High Speed Performance Estimation of Embedded Hard-core Processors in FPGA-based SoCs

Deshya Wijesundera†, Achintha Ihalage*, Alok Prakash†, Thambipillai Srikanthan†
†Nanyang Technological University, Singapore
*University of Moratuwa, Sri Lanka
deshyase001@e.ntu.edu.sg, 130212f@mrt.ac.lk, {alok, astsrikan}@ntu.edu.sg

ABSTRACT

The embedded hard-core processors beside the traditional FPGA fabric in FPGA-based System-on-Chip (SoC) devices make them an attractive alternative for realizing the software portions of the application while using the FPGA fabric for hardware acceleration. However, several hard-core processor options are becoming available from different manufacturers or even from a single vendor. This necessitates methodologies for rapid and reliable performance estimation of such embedded processors so as to enable rapid selection of a processor given an application at an early design stage. Architectural features such as superscalar, multi issue and out-of-order processing, however, make it challenging to accurately estimate their performance for a given application. In this paper, we propose a high speed performance estimation framework for such processors. Experimental results on the *ARM Cortex-A9* processor in a *Xilinx Zynq SoC FPGA* executing applications from the widely used CHStone benchmark suite show an average error of less than **6%**, completed in just over a minute.

Keywords

HW-SW codesign; HW-SW partitioning; performance estimation

1. INTRODUCTION

Hardware-software partitioning is an important step in the hardware-software codesign process, allowing designers to exploit the benefits offered by both worlds. The flexibility and short development time of software is effectively combined with performance and low power/low energy consumption of hardware [21]. Modern FPGA-based System-on-Chip (SoC), such as the ones manufactured by Xilinx [26] and Altera [11], provide designers the option of partitioning the design into many computing platforms including both hard-core and soft-core processors. Figure 1 shows the high level layout of a typical FPGA-based SoC, which integrates a traditional hard-core processor alongside the FPGA logic resources. At the same time, a soft-core processor can also be implemented using the FPGA logic resources such as look-up-tables (LUT) and Block RAMs (BRAMs). However, the characteristics of the underlying architecture enables hard-core processors to dominate soft-cores in terms of performance [23] [20], which makes them an obvious choice to implement the software portions in designs where achieving high performance is paramount. However, with a variety of FPGAs in the market there are several choices of hard-core processors used in FPGA-based SoCs, varying across vendors [22] [26] [11] or even from a single vendor that provides different hard-cores on different FPGA platforms [15] [25].

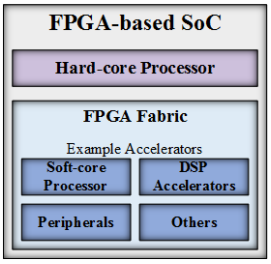

Figure 1: Layout of a Modern FPGA-based SoC

Hence, the task of selecting the best processor that provides optimal performance under user constraints for an application is a daunting task for the designer as executing the application on the multitude of processors is practically infeasible. Alternatively, a designer could rely on simulators to estimate performance. However, simulators are typically slow, complex and require specialized knowledge [16].

Estimating the application performance on an embedded hard-core processor is itself challenging due to the sophisticated branch prediction schemes, pipeline hazards, etc.. The advent of superscalar architectures and out-of-order processing further exacerbates this problem [7]. Existing work in this area has mostly been focused on single-issue, in-order processors [4] [24] or need significant effort for every new target processor [4]. Hence, in this paper, we present a *methodology for rapid performance estimation of applications on dual-issue, out-of-order, superscalar embedded hard-core processors* that relies solely on estimation based techniques. The proposed methodology uses a one-time, offline processor characterization step to allow rapid and accurate performance prediction of any application and can be easily re-targeted for new processors. This can also be used along with existing estimation tools for FPGAs, e.g. [28], to enable rapid design space exploration on FPGA-based SoCs at an early design stage during hardware-software codesign step.

Next, section 2 discusses the existing literature followed by the methodology in section 3. Section 4 discusses the results with our conclusions in section 5.

2. RELATED WORK

Performance estimation could be carried out using ana-

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2017) Bochum, DE, June 7-9, 2017.

lytical models or by simulation. Analytical models can be linear or non-linear. Simulation based models are classified as instruction set simulation based (ISSB), virtual platform based (VPB) and native execution based (NEB) [18].

Giusto et al.[9] and Lattuada et al.[14] have proposed analytical estimation models based on regression techniques and feature extraction. However, these methods are accurate only when considerably large benchmarks are used to train the models or when the characteristics of the application closely resemble the training set. ISSB methods could provide either cycle accurate or instruction accurate performance but have slow simulation speeds and do not allow the system designer to modify the behavior of the tool. VPB methods are based on either hardware VPs or software VPs. The hardware VP modelling process is complex, time consuming and requires specialized knowledge. Software VPs are not cycle accurate and thus, it is hard to deploy for performance estimation. NEB methods can be heavily dependent on the target processor architecture [16].

Obeidat [17] proposed a framework for embedded processor performance estimation, however, the method relies on hardware profilers. Aung et al. [4] relied on the complex binary level control flow graph (CFG) to extract information and back annotate the details to estimate performance using intermediate representation (IR) level CFG for a PowerPC 405 processor. The authors stated that the availability of LLVM backend for the target processor is critical for their approach. Hence, unlike the method proposed in this paper, their technique is not readily transferable to new processor architectures. Wijesundera et al. [24] proposed an estimation methodology for processor performance estimation based on the LLVM IR. However, their methodology only considers single-issue in-order processing and is limited to soft-core processors. Eyerman et al. [8] proposed a method to model various characteristics of processors to estimate its performance. But, their approach requires the user to possess a deep understanding of the micro-architectural details that might not be feasible for commercially available processors. Ivoševic et al. [12] proposed a model for function-level performance estimation for heterogeneous MPSoC platforms. However, they only showed results for a few functions, unlike the work proposed in this paper that estimates the performance of the entire application instead of just a few functions in the application.

From the discussion above, it can be observed that existing methodologies for performance estimation are focused on single-issue, in-order processors [24] or are limited to specific processor architectures [4]. Methodologies for estimation of out-of-order processors require high levels of architectural knowledge [8]. In this paper, we propose a performance estimation technique for dual-issue, out-of-order, superscalar processors that are typically found in modern FPGA-based SoCs [11] [26] and even in other current embedded systems. The proposed work also performs estimation at LLVM IR level instead of the binary level, which lends well to easier re-targeting to new processors, unlike the existing work [4].

3. METHODOLOGY

In this section, we discuss the proposed methodology, shown in Figure 2, to estimate application performance on dual-issue out-of-order superscalar processors. The technique is based on the application's IR obtained from the LLVM compiler and can be extended to other similar processor architectures. The methodology is implemented using *Shell* scripting and *C* programming languages.

We estimate the clock cycle count of each instruction, hereafter referred to as execution count, considering data hazards, control hazards etc. and accumulate them to obtain the full application execution time. The method constitutes of 4 phases, Processor Characterization, Application Profiling, Instruction Analysis and Performance Estimation.

3.1 Processor Characterization

This is a *onetime phase for each processor*. In this phase, we map the instruction set architecture (ISA) of the target processor to the LLVM ISA used in the IR. We have classified these LLVM instructions into 4 categories for clarity:

- Data Processing Instructions: Instructions corresponding to ALU operations
- Data Transfer Instructions: Instructions corresponding to loading/storing data between memory and registers
- Flow Control Instructions: Instructions which determine the value of the program counter
- Remaining Instructions: Instructions which could not be mapped to any category mentioned above

However, the process of mapping instructions under each category is non-trivial. We identify the following types of mapping between processor ISA and LLVM ISA:

- One-to-one mapped instructions: One processor instruction maps to one instruction in LLVM ISA
- One-to-many mapped instructions: One processor instruction maps to many instructions in LLVM ISA
- Many-to-many mapped instructions: Many processor instructions map to many instructions in LLVM ISA
- Other instructions: Type of mapping which does not belong to the categories above. For example, in the case of the ARM Cortex-A9 any number of subsequent *alloca* instructions in the LLVM ISA maps to a single sequence of *push, move, sub, bic* in the processor ISA

An example instruction mapping under the classified categories for the case of the *ARM Cortex-A9* processor [3] is shown in Table 1, where M is a positive integer. In order to obtain the execution count of each instruction we either use datasheets or run micro-benchmarks. We use the LLVM reference manual to create micro-benchmarks [1]. Figure 3 shows a sample output ISA mapping file.

3.2 Application Profiling

In this phase, the application is first profiled using the LLVM compiler to obtain an IR of the application. The profiling information is used to extract basic blocks, instructions in each basic block and execution frequency of basic blocks in the application. This information is used during scheduling in phase 4.

3.3 Instruction Analysis

We use the IR and other profile information from section 3.2 to identify the pipeline hazards between instructions and annotate each instruction with this information. Also, we identify multi-cycle instructions as this is a special case of processor pipeline behavior. Further, we annotate each instruction with the execution count obtained in section 3.1.

As mentioned in section 1, modern processors incorporate sophisticated execution mechanisms and advanced micro-architectural features, which makes the process of estimating

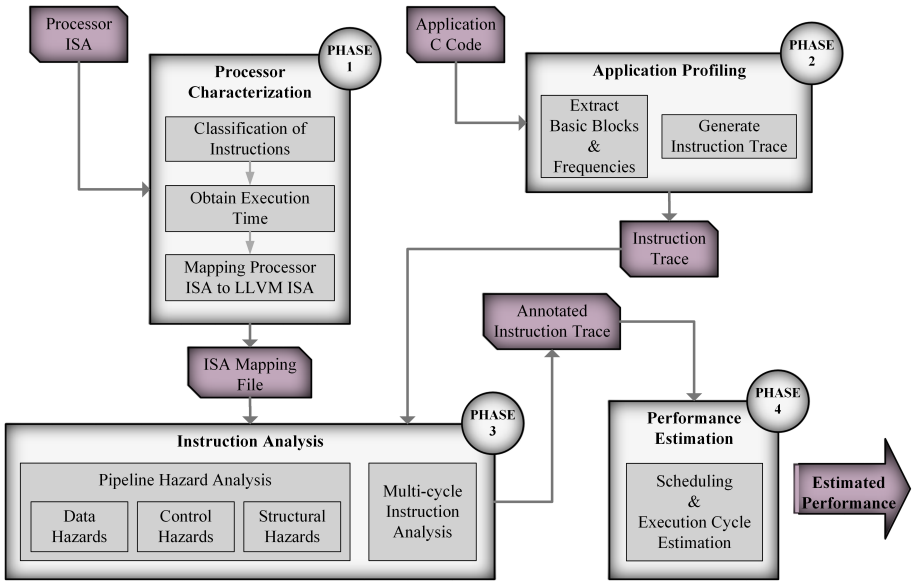

Figure 2: Framework for Out-of-Order Processor Performance Estimation

Table 1: Classification of LLVM Instructions and Example Mapping for *ARM Cortex-A9* Processor

Instruction Category	Processor Instruction	LLVM Instruction
Data Processing Instructions	add	add
	mls	[mul,sub]
	[bfc,movw,movt]	getelementptr
	cmp	icmp
	[cmp,movne]	icmp
	[bic,pop]	[srem,ret]
	[smmul,mov]	[sext,mul]
	[push,mov,sub,bic]	alloca x M
	...	...
Data Transfer Instructions	ldr	load
	ldrb	load
	str	store
	stm	store
	...	...
Flow Control Instructions	b	br
	bl	call
	[b,blt]	br
	[beq,b]	br
	...	...
Remaining Instructions	vadd	fadd
	vcvt	sitofp
	vmla	[fmul,fadd]
	vfma	[fmul,fadd]
	...	...

performance of an application on a given processor significantly challenging. Thus, analysing the inter-dependencies between the instructions is of utmost importance to improve the accuracy of the estimation process. Therefore, the proposed methodology incorporates the following steps in order to analyze the instruction profile.

- Pipeline Hazard Analysis
- Multi-cycle Instruction Analysis

3.3.1 *Pipeline Hazard Analysis*

ISA Mapping File: ARM Cortex-A9

********** Data Processing Instructions **********

[Instruction]	[Type]	[Clock Cycle Count]
add	i32	1
add	i64	1
mul	i32	3
mul	i64	4
icmp	n/a	1
alloca	i32	1
alloca	i64	2
sdiv	i32	4
------	----	--

********** Data Transfer Instructions **********

[Instruction]	[Type]	[Clock Cycle Count]
load	i32	1
load	i64	2
store	i32	1
store	i64	2
------	----	--

********** Flow Control Instructions **********

[Instruction]	[Type]	[Clock Cycle Count]
br	n/a	1
br (mis-predicted)	n/a	11
call	n/a	1
-------	----	--

********** Remaining Instructions **********

sitofp	float	4
fptosi	float	4
-------	----	--

Figure 3: Sample ISA Mapping File

A modern processor typically integrates deep pipelines within the processor execution path in order to achieve instruction-level parallelism. However, due to the interaction of instructions with each other leading to data, control and structural hazards, the execution count of a given application varies significantly in each scenario. These hazards affecting the processor pipeline could result in flushing or stalling of the processor pipeline. With deep pipelines (for an example the ARM Cortex-A9 processor pipeline has 8-11 stages) the number of clock cycles required to re-fill the pipeline would incur a significant penalty adding to the execution count of the application [3]. Thus, identifying the aforementioned hazards from the IR level instruction profile

is a critical step in the estimation process. We propose analytical techniques in order to identify pipeline hazards in the processor. Subsequent sections explain these techniques.

- Data Hazards

A data hazard is created whenever there is a dependency between instructions, and the overlap caused by pipelining of spatially localized instructions would change the order of access to an operand [19]. The proposed methodology analyses data hazards using the dependency analysis technique.

Dependency analysis is the process of analysing the instruction profile to identify instructions which are dependent on the data produced by the previous instructions or subsequent instructions which have the same destination address [19] [3]. A single-issue processor would only execute a single instruction at any given time. Also, an in-order processor would only consider dependencies with the subsequent instruction, and hence does not require extensive dependency analysis. However, an out-of-order processor requires an extensive dependency analysis within the instruction window. This is further exacerbated by the dual-issue feature compared to a single-issue processor. Hence, since the proposed model is focused on dual-issue, out-of-order processors, we analyze the dependencies among instructions in order to identify suitable candidate instructions that could be processed in parallel and also to identify their order of execution. We annotate instructions with identified dependencies and use this information later in the scheduling phase.

For example, load-use dependency is a common data dependency occurring in processors. Load-use dependencies occur when the data to be loaded is used in subsequent instructions prior to the correct data being written to the registers. The proposed model analyses such dependencies and compensates the execution count in the scheduling phase.

- Control Hazards

Control hazards occur due to the instructions that alter the control flow of a program by modifying the program counter. Therefore, we analyze instructions such as conditional branches and calculate the additional latency incurred due to branch mis-predictions.

Branches could be either conditional or unconditional. Unconditional branches do not stall or flush the pipeline. In the case of conditional branches, the execution count depends on whether or not the branch was correctly predicted. Correctly predicted branches behave similar to unconditional branches. However, mis-predicted branches cause pipeline flushing. Thus, it is important to identify branch mis-predictions. In this work, we use the branch prediction accuracy for the relevant processor to compensate for branch mis-prediction. This value can be obtained from the datasheet of the relevant processor [13]. We make an approximation by multiplying the frequency of the basic block by the accuracy of prediction to obtain the number of times an instruction is predicted correctly and the remaining number is considered as the number of times it is mis-predicted.

- Structural Hazards

Structural hazards occur due to the limited resources in the processor. For example, due to instruction-level parallelism there could be multiple instructions requesting the service of a common resource. In such a scenario, the processor pipeline will be stalled while the hazard is resolved.

The proposed method analyses load/store instructions to identify structural hazards. Load/store analysis identifies load and store instructions in the instruction profile as some processors could have limitations in dual issuing load/store instructions [3]. The proposed method analyses the load/store instructions and annotates the information into the instruction trace. The scheduler in the subsequent phase analyses the annotated information in order to compensate, whenever necessary, during the scheduling process.

3.3.2 *Multi-cycle Instruction Analysis*

Multi-cycle instructions may or may not be handled in parallel, depending on the processor architecture [3]. Thus, our models identify these instructions, and annotate them for identification by the scheduler.

3.4 Performance Estimation

In this phase, we use the annotated instruction trace from section 3.3 to model the schedule of instructions. Firstly, the model checks if the pipeline is free and if so the instructions are scheduled. The instruction schedule is later used to estimate the performance of the application.

3.4.1 *Scheduling and Execution Cycle Estimation*

Scheduling is performed on each basic block obtained from the application profiling phase. We use Algorithm 1 for scheduling the application. Here, PQ stands for priority queue, N is the size of the instruction window and t_i is the execution count of the instruction under consideration.

Typically, out-of-order processing is performed within a specified instruction window which is dependent on the processor [6]. Our implementation utilizes a configurable instruction window size. During dual issue, if the subsequent instruction is not feasible for scheduling, the scheduler checks the next instruction and continues this process within the specified instruction window until an independent schedulable instruction is identified. The proposed model implements a priority queue to store the dependent instructions as obtained from the previous phase. This process is repeated until the end of the basic block.

The schedule provides the execution count of each basic block on the two pipelines. However, we select the higher execution count as the execution count of the basic block, since execution requires a minimal time equivalent to the pipeline which consumes the larger amount of time.

$$N_{bb} = \begin{cases} t_p * f_{bb}, & br = 0 \\ t_p * f_{bb} * P_p + t_{mp} * f_{bb} * P_{mp}, & br = 1 \end{cases} \quad (1)$$

where,

f_{bb}	=	Execution frequency of the basic block,
P_p	=	Branch prediction probability,
P_{mp}	=	Branch mis-prediction probability ($P_{mp} = 1 - P_p$),
t_p	=	Execution count of the basic block,
t_{mp}	=	Execution count of the mis-predicted basic block ($t_{mp} = t_p$ + *branch mis-prediction penalty*),
N_{bb}	=	Total execution count of the basic block

We use the method proposed in section 3.3.1 to estimate the execution count of a basic block incorporating branch prediction. According to the definition of a basic block [1], a basic block could contain only one terminator instruction such as a branch instruction. Thus, we use Equation 1 to

Algorithm 1 Scheduling Algorithm

Input: Annotated Instruction Trace
Output: Instruction Schedule of Basic Blocks

```
1: loop:
2: if initial or last issue is a dual issue then
3:     if PQ is non-empty then
4:         issue 1st instruction of PQ;
5:         stall other pipeline for (t_i-1);
6:         issued pipeline = busy;
7:         other pipeline = non-busy;
8:         dual-issue ← 0;
9:         remove issued instruction from PQ;
10:        endif
11:    else
12:        issue next instruction from instruction window;
13:        stall other pipeline for (t_i-1);
14:        issued pipeline = busy;
15:        other pipeline = non-busy;
16:        dual-issue ← 0;
17:    endif
18:    if issued instruction = load or store then
19:        load/store ← 1
20:    endif
21: else if last issue is a single issue then
22:    if PQ is non-empty then
23:        while read PQ do
24:            check for dependencies;
25:            check previous issue for a load/store;
26:            check previous issue for multi-cycle;
27:            if current instruction is dual issuable then
28:                issue instruction;
29:                stall other pipeline for (t_i-1);
30:                issued pipeline = busy;
31:                other pipeline = non-busy;
32:                remove issued instruction from PQ;
33:                dual-issue ← 1;
34:            endif
35:        done
36:    endif
37:    if no instruction from PQ is issued then
38:        for i in next N instructions do
39:            check for dependencies;
40:            check previous issue for a load/store;
41:            check previous issue for a multi-cycle;
42:            if current instruction is dual issuable then
43:                issue instruction;
44:                stall other pipeline for (t_i-1);
45:                issued pipeline = busy;
46:                other pipeline = non-busy;
47:                remove issued instruction from PQ;
48:                dual-issue ← 1;
49:                break
50:            else
51:                add current instruction to PQ;
52:            endif
53:        done
54:        if not issued from next N instructions then
55:            stall non-busy pipeline;
56:        endif
57:    endif
58: endif
59: goto loop.
```

compute the total execution count of a basic block depending on the presence of a branch instruction. Here, the value of br represents the presence of a branch instruction.

Next, we accumulate the total execution count of all the basic blocks to obtain the execution count of the full application using Equation 2. Equation 3 is used to obtain the estimated execution time of the application in seconds.

$$N_{cc} = \sum_{i=1}^{n} N_{bb_i} \tag{2}$$

$$T_{ex} = N_{cc} * \frac{1}{f} \tag{3}$$

where,

f	=	Frequency of the processor,
N_{bb_i}	=	Total execution count of basic block i,
N_{cc}	=	Total execution count of the application,
T_{ex}	=	Estimated execution time of the application

4. RESULTS

Next, we discuss the results obtained by the proposed methodology. We use the AVNET ZedBoard development kit [2] with Xilinx Zynq-7000 All Programmable SoC platform and the Xilinx Vivado 2015.3 development environment for the experiments. This FPGA-based SoC contains an ARM Cortex-A9 hard-core processor, which is a dual-core, dual-issue, out-of-order, superscalar processor running at $667MHz$ [2]. Level 1 instruction and data caches are $32kB$ in size and the Level 2 cache is $1MB$ [5]. The instruction window is set to hold 50 instructions [6]. We have assumed a branch prediction accuracy of 95% similar to [13]. Similarly, branch mis-prediction penalty is set to 11 clock cycles. We run applications from the CHStone Benchmark suite in bare-metal mode on the processor [10]. The *gprof* profiler of Vivado SDK is used to measure the runtimes of applications [27]. The proposed methodology was run on a 6-core virtual machine with 8GB RAM, running OpenSUSE 13.2 on an Intel Xeon E5-1650V2 CPU host at 3.5 GHz.

Figure 4 shows a comparison between the actual and estimated results. Both axes of the graph are represented in log_2 scale for clarity. Here, blue circles represent actual results depicted in the horizontal axis while the orange triangles represent estimated results depicted in the vertical axis. The distance of the data points from the $(y = x)$ line shown in green indicates the accuracy of results. A lower deviation of the data points from the line depicts higher accuracy.

The proposed technique has an average estimation error of **5.84%** averaged across all 9 applications from the CHStone suite. The estimated results for the *MOTION* application initially exhibited a relatively higher error. Upon closer inspection of the actual results by accessing the processor performance counters, we identified that it was due to the branch prediction accuracy for the application being lower than the value stipulated in the data sheets and not due to a short coming of our approach. Thus, for the *MOTION* application we adjusted the branch prediction accuracy to reflect the actual numbers. In future, we will automate this manual adjustment process.

The runtime of the proposed technique averaged across all these applications is **1** minute and **8** seconds. Table 2 shows the individual estimation errors and run times for all applications. It can be observed that unlike [4], we do not need to obtain the binary level information for each application,

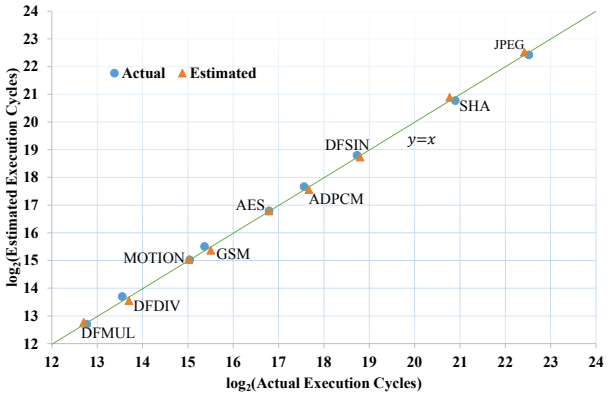

Figure 4: Comparison of Results

Table 2: Estimation Error and Runtime

Application	ADPCM	AES	DFDIV	DFMUL	DFSIN	GSM	JPEG	MOTION	SHA	**AVG.**
Estimation Error (%)	7.03	0.10	9.84	4.90	4.57	9.04	7.18	1.14	8.76	**5.84**
Runtime (min:sec)	1:20	2:09	0:42	0:31	1:25	1:11	1:46	0:39	0:33	**1:08**

that necessitates a LLVM backend for the target processor. Instead, the proposed method relies only on the target-independent LLVM IR information and yet obtains rapid and accurate performance estimation for more complex and widely prevalent dual-issue, out-of-order processor.

5. CONCLUSION

This paper proposes a rapid technique for performance estimation of dual-issue, out-of-order superscalar processors, typically integrated in FPGA-based SoCs. The approach can also be adapted to other out-of-order embedded processor architectures. The proposed technique has been shown to be accurate with an average estimation error of only 5.84% for 9 test applications. The runtime of the approach is in the order of minutes. In future, we will consider estimating power consumption in order to generate not only performance efficient designs but also power and energy efficient systems.

6. REFERENCES

[1] LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`.

[2] Zed Board Hardware User's Guide. `http://zedboard.org/sites/default/files/documentations/ZedBoard\_HW\_UG\_v2\_2.pdf`.

[3] ARM Ltd. Cortex-A9 Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F\_cortex\_a9\_r2p2\_trm.pdf`.

[4] Y. L. Aung et al. Compiler-assisted Technique for Rapid Performance Estimation of FPGA-based Processors. In *SOCC*, 2011.

[5] C. Celio. Kayla (1.4 GHz ARM Cortex A9). https://github.com/ucb-bar/ccbench/wiki/Kayla-(1.4-GHz-ARM-Cortex-A9).

[6] Y. Etsion et al. Computer Architecture - Out-of-order Execution. `https://webcourse.cs.technion.ac.il/234267/Spring2014/ho/WCFiles/5-ca-oooe.pdf`, 2014.

[7] S. Eyerman et al. Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors. In *DATE*, volume 1, 2006.

[8] S. Eyerman et al. A Mechanistic Performance Model for Superscalar Out-of-order Processors. In *TOCS*, 2009.

[9] P. Giusto et al. Reliable Estimation of Execution Time of Embedded Software. In *DATE*, 2001.

[10] Y. Hara et al. CHStone: A Benchmark Program Suite for Practical C-based High-level Synthesis. In *ISCAS*, 2008.

[11] Intel Inc. SoCs - SoCs - Overview. `https://www.altera.com/products/soc/overview.html`.

[12] D. Ivoevic et al. Function-level Performance Estimation for Heterogeneous MPSoC Platforms. In *ZINC*, 2016.

[13] J. A. Langbridge. *Professional Embedded ARM Development*. Wiley, 2013.

[14] M. Lattuada et al. Performance Modeling of Embedded Applications with Zero Architectural Knowledge. In *CODES+ISSS*, 2010.

[15] J. Lazzaro. Xilinx Parts Family History. `http://www-inst.eecs.berkeley.edu/\textasciitidlecs294-59/fa10/resources/Xilinx-history/Xilinx-history.html`.

[16] T. Nakada et al. Design and Implementation of a Workload Specific Simulator. In *ANSS*, 2006.

[17] F. Obeidat. *Embedded Processor Selection /Performance Estimation using FPGA-based Profiling*. PhD thesis, Virginia Commonwealth University, 2010.

[18] R. Patel et al. Recent Trends in Embedded System Software Performance Estimation. *Des. Autom. Emb. Syst.*, 2013.

[19] G. Prabhu. Data Hazard Classification. `https://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/dataHazClass.html`.

[20] A. K. B. Salem et al. Hard and Soft-core Implementation of Embedded Control Application Using RTOS. In *ISIE*, 2008.

[21] M. U. Sharif et al. Hardware-software Codesign of RSA for Optimal Performance vs. Flexibility Trade-off. In *FPL*, 2016.

[22] SourceTech411. Top FPGA Companies For 2013 - Sourcetech411. `http://sourcetech411.com/2013/04/top-fpga-companies-for-2013/`.

[23] D. Wijesundera et al. Exploiting Configuration Dependencies for Rapid Area-efficient Customization of Soft-core Processors. In *SCOPES*, 2016.

[24] D. Wijesundera et al. Rapid Design Space Exploration for Soft Core Processor Customization and Selection. In *FPT*, 2016.

[25] Xilinx Inc. Embedded Processors. `https://www.xilinx.com/products/intellectual-property/embedded/nav-embedded-processors.html`.

[26] Xilinx Inc. Zynq-7000 All Programmable SoC. `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`.

[27] Xilinx Inc. EDK Profiling User Guide. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/edk_prof.pdf`, 2009.

[28] G. Zhong et al. Design Space Exploration of FPGA-based Accelerators with Multi-level Parallelism. In *DATE*, 2017.