Programming Language Assisted Waveform Analysis: A Case Study for RISC-V Instruction Performance

Lucas Klemmer Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
{lucas.klemmer, daniel.grosse}@jku.at

***Abstract*—RISC-V's growing traction leads to the release of new *RISC-V* cores on a near monthly basis. In this growing and diverse ecosystem, understanding the performance and other properties of a RISC-V core is of great importance since selecting the best fitting core is mandatory for a successful project. Analyzing RISC-V cores by hand is not possible due to the ever-increasing number of available cores and available software benchmarks might not be fine-grained enough to understand a core completely. Programming and powerful programming languages have proven to provide the productivity that is required to keep pace with these fast developments.**

In this paper we present a case study in which we use the Domain Specific Language WAWK to analyze the performance of all instructions of SERV, a well known bit-serial RISC-V core. With WAWK, only a few lines of code are necessary to calculate the respective metric on the waveform generated during simulation.

I. Introduction

RISC-V is an open and royalty free ISA [1] striving for innovation through collaboration, thus enabling even small companies as well as community projects to develop their own processors which take advantage from RISC-V's permissive license and its extensibility to explore new ideas and markets with often highly specialized hardware. However, this openness and extensibility of RISC-V brings its own set of challenges, since the sheer number of available RISC-V cores, which are often highly configurable and extensible, makes it very hard and time-consuming for both, designers and users, to compare different cores against each other [2] [3].

In this paper, we use the open-source *Domain Specific Language* (DSL) WAWK [4] to analyze the the number of cycles that is required to execute one RISC-V instruction on the well-known SERV core [5]. WAWK programs have direct access to all signal values of a waveform. Accessing signals in a WAWK program is similar to accessing variables in regular programming languages with the difference that the value returned depends on the loaded waveform and the time at which the signal is accessed. Since WAWK is compiled down to the *Waveform Analysis language* (WAL) [4] it has access to a very rich feature set that provides a large collection of functions which can be used to analyze waveforms. Thus WAWK allows creating analysis programs using the values from the VCD waveforms generated during simulation of a RISC-V core.

With WAWK, we show that the analysis of RISC-V cores is possible with only a handful of lines which give developers crucial information about the performance of their or others cores.

II. Related Work

In the context of processor architecture research several processor simulators have been proposed. A prominent example is gem5 [6] or multi2sim [7]. A complimentary direction are emulators, such as qemu [8] or OVPSim [9]. Both simulators, and emulators can partially be used to calculate (performance) metrics. However, they are not as flexible as a programmable approach such as WAWK for the problems considered. WAWK allows developers to create programs and tailor them to the exact requirements, such as IPC count, pipeline analysis and more.

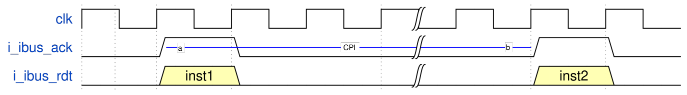

Fig. 1: SERV Instuction Lifecycle

III. Preliminaries

A. WAWK

In this paper we use WAWK, a language inspired from the popular text-processing language *AWK*, that is compiled down to WAL. In general, all WAWK scripts consist of multiple statements that follow a `condition: { action }` scheme. For each time index in a waveform, WAWK evaluates the `condition` of each statement, and, if satisfied, executes the associated `action`.

B. SERV

SERV is a bit-serial implementation of the RISC-V ISA with a heavy focus on very low area usage. The bit-serial implementation dictates that all instructions are executed bit by bit such that for example the RV32I *add* instruction is split over at least 32 cycles, one for each bit of the result.

To analyze the runtime of individual instructions in the SERV core we can observe the values of the instruction fetch bus. The start of a new instruction is signalized by the core when the signal `i_ibus_ack` rises. Therefore, the runtime of one instruction can be calculated by observing two of the rising events on `i_ibus_ack` and calculating the difference between the occurrence of both events $event_2 - event_2$. This is shown in Figure 1 where the first instruction is started at time `a`. At time point `b` the next instruction is started thus the `CPI` for this instruction can be calculated by the difference between `b` and `a`.

IV. CPI Analysis

The WAWK program for the CPI analysis is shown in Listing 1. The analysis program consists of four WAWK statements. First, after the program is started and before the waveform analysis is started the first statement (Line 1) is executed once. This is triggered by the **`BEGIN`** condition which is a special variable which is only true after the program starts. This Python script uses the *riscvmodel* package to decode RISC-V instructions. Therefore, the external Python script containing a few lines of Python glue code is imported (Line 2) and the list of results which will store all measured CPI values is initialized. Next, some aliases are declared which can be used instead of the long full signal names.

The second statement (Line 9) is executed at the last cycle of each instruction. However, the statement is only executed when the opcode of the current instruction matches the opcode which we want to analyze. This opcode has to be passed to the program as an command line argument and it is available to the WAWK program in the `args` variable. At this point, the difference to the starting time of this instruction is calculated and added to the list of measured CPI values.

The next statement (Line 13) is executed whenever a new instruction starts executing. At this moment, the current time index, which is used to calculate the runtime of this instruction in the previous statement, is stored and the current opcode is decoded and stored in the `op` variable.

Finally, after all indices were processed by the program, the final results, which consist of the average, minimum, and maximum runtimes, are printed by the last statement (Line 18).

```
1   BEGIN: {
2     import(extern);
3     cpis = [];
4     alias(clk, TOP.servant_sim.dut.cpu.clk);
5     alias(fire, TOP.servant_sim.dut.cpu.i_ibus_ack);
6     alias(instruction,
         TOP.servant_sim.dut.cpu.i_ibus_rdt);
7   }
8
9   clk, !fire, fire@2, op == args[0]: {
10    cpis = cpis + ((INDEX - start) / 2);
11  }
12
13  clk, fire: {
14    start = INDEX;
15    op = call(extern.decode, instruction);
16  }
17
18  END: {
19    if (cpis) {
20      if (min(cpis) == max(cpis)) {
21        printf("%10s %10d\n", args[0], average(cpis));
22      } else {
23        printf("%10s %10d %10d %10d\n", args[0],
         average(cpis), min(cpis), max(cpis));
24      };
25    };
26  }
```

Listing 1: WAWK code for CPI the analysis

V. Results

Table I shows the analysis results for each of the instructions of the SERV core. The first column lists the name of the instruction, the second lists the average number of cycles this instruction requires to be executed, and the last two columns show the minimum and maximum number of instructions required to execute this instruction respectively. The table shows that the instructions mainly fall into two categories. One half of the instructions always finishes execution in constant time while the over half requires a variable number of cycles.

VI. Conclusion

In this paper we presented a programmable analysis of the performance of individual instructions as implemented in the well-known RISC-V core SERV. The analysis program is written in WAWK, a DSL for waveform analysis inspired by the AWK text-processing language. In only a few lines of code the program analyzes the number of cycles per instruction required to execute a specific instruction.

Opcode	Avg.	Min	Max
auipc	35		
lui	35		
add	35		
addi	35		
sub	35		
and	35		
andi	35		
or	35		
ori	35		
xor	35		
xori	35		
ecall	35		
slt	68		
slti	68		
sltu	68		
sltiu	68		
sll	68		
slli	68		
sra	75	68	99
srai	70	68	99
srl	75	68	99
srli	75	68	99
jal	68	68	70
jalr	69	68	70
beq	68	68	70
bge	69	68	70
bgeu	69	68	70
blt	68	68	70
bltu	69	68	70
bne	68	68	70
lb	69		
lh	69	69	70
lhu	69	69	70
lw	69	69	70
sh	69	69	70
sw	69	69	70

TABLE I: SERV CPI on Compliance Tests

References

[1] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[2] A. Dörflinger, M. Albers, B. Kleinbeck, Y. Guan, H. Michalik, R. Klink, C. Blochwitz, A. Nechi, and M. Berekovic, "A comparative survey of open-source application-class risc-v processor implementations," in *Proceedings of the 18th ACM International Conference on Computing Frontiers*, ser. CF '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 12–20.

[3] E. Sperling, "Which Processor Is Best?" https://semiengineering.com/which-processor-is-best, 2022.

[4] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP Design Automation Conf.*, 2022.

[5] "GitHub - SERV: The SErial RISC-V CPU," https://github.com/olofk/serv, 2022.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[7] R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez, "Multi2sim: A simulation framework to evaluate multicore-multithreaded processors," in *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, 2007, pp. 62–68.

[8] "QEMU a generic and open source machine emulator and virtualizer," https://www.qemu.org/, 2022.

[9] "Technology OVPsim," https://www.ovpworld.org/technology_ovpsim, 2022.