Modeling Instruction cache and instruction buffer for performance estimation of VLIW architectures using native simulation

Omayma Matoussi and Frédéric Pétrot
TIMA Laboratory, Univ. Grenoble Alpes/CNRS/Grenoble INP,
F-38000, Grenoble, France
omayma.matoussi@imag.fr, frederic.petrot@imag.fr

***Abstract*—In this work, we propose an icache performance estimation approach that focuses on a component necessary to handle the instruction parallelism in a very long instruction word (VLIW) processor: the instruction buffer (IB). Our annotation approach is founded on an intermediate level native-simulation framework. It is evaluated with reference to a cycle accurate instruction set simulator leading to an average cycle count error of 9.3% and an average speedup of 10.**

1. Introduction

Software evaluation is usually performed using simulation approaches as they help recreate the dynamic behavior of the system as well as the interactions between the different hardware and software components. Many reserachers have resorted to native simulation [4], [7] as it offers a high simulation speed as opposed to conventional instruction set simulation (ISS). However, native simulation lacks information about non-functional properties like execution time.

These non-functional aspects are tightly influenced by the behavior of the micro-architectural components, like caches, and their inter-dependency with software. Simulating cache behavior is a key factor to accurate performance estimates [12], [13]. Most existing native simulation techniques use generic tag-search based icache models.

Although these generic cache models may work for scalar architectures, they lead to erroneous estimations when used for VLIW processors. The current breed of processors, *e.g.* MPPA manycore by Kalray, ST200 series by ST microelectronics, Tilera's TILE-Gx, etc, makes use of VLIW architectures to achieve high execution speed at low energy [6].

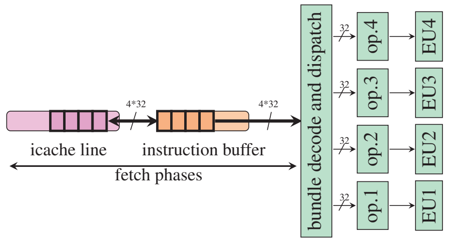


Figure 1. Overview of a VLIW architecture

In a VLIW architecture, a bundle is a set of independent instructions that can be executed in parallel (also called execute packets or VLIW instructions [1]), each instruction being composed of syllables. During execution, a contiguous sequence of syllables (a.k.a fetch packet), is fetched from the icache and placed in a buffer (referred to as instruction buffer [1], [5], [11] or prefetch buffer-PFB [3]) as shown Fig. 1. A fetch packet is thus composed of either a subset of a bundle, a single bundle or multiple bundle(s).

In the example of Fig. 1 four 32-bit instructions (*i.e.* a fetch packet of 128 bits) are fetched from the icache and pushed in the instruction buffer (IB). Then, during the decode and dispatch stage, the fetch packet is decoded, exposing the constituent bundle(s). The fetch packet in the example of Fig. 1 is composed of one bundle, that is, all four instructions of the fetch packet are dispatched to the four existing functional units and executed simultaneously.

Although the VLIW technique does not have an impact on the behavior of the cache in terms of hits/misses, it certainly impacts the performance in terms of CPI (cycle per instruction).

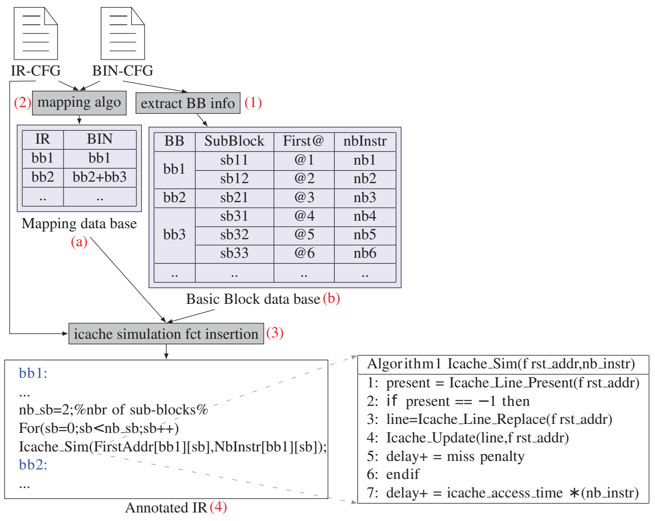
In case of scalar processors, computing the number of hits and misses using the traditional cache models and annotating the software with their respective time delays is a feasible approach and it has been abundantly adopted in literature. Yet, a naïve estimation of the number of cycles based solely on the number of cache hits and misses will lead to unreliable results for VLIW architectures that exhibit a complex timing behavior.

The objective of our work is to simulate the behavior of an instruction cache and take into account the effects of the instruction buffer on the performance of the cache.

2. Generic instruction cache simulation

In order to simulate the instruction cache behavior dynamically, a cache model identical to the real cache is used. Thus, we replicated all the characteristics of the hardware cache such as associativity, size, number of lines and replacement policy. This icache model is widely adopted in literature [10], [13] and [12]. However, existing approaches that simulate the icache dynamically settle for classic architectures and do not take into account common performance-enhancing features such as the ones used in VLIW processors which will be addressed in this paper.

The instructions' addresses are also required. These addresses are known at compile time and are extracted from



IR	BIN
bb1	bb1
bb2	bb2+bb3
..	..

BB	SubBlock	First@	nbInstr
bb1	sb11	@1	nb1
	sb12	@2	nb2
bb2	sb21	@3	nb3
bb3	sb31	@4	nb4
	sb32	@5	nb5
	sb33	@6	nb6
..	..	..	..

```
bb1:
...
nb_sb=2;%nbr of sub-blocks%
For(sb=0;sb<nb_sb;sb++)
Icache_Sim(FirstAddr[bb1][sb],NbInstr[bb1][sb]);
bb2:
...
```

```
Algorithm1 Icache_Sim(f rst_addr,nb_instr)
1: present = Icache_Line_Present(f rst_addr)
2: if present == −1 then
3:     line=Icache_Line_Replace(f rst_addr)
4:     Icache_Update(line,f rst_addr)
5:     delay+ = miss penalty
6: endif
7: delay+ = icache_access_time ∗(nb_instr)
```

Figure 2. The instruction cache simulation process

the target binary code (Fig. 2-(1)). To reduce tag search, each basic block is divided into sub-blocks such that all the instructions of a sub-block fit in the same cache line. A basic block data base is created (Fig. 2-(b)) to hold the statically collected information about each binary basic block: number of sub-blocks, first address and number of instructions of each sub-block.

Our annotation approach is based on ILS (intermediate level simulation [2], [4]). So the annotations are inserted inside the basic blocks of the IR (intermediate representation) code. It should be noted that the IR and binary control flow graphs (CFGs) are not always isomorphic due to compiler optimizations [2], [4]. So, a mapping between the binary CFG and the IR CFG is conducted (Fig. 2-(2)) resulting in a mapping data base (Fig. 2-(a)). A detailed explanation about ILS and the mapping process can be found in [8].

The annotation function $Icache\_Sim$ is then inserted in the IR basic blocks (Fig. 2-(3)). Algo.1 (Fig. 2) illustrates the instruction cache annotation function. $Icache\_Sim$ takes as parameters the first address and the number of instructions of a given sub-block. Only the tag and the index are needed as no real data will be loaded in the cache.

3. Instruction cache and instruction buffer performance estimation in a VLIW architecture

In a generic cache simulation approach, the delay is computed as follows: (1) $delay \mathrel{+}= nbr\_misses * miss\_penalty + nbr\_accesses * icache\_access\_time$. This formula is based on a simple association of number of hits and misses to their respective delays.

This approach, though efficient with scalar processors, is incapable of accurately reflecting the impact of VLIW architectures on the performance of instruction caches.

Operations	Scalar processor	WLIW processor with 2 load/store units 1 add unit,1 mul unit
op1: m=a+b	load a r1	load a r1 // load b r2
op2:n=c*d	load b r2	load c r3 // load d r4 // add r1 r2 r5
	add r1 r2 r5	mul r3 r4 r6 // store r5 m
	store r5 m	store r6 n
	load c r3	
	load d r4	
	mul r3 r4 r6	
	store r6 n	

Figure 3. Scalar vs. VLIW

3.1. The effect of VLIW on instruction cache performance estimation

Fig. 3 shows a comparison between the execution of two simple operations using a scalar processor and a rudimentary VLIW processor with two load/store units, one multiply and one add unit. The symbol ”//” portrays parallelism between instructions.

As can be noticed, the VLIW processor executes up to three instructions simultaneously. In our example, instructions are grouped in 4 bundles. Assuming each instruction can be completed in one unit of time then the VLIW processor takes 4 units which is half the time taken by the scalar processor.

At the basic block information extraction step, instead of dividing basic blocks into sub-blocks like we did in the generic cache simulation method (Fig. 2-(b)), we divide basic blocks into bundles. Each bundle is then divided into sub-bundles. A sub-bundle has the same definition as a sub-block. The cache annotation function is called for each sub-bundle. If the first instruction of the sub-bundle is in the cache then $delay \mathrel{+}= icache\_access\_time$, otherwise $delay \mathrel{+}= miss\_penalty + icache\_access\_time$.

Since the instructions in a sub-bundle are executed synchronously, only the first instruction is considered in the delay computation as opposed to eq. (1).

3.2. Instruction buffer impact on instruction cache performance estimation

At this level, we handled parallelism inside bundles. However, we considered bundles to be executed sequentially. In many VLIW architectures ([3], [5], [11]), the icache is

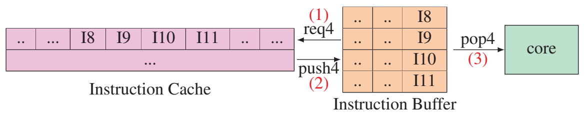


Figure 4. Instruction Buffer

further enhanced with an instruction buffer (Fig. 4). An instruction buffer is in charge of fetching a sequence of instructions (*i.e.* a fetch packet) from the icache and providing it to the core. The interesting aspect of an IB is the ability to fetch instructions ahead of time (before the core even requests them) which offers a sort of parallelism among fetch packets.

In the example of Fig. 4, the IB is composed of four FIFOs with three stages each. The IB can hold four 32-bit instructions in each one of its three stages. Each clock cycle, the IB requests four instructions from the icache (Fig. 4-(1)). The cache pushes the four requested instructions into the instruction buffer's FIFOs (Fig. 4-(2)) which are then popped into the next stages of the pipeline (Fig. 4-(3)) to be decoded and executed as bundles. In the remaining of this article, the IB example (Fig. 4) will be used to explain the IB behavior and its influence on the execution time. For clarity reasons and without loss of generality, we assume that a fetch packet is composed of one bundle. Thus, the terms "bundle" and "fetch packet" will be used interchangeably.

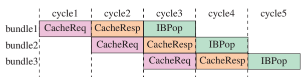


Figure 5. Nominal Case

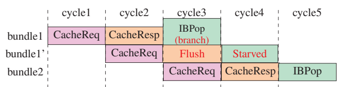


Figure 6. Branch Case

Breaking up the execution of bundles into sub-steps enables the bundles to overlap (to be executed partially at the same time) and offers better performances. As can be depicted Fig. 5, the sub-steps are: *CacheRequest*, *CacheRespond*, and *IBPop* which respectively correspond to (1), (2) and (3) in Fig. 4. These sub-steps constitute the 3 phases of the pipeline's fetch stage.

Given the VLIW architecture and with the instruction buffer in the picture, the parallelism is both intra- and inter-bundles. So, the delay computation is more complicated than explained in subsection 3.1 as several cases arise:

- In the nominal case (Fig. 5), the IB requests 4 instructions from the cache in the first cycle. In the following cycle, assuming that the instructions hit in the cache and are in the same cache line, the cache pushes the requested instructions in the buffer and the buffer initiates a new request during the same cycle. In the third cycle, the core requests a bundle of four 32-bit instructions which is already in the IB. So, the bundle is popped into the pipeline. During the third cycle, three bundles are handeled simultaneously: the first bundle is already in the pipeline, the second bundle is in the IB and the third bundle is fetched from the cache. Starting from the fourth cycle a steady state is reached where a bundle is executed per cycle.

To be able to reflect the overlapping of bundles in the delay computation, the delay formula of the nominal case (i.e. cache hit, the requested instructions reside in the same cache line and the core requests a 4-instruction bundle) is: $(2)\ delay_{nom} = k + nbr\_bundles - 1$, where k is the number of phases of the fetch stage (3 in our case), and $nbr\_bundles$ is the number of the executed bundles. For example, the delay of the 3 bundles Fig. 5 is 5 cycles. In order to keep track of the number of executed bundles, a bundle counter is introduced in the IR code. During simulation, each time a basic block is visited, the counter is incremented with the number of bundles of the visited basic block.

- When a branch is executed (Fig. 6), the buffer has to flush its FIFOs because it has already filled them with sequentially fetched instructions that may not be valid anymore. In cycle 3, bundle1 (the bundle that comprises the branch instruction) is popped into the pipeline to be decoded and dispatched to the different execution units. When a branch is executed, the IB flushes its FIFOs and sends a request to the icache in order to retrieve four instructions starting from the address of the branch target. During the fourth cycle, the core attempts to pop a bundle from the buffer but fails as the buffer is still empty. So, the pipeline is starved for one cycle. In the fifth cycle, the core can finally retrieve a bundle since it is available in the IB.

As a consequence, a bundle that is a target of a branch (a branch can be a function call) takes 1 extra cycle to be brought to the pipeline, under the same conditions as the nominal case. Accordingly, the starting bundle of each basic block, except the entry basic block, is a target of a branch (by definition of a basic block) and as a result takes 1 extra cycle: $(3)\ delay_{branch} = nbr\_basic\_blocks - 1$

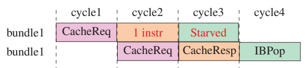


Figure 7. A line-crossing bundle case

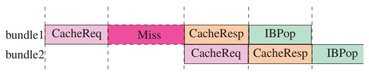


Figure 8. A miss case (a blocking cache)

- In case of a line-crossing bundle (Fig. 7), the cache responds with less instructions than requested by the IB because the bundle crosses the cache line boundary. Thus, more than one cache access is needed. In the example of Fig. 7, only one instruction of bundle1 is pushed into the buffer in the second cycle. Consequently, the pipeline is starved during the third cycle since it wants a bundle of four instructions and only one instruction is available in the buffer. The three remaining instructions are requested by the buffer in the second cycle, pushed in its FIFOs in the third cycle and popped into the pipeline in the fourth cycle. So, each sub-bundle introduces an extra cycle:

$$(4)\ delay_{cross} = \sum_{i=1}^{nbr\_bundles} nbr\_sub\_bundles_i - 1$$

For each bundle, the number of added cycles is the number of its sub-bundles minus one. We subtract 1 for each visited bundle because its first sub-bundle is already included in (2). Assuming that a cache line can hold more than four instructions, a bundle can span two cache lines at most. So, it can be divided into two sub-bundles, each one resides in a different cache line. Thus, the added delay of a line-crossing bundle is 1 cycle (in case both sub-bundles hit in the cache).

- In case a cache miss occurs (Fig. 8) (in a blocking cache), the cache cannot respond to the ensuing requests issued by the IB until the missing bundle is brought to the cache. So, subsequent requests are delayed by the miss penalty of the missing bundle. $(5)\ delay_{miss} = miss\_cycles * (nbr\_misses)$
The number of misses is computed during simulation using the cache model and the cache annotation approach explained hereinbefore. The number of misses is recorded in a counter and updated whenever a basic block is visited.

The overall delay (in cycles) caused by the icache and the IB is the sum of (2), (3), (4) and (5):

$$(6)\ delay = delay_{nom} + delay_{branch} + delay_{cross} + delay_{miss}$$

4. Experimental results

We evaluated our instruction cache model by simulating 8 programs from Polybench [9] using an in-house developed native HW/SW co-simulation platform that offers the possibility to communicate with transaction-level hardware models described in SystemC. This native simulation platform is executed on an Intel x86-64 host processor that runs at 3.476Hz.

As for the target architecture, it is the Kalray k1 core [3] which is a 5-issue 32-bit VLIW architecture that has an 8K-byte 2-way set associative icache and an instruction buffer (called PFB) composed of four 3-stage FIFOs. Each stage of the PFB can hold up to four 32-bit syllables.

To validate our icache model, which is a replica of the real cache, we used an ISS platform provided by Kalray in its design kit as a reference. We also used the ISS to collect the information necessary to build the basic block data base.

TABLE 1. COMPARISON OF ICACHE PERFORMANCE

	ISS			ILS+icacheVLIW			
	accesses	misses	miss_rate	accesses	misses	miss_rate	miss_error
gemm	103835	32	0.0003	104950	30	0.0003	-6.25%
reg_detect	3687	16	0.0043	3706	16	0.0043	0.00%
BubbleSort	3252	5	0.0015	3205	5	0.0016	0.00%
atax	15059	89	0.0059	14897	88	0.0059	-1.12%
3mm	252328	47	0.0002	261634	44	0.0002	-6.38%
jacobi_2d	24245	14	0.0006	24703	14	0.0006	0.00%
trisolv	11280	12	0.0011	11426	13	0.0011	8.3%
syr2k	158212	14	0.0001	156864	14	0.0001	0.00%

Table 1 shows the miss count and the cache access count recorded for different programs executed on the ISS platform and natively executed with the ILS approach that we enhanced with an icache model. The model takes into account VLIW and instruction buffer effects. The average miss error has an absolute value of 2.76%.

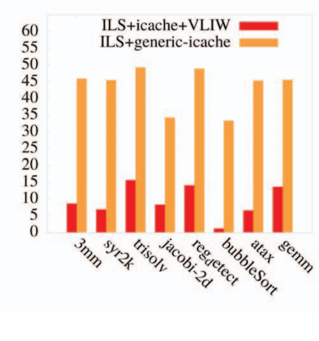


Figure 9. Cycle count error (%)

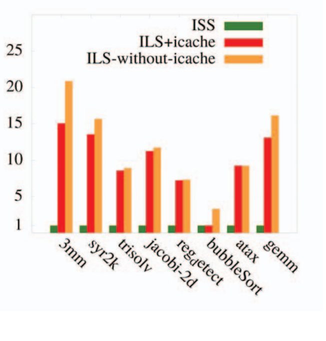


Figure 10. Simulation speedup

We also used our cache simulation approach to compute the cycle count and compared it to the ISS and the generic icache simulation approach (i.e. no consideration for the VLIW aspect, instructions are used instead of bundles as described Sec. 2). As depicted Fig. 9, the proposed icache simulation approach offers more accurate cycle count estimates than the generic approach for the different simulated programs.

Fig. 10 plots the speedup of ILS-with-icache and ILS-without-icache compared to ISS. Since simulating an icache requires the insertion of annotation functions as well as the introduction of several counters in the software code, a simulation overhead is expected compared to ILS-without-icache. The maximum slow down that we have is under 6 for application *3mm* (which is the largest application among the simulated ones in terms of icache accesses).

5. Conclusion

In this paper we presented an approach to estimate the performance of an instruction cache in a VLIW architecture with an instruction buffer. Our approach achieves an average simulation speedup of 10, a precise icache miss count (average of 2.76%) and an average of cycle count error of 9.3%.

References

[1] J. A.Fisher, P. Faraboschi, and C. Young, *Embedded Computing, a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[2] A. Bouchhima, P. Gerin, and F. Pétrot, “Automatic instrumentation of embedded software for high level hardware/software co-simulation,” in *ASP-DAC ’09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, 2009, pp. 546–551.

[3] B. Dupont de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. Guironnet de Massas, F. Jacquet, S. Jones, N. Morey Chaisemartin, F. Riss, and T. Strudel, “A clustered manycore processor architecture for embedded and accelerated applications,” in *IEEE High Performance Extreme Computing Conference*. IEEE, 2013, pp. 1–6.

[4] A. Gerstlauer, S. Charkravarty, and Z. Zhao, “Automated, retargetable back-annotation for host compiled performance and power modeling,” in *International Conference on Hardware/Software Codesign and System Synthesis*, Sep. 2013, pp. 1–10.

[5] Intel, *Intel Itanium Processor 9300 Series Reference Manual for Software Development and Optimization*. Intel, March 2010.

[6] R. A. Lethin, “How vliw almost disappeared - and then proliferated,” *IEEE Solid-State Circuits Magazine*, vol. 1, no. 3, pp. 15–23, summer 2009.

[7] K. Lu, D. Muller-Gritschneder, U. Schlichtmann, and O. Bringmann, “Fast cache simulation for host-compiled simulation of embedded software,” *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 637–642, March 2013.

[8] O. Matoussi and F. Pétrot, “Loop aware ir-level annotation framework for performance estimation in native simulation,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016.

[9] L. Pouchet, “Polybench benchmark,” *http://web.cse.ohio-state.edu/ pouchet/software/polybench/*.

[10] J. Schnerr and O. Bringmann, “High-performance timing simulation of embedded software,” *Design Automation Conference (DAC)*, June 2008.

[11] STMicroelectronics, *ST200 VLIW Series ST231 Core and Instruction Set Architecture Manual*. STMicroelectronics, March 2004.

[12] Z. Wang and J. Henke, “Fast and accurate cache modeling in source-level simulation of embedded software,” *Design, Automation and test in Europe Conference and Exhibition (DATE)*, March 2013.

[13] R. Yan, D. Ma, K. Huang, X. Zhang, and S. Xiu, “Annotation and analysis combined cache modeling for native simulation,” *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 406–411, January 2014.