Using execution graphs to model a prefetch and write buffers and its application to the Bostan MPPA

Wei-Tsun Sun, Hugues Cassé, Christine Rochange, Hamza Rihani, Claire Maiza

Using execution graphs to model a prefetch and write buffers and its application to the Bostan MPPA*

Wei-Tsun Sun[1], Hugues Cassé[1], Christine Rochange[1], Hamza Rihani[2], and Claire Maïza[2]

[1] IRIT - University of Toulouse
[2] Verimag - Univ. Grenoble Alpes

{*wsun, casse, rochange*}@irit.fr
{*rihanih, claire.maiza*}@univ-grenoble-alpes.fr

Abstract

Verifying the temporal properties of critical systems embedded in vehicles, like planes or cars, is crucial to avoid catastrophic issues. A key component of this verification is the *Worst Case Execution Time* (WCET) of the programs composing these systems. A common and sound approach to compute WCET is based on static analysis of the programs that requires, in turn, to precisely model the behavior and the timings of the hardware.

Processor-specific features such as pipelines, caches, and buffers influence the hardware performances significantly. Hence taking processor features into account when estimating WCET is essential. Modeling the processor's features formally to ensure safe and accurate estimation is then a must. In this paper, we present the methodology applied to capture the behavior of prefetch and write buffers of the Kalray Bostan MPPA microprocessor, and to incorporate the established models with the *Execution Graph* (XG) to obtain WCET estimation. These analyses are then applied to the Mälardalen benchmark suite and the experimentation results validate the feasibility of our approach.

Keywords Real-Time, WCET, static analysis, VLIW, pipeline analysis

1 Introduction

Verifying the temporal properties of critical systems embedded in vehicles, like planes or cars, is crucial to avoid catastrophic issues. A key component of this verification is the *Worst Case Execution Time* (WCET) of the programs composing these systems. In this paper, we focus on the WCET computation by static analysis that requires to model the software and the host hardware in order to obtain a sound overestimation of the WCET. This computation is usually made of several analyses performed on the binary code and depending on the considered micro-architecture.

This paper presents the adaptation of the Execution Graph (XG) [14] approach to model two pipeline acceleration mechanisms: the prefetch buffer and the write buffer.

As a platform example containing write buffer and prefetch buffer we target the Kalray's Bostan MPPA core, where we will use its short name, k1b, throughout the paper. This architecture is designed for massive parallel computing and hence provides strong computational power. Yet, its design should also address the requirements for real-time applications, i.e. the deterministic and predictable behavior of the hardware. Therefore, tight WCET static estimation should naturally be achieved for this architecture: this paper also verifies this assumption.

Contributions: (a) the analysis of the *Prefetch Buffer* (PFB) and the *WriteBuffer* (WB), (b) the implementation in XG of their temporal behavior, (c) an approach based on a mix of static analyses and XG customization to cope with micro-architecture elements exhibiting complex behaviors.

Organization: The next section presents the state-of-the-art of WCET computation by static analysis and introduces the XG method and the concept of temporal events. Section 3 and 4 show the support, respectively, of PFB and WB, in the block time computation by XG. Related works are discussed in Section 5 and we conclude in Section 6.

2 Background

In this paper, we use the *Implicit Path Enumeration Technique* (IPET) [10] approach to compute an upper-bound on the WCET of programs in machine code. This approach is currently the most used as it is the most flexible to combine the timing effects of

*This work is supported by the project CAPACITES, funded by French DGE and BPI.

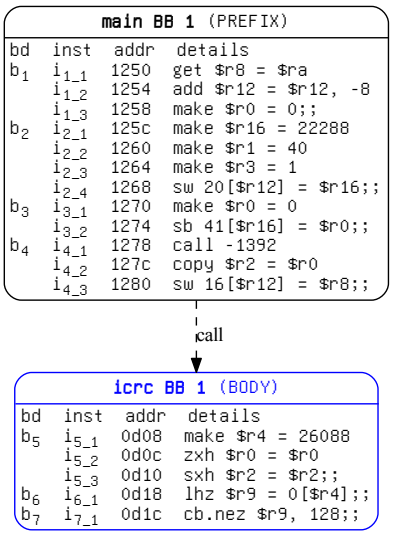

Figure 1: The partial CFG from CRC

the different components of the micro-architecture. It is often split into three passes: (a) extraction of the execution paths in the binary code as a *Control Flow Graph* (CFG) and analysis of flow facts (e.g. loop bounds), (b) analysis and qualification of the behavior of the acceleration mechanisms (like caches) and (c) combination of information provided by (a) and (b) to obtain the block times and the WCET.

The program is represented by a CFG: $G = (V, E, \epsilon)$ where V is the set of *Basic Blocks* (BB), $E = V \times V$ is the set of transitions between BB (by sequential execution or branches) and $\epsilon \in V$ is the entry point of the program. A basic block is a sequence of instructions with one entry point and one exit point: a branching instruction is always the last instruction of the BB.

In this paper we consider a *Very Long Instruction Word* architecture (VLIW). The instructions of a VLIW are grouped in bundle.

Following example: Throughout this paper, we use a code segment of the program *crc*, from the Mälardalen benchmarks [1], whose CFG is shown in Figure 1. In this example, function icrc (whose we only show the first BB) is called from BB 1 of the main function. Since the processor is implemented with VLIW architecture, a *bundle*, b_i, consists of multiple instructions executed together. The bundles are shown at the most left columns in the BB, where each individual instruction is labeled as i_{m_n}, indicating the n_{th} instruction of bundle b_m. The program address and the details of each instruction are given at the right of each BB.

2.1 The IPET approach

The basic formulation of IPET considers the WCET as the maximized objective function of an *Integer Linear Programming* (ILP) problem:

$$\text{WCET} = max \sum_{v \in V} t_v \; x_v \tag{1}$$

where x_v is the occurrence number of BB v on the longest path and coefficient t_v is the execution time of BB.

The WCET objective function is bound by structural constraints representing the aspect of the program execution. They encompass, at least, constraint representing the structure of the CFG: ϵ is executed once and a BB is executed as many times it is entered or left:

$$x_\epsilon = 1$$
$$\forall v \in V, x_v = \sum_{w \in SUCC(v)} x_{v \to w} = \sum_{w \in PRED(v)} x_{w \to v}$$

where $x_{v \to w}$ is the execution number of edge $v \to w \in E$ on the longest path, $SUCC(v)$, respectively $PRED(v)$, are the set of successors, predecessors, of $v \in V$ in CFG G.

If G contains a loop L_h (a loop headed by BB $h \in V$), we have also to bound the count of back-edge traversals ($BACK(h)$ provides the set of back edges of G sinking to h) with N, the maximum number of iteration of L_h:

$$\sum_{v \to h \in BACK(h)} x_{v \to h} \leq N$$

The ILP problem is then maximized by an ILP solver and the result is the estimated WCET of the program.

2.2 Execution Graphs

The IPET formulation of the previous paragraph uses coefficient t_v as the execution time of BB $v \in V$, that are the result of running the instructions of the BB on the micro-architecture (pipeline and acceleration mechanisms). This paper uses the *Execution Graph* (XG) [9, 14] approach to compute t_v.

An XG $G_X = (V_X, E_X)$ models the traversal of the pipeline by the bundles of a BB v: the set of vertices, $V_X = B_v \times S$, denoted $[b/s]$, represents the occupation of bundle $b \in B_v$ in the pipeline stage $s \in S$. The edges $E_X = V_X \times V_X$ represents the timing dependency between two vertices of V_X. Examples of XG edges include but are not limited to pipeline order, instruction order or data dependencies.

Example: Figure 2 shows the XG created from the sequence of two BB in the partial CFG shown in Figure 1. As target platform with write buffer

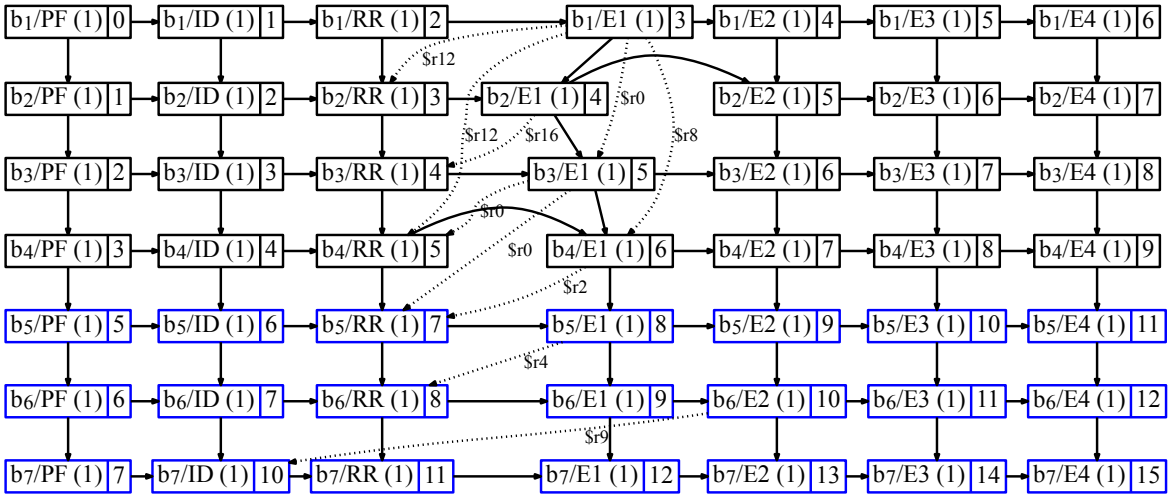

Figure 2: The XG without considering PFB and WB

and prefetch buffer we use the MPPA core k1b. k1b implements a 7-stage pipeline, that is, $S = \{PF,\ ID,\ RR,\ E1,\ E2,\ E3,\ E4\ \}$ for which details are given later in the paper. This example illustrates the usual types of arrows in XG: (a) the solid vertical arrows represents the sequential flow of bundles in the program, (b) the solid horizontal arrows represents the order of pipeline stages and (c) the slashed arrows represents data dependencies. The slashed arrows are applicable when a bundle uses the content of a register produced by another bundle: it links the vertex where the register value is computed to the vertex where it is used.

Not illustrated in Figure 2 but used later in the paper, the XG edges may be solid or slashed. Solid edges, $[b_i/s_i] \rightarrow [b_j/s_j]$ means that $[b_j/s_j]$ starts just after the end of execution of $[b_i/s_i]$. A slashed edge, $[b_i/s_i] \nrightarrow [b_j/s_j]$, means that $[b_j/s_j]$ can start as soon as $[b_i/s_i]$ starts.

The XG is used to compute the starting and the ending times of each vertex (a) considering the first vertex starts at time 0 and (b) a vertex can only start when all its predecessors have started (slashed edge) or ended (solid edge): the default duration of each stage is considered to be 1 cycle. As the XG does not contain any cycles, a calculation following the topological order of the graph is enough to obtain the start time of each vertex.

A naive approach to compute block time would consider that the block times is the difference between the starting time of the first bundle of the block and the ending of its last bundle: yet, this would lead to a significant overestimation as, because of the pipeline, the block executions overlap. Instead, we prefer to build an XG for a block (*body*) and each of predecessors (called *prefixes*) in the CFG (as in Figure 1). From there, the block time is more precisely defined as the difference between the ending time of last vertex of the body and the ending of the last vertex of the prefix.

To use a time based on the predecessor blocks, the Equation 1 is reformulated as:

$$\text{WCET} = max \sum_{v \rightarrow w \in E} t_{v \rightarrow w}\ x_{v \rightarrow w} \qquad (2)$$

where $t_{v \rightarrow w}$ is the time computed with v considered as the prefix and w as the block.

2.3 Support of variable execution times

The times $t_{v \rightarrow w}$ computed in the previous paragraph assume that the execution time of a bundle in a pipeline stage is fixed: 1 cycle. Yet, in order to ensure a good execution performance, acceleration mechanisms are introduced to counterbalance the relative slowness of the memory. These mechanisms (instruction cache, data cache, etc) make memory accesses on a statistical basis: most of the time, they allow faster accesses (*hit*) and, more rarely, they have to perform a slow access to the memory (*miss*).

In order to account for such an effect e, called an *event*, one has (a) to bound the number of occurrences of *hits*, $x^{h,e}_{v \rightarrow w}$, and *misses*, $x^{m,e}_{v \rightarrow w}$, and (b) to get the time in cycles of this access, $t^{h,e}_{v \rightarrow w}$ and $t^{m,e}_{v \rightarrow w}$. $t^{h,e}_{v \rightarrow w}$ and $t^{m,e}_{v \rightarrow w}$ are usually constants with $t^{h,e}_{v \rightarrow w} = 1$ and $t^{m,e}_{v \rightarrow w}$ depending on the hardware properties. On the opposite, an analysis, that depends on the acceleration mechanism itself, must be performed to obtain $x^{h,e}_{v \rightarrow w}$ and $x^{m,e}_{v \rightarrow w}$. An event is made of elements including $t^{h,e}_{v \rightarrow w}$, $t^{m,e}_{v \rightarrow w}$, $x^{h,e}_{v \rightarrow w}$, $x^{m,e}_{v \rightarrow w}$, and the component of the XG (a vertex or an edge) that each event applies to. An activation of an event indicates that the XG component time is set to $t^{m,e}_{v \rightarrow w}$.

Until now, we have just considered an event alone. Yet, a single BB may exhibit several events based on the effects of acceleration mechanism triggered by the bundles composing the BB. To soundly support

timing anomalies, we have to consider all combinations of activation / inactivation of events produced within the BB. The all possible combinations of events, $C_{v\to w}$, for a sequence of BB $v \to w$, the Equation 2 is rewritten to:

$$\text{WCET} = max \sum_{v\to w \in E} \sum_{c \in C_{v\to w}} t^c_{v\to w}\ x^c_{v\to w} \quad (3)$$

where $t^c_{v\to w}$ and $x^c_{v\to w}$ respectively represent the time and the count of occurrences of configuration c along the WCET path. New constraints on $x^c_{v\to w}$ must be set up to cope with the occurrences variables of each event e of $v \to w$, that is $x^{h,e}_{v\to w}$ and $x^{m,e}_{v\to w}$.

The remaining of the paper analyzes $x^{h,e}_{v\to w}$, $x^{m,e}_{v\to w}$, $t^{h,e}_{v\to w}$ and $x^{m,e}_{v\to w}$ for two accelerations mechanisms: the PFB and the WB.

3 Capturing the PFB in XG

As the VLIW pipeline consumes several instructions words at each cycle, a PFB is used to increase the throughput of instruction fetching from the instruction cache (IC). Assuming that most of the code is sequential, the PFB speculatively loads as many instructions as possible to amortize possible miss times in the IC. However, when a branch is performed, its content is discarded for refilling the actual instructions of the branching target, which introduces a delay in execution.

To summarize, possible delays from the PFB are due to: (1) the loading of the speculative instruction which may in turn trigger IC misses, and (2) clearing the PFB and refilling instructions from the actual target address. In this section, we will use the example illustrated in Figure 2 to show the behaviors of the PFB and how to capture them into the XG.

3.1 The relationship between the IC and the PFB

Due to the VLIW feature in k1b, one or more instructions are combined to form a bundle. The IC is organized into 64 sets of 64-byte cache blocks (CB). Figure 3(a) illustrates how our example is located in the IC. Note that each block in Figure 3(a) is of 1 word, and some instructions may span over more than one word, e.g. the i2_4, for which we use $i2\_4_1$ and $i2\_4_2$ to present the parts of the instruction. We use different colors to show the words are in different CBs. Words which are not part of the prefix nor the body are shaded and identified in the form of s_n to show how the PFB behaves.

k1b is equipped with a 48-byte prefetch buffer (PFB) organized as a set of 4 FIFOs, each FIFO consisting of 3 stages (elements). The size of each element is 1 word (4 bytes). The PFB fetches instructions from the IC in a request-reply manner. The data provided by the IC, to the PFB, may be an incomplete (partial) bundle, a full bundle, or multiple bundles whose total size is of 4 words. The PFB does not have any knowledge of the data received, in other words, it is bundle-unaware.

When each FIFO of the PFB has at least one available space, the PFB fetches 4 words from the IC on each request with a specified address. Each word provided by the IC will be fetched to the designated FIFO according to its address. For example, the word of instruction i1_1 located at the address 0x1250 will be assigned to the first FIFO, i1_2 at the address 0x1254 will be stored in the second FIFO and so forth.

Since only the prefix-body pair of BBs is considered when creating the corresponding XG, the state of the PFB at the beginning of the prefix is assumed with the worst-case scenario, i.e. the PFB is full. The words in the PFB will be taken by the ID stage, one after another, along with loading the prefix into the PFB. This leads to the state shown in Figure 3(b), which the PFB is filled by the instructions of the prefix.

3.2 From the PFB to the ID stage

Along with the fetching process, the ID stage extracts a bundle from the PFB, which in turn frees spaces for the next fetch. However, the newly available spaces may not be sufficient for the PFB to request the data from the IC. Figure 3(c) shows such a case: even though bundle b_1 (i1_1 to i1_3), which occupies the first elements of the 3 FIFOs, is fed to the ID, the 4th FIFO is still full and thereby no request will be sent to the IC. Once b_2 is extracted from the PFB, i4_3 will then be fetched, along with s1-s3 as shown in Figure 3(d). Once bundle b_3 is fully entered into the PFB given that b_2 leaves the PFB (extracted by the ID). Such a relationship between b_2 and b_4 is called a ***contention***. Note that in Figure 3(d) s4 to s7 are also fetched into the PFB to capture the worst-case scenario where the pipeline does not consume instructions faster than the PFB fetches them. Similarly, the extraction of b_3 does not provide sufficient space for the next fetch, as shown in Figure 3(e).

Figure 3(f) shows that once bundle b_4 is transfered to the ID, data s8 to s11 are loaded into the PFB. Even though data s1-s11 are of the same color as i4_3, they are shaded because they will be abandoned once the function call (i4_1) takes place. To address such a behavior, we call the loading of s1-s11 to the PFB ***speculation*** throughout this paper.

Function calls as well as branches will make the PFB clears its content and start to request the data of the target address from the IC. We identify this behavior as ***branch***. Note that the first instruction

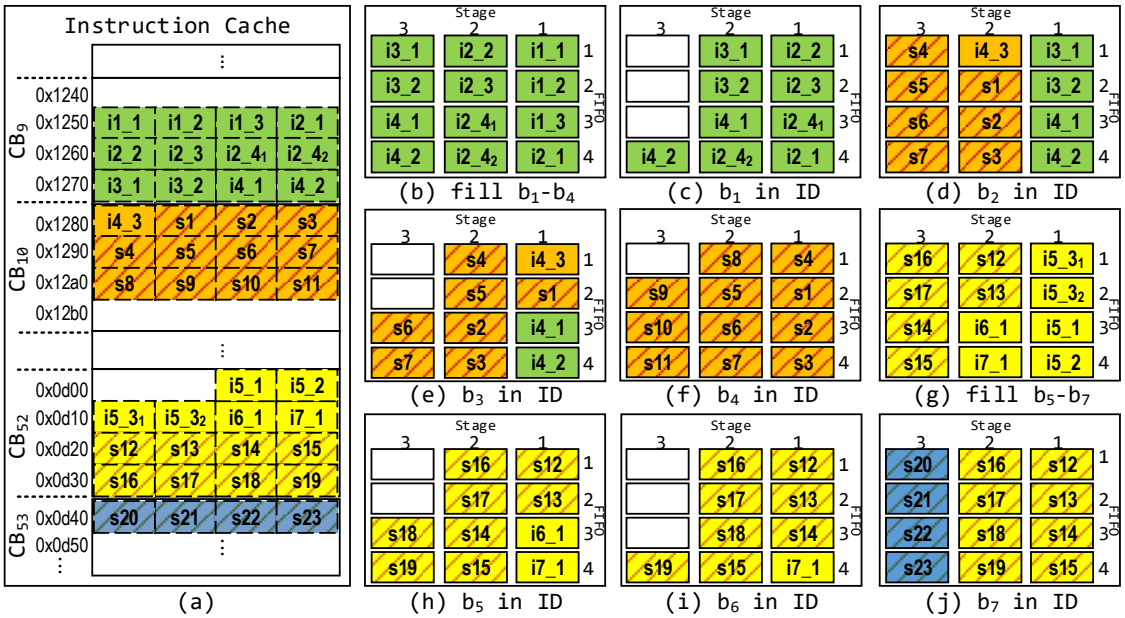

Figure 3: The layout of the IC and the state of the PFB when fetching

of the body BB, i5_1, is stored in the 3rd FIFO according its address. The ID is aware of the current program address, hence the correct FIFO will be used for bundle extraction. As result, in Figure 3(g), b_5-b_7 (i5_1-i_1) are fetched into the PFB, and similarly s12-s19 are fetched as the speculation. Figure 3(h) to (j) illustrate the state of the PFB as the BODY is progressively extracted to the ID. As in (j), since the conditional branch (bundle b_7, or the instruction i7_1) is executed in the RR stage, the PFB will not be cleared when the bundle reaches the ID. Meanwhile, the speculative s20-s23 (shaded in blue) are fetched into the PFB, as another example of speculation. Finally, note that bundles b_6 and b_7 are loaded into the PFB together: we call such relationship a ***chunk***. In contrast to the contention, bundles in the same chunk are loaded into the PFB with delay.

3.3 Formulating the PFB for XG

Before formulating the behavior of the PFB, the accesses to the IC must be first identified by the IC analysis. The analysis also determines whether the requests from the PFB to the IC will cause an IC hit or miss. We are interested in possible IC misses which introduce delays in the execution.

We first introduce some helper functions to simplify the formulas describing the PFB:

1. $chunk(b_i)$, returns the chunk in which b_i resides.
2. $chunkHead(b_i)$, returns true if a bundle is the head of the associated chunk.
3. $nextChunk(b_i)$, provides the next chunk from the current chunk contains b_i.
4. $latency(v)$, sets the latency of an XG vertex.
5. $contention(b_i, b_j)$, determines if b_i and b_j are in contention.

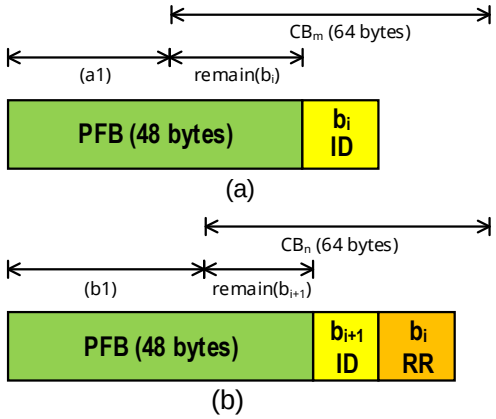

Figure 4: Calculate the number of CBs can fit into the PFB for (a) b_i enters the ID, and (b) b_i enters RR and b_{i+1} enters ID

6. $speculation(b_i)$, provides a set of CBs which might be accessed due to the speculation.
7. $remain(b_i)$, gives the remaining size (in byte) from the end of the b_i (non-inclusive) till the end of its associated CB.
8. $|b_i|$, provides the size of b_i.
9. $CB_{cross}(b_i)$, determines if b_i crosses two CBs.

The formulas applied on the XG that captures the behaviors of the PFB are categorized into: (1) chunks, (2) contention, (3) branch, (4) speculation, and (5) the access time of the IC. In the formulas, we use $[b_i/stage]$ to represent a vertex, a $\nrightarrow$ for a slashed arrow, and a $\rightarrow$ for a solid arrow in the XG.

For chunk:

(1a) $[PF/b_i] \rightarrow [PF/b_{i+1}]$ | $(chunk(b_i) = chunk(b_j)) \wedge chunkHead(b_i)$
In order to propagate the cost within the chunks in XG, a solid line is required for the first bundle (head) of the chunk and the next chunk.

(1b) $[b_i/PF] \nrightarrow [b_{i+1}/PF]$ |

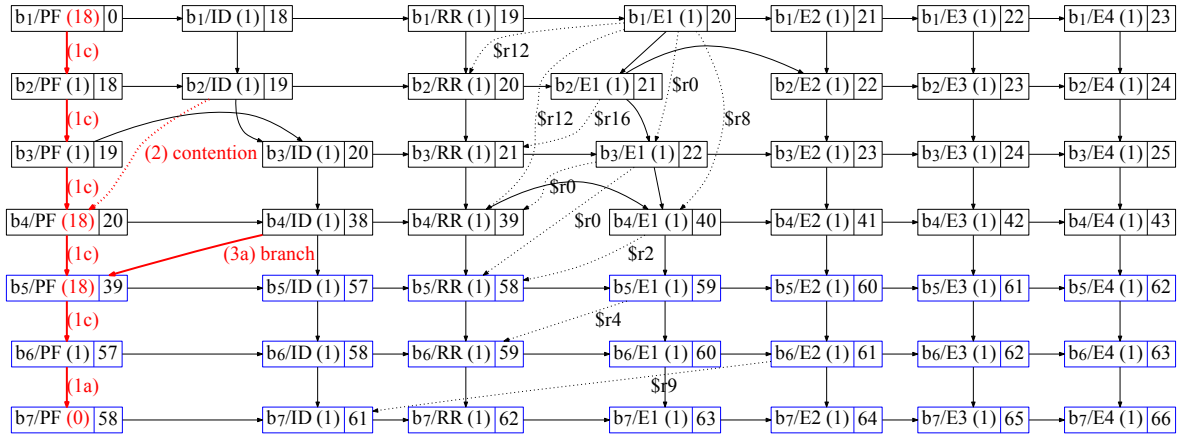

Figure 5: The XG including effects from PFB and the IC misses

$(chunk(b_i) = chunk(b_j)) \wedge \neg chunkHead(b_i)$
Since the bundles in the same chunk are fetched into the PFB together at the same time, a slashed arrow is used to capture this feature.

(1c) $[b_i/PF] \rightarrow [b_{i+1}/PF] \mid chunk(b_i) \neq chunk(b_{i+1})$
This captures the loading sequence of the bundles in the different and consecutive chunks. This arrow also carries the latency, if any, for loading the next chunk.

(1d) $[b_i/PF] \rightarrow [b_j/PF] \mid chunkHead(b_i) \wedge chunkHead(b_j) \wedge (b_i \neq b_j) \wedge (nextChunk(b_i) = chunk(b_j))$
This is a specialized case of (1c) in when b_i is the only bundle in the chunk and b_j is the next bundle of b_i.

(1e) $l([b_i/]) = 0 \mid \neg chunkHeader(b_i)$
As for the non-head bundles in a chunk, the latencies of the corresponding XB vertices will be set to 0, as they are loaded into the PFB at the same time without any delay in between.

For contention:

(2) $[b_h/ID] \Rightarrow [b_i/PF] \mid chunkHead(b_i) \wedge contention(b_h, b_i)$
This tells that, as soon as that b_h enters the ID stage, b_i will be pushed into the PFB. Because the rest of the chunk will share the same timing latency, according to the formula (1e), therefore it is not necessary to create the slashed arrows to the other bundles of the same chunk.

For branch:

(3a) $[b_i/ID] \rightarrow [b_{i+1}/PF]$
The target of the unconditional branches or the function calls are determined at the ID stage.

(3b) $[b_i/RR] \rightarrow [b_{i+1}/PF]$
Similarly, the conditional or the indirect branches require one extra cycle to compute the target address, therefore the creation of the solid arrow from the RR.

For speculation:

(4) $l([b_i/PF] \rightarrow [b_{i+1}/PF]) = sum(T_{IC}\_k), k \in speculation(b_i)$
Given that b_i contains a branch instruction, and the PFB can fit k CBs before it is emptied due to branching. The total IC access time for k CBs are associated with the arrow indicating the cost of the speculation before loading the branch target into the PFB.

We use Figure 4 to illustrate the scenarios of the speculation when the bundle containing the branching instruction that leaves the PFB. The number of the speculative CB accesses are computed as the follows:

For unconditional branches:
$ceil(max(0, |PFB| - remain(b_i)) / |CB|)$

For conditional or indirect branches:
$ceil(max(0, |PFB| + |b_{i+1}| - remain(b_i)) / |CB|)$

As shown in Figure 4(a), the PFB contains b_i and the remaining of the associated CB. Once b_i enters the ID, the available space of the PFB, a1, can be computed as $|PFB| - remain(b_i)$. The number of the CBs can fit into a1 is simply $a1 / |CB|$. We use the *ceil* function to round-up the integer result. For indirect branches, as depicted in Figure 4(b), the next bundle b_{i+1} enters the ID while b_i enters the RR. The available space (b1) can be calculated as $|PFB| - remain(b_{i+1})$ while $remain(b_{i+1}) = remain(b_i) - |b_{i+1}|$, we leave the rest for the readers.

For the access time of the IC:

(5) $l([b_i/PF]) = T_{IC} \mid CB_{cross}(b_i) \vee (i = 0)$
The latency of accessing a CB will be associated with the vertex with possible IC misses determined by the IC analysis.

Figure 5 illustrates the resulted XG that takes the PFB's behaviors into account, where the newly

augmented vertices and arrows are in red with the corresponding formulas. The latencies of the vertices $[b_1/PF]$, $[b_4/PF]$, and $[b_5/PF]$ are set to 18 cycles, which is the IC miss penalty, according to the rule (5). $[b_7/PF]$'s latency is set to 0 due to the rule (1e). The cost of the body becomes 66-43=23 cycles instead of 15-9=6 cycles in the original XG.

4 Capturing the WB in XG

A write buffer (WB) is used to reduce the time that takes to write data to the slower main memory preventing stalling the pipeline. Eventually, the data is written to the main memory following a specific retirement policy (such as FIFO or LRU). In this paper, we focus on a WB with an LRU (least recently used) retirement policy. We also consider that data can be updated directly in the WB. According to [5], such configuration is more predictable to use in

real-time systems. This also corresponds to the adopted WB in k1b used by our motivating example.

4.1 Description of the write buffer

The WB operates with the write-merge policy. On each store, the WB checks if it already contains the data associated to the store address. If so, the entry is updated and marked as the most recently used (MRU). If the WB does not contain such address and it is not full, an unoccupied entry is taken and marked with MRU. If the buffer is full, then the LRU entry is evicted to the main memory (MEM).

The WB works closely with the data cache (DC). For example, when a DC-miss occurs, the contents of the WB is checked to see if it contains the data associated with the load address. If so, to ensure data coherence, the contents of the WB is flushed to the MEM, prior to the DC refill from the MEM. Otherwise the WB stays untouched and the DC performs the cache-refill without delay. However, if an eviction due to store was in the process of writing to the MEM, the WB flush and DC refill needs to wait the completion of the eviction. For better performance, the *critical-word first* (or CWF) is featured to enable later instructions to obtain the loaded value as soon as it is available to the processor. However, such a feature will introduce a possible delay to any following store/load instructions that tries to access the MEM. A load instruction with a DC-hit will have no effect on both WB and DC, where the data is read directly from the DC and made available on stage E2.

Figure 6 illustrates the behaviors of the WB and the DC in one automaton. The represented WB consists of eight 64-bit entries. Notations DC_h and DC_m indicate the hit or the miss of the DC respectively. Similarly WB_h and WB_m represent whether the address to load is present (hit) or not (miss) in the WB. Five states are identified: (1) not-full, where the WB contains less than 8 entries, (2) eviction, when the WB is full and the LRU entry is to evict to the MEM, (3) refill, when load with DC-miss and miss in WB, (5) purge+refill, similar to the refill, with the flush due to the hit in WB, and (5) evict+refill, which is created due to the concurrent activities of the WB and the DC, represents the scenario when a store occurs after a refill resulted by DC_m and WB_m such that $|WB| = 7$. The WB has to wait for the CWF delay to perform the eviction.

The costs of accessing the MEM in the above scenarios are identified as the following:

1. t_e: evicting the LRU entry to the MEM.
2. $t_p(n)$: purging n occupied entries to the MEM
3. t_r: refilling the DC by loading from the MEM
4. t_c: the penalty of the critical-word first loading

4.2 WB Transition Graph

The contents of the WB plays a crucial role in estimating the access time to the MEM, i.e., the occupancy decides the flush time, and the contained addresses determine whether the WB needs to be flushed. To accurately capture the state of the WB, two analyses are performed: first, the address analysis to determine the targets of the stores, which are the entries of the WB; then the DC analysis to determine the activity of the DC and consequently the state transitions according to the mentioned automaton. The operations of the WB depend on its previous state, where the building of the XG only considers the prefix and the body BBs. It is then essential to collect the possible configurations of the WB, so that its actual behavior can be extracted and used when considering a segment of the program. Here we introduce the Write Buffer Transition Graph, WBTG, to represent the global configuration of the WB.

The $WBTG = (V_{WB}, E_{WB}, \epsilon_{WB})$ where: (i) $V_{WB} = (ACC_{mem}, WB_{conf})$ are vertices associated with memory accesses and the resulting WB configuration. (ii) $E_{WB} = (V_{WB} \times V_{WB}, Trans)$ are the edges connecting two WBTG vertices and the possible WB/DC state transitions. (iii) ϵ_{WB} is the entry point of the WBTG, which is not associated to any memory access, is associated with the initial configuration of the WB. We assume that, initially, the WB is full and each entry contains the address *top* (any value) so that the worst-case scenario can be captured with a combination of DC-miss and WB-hit. The construction of the WBTG is done by propagating the WB configurations throughout the memory accesses. The WB/DC state transitions are determined by referring the aforementioned automate.

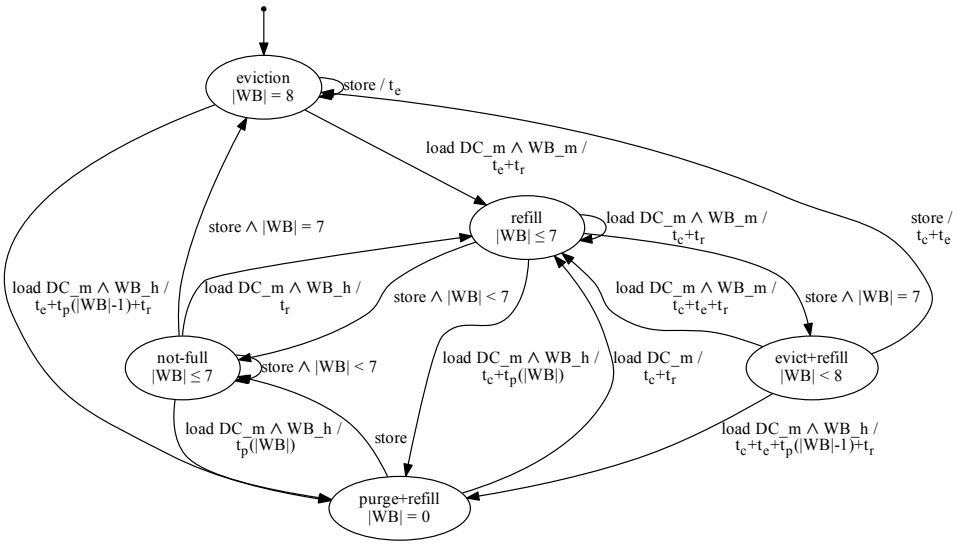

Figure 6: The automaton showing the behaviors of the WB and the DC

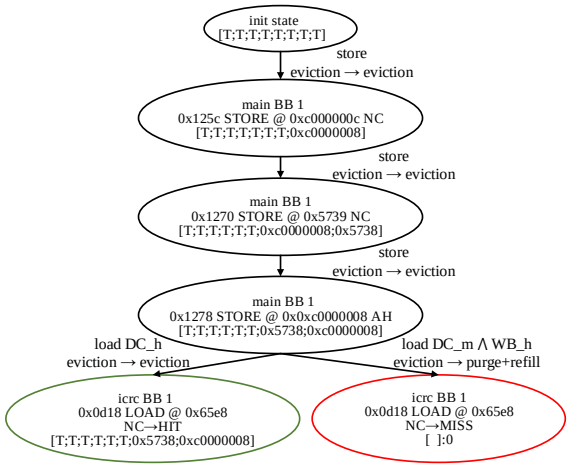

Figure 7: The WBTG of the sequence in Figure 2

Each time a new vertex is created, its associated WB configuration is used to obtain the next WB configurations in the upcoming memory accesses. The construction of the WBTG terminates once there is no more newly created WB configurations.

The partial WBTG for our example is illustrated in Figure 7. Each vertex is labeled with (for instance, the vertex below the entry vertex): (1) the BB (main BB1) where the memory access occurs, (2) the address of the memory access instruction (0x125c), (3) the type (store) of the access, (4) the target address of the access (0xc000000c), (5) the outcome of the access (NC - not classified), and (6) the WB configuration where the LRU entry is positioned at the left and the MRU is at the right.

It is worth noting that both the green and the red vertices are associated to the same memory access but with different WB configurations. This is due to the fact that the outcomes of some of the memory-load accesses can not be determined by the DC analysis. Hence they are assigned with category not-classified (NC). To cover all the possible cases, we create the green vertex indicating the DC-hit and the red vertex for the DC-miss.

4.3 Formulating the WB for XG

To account for the behaviors of the WB in the XG, we apply a similar approach to the one used for the PFB. The XG that includes the effects from the PFB and the WB is shown in Figure 8 where the arrows and the costs, created due to the WB effects, are green colored (effects from PFBs are in red).

By referring to the WBTG, the state transition between two memory accesses can be used to determine the order and the latency between two XG vertices. For example, from Figure 7 the first store (at 0x125c) in the main function is in the eviction state, and the next store (at 0x1270) leads to the same eviction state, with a cost of t_e, as shown in Figure 6. Because the eviction happens in stage E1, an arrow is created between the two XG vertices, $[b_2/E1]$ and $[b_3/E1]$, with latency $= t_e = 17$ cycles. Similarly, an arrow is created between $[b_3/E1]$ and $[b_4/E1]$.

Note that the arrow between $[b_4/E1]$ and $[b_6/E3]$ has a condition, b_6, which indicates that the latency of this arrow is only valid when the event on b_6, which is a DC-miss when loading, occurs in stage E3. The state will transit from *eviction* to *purge_refill*, by referring to the WBTG, with cost t_e on the arrow and the rest of the cost $(t_p(|WB|-1) + t_r)$ applied on the target vertex. The arrow from $[b_6/E3]$ to $[b_7/ID]$ represents the data-dependency where the data is made available after the DC refills.

5 Experiments

The Mälardalen benchmarks [1] compiled with the -O2 flag and are tested on Intel core i7-4810MQ processor at 2.8 GHz, with the results shown in Table 1. For each benchmark, we have collected the WCET, the time for performing the analyses, the total number of BB and the number of memory

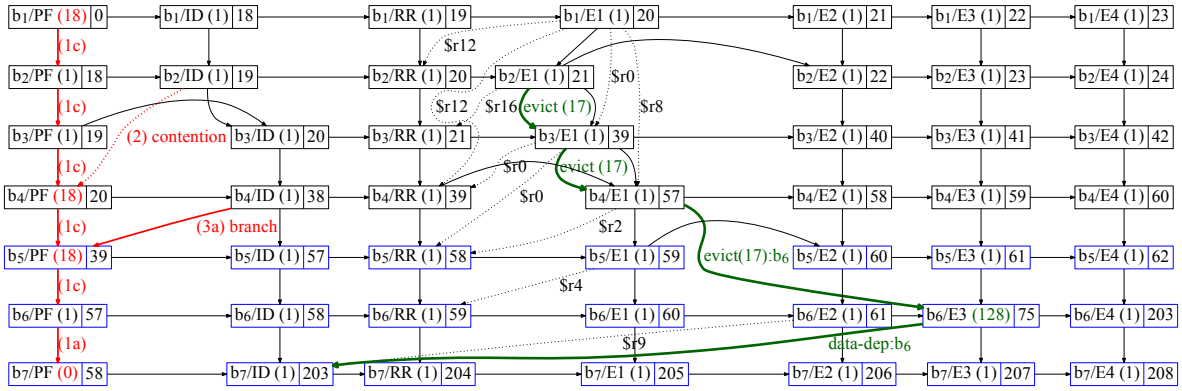

Figure 8: The XG including effects from both PFB and WB and events

accesses for each benchmark, followed by the number of the WBTG vertices created. We are able to obtain the WCETs for 20 out of 35 benches. The binaries of 3 benchmarks (expint, fibcall, and ns) only contain a single return value due to compiler optimizations. We tested these 3 benchmarks compiled with -O0 (non-optimized) flag. Due to a large numbers of events, resulted from the combinations of IC- and DC- misses, we were not able to compute the WCETs for 13 benchmarks (marked with §). The binaries of the 4 benchmarks (cover, duff, fac, and recursion, marked with ¶) contain irregular CFG structures which are currently not handled by our framework OTAWA [2]; however we were able to estimate the WCETs by switching to the -O0 flag for 2 of these 4 benchmarks (cover and fac).

Bench	WCET	time	BB	Accs	$\|WBTG\|$
adpcm§			369	677	5,948
bs	981	0.052	10	1	3
bsort100	3,097,290	0.06	14	7	130
cnt	6,030	0.14	21	13	153
compress§			148	167	98,306
cover¶	32,716	3.43	216	397	1,548
crc	189,786	0.52	53	23	572
duff¶					
edn§			107	102	59,372
expint*	129,056	0.36	96	16	56
fac¶	834	0.08	9	4	12
fdct	12,151	59.51	12	29	83
fft1§			1,661	761	4,043
fibcall*	2,751	0.09	12	24	76
fir	59,964	0.53	37	23	275
insertsort	12,223	0.06	8	10	49
janne	1,093	0.06	8	0	4
jfdctint	22,511	101.96	10	33	109
lcdnum	762	0.06	9	3	11
lms§			1,119	533	5,871
ludcmp§			336	167	249,124
matmult	2,387,321	0.82	29	20	1,050
minver§			347	1,980	244,628
ndes§			64	113	17,026
ns*	46,727	0.32	23	35	125
nsichneu§			756	2,400	11,737
prime	98,009	0.11	55	10	25
qsort	904,275	38.98	35	33	453
qurt§			2,102	912	5,060
recursion¶					
select	104,345	167.60	30	31	222
sqrt	34,567	5.90	189	47	271
st§			1,543	589	4,431
statemate§			321	604	18,548
ud§			107	174	18,582

Table 1: The WCETs for the Mälardalen benchmarks §: too many events ¶: irregular structure *: optimized

6 Related works

The static analysis of acceleration mechanisms of a pipeline represents a significant set of publications on WCET analysis. Roughly, they may be classified as working on graphs or on categories. The graph approach has mainly be applied on the earlier definition of IPET problem to model, in the ILP system, the effects of the caches [11, 12] but they were more recently used to model the complex behavior of branch predictors [13]. They consist in using a graph whose vertices are the possible states of the mechanism and edges are the transition between these states. As the graph is the result of possible executions of the mechanism on the CFG paths, (a) the transitions are related to the ILP variables and allow bounding the occurrences of long-time accesses (like misses) and (b) it may be translated into the ILP as it is done for the CFG. The main drawback is the possible explosion of the graph size and the resulting increase of the ILP system size.

On the opposite, the category approach classifies the behavior of the considered mechanism considering a small set of categories. These categories are then translated into the ILP with a limited set of constraints and variables, preventing the ILP size blowup of graph approach. These approaches have been successfully developed and applied for instruction and data caches [6, 7], multi-level caches [8], branch prediction [4], write-back caches [3], etc. The downside of the approach is that too complex behaviors have to be categorized as *Not Classified* and the resulting bound of long-time event occurrences leads to an overestimation. This may come from the program itself or from the intrinsic behavior of the mechanism. For example, the analyses of *round-robin* and *pseudo-LRU* replacement policies for the cache are less precise than the LRU policy.

The PFB analysis proposed here does not use any CFG-level analysis as the effects are local to the BB: expressing them in the XG is enough to cope with the PFB even if this causes a little overestimation for the *prefix* BB. Yet, it relies on the analysis of the instruction cache and even modifies it a bit to account for speculative block loading. On the opposite, analyzing the WB requires a complex CFG-level analysis requiring data cache analysis. Yet, the use of the graph to calculate the WCET: the built graph is the same as used in the graph approach but the full graph is not translated to the ILP system: an approximation is performed to limit the quantity of additional variables and constraints.

7 Conclusion

Execution Graphs provide an efficient and precise approach to compute the execution time of an instruction block in a pipeline, considering the overlapping with preceding blocks. This paper applies successfully this approach to model the VLIW pipeline of the Kalray Bostan MPPA including effects of instruction and data caches as well as two specific acceleration mechanisms, the *Prefetch Buffer* and the *Write Buffer*. These components have a straight impact on the performances of the pipeline but require a special processing, which consists of XG tuning and modeling their behavior along the execution paths of the program.

The application to the Mälardalen benchmark suite is successful in most cases. In the future, we plan to relax the encountered shortcomings. First, the well-known issue of size blowup of graphs modeling an acceleration mechanism can be fixed or reduced considering more abstract states at the cost of precision loss: the abstraction level may be tuned according to the size of the graph or the importance of the information.

Another important limitation concerns the production of events that, in turn, causes the number of times to compute for a BB to grow. Either, we can try to limit as much as possible the number of events by aggregating correlated events; yet, the soundness of the calculation might be preserved, that is, to maintain the right support of timing anomalies. Or the block time calculation method must be improved to support so many events: (a) blocks with too many events may be split, at the cost of loosing precision, (b) more correlations between events must be found that means, in turn, to improve the precision of static analyses producing the events or (c) the XG calculation algorithm could take into account the time variations.

In the end, the precision of WCET calculation by static analysis requires to cope with the full complexity of the pipeline meaning precise static analyses and support of the time variation of blocks. The XG approach seems to be a promising solution to this problem.

References

[1] The mälardalen wcet benchmarks: Past, present and future.

[2] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: an open toolbox for adaptive wcet analysis. In *SEUS*, pages 35–46, 2010.

[3] Tobias Blaß, Sebastian Hahn, and Jan Reineke. Write-back caches in WCET analysis. In *ECRTS*, pages 26:1–26:22, 2017.

[4] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. Technical Report PI-1236, IRISA, March 1999.

[5] Robert I. Davis, Sebastian Altmeyer, and Jan Reineke. Analysis of write-back caches under fixed-priority preemptive and non-preemptive scheduling. technical report. Technical report, University of York, 2016.

[6] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction, 1997.

[7] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. *Lecture notes in computer science*, pages 16–30, 1998.

[8] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. *RTSS'08*, 2008.

[9] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *Real-Time Systems Symposium, 2004.*, pages 92–103, 2004.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 88–98, 1995.

[11] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modelling and path analysis for real-time software. *Real-Time Systems Symposium*, pages 254–263, December 1995.

[12] Y-TS Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Real-Time Systems Symposium, 1996*, pages 254–263, 1996.

[13] C. Maiza and C. Rochange. History-based schemes and implicit path enumeration. In *WCET*, 2006.

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