Late Breaking Results: Waveform-based Performance Analysis of RISC-V Processors

Lucas Klemmer
Institute for Complex Systems
Johannes Kepler University Linz
Linz, Austria
lucas.klemmer@jku.at

Daniel Große
Institute for Complex Systems
Johannes Kepler University Linz
Linz, Austria
daniel.grosse@jku.at

ABSTRACT

In this paper, we demonstrate the use of the open-source domain specific language WAL to analyze performance metrics of RISC-V processors. The WAL programs calculate these metrics by evaluating the processors signals while "walking" over the simulation waveform (VCD). The presented WAL programs are flexible and generic, and can be easily adapted to different RISC-V cores.

1 INTRODUCTION

Today, the processor market is dominated by few proprietary *Instruction Set Architectures* (ISAs) and only a handful of very large corporations. RISC-V is an open and royalty free ISA [10] striving for innovation through collaboration. The open nature of RISC-V enabled 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 development 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 and core configurations against each other [6, 9]. A sophisticated analysis of the cores is needed to obtain relevant performance metrics. Since a wide range of cores has to be evaluated, the analysis solution must satisfy several requirements: (1) the analysis must be powerful enough to cover complex analysis tasks, (2) it must be implementation-agnostic and easy to port to new cores, and (3) it must be easy to integrate into existing workflows.

In this paper, we use the open-source *Waveform Analysis Language* (WAL) [5] to analyze relevant metrics for several RISC-V implementations ranging from extremely area efficient ones to pipelined cores with higher performance. WAL has been realized as a *Domain Specific Language* (DSL) [8]. The language allows creating analysis programs using the values from the VCD waveforms generated during simulation of a RISC-V core. Our contributions are flexible WAL programs for different performance metrics. The programs can be adapted and used on a wide variety of RISC-V microarchitectures.

Our experimental results demonstrate that the WAL-based analysis can clearly highlight the differences between the analyzed cores. In addition, we can quantify the performance improvements of different core configurations that can be set by enabling additional features, such as instruction caches or branch prediction.

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530623

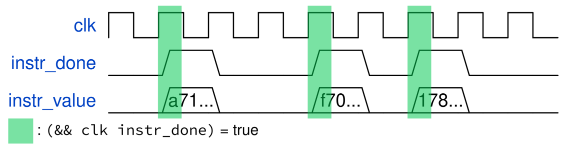

Figure 1: Waveform of instruction control and data signals.

2 PROCESSOR ANALYSIS WITH WAL

We demonstrate how flexible and generic processor analysis programs can be created in WAL. To this end, we first illustrate how WAL programs work (Section 2.1). Then, we introduce WAL programs to analyze the number of executed instructions per cycle (Section 2.2) and to calculate the percentage of cycles with stalled pipeline stages (Section 2.3). We consider four well known RISC-V cores. Two of the cores are small and area efficient [2, 3], while the other two cores are more sophisticated, pipelined, and capable of running Linux in some configurations [1, 4, 7].

2.1 WAL Program Principle

In comparison to other programming languages, WAL programs have direct access to all signal values of a waveform. Accessing signals in WAL is similar to accessing variables with the difference that the value returned depends on the loaded waveform and the time at which the signal is accessed. Consider the waveform in Figure 1. The WAL expression `(&& clk instr_done)`[1] returns true at a given time point in the waveform if and only if the *clk* and *instr_done* signals are both set to 1. In Figure 1, all time-points at which the expression evaluates to true are highlighted in green. WAL provides a large collection of functions that can be used to analyze waveforms. For example, the count function can be used to count how many instructions are executed on the waveform with the WAL expression `(count (&& clk instr_done))`.

2.2 Instructions Per Cycle

First, we analyze the raw performance of each core in terms of executed *Instructions Per Cycle* (IPC). Since all analyzed cores are single core architectures, the best theoretical IPC score is 1.0. This means that the core executes and commits one instruction in each clock cycle. However, this is almost impossible to achieve, for example, due to branching and memory induced delays.

The WAL program for IPC analysis is split into two separate parts, a generic and core-independent analysis part and the core-specific code which has to be provided by the user.

The generic WAL program to perform the IPC analysis is shown in Listing 1. The function performs the IPC analysis for all waveforms passed in the *traces* parameter. For each trace, first, the trace is loaded in Line 3 and then the optional *setup* function is called in Line 4. The optional *setup* and *clean-up* functions can be defined by the users to perform core-specific setup and clean operations. Then, the number of executed instructions is calculated in Line 5 using the

[1]WAL uses a LISP style prefix notation.

```
1  (defun calc-ipc [traces]
2    (for [trace traces]
3      (load trace t)
4      (setup)
5      (set [instructions (count (&& (is-valid) (instr-done)))])
6      (set [ipc (fdiv 1 (fdiv (count (is-valid)) instructions))])
7      (printf "%40s:␣%15.2f\n" trace ipc)
8      (clean-up)
9      (unload t)))
```

Listing 1: Generic WAL Function for IPC Analysis

```
1  (defun is-valid [] (&& TOP.IO_CLK TOP.IO_RST_N))
2  (defun instr-done []
3    TOP.ibex_simple_system.u_top.u_ibex_top.u_ibex_core.id_stage_i.instr_done)
4
5  (require riscv-lib)
6  (calc-ipc '("ibex-default.vcd" "ibex-icache.vcd"))
```

Listing 2: IBEX specific code for the IPC analysis

```
1  (defun setup [(stages stages)] (set [stages (groups "isMoving")]))
2  (defun is-valid [] (&& TOP.clk (! TOP.reset)))
3  (defun is-stalled [(stages stages)]
4    (in 0 (map (lambda [stage] (in-group stage #isMoving)) stages)))
5
6  (require riscv-lib)
7  (calc-pipeline-stall '("vexrv-smallest.vcd" "vexrv-full.vcd"))
```

Listing 3: VexRiscv specific pipeline-stall analysis code

user-supplied *is-valid* and *instr-done* functions (see below). The idea is to count how often the predicates *is-valid* and *instr-done* evaluate to 1 on the waveform and then to assign the result to the variable *instructions* via the `set` function of WAL. Next, the resulting IPC value is calculated in Line 6. We divide the number of total valid cycles by the number of executed *instructions*, take the reciprocal value, and print it in Line 7. Finally, the optional *clean-up* function is called and the trace is unloaded from the WAL environment in Line 9.

To perform the IPC analysis on a new RISC-V core, users only have to provide the two *is-valid* and *instr-done* functions. Lines 1-3 in Listing 2 show the implementations of these functions for the IBEX processor. The IBEX processor always sets the *instr_done* signal inside the *id_stage_i* module to 1 whenever an instruction is completed. Therefore, the *instr-done* function only has to return the value of this signal. After the definition of the required functions, the IPC analysis can be started. First, in Line 5 of Listing 2 we import our WAL RISC-V library to have access to the WAL definitions of the IPC analysis. Then, in Line 6 we call the *calc-ipc* function with a list of waveform files for which we want to perform the analysis (in the example the two IBEX VCDs default and icache, respectively).

2.3 Pipeline Stall Activity

In addition to the IPC analysis, the WAL RISC-V library also supports calculating the percentage of cycles with stalled pipeline stages. For example, this metric is useful to access how efficient the branch prediction is working. The pipeline stall analysis works similar to the IPC analysis in the sense that users can use a generic library function and only have to provide certain processor specific functions themselves. The users have to provide the function *is-valid* and *is-stalled*. Listing 3 shows the processor specific code required for the pipeline stall analysis on the VexRiscv processor. The *is-stalled* function should return true at each time-point where some part of the pipeline is stalled. Consider the VexRiscv pipeline: Each stage has an *isMoving* signal which can be used to determine if the stage is currently stalled. We get a list of all groups associated with the pipeline stages in the *setup* function, which is called before the main analysis. The *is-stalled* function is defined in Line 3. This function creates a list with the values of each *isMoving* signal and checks if this list contains a 0 which indicates that this pipeline stage is currently stalled. By using the `in-group` function we can get the value of all *isMoving* signals even if the actual location of the signal changes or the number of pipeline stages varies. In Line 2 the *is-valid* function, which is similar to the implementation for the IBEX core, is implemented. Finally, the RISC-V library is imported in Line 6 and the analysis is started on two waveforms in Line 7.

Table 1: Analysis Results

Core	Configuration	IPC	Stalled Cycles
SERV	servant	0.02	*not pipelined*
PicoRv32	default	0.24	*not pipelined*
VexRiscv	microNoCsr	0.33	63%
VexRiscv	smallest	0.33	66%
VexRiscv	smallAndProductive	0.42	54%
VexRiscv	smallAndProductiveICache	0.47	51%
VexRiscv	twoThreeStage	0.47	48%
VexRiscv	secure	0.57	42%
VexRiscv	linux	0.59	38%
VexRiscv	full	0.57	35%
VexRiscv	fullNoMmuMaxPerf	0.63	33%
IBEX	default	0.63	48%
IBEX	icache	0.89	19%

3 EXPERIMENTAL RESULTS

The results for the IPC and pipeline stall activity are summarized in Table 1. To get the experimental results we analyzed the waveforms produced by running the Dhrystone benchmark on each core.

The *Core* column shows the name of the analyzed RISC-V core and the *Configuration* column shows the core configurations. Columns *IPC* and *Stalled Cycles* show the number of instructions per cycle and the percentage of cycles with stalled pipeline stages, respectively. The IPC results show large differences in the performance of the evaluated cores. For example, the IBEX *icache* configuration is more than 44 times faster than the SERV core. Enabling more sophisticated features, e.g. better branch prediction, for the VexRiscv and IBEX cores clearly shows that the number of cycles in which the pipeline is stalled decreases significantly. This also correlates with the performance improvements seen in the *IPC* column.

4 CONCLUSIONS

In this paper, we have demonstrated the use of the open-source language WAL to analyze performance metrics. We have shown that the analysis programs can be written in a generic form and microarchitecture specific details can be handled via user-defined functions. In the experiments our analysis programs highlighted large performance differences on RISC-V cores with diverse microarchitectures.

ACKNOWLEDGMENTS

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

REFERENCES

[1] 2022. GitHub - IBEX. https://github.com/lowRISC/ibex.
[2] 2022. GitHub - PicoRV32. https://github.com/YosysHQ/picorv32.
[3] 2022. GitHub - SERV. https://github.com/olofk/serv.
[4] 2022. GitHub - VexRiscv. https://github.com/SpinalHDL/VexRiscv.
[5] 2022. WAL the Waveform Analysis Language. https://github.com/ics-jku/wal.
[6] Dörflinger et al. 2021. A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations. In *CF*. 12–20.
[7] Davide Schiavone et al. 2017. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In *PATMOS*. 1–8.
[8] Lucas Klemmer and Daniel Große. 2022. WAL: A Novel Waveform Analysis Language for Advanced Design Understanding and Debugging. In *ASP-DAC*. 358–364.
[9] Ed Sperling. 2022. Which Processor Is Best? https://semiengineering.com/which-processor-is-best.
[10] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley.