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

To cite this version:
hal-02441594

HAL Id: hal-02441594
https://hal.science/hal-02441594
Submitted on 15 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Using execution graphs to model a prefetch and write buffers and its application to the Bostan MPPA∗

Wei-Tsun Sun1, Hugues Cassé1, Christine Rochange1, Hamza Rihani2, and Claire Maïza2

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
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 \( i_{mn} \), 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:

\[
WCET = \max \sum_{v \in V} t_v \times x_v
\]

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: \( \epsilon \) is executed once and a BB is executed as many times it is entered or left:

\[
x_{\epsilon} = 1
\]

\[
\forall v \in V, x_v = \sum_{w \in SUCC(v)} x_{v \rightarrow w} = \sum_{w \in PRED(v)} x_{w \rightarrow v}
\]

where \( x_{v \rightarrow w} \) is the execution number of edge \( v \rightarrow w \in E \) 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 \( L_h \):

\[
\sum_{v \rightarrow h \in BACK(h)} x_{v \rightarrow h} \leq N
\]

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

$$\text{WCET} = \max \sum_{v \rightarrow w \in E} t_{v \rightarrow w} \cdot x_{v \rightarrow w}$$  \hspace{1cm} (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, called an event, one has (a) to bound the number of occurrences of hits, $x_{v \rightarrow w}^{h,c}$, and misses, $x_{v \rightarrow w}^{m,c}$, and (b) to get the time in cycles of this access, $t_{v \rightarrow w}^{h,c}$ and $t_{v \rightarrow w}^{m,c}$. $t_{v \rightarrow w}^{h,c}$ and $t_{v \rightarrow w}^{m,c}$ are usually constants with $t_{v \rightarrow w}^{h,c} = 1$ and $t_{v \rightarrow w}^{m,c}$ depending on the hardware properties. On the opposite, an analysis, that depends on the acceleration mechanism itself, must be performed to obtain $x_{v \rightarrow w}^{m,c}$ and $x_{v \rightarrow w}^{m,c}$. An event is made of elements including $t_{v \rightarrow w}^{h,c}$, $t_{v \rightarrow w}^{m,c}$, $x_{v \rightarrow w}^{h,c}$, $x_{v \rightarrow w}^{m,c}$, 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 $t_{v \rightarrow w}^{m,c}$.

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, $C_{v \rightarrow w}$, for a sequence of BB $v \rightarrow w$, the Equation 2 is rewritten to:

$$WCET = \max \sum_{v \rightarrow w \in E \in C_{v \rightarrow w}} t_{v \rightarrow w}^c x_{v \rightarrow w}^c$$

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

The remaining of the paper analyzes $x_{v \rightarrow w}^{h,e}$, $x_{v \rightarrow w}^{m,e}$, $t_{v \rightarrow w}^{h.e}$, and $x_{v \rightarrow w}^{m.e}$ 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 reffilling 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.a, for which we use i2.a1 and i2.a2 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 b1 (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 b2 is extracted from the PFB, i4.3 will then be fetched, along with s1-s3 as shown in Figure 3(d). Once bundle b3 is fully entered into the PFB given that b2 leaves the PFB (extracted by the ID). Such a relationship between b2 and b4 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 b3 does not provide sufficient space for the next fetch, as shown in Figure 3(e).

Figure 3(f) shows that once bundle b4 is transferred 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 clear 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
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), b5-b7 (i5_1-i1_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 b7, 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 b6 and b7 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(bi)**, returns the chunk in which bi resides.
2. **chunkHead(bi)**, returns true if a bundle is the head of the associated chunk.
3. **nextChunk(bi)**, provides the next chunk from the current chunk contains bi.
4. **latency(v)**, sets the latency of an XG vertex.
5. **contention(bi, bj)**, determines if bi and bj are in contention.

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 [bi/stage] to represent a vertex, a → for a slashed arrow, and a \( \neg \rightarrow \) for a solid arrow in the XG.

**For chunk:**

1. \[ \text{chunk}(b_i) \rightarrow \text{chunk}(b_{i+1}) \] \( \land \) \( \text{chunkHead}(b_i) \)

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

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

6. **speculation(bi)**, provides a set of CBs which might be accessed due to the speculation.
7. **remain(bi)**, gives the remaining size (in byte) from the end of the bi (non-inclusive) till the end of its associated CB.
8. |bi|, provides the size of bi.
9. **CB_cross(bi)**, determines if bi 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 [bi/stage] to represent a vertex, a → for a slashed arrow, and a \( \neg \rightarrow \) for a solid arrow in the XG.
(chunk(b_i) = chunk(b_j)) ∧ ¬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) \[ b_i/\text{PF} \rightarrow b_{i+1/\text{PF}} \mid \text{chunk}(b_i) \neq \text{chunk}(b_{i+1}) \]
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) \[ b_i/\text{PF} \rightarrow b_j/\text{PF} \mid \text{chunkHead}(b_i) \land \text{chunkHead}(b_j) \land (b_i \neq b_j) \land (\text{nextChunk}(b_i) = \text{chunk}(b_j)) \]
This is a specialized case of (1c) in which b_i is the only bundle in the chunk and b_j is the next bundle of b_i.

(1e) \[ l([b_i/\text{PF}]) = 0 \mid \neg \text{chunkHeader}(b_i) \]
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) \[ b_h/\text{ID} \Rightarrow b_i/\text{PF} \mid \text{chunkHead}(b_i) \land \text{contention}(b_h, b_i) \]
This tells that, as soon as 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) \[ b_{i/\text{ID}} \rightarrow b_{i+1/\text{PF}} \]
The target of the unconditional branches or the function calls are determined at the ID stage.

(3b) \[ b_{i/\text{RR}} \rightarrow b_{i+1/\text{PF}} \]
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) \[ l([b_i/\text{PF}]) = T_{IC} \cdot k, \quad k \in \text{speculation}(b_i) \]
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:

\[ \text{ceil}(\text{max}(0, |\text{PFB}| - \text{remain}(b_i)) / |\text{CB}|) \]

For conditional or indirect branches:

\[ \text{ceil}(\text{max}(0, |\text{PFB}| + [b_{i+1} - \text{remain}(b_i)] / |\text{CB}|) \]

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 |\text{PFB}| - \text{remain}(b_i). The number of the CBs can fit into a1 is simply a1 / |\text{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 |\text{PFB}| - \text{remain}(b_i+1) while \text{remain}(b_{i+1}) = \text{remain}(b_i) - [b_{i+1}] , we leave the rest for the readers.

For the access time of the IC:

(5) \[ l([b_i/\text{PF}]) = T_{IC} \cdot CB\text{cross}(b_i) \text{ or } i = 0 \]
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
\([b_1/\text{PF}], [b_6/\text{PF}], \text{and} [b_9/\text{PF}]\) are set to 18 cycles,
which is the IC miss penalty, according to the rule
(5). \([b_7/\text{PF}]\)'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 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 DCₙₜₜ and DCₙₘₘ denote the hit or the miss of the DC
respectively. Similarly WBₙₜ and WBₙₘ 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, (4) 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 DCₙₚ and WBₙₚ 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. \(t_c\): 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) \(E_{WB}=(V_{WB} \times V_{WB}, \text{Trans})\) are the
edges connecting two WBTG vertices and the possible
WB/DC state transitions. (iii) \(\epsilon_{WB}\) 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.
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 $[b_2/E1]$ 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 $[br/ID]$ 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
The analyses of round-robin and pseudo-LRU replacement policies for the cache are less precise than the LRU policy.

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 been 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 accesses for each benchmark, followed by the number of the WBTTG 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).

<table>
<thead>
<tr>
<th>Bench</th>
<th>WCET</th>
<th>time</th>
<th>BB</th>
<th>Accs</th>
<th>WBTG</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bs</td>
<td>981</td>
<td>0.052</td>
<td>10</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>bsort100</td>
<td>3,097,290</td>
<td>0.06</td>
<td>14</td>
<td>7</td>
<td>130</td>
</tr>
<tr>
<td>cut</td>
<td>6,030</td>
<td>0.14</td>
<td>21</td>
<td>13</td>
<td>133</td>
</tr>
<tr>
<td>compress</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cover</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crc</td>
<td>189,786</td>
<td>0.52</td>
<td>53</td>
<td>23</td>
<td>572</td>
</tr>
<tr>
<td>duff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>edn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expint</td>
<td>129,056</td>
<td>0.36</td>
<td>96</td>
<td>16</td>
<td>56</td>
</tr>
<tr>
<td>fac</td>
<td>834</td>
<td>0.08</td>
<td>9</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>fdict</td>
<td>12,151</td>
<td>0.31</td>
<td>12</td>
<td>19</td>
<td>83</td>
</tr>
<tr>
<td>fft</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fibcall</td>
<td>2,751</td>
<td>0.09</td>
<td>12</td>
<td>24</td>
<td>76</td>
</tr>
<tr>
<td>fir</td>
<td>59,964</td>
<td>0.53</td>
<td>37</td>
<td>23</td>
<td>275</td>
</tr>
<tr>
<td>insertsort</td>
<td>12,223</td>
<td>0.06</td>
<td>8</td>
<td>10</td>
<td>49</td>
</tr>
<tr>
<td>janne</td>
<td>1,093</td>
<td>0.06</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>jfdctint</td>
<td>22,511</td>
<td>101.96</td>
<td>10</td>
<td>33</td>
<td>109</td>
</tr>
<tr>
<td>lednum</td>
<td>762</td>
<td>0.06</td>
<td>9</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>lns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lucencp</td>
<td>566</td>
<td>107</td>
<td>249,124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>matmul</td>
<td>2,387,321</td>
<td>0.82</td>
<td>29</td>
<td>20</td>
<td>1,050</td>
</tr>
<tr>
<td>minver</td>
<td>347</td>
<td>1,980</td>
<td>244,628</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ndes</td>
<td>64</td>
<td>113</td>
<td>17,026</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ns</td>
<td>46,727</td>
<td>0.32</td>
<td>23</td>
<td>35</td>
<td>125</td>
</tr>
<tr>
<td>nschneu</td>
<td>756</td>
<td>2,400</td>
<td>11,737</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prime</td>
<td>98,009</td>
<td>0.11</td>
<td>55</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>qsort</td>
<td>904,275</td>
<td>38.98</td>
<td>35</td>
<td>33</td>
<td>453</td>
</tr>
<tr>
<td>qurt</td>
<td>2,102</td>
<td>912</td>
<td>5,060</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recursion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>104,345</td>
<td>167.60</td>
<td>30</td>
<td>31</td>
<td>222</td>
</tr>
<tr>
<td>sqrt</td>
<td>34,567</td>
<td>5.90</td>
<td>189</td>
<td>47</td>
<td>271</td>
</tr>
<tr>
<td>st</td>
<td>1,543</td>
<td>589</td>
<td>4,431</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stateate</td>
<td>321</td>
<td>604</td>
<td>18,548</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ud</td>
<td>107</td>
<td>174</td>
<td>18,582</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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