 10\gamma(DC_2)$ is always greater than $max(1, 10\gamma(IC_1))$. This property is exploited in the XDD by removing IC_1 nodes from the right-side sub-XDD of node DC_2. Although the same property is verified in the corresponding symbolic representation, it cannot be used to simplify the expression.



Instantiation. The basic use of an XDD is to evaluate a time, given a particular configuration of events. In this sense. Based on the structure of the XDD, we define the instantiation 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 that matches the configuration is selected and the search continues down 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 canonicity of XDDs.

Order. As for BDDs, an order on the events is necessary to enforce 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 $\preceq$: $\forall e_1, e_2 \in \mathcal{E}, e_1 \preceq e_2 \vee e_2 \preceq e_1$.

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 be at the deepest level, we define $e_\perp$, satisfying $\forall e \in \mathcal{E} \setminus \{e_\perp\}$, $e_\perp \preceq e$. To simplify the notation, we define the function evt : XDD $\rightarrow \mathcal{E}$ s.t. $evt(\text{NODE}(e, \overline{g}, g)) = e$ and $evt(\text{LEAF}(k)) = e_\perp$. The method that we use to find such an order is further discussed in Section 4.3.

To ensure that the events of XDD nodes are ordered, 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.

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

$\forall f_1, f_2 \in \text{XDD}, f_1 \odot f_2 =$

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

The extension consists in combining XDDs according to their nature. If two leaves are added, the result is a leaf which value is the application of $\boxdot$ on both leaves values **(a)**. If two nodes with the same event are combined, the operation is propagated equally on each side of the node **(b)** and **(e)**. If

the events are different, the operation is propagated according to the order of events **(c)**, **(d)**, **(f)** and **(g)**. Particularly, applying the operation 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.

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 $\rho_w^\# \in \text{XDD}$. An $\rho_v^\#$ 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 in this calculation stems in the mix of events handling and of the minimization of XDD representation based on the *Compactness* property.

3.5 Equivalence between XDD and T

Lemma 3.9. *Consider an operation $\boxdot : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and its derivative on XDD, $\odot$: XDD $\times$ XDD $\to$ XDD. The following property holds:*

$$\forall f_1, f_2 \in \text{XDD}, \forall \gamma \in \Gamma, (f_1 \odot f_2)^{[\gamma]} = f_1^{[\gamma]} \boxdot f_2^{[\gamma]}$$

Proof. The proof of Lemma 3.9 makes an induction on the structure of an XDD from the leaves to the root. As an XDD is implemented as a *Directed Acyclic Graph*, a node may have several predecessors and the induction requires to find back the relevant predecessor corresponding to the path induced by configuration γ.

Let $\pi_g^\gamma(f)$ be a function that returns the predecessor of g in f along the path induced by the configuration γ, or $\perp$ if g is not on the path of f along γ. It is defined by:
$\forall f, g \in \text{XDD}, \forall \gamma \in \Gamma$

$$\pi_g^\gamma(f) = \begin{cases} f & \text{if } (f = \text{NODE}(e, g, \_) \wedge \gamma(e) = 0) \\ & \vee\ (f = \text{NODE}(e, \_, g) \wedge \gamma(e) = 1) \\ & \vee\ f = g \\ \pi_g^\gamma(\overline{h}) & \text{if } f = \text{NODE}(e, \overline{h}, h) \wedge \gamma(e) = 0 \\ \pi_g^\gamma(h) & \text{if } f = \text{NODE}(e, \overline{h}, h) \wedge \gamma(e) = 1 \\ \perp & \text{else} \end{cases}$$

Initial case: Consider the initial case with $g_1 = \text{LEAF}(f_1^{[\gamma]})$ and $g_2 = \text{LEAF}(f_2^{[\gamma]})$:

$$\begin{aligned} (\text{LEAF}(f_1^{[\gamma]}) \odot \text{LEAF}(f_2^{[\gamma]}))^{[\gamma]} &= \text{LEAF}(f_1^{[\gamma]} \boxdot f_2^{[\gamma]})^{[\gamma]} \\ &= f_1^{[\gamma]} \boxdot f_2^{[\gamma]} \end{aligned}$$

Induction case: Let g_1 and g_2 be, respectively, the sub-XDDs of f_1 and f_2 along the path induced by γ. Let us assume that $(g_1 \odot g_2)^{[\gamma]} = g_1^{[\gamma]} \boxdot g_2^{[\gamma]}$. The proof is completed if $g_1 = f_1$ and $g_2 = f_2$. Otherwise we have different ways to perform the induction. Disregarding the initial case, $\pi_g^{[\gamma]}$ always results in a node denoted $\text{NODE}(e, g, \_)$ if $\gamma(e) = 0$, and $\text{NODE}(e, \_, g)$ otherwise.

1. if $evt(\pi_{g_1}^\gamma(f_1)) = evt(\pi_{g_2}^\gamma(f_2)) = e$ and $\gamma(e) = 0$ then

$$\begin{aligned} (\pi_{g_1}^\gamma(f_1) \odot \pi_{g_2}^\gamma(f_2))^{[\gamma]} &= (\text{NODE}(e, g_1, \_) \odot \text{NODE}(e, g_2, \_))^{[\gamma]} \\ &= \text{NODE}(e, g_1 \odot g_2, \_)^{[\gamma]} \\ &= (g_1 \odot g_2)^{[\gamma]} \\ &= g_1^{[\gamma]} \boxdot g_2^{[\gamma]} \\ &= \pi_{g_1}^\gamma(f_1)^{[\gamma]} \boxdot \pi_{g_2}^\gamma(f_2)^{[\gamma]} \end{aligned}$$

2. if $evt(\pi_{g_1}^\gamma(f_1)) = evt(\pi_{g_2}^\gamma(f_2)) = e$ and $\gamma(e) = 1$: similar to (1) using the right-side sub-XDDs in place of g_1 and g_2

3. if $evt(\pi^{\gamma}_{g_1}(f_1)) \preceq evt(\pi^{\gamma}_{g_2}(f_2))$ and $\gamma(evt(\pi^{\gamma}_{g_2}(f_2))) = 0$ then $\pi^{\gamma}_{g_2}(f_2) = \text{NODE}(e_2, g_2, \_)$

$$\begin{aligned}(\pi^{\gamma}_{g_1}(f_1) \odot \pi^{\gamma}_{g_2}(f_2))^{[\gamma]} &= (\pi^{\gamma}_{g_1}(f_1) \odot \text{NODE}(e_2, g_2, h))^{[\gamma]} \\ &= \text{NODE}(e, \pi^{\gamma}_{g_1}(f_1) \odot g_2, \pi^{\gamma}_{g_1}(f_1) \odot h)^{[\gamma]} \\ &= (\pi^{\gamma}_{g_1}(f_1) \odot g_2)^{[\gamma]} \\ &= \pi^{\gamma}_{g_1}(f_1)^{[\gamma]} \boxdot g_2^{[\gamma]} \\ &= \pi^{\gamma}_{g_1}(f_1)^{[\gamma]} \boxdot \pi^{\gamma}_{g_2}(f_2)^{[\gamma]}\end{aligned}$$

4. if $evt(\pi^{\gamma}_{g_1}(f_1)) \preceq evt(\pi^{\gamma}_{g_2}(f_2))$ and $\gamma(evt(\pi^{\gamma}_{g_2}(f_2))) = 1$: similar to (3) using the right-side sub-XDDs
5. else $evt(\pi^{\gamma}_{g_1}(f_1)) \preceq evt(\pi^{\gamma}_{g_2}(f_2))$: same as (3) and (4) swapping g_1 and g_2 (immediate if $\odot$ and $\boxdot$ are commutative).

□

Proposition 3.10. *The domains $\langle XDD, \oplus, \otimes\rangle$ and $\langle T, max, +\rangle$ are semi-rings.*

Proof. The demonstration is straightforward considering that both XDD and T are structures embedding the well-known semi-ring $\langle \mathbb{Z}, max, +\rangle$. □

To show that XDD and T are equivalent, we define function $F : \text{XDD} \rightarrow T$ ensuring that the semi-rings $\langle \text{XDD}, \oplus, \otimes\rangle$ and $\langle TS, max, +\rangle$ are isomorphic.

Definition 3.11.

$$\forall f \in \text{XDD}, \forall \gamma \in \Gamma, [F(f)](\gamma) = f^{[\gamma]}$$

Proposition 3.12. *$F : XDD \rightarrow T$ and its inverse F^{-1} form an isomorphism between semi-rings ($\langle XDD, \oplus, \otimes\rangle$ and $\langle T, max, +\rangle$), i.e. : $\forall f_1, f_2 \in XDD$,*

- $F(f_1 \oplus f_2) = \max(F(f_1), F(f_2))$
- $F(f_1 \otimes f_2) = F(f_1) + F(f_2)$
- *F is bijective*

Proof. F is bijective because we can exhibit $F^{-1} : T \rightarrow \text{XDD}$ as: $\forall t \in T$, $F^{-1}(t) = \bigoplus_{\gamma \in \Gamma} \mu^{\gamma}_n(t(\gamma))$ with:

$$\mu^{\gamma}_i(k) = \begin{cases} \text{LEAF}(k) & \text{if } i = 0 \\ \text{NODE}(e_i, \text{LEAF}(0), \mu^{\gamma}_{i-1}(k)) & \text{if } \gamma(e_i) = 1 \\ \text{NODE}(e_i, \mu^{\gamma}_{i-1}(k), \text{LEAF}(0)) & \text{else} \end{cases}$$

This ensures (a) that $\forall \gamma \in \Gamma$, $[F^{-1}(t)]^{[\gamma]} = t(\gamma)$ and (b) that $\forall f \in \text{XDD}$, $F^{-1}(F(f)) = f$ because of the canonicity condition.

$\forall \gamma \in \Gamma$,

$$\begin{aligned}[F(f_1 \oplus f_2)](\gamma) &= (f_1 \oplus f_2)^{[\gamma]} \\ &= \max(f_1^{[\gamma]}, f_2^{[\gamma]}) \qquad \text{(by Lemma 3.9)} \\ &= \max([F(f_1)](\gamma),\ [F(f_2)](\gamma))\end{aligned}$$

The proof of $[F(f_1 \otimes f_2)](\gamma) = [F(f_1)](\gamma) + [F(f_2)](\gamma)$ can be derived similarly. □

Since F and its inverse F^{-1} form an isomorphism between $\langle \text{XDD}, \oplus, \otimes\rangle$ and $\langle T, max, +\rangle$, the computations on an XG with T and XDD are equivalent. The following section presents additional enhancements to improve the efficiency of XDDs.

4 Enhancing the performances of XDD

In the previous section, we have formally defined the XDDs, and adapted the basic operations on the BDD introduced in [1]. This algorithm is designed to be general but $\otimes$ and $\oplus$ also support specific optimizations that are exposed in this section. First, we present a *cutting* technique that allows to stop the recursive application of $\oplus$ as soon as a particular condition is satisfied. We then show how *memoization* can be used to compactly store the tree structure of an XDD and to prevent redundant calculi in the computation of the $\otimes$ and $\oplus$ operators. Finally, we discuss the impact of event ordering on the performance of XDDs.

4.1 Cutting the computation of $\oplus$

According to Def. 3.6, any operator on an XDD performs a recursive descent in the tree structure and applies the operands to each leaf. However, when $f_1 \oplus f_2 = f_1$, f_1 can directly be returned as the result of the operator application without having to propagate the recursion further on f_1 and f_2, thus *cutting* the computation.

$$\forall f_1, f_2 \in \text{XDD}, f_1 \oplus f_2 = f_1 \text{ iff } \forall \gamma \in \Gamma, f_1^{[\gamma]} \geq f_2^{[\gamma]} \tag{4}$$

This condition requires examining all the configurations to perform the cut, but a simple yet stronger condition is:

$$\forall f_1, f_2 \in \text{XDD}, f_1 \oplus f_2 = f_1 \text{ iff } min^{\#}(f_1) > max^{\#}(f_2) \tag{5}$$

with $max^{\#}$, $min^{\#} : \text{XDD} \rightarrow \mathbb{Z}$ defined as follows:

Definition 4.1. $\forall f \in \text{XDD}$,

$$min^{\#}(f) = \begin{cases} min(min^{\#}(\overline{g}), \min^{\#}(g)) & \text{if } f = \text{NODE}(e, \overline{g}, g) \\ k & \text{if } f = \text{LEAF}(k) \end{cases}$$

$$max^{\#}(f) = \begin{cases} max(max^{\#}(\overline{g}), max^{\#}(g)) & \text{if } f = \text{NODE}(e, \overline{g}, g) \\ k & \text{if } f = \text{LEAF}(k) \end{cases}$$

Since the definitions of $max^{\#}$ and $min^{\#}$ are recursive, we can associate a pair (min, max) to each node representing the minimum and maximum leaf time. This pair can be simply built during the construction of the XDD, and allows testing the condition of Equation 5 conveniently at each step of computation without going recursively down to the leaves. Once the cut condition is satisfied, we can stop the computation right away and take the strict superior operand. To do so, we insert the two following rules into Definition 3.6 between rule **(a)** and rule **(b)**, only for operator $\oplus$.

$$f_1 \oplus f_2 = \begin{cases} f_1 & \text{if } min^{\#}(f_1) \geq max^{\#}(f_2) \textbf{ (a')} \\ f_2 & \text{if } min^{\#}(f_2) > max^{\#}(f_1) \textbf{ (a")} \end{cases} \tag{6}$$

4.2 Memoization

Memoization is the key for the performance of XDDs. We use two types of memoization:

- A *Uniqueness table* is used to store each instance of XDD to ensure its canonicity (compactness and events ordering).
- An *Operation table* stores the results of operations performed on the XDD sub-trees during the recursive calls implementing $\oplus$ and $\otimes$.

The *Uniqueness table* is implemented as a hash table to store all created nodes and leaves. Explicitly, it maps a node or a leaf to a unique instance. When creating new nodes or leaves, we check if a corresponding instance exists: if so, the formerly-created instance is re-used and hence kept unique. Considering the nature of an XG (and the underlying pipeline), the XDD used in the calculation of one XG are likely to be unrelated to XDDs of a different XG. Hence, we use one *Uniqueness table* per XG.

The operators $\oplus$ and $\otimes$ are applied recursively on the sub-XDDs. Since the XDD corresponding to one XG node could be close to the XDD of its predecessors, the partial results (e.g. the result of recursive call to sub-XDDs) are often similar. Hence, we use two global maps (one per operator) to record those results and check if they could be re-used upon a subsequent operation application.

As observed in the calculation of XG, the events are likely to compensate themselves leading to an important re-use. In this context, the use of *Uniqueness table* and of the *Operation table* is critical in order to speed up the computations.

4.3 Events ordering

As explained in the definition of XDD, an order $\preceq$ on the events is necessary to define a canonical XDD. However, such an order is not unique. As for BDDs, the chosen order has a significant impact on performance. Yet determining the best order for a BDD has been proven to be very complex [18]. Fortunately, we are able to propose a heuristic order. It is based on (a) the topological order of the first occurrence of an event in XG and (b) the indices associated to events to solve the case when two events are applied to the same node.

More precisely, let two events e_{k_1} and e_{k_2} arising on XG nodes $[I_{i_1}/S_{j_1}]$ and $[I_{i_2}/S_{j_2}]$ respectively. $e_{k_1} \preceq e_{k_2}$ holds iff the triple $\langle i_1, j_1, k_1\rangle$ is smaller than $\langle i_2, j_2, k_2\rangle$ in the lexicographic order.

As the XG analysis is performed in topological order, an event e arising on a node v is usually smaller than any event arising inside the XDD f of predecessors of v. Thus, the performed computation, $f \otimes \text{NODE}(e, \text{LEAF}(0), \text{LEAF}(k_e))$, results in $\text{NODE}(e, f, f \otimes \text{LEAF}(k_e))$: f is re-used as-is in the resulting XDD, almost halving the amount of computation to perform.

5 Evaluation

We now present the experiments performed to evaluate the efficiency of XDDs. We used *OTAWA*, a framework dedicated to static WCET analysis [2], that includes analysis engines able to identify events (e.g. cache and branch prediction analyses). We have implemented the XDD approach and compared it to the approach currently existing 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 analysed basic blocks.

Simple	Complex
5 stages	4 stages
FE, DE, EX, MEM, WB	FE, DE, EX, WB
no fetch queue	fetch queue width = 3
1 instruction/cycle	3 instructions/cycle (3-ways superscalar)
1 Execution Stage	1 ALU, 1 FPU, 1 MU
branch prediction: yes	
2-way 16KB LRU instruction cache	
2-way 8KB LRU data cache	

Table 1. Target hardware architecture details

5.1 Experimental framework

We considered 81% of the TACLe benchmark suite [8]. The remaining 19% had to be discarded due to restrictions imposed by the current version of *OTAWA* analyses.

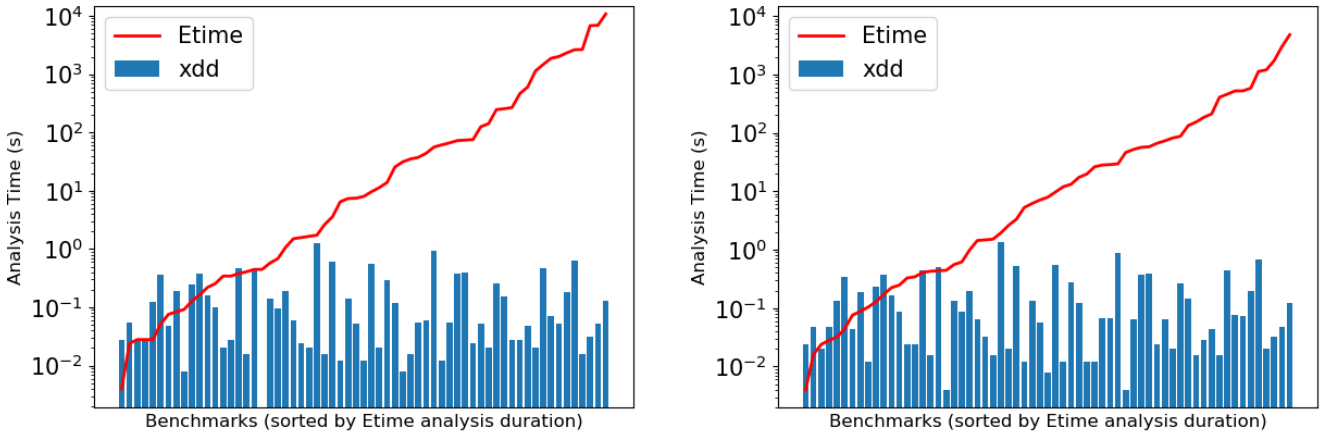
We modeled two in-order pipelined architectures: one representative of simple embedded processors and a more complex Tricore Aurix-like one. They cover both ends of processor families typically used in embedded systems andA2 will provide an insight into the influence of the pipeline complexity on the XDD computation performances. Table 1 provides a more detailed specification of both architectures. The simple architecture is composed of a classical 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 functional units composing the execution stage.

All the xperiments were performed on a server composed of 8 Intel Xeon E25 cores (@2.4GHz) sharing 32GB of RAM. Our implementation of XDD is single-threaded but multiple experiments were executed in parallel.

All details about the experiment are available on Zenodo[2].

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

[2]https://zenodo.org/record/3756621/files/LCTES.tar?download=1



(a) Simple architecture

(b) Complex architecture

Figure 2. Analysis time

a reasonable time 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. We suspect that this technique introduces additional imprecision since it does not consider the overlap of BBs inside the pipeline at the split boundaries. However, a complete and sound investigation on this topic would take too much room and is 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, allowing to cover 99% of their BBs without split. Only `rijndael_enc` and `gsm_dec`, that 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 Analysis Time

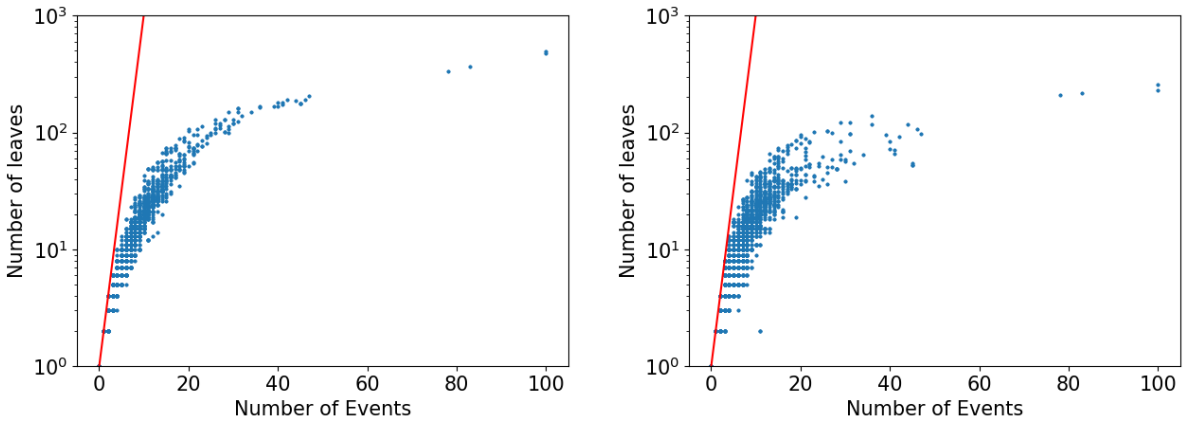
Figures 2a and 2b plot the analysis time of the *Etime* and XDD approaches. The x-axis represents the benchmarks ordered by their analysis time using *Etime*, which provides a raw experimental estimation of their complexity.(*Etime* being an exhaustive computation) The y-axis shows the analysis time 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 blue 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 restrains the exponential blowup. Whatever, the execution time depends mainly on the total number of events of the benchmark and the number of block containing more than 15 events.The most time consuming benchmarks, *rijndael_enc* and *statemate*, are also the ones that have the most of events in total, and have blocks containing the most of events. This observation applies well to most benchmarks but more details can be found in published experiment data.A4

As the analysis time for *Etime* grows steadily, no pattern emerges for the analysis time using XDD.his is particularly striking with one benchmark in Figure 2a 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.

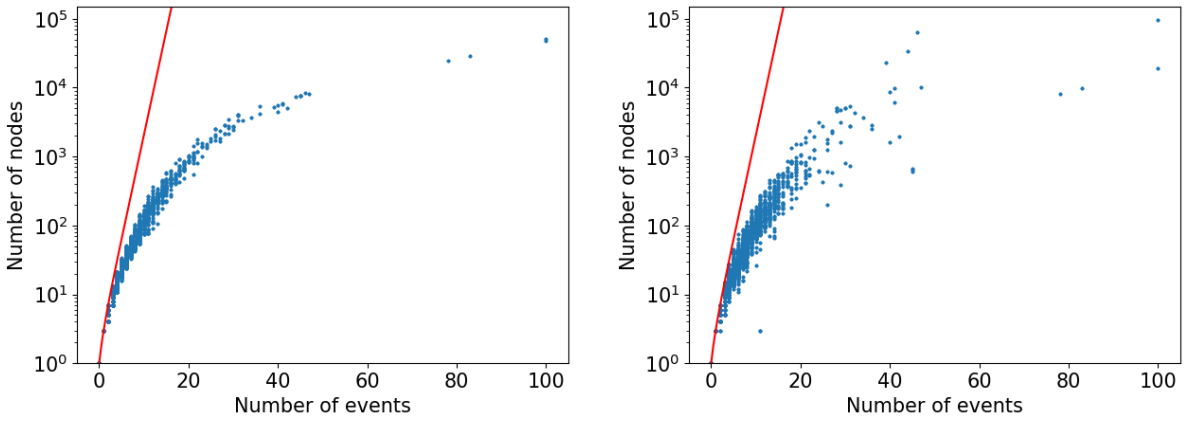
5.3 XDD Compactness

The idea of compactness comes from the observation that the amount of possible execution times of a BB is generally much less 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 3a and 3b show the results for both the simple and complex architectures. Each dot



(a) Simple architecture **(b)** Complex architecture

Figure 3. Number of leaves of resulting XDD with respect to the number of Events



(a) Simple architecture **(b)** Complex architecture

Figure 4. Number of nodes of resulting XDD with respect to the number of Events

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 does clearly 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 4a and 4b 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 the increase of

Instruction Level Parallelism induced by superscalar architectures which allow more variable patterns of instruction execution inside the pipeline.

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 [10, 11, 13], a simulator-based approach was proposed by Engblom et al. [6, 7]. Although it takes into account the overlapping of blocks in the pipeline, time events are added as-is to the final time (a) inducing an overestimation and (b) preventing the support for timing anomalies. Healy et al. in [9] 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 be applied to very simple microprocessors.

A first kind of generic method to compute basic block execution times was proposed by Kassem et al. in [12]. It uses automata to represent the different states of the pipeline. Transition between states are triggered by a mix of events recording the instruction execution phases and other hardware effects. Cassez et al. in [3, 5] 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 [14, 23] 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* [22] 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.

Basically used to optimize Boolean functions, BDDs have been successfully used to avoid explicit computation in symbolic model checking [4]. In a different context, Wilhelm et al. proposed to use BDDs to compact the pipeline state representation to perform abstract interpretation for pipeline analysis.[25]C1, C2, C3

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 speed up the analysis of the WCETs of basic blocks. The increase in performance comes from the exploitation of the latency absorbing properties of microprocessor pipelines. Moreover, we proved that this improvement comes at no cost with respect to the precision of the analysis. We also showed that using XDDs significantly reduces the empirical complexity of the analysis compared to the existing *Etime* method, allowing WCET analysis to be performed on larger and more complex applications. Experimentally, the analysis time was reduced to less than 1 second for all analyzed benchmarks from the *TACLe* suite (for a *split threshold* of 15), while the *Etime* method can take up to 7 minutes. Moreover, by observing the number of nodes and leaves in the XDDs, we confirmed our initial assumption that the pipeline mechanisms hide some execution latencies and can be efficiently 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 basic block has more than 15 events.

As future work, we plan to extend the applicability of XDDs to other models of architectures, like out-of-order pipelines,C5 and to further increase their performances. First, although XDDs significantly speed up the analysis of BBs possible execution times, the number of execution times for a complete application may still be too large for the the IPET ILP resolution. A method must be found to reduce the number of variables in the ILP system, while at the same time not increasing too much the pessimism of the estimated WCET. This issue is already addressed in the original *Etime* approach but the easy-to-handle structure of XDDs might open new ways to tighten the precision of the WCET.C4 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 a L1 cache could later cause a Miss in a 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 allowing the practical analysis of more complex architectures.

References

[1] H. R. Andersen. An introduction to binary decision diagrams. *Lecture notes, IT University of Copenhagen*, 1997.

[2] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, pages 35–46, 2010.

[3] J.-L. Béchennec and F. Cassez. Computation of WCET using program slicing and real-time model-checking. *CoRR*, abs/1105.1633, 2011.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 States and beyond. *Information and Computation*, 98:142–170, June 1992.

[5] F. Cassez and P.G.A. Marugán. Timed automata for modelling caches and pipelines. *arXiv preprint arXiv:1511.04172*, 2015.

[6] J. Engblom. *Processor Pipelines and and Static Worst-Case Execution Time Analysis*. PhD thesis, University of Uppsala, 2002.

[7] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proc. of 6th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, December 1999.

[8] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wägemann, and S. Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[9] C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[10] N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution time analysis for digital signal processors. In *10th European Signal Processing Conference*, pages 1–4, 2000.

[11] N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. *Space Systems Finland Ltd*, 2002.

[12] R. Kassem, M. Briday, J.-L. Béchennec, Y. Trinquet, and G. Savaton. Simulator generation using an automaton based pipeline model for timing analysis. In *International Multiconference on Computer Science and Information Technology*, pages 657–664. IEEE, 2008.

[13] R. Kirner. The wcet analysis tool calcwcet167. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 158–172. Springer, 2012.

[14] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *International Static Analysis Symposium*, pages 294–309. Springer, 2002.

[15] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 34(3):195–227, 2006.

[16] Y.-T. Steven Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, volume 30.11, pages 88–98, 1995.

[17] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[18] C. Meinel and A. Slobodová. On the complexity of constructing optimal ordered binary decision diagrams. In *Mathematical Foundations of Computer Science 1994*, pages 515–524. Springer, 1994.

[19] S.-I. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *27th ACM/IEEE Design Automation Conference*, pages 52–57, 1990.

[20] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, 2006.

[21] C. Rochange and P. Sainrat. A context-parameterized model for static analysis of execution times. *Transactions on High-Performance Embedded Architectures and Compilers II*, pages 222–241, 2009.

[22] I. J. Stein. *ILP-based path analysis on abstract pipeline state graphs*. PhD thesis, Saarland University, 2010.

[23] S. Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models*. PhD thesis, Saarland University, 2004.

[24] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[25] S Wilhelm. Efficient Analysis of Pipeline Models for WCET Computation. In *WCET'05*, 2007.