Blekinge Institute of Technology
Faculty of Software Engineering

Bachelor's Thesis

RISC-V Compiler Performance: A Comparison between GCC and LLVM/clang

Author
Johan Bjäreholt
johan@bjareho.lt

Supervisor
Richard Berntsson Svensson
richard.berntsson.svensson@bth.se

2017/06/22

urn:nbn:se:bth-14659

Abstract

RISC-V is a new open-source instruction set architecture (ISA) that in December 2016 manufactured its first batch of mass-produced processors. It focuses on both efficiency and performance and differs from other open-source architectures by not having a copyleft license permitting vendors to freely design, manufacture and sell RISC-V chips without any fees nor having to share their modifications on the reference implementations of the architecture.

The goal of this thesis is to evaluate the performance of the GCC and LLVM/clang compilers support for the RISC-V target and their ability to optimize for the architecture. The performance will be evaluated from executing CoreMark and Dhrystone which are both popular industry standard benchmarks for evaluating performance on embedded processors. They will be run on both the GCC and LLVM/clang compilers on different optimization levels and compared in performance per clock to the ARM architecture which is in comparison to RISC-V more mature yet the most similar widely used CPU architecture. The compiler support for the RISC-V target is still in development and the focus of this thesis will be the current performance differences between the GCC and LLVM compilers on this architecture. The platform we will execute the benchmarks on wil be the Freedom E310 processor on the SiFive HiFive1 board for RISC-V and a ARM Cortex-M4 processor by Freescale on the Teensy 3.6 board. The Freedom E310 is almost identical to the reference Berkeley Rocket RISC-V design and the ARM Coretex-M4 processor has a similar clock speed and is aimed at a similar target audience.

The results presented that the -O2 and -O3 optimization levels on GCC for RISC-V performed very well in comparison to our ARM reference. On the lower -O1 optimization level and -O0 which is no optimizations and -Os which is -O0 with optimizations for generating a smaller executable code size GCC performs much worse than ARM at 46% of the performance at -O1, 8.2% at -Os and 9.3% at -O0 on the CoreMark benchmark with similar results in Dhrystone except on -O1 where it performed as well as ARM. When turning off optimizations (-O0) GCC for RISC-V was 9.2% of the performance on ARM in CoreMark and 11% in Dhrystone which was unexpected and needs further investigation. LLVM/clang on the other hand crashed when trying to compile our CoreMark benchmark and on Dhrystone the optimization options made a very minor impact on performance making it 6.0% the performance of GCC on -O3 and 5.6% of the performance of ARM on -O3, so even with optimizations it was still slower than GCC without optimizations.

In conclusion the performance of RISC-V with the GCC compiler on the higher optimization levels performs very well considering how young the RISC-V architecture is. It does seems like there could be room for improvement on the lower optimization levels however which in turn could also possibly increase the performance of the higher optimization levels. With the LLVM/clang compiler on the other hand a lot of work needs to be done to make it competetive in both performance and stability with the GCC compiler and other architectures. Why the -O0 optimization is so considerably slower on RISC-V than on ARM was also very unexpected and needs further investigation.

Keywords: RISC-V, Compiler Benchmarking, Code Optimization, Microprocessors

Contents

1 Introduction 4

2 Background 5
2.1 Related Work 7
2.1.1 Previous RISC-V Compiler Comparisons 7
2.1.2 Previous RISC-V Benchmarks 7
2.2 Compiler Optimization on Other Architectures 9

3 Focus and Goals 10

4 Research Methodology 12
4.1 Independent Variables 12
4.1.1 Platform and Processor 13
4.1.2 Benchmarks 13
4.1.3 Compilers and Optimization 14
4.2 Execution 15
4.2.1 Experiment 1: Dhrystone 16
4.2.2 Experiment 2: CoreMark 17
4.3 Hypothesis 17
4.4 Validity Threats 18

5 Results 20
5.1 Dhrystone 20
5.2 CoreMark 22

6 Analysis 24
6.1 How well does compilers currently optimize for the RISC-V target? (RQ1) 24
6.2 How does compilers optimization flags affect performance on RISC-V? (RQ2) 24
6.3 Does GCC or LLVM have any RISC-V specific performance issues? (RQ3) 25

7 Conclusion 26
7.1 Future Work 27
7.1.1 Research 27
7.1.2 Development 27

8 References 29

1 Introduction

The processor market is has for many years been dominated by two architectures, x86_64 and ARM. At University of Californa, Berkeley at the faculty of Computer Architecture and Engineering (ARC)[1]they have been researching and designing CPU architectures since the 1980's, and in 2010 they started working on an open instruction set architecture (ISA) named RISC-V[3]. An ISA is a standard of the instructions, registers, calling conventions and other execution behaviours without specifying how the microarchitecture should execute these instructions. Reference implementations using the ISA has also been created for anyone to freely use and modify to make it easy to start designing custom designs built upon the ISA. RISC-V has now been in development for 6 years and the first commercial hardware implementations has now been released.

In this thesis we will evaluate how good the compiler performance for RISC-V is now that the architecture is starting to become widely used. It would be valuable to see where how the performance of the RISC-V processors differ depending on compiler and optimization flags to see if there is any room for performance improvement through optimization in the compiler output. Benchmarking has been done on RISC-V for very long on simulators to find out performance per clock, however only limited benchmarking has been done directly on hardware due to just recently starting with mass production of RISC-V chips. What noone has done yet however is compare the RISC-V compilers performance against eachother on different compiler optimization levels which has the potential to highlight performance differences and potentially find improvements through optimization for the compilers.

So the aim of this thesis will be to find how RISC-V performs depending on compilers and their optimization levels. As a reference our RISC-V results will be compared to an ARM processor with similar performance, features and price and compare relative performance differences between optimization levels. The benchmarks to be used will be the CoreMark and Dhrystone benchmark suites which will be compiled with the GCC and LLVM/clang compilers on RISC-V and only with GCC on ARM.

2 Background

The current state of the CPU market is that it is dominated by only a couple of architectures, both of them commercially licensed to multiple manufacturers with their own modifications of the same core. The desktop market is dominated by the x86_64 architecture used by Intel and AMD while the mobile market is dominated by the ARM architecture licensed by companies such as Apple, Qualcomm, Samsung and Mediatek. For many years there has been interest for an open-source CPU architecture designs, none of which have had a successful breakthrough however.

At University of California, Berkeley they have for many years been designing in-house CPU architectures used both as an educational tool and for research. At the Computer Architecture and Engineering faculty (ARC)[1] David Patterson designed the Berkeley RISC[2] architecture in the 1980's which have inspired many of the major CPU architectures of today such as ARM, Power and SPARC. In 2010 5 professors including David Patterson started working on a new standard instruction set architecture (ISA) to be the spiritual successor to Berkeley RISC. Open reference implementations of the ISA has also been created with the intention to make it easy for anyone to modify and potentially produce their own processors based on the RISC-V ISA. This has since gained a lot of interest not only in academia but also from the industry. Companies supporting the RISC-V foundation includes many industry leaders such as Intel, AMD, Nvidia, Google, Microsoft, Samsung, Huawei and Qualcomm[4]. The first mass-produced commercially available hardware implemetations have just recently begun to be sold and many more companies are planning to release and manufacture their own versions of it in late 2017 and 2018.

RISC-V is not the first open ISA however and projects such as OpenRISC[5] is an alternative for embedded systems but is still not a widely used architecture. Where RISC-V differs is that in contrast to OpenRISC it has a BSD license which allows companies to develop their own implementations of RISC-V without having to share their modifications back, while OpenRISC uses the GPL license which forces you to share your modifications back to the community (also known as copyleft). This has resulted in that a lot of companies have showed and interest and are backing the RISC-V foundation and therefore development of this architecture since they are doing the hard and expensive work of developing an open ISA and implementation that they later can modify to their own liking without having to share their intellectual property.

As Moore's law starts to slow down, the performance and efficiency improvements in computational circuits will start to rely less on the process node and more on the architecture[6]. 2016 was the first year in two decades where Intel had to delay the release of their new processor due to their transition to a smaller fabrication node took longer than expected. If this trend continues due to the difficulty of shrinking the size of transistors past 8nm this will shift the processor market to be less focused on the processing node and more focused on the processor architecture. RISC-V has a potential to gain a lot of marketshare if it could become of a greater benefit for corporations to make their own specialized variation of an architecture for their specific workload instead of choosing the best general purpose processor with the best processing node. A free and open CPU architecture also makes it easier for academia to do architectual research, and while other open architectures have existed for many years there have been none that have been popular in the industry which RISC-V has the ability to potentially become.

2.1 Related Work

Previous work regarding RISC-V is mostly either related to the low-level architecture such as specifications of the architecture[3], general design[7] and efficiency improvents[8] or custom processors built upon RISC-V specialized at specific workloads[9][10]. General performance benchmarking on RISC-V has been done previously but never with the purpose of comparing compilers due to compiler support for RISC-V is still being new. That the architecture is still in development has to be considered in the results, however but the point for this thesis is to try and find performance flaws in its design to help future development as constructive criticism rather than comparing it to the competition. In the forthcoming subsections we will discuss relevant previous related work about the compilers with RISC-V target support as well as general RISC-V performance benchmarks.

2.1.1 Previous RISC-V Compiler Comparisons

Regarding compiler benchmarks of RISC-V the only related study was a blog post[11] from the PhD student Graham Markall at the Imperial College London. The catch is that his post only shows differences in executable sizes which is interesting but it rarely correlates with the actual execution time. The results pointed that executables for RISC-V compiled with the GCC compiler was smaller than those compiled with the LLVM/clang compiler, especially on 32-bit. It was found that on average over multiple programs the 32-bit executables were 35% larger while on 64-bit executables only 13%. Executable size was also compared to the ARM architecture and both 32-bit and 64-bit RISC-V executable size was very similar to ARMv7m and ARMv8m while being 31% larger on the more efficiency focused ARMv8a. It is also worth noting that the title of his post contains "Part 1:", but there are no other relating posts on his blog. Since this "Part 1" was posted in May of 2016 and no follow-up post had been written over a year later it is likely that this blog post series was abandoned.

2.1.2 Previous RISC-V Benchmarks

Regarding ordinary performance benchmarks of the RISC-V architecture there are a few comparison tables of questionable reliability available online. The first performance table [1] is in the introduction file for the Rocket RISC-V project on Github where they compare the Rocker RISC-V imple-

[1] `https://github.com/ucb-bar/rocket/blob/master/README.md`, table 1 (last accessed: 2017-05-17)

mentation to a ARM Cortex-A5, but what chip is actually used and exactly what clock frequency it was running at was not specified (only specified that it was >1Ghz). Since both authors of that document are working at Berkeley it is very likely that the chip was one of the RISC-V prototypes made for Berkeley, but we still do not know which one. The second performance table [2] is from the Freedom E310 crowdfunding page comparing itself with other microcontrollers. The only benchmark they run here is Dhrystone where they measure both performance, performance per clock and efficiency. On the performance per clock and efficiency scores they have underclocked the board to 200Mhz however which makes you wonder what the scores would be at its stock speeds.

On the third RISC-V Workshop there was also a presentation about the out-of-order RISC-V processor BOOM (Berkeley our-of-order Machine)[12] where they presented a chart comparing both the BOOM and Rocket RISC-V processors to other processor architectures in CoreMark/Mhz. They did not mention however who measured these results, what compilers or optimization options were used on each architecture which makes it hard to evaluate the validity of the results. It is also impossible that they all used a single compiler of a single version on these tests since no compiler had upstream RISC-V support during the time that they had this presentation.

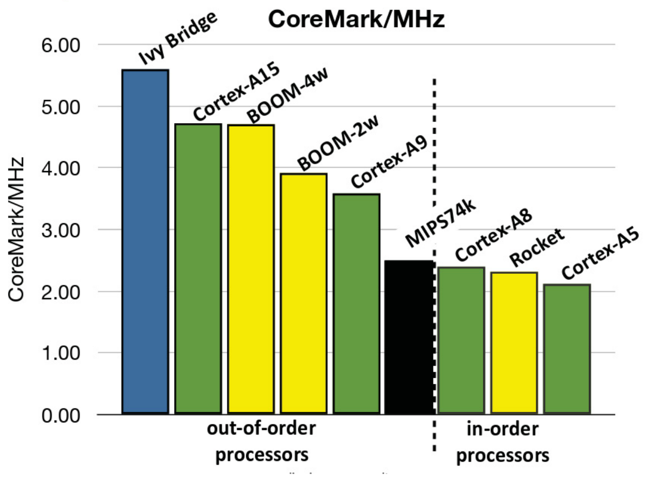

Figure 1: Performance per Mhz of RISC-V in comparison to other CPU architectures[13]

[2]`https://www.crowdsupply.com/sifive/hifive1` (last accessed: 2017-05-17)

2.2 Compiler Optimization on Other Architectures

Ther has been a lot of previous research on comparing compilers and their optimization potential on other architectures.

A comparisons between GCC and LLVM on the ARM platform has been done previously in the paper 'Comparison of LLVM and GCC on the ARM Platform'[14] with the results that the performance of LLVM was 97.2% of GCC on average on the 6 workloads which the MiBench benchmark consists of. From looking at each result from MiBench however we see that the performance is very inconsistent between the compilers from LLVM being 128% the performance of GCC on the djikstra alrogitm and 66% in susan tells us that the slight average win for GCC could easily be given to LLVM if they would have chosen some other workloads. The conclusion is that the performance between GCC and LLVM is very dependent on the workload, and since the average performance difference is so close it is not clear that GCC is always faster on average.

Comparisons between what optimizations are most effective at what workloads has also been tested in different ways in multiple papers. In the paper 'Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms'[15] they benchmarked the Cortex M0, M3 and A8 ARM processors and compared performance and efficiency to which they concluded that there is no optimization that generally improves performance on most workloads. This tells us that the default optimization levels -O1, -O2, -O3 and -Os are not optimal and should only be used as a general set of optimizations that will be beneficial in a majority of workloads.

3 Focus and Goals

The RISC-V project is very interesting both from a commercial and academic standpoint. Compared to the other existing open source instruction set architectures it unique in not having a copyleft license, has reference implementations focused on both performance (BOOM) and effiency (Rocket), is customizable due to having a small core with extensions and while still being in early development seems to be competitive with the current major architectures. These attributes makes it a very attractive architecture for large comanies who want to make their own processors without having to buy expensive licenses nor having to share their modifications to the architecture. Now that RISC-V is starting to go into mass production it is a perfect time to start evaluating it more thoroughly. One of the key components for a performant and efficient architecture is properly optimized compiler support for an architecture which will be the focus of this thesis.

Compilers that currently have RISC-V targets are GCC and LLVM/clang. While GCC has complete support for RISC-V, the LLVM compiler is still in early development. The GCC compiler has complete support for RISC-V and was added in upstream GCC version 7.1 (released 2017/05/02). LLVM/clang oon the other hand is still in development and only supports compiling source to assembly, so linking has to be done by GCC even when using LLVM.

The focus of this thesis will be direct performance comparisons between GCC and LLVM/clang. Noone has previously published research about RISC-V compiler performance and if unexpected performance differences are found it could help the development of these compilers and the RISC-V architecture. Comparing different optimization levels on RISC-V with both LLVM and GCC is also something that noone else has done before, and while it is possible that it will not differ much from other architectures it is still worth investigating. For reference we will be comparing all results with an ARM processor to find where major differences occur and pinpoint in what scenarios the compilers might need to improve their RISC-V target.

The value of the results of this thesis will be that they will help compilers find where to focus their compiler optimizations for the RISC-V target. Due to the compiler support for RISC-V being so young we expect there to be a lot of room for performance improvement. Compiler optimization is also a way to get free performance improvements which benefits everyone. The fact that RISC-V is a young architecture and just recently implementations started to get manufactures also means that compiler optimization support will become significantly more important in at least the forthcoming few years and possibly longer depending on how popular RISC-V becomes.

For the focus of this thesis to have a a value, we are aiming to fulfill these three goals:

- G1: To find the performance differences between GCC and LLVM on the RISC-V architecture
- G2: To evaluate how different compilers and optimization levels impact performance on RISC-V
- G3: Run the same benchmarks on ARM as a reference to compare with a more mature compiler infrastructure

Goal 1 is the primary goal but to give a in-depth result for it we need to better understand why the performance differs and put into context by comparing it to the competition, and that is where goal 2 and 3 comes in. Goal 2 helps us seeing how much impact optimization level actually does, and if the performance is lower than expected at some optimization level it can help developers better guess exactly which optimization flag it is that couses it to be able to fix it. Goal 3 puts the performance of RISC-V into perspective which is needed to understand what performance RISC-V should be aiming at to be able to compete with its competitors.

To achieve the goals specified we have written three research questions which the analysis will reflect upon to satisfy the goals.

- RQ1: How well does compilers currently optimize for the RISC-V target?
- RQ2: How does compilers optimization flags affect performance on RISC-V?
- RQ3: Does GCC or LLVM have any RISC-V specific performance issues?

Research question 1 is the primary question for our experiment. Research question 2 gives us a better understanding of the answer to research question 1 and is dependent on goal 2. Research question 3 is the most valuable question because while research question 1 answers the current state of the compiler optimizations for RISC-V, research questions 3 will go through where RISC-V currently has performance issues which pinpoints where the compilers could potentially improve in their support for RISC-V.

4 Research Methodology

The research method used in this thesis will be an experiment[16] utilizing two benchmarks compiled with multiple compilers and optimization options on two processors. Making a experiment by measuring performance by execution time is a reliable way to make a quantivative evaluation of performance as long as you are aware of the traits and impact of the independent variables. Benchmarks are used to measure performance on hardware and by compiling our benchmarks with different compilers and optimization options we will be able to see how the compilation affects the performance.

To run our experiment on our respective platforms we need to make a executable of our benchmarks. We feed the compiler with the source code from one of the benchmarks for compilation and linking into a executable which we can upload to a platform and execute to aquire a benchmark score. This will then be repeated with different optimization options to we in turn can measure the impact and compare it between our different compilers and platforms. Figure 2 illustrates the process of how the experiments were executed and gives an overview of all the independent variables for the resulting performance metric.

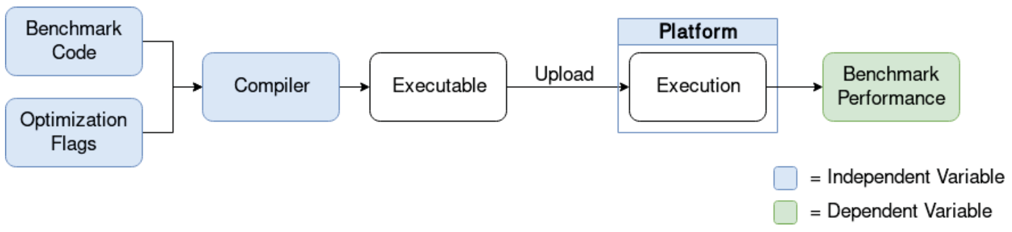

Figure 2: Overview of research methodology

In subsection 4.1 we will go through each independent variable in this process, why they were chosen and their impact on the results and in subsection 4.2 we will go through the experiments themselves and what configuration each will execute with.

4.1 Independent Variables

In our experiment we have executed 2 different benchmarks on 2 different CPU architectures with multiple compilers using multiple optimization settings. This makes a lot of possible combinations, but will be restricted slightly due to all options not being available on all platforms. In the coming subsections we will go through all independent variables and argue for why we chose each one specifically and in the next section we will present which combinations we executed.

4.1.1 Platform and Processor

For our platforms we need one RISC-V CPU as well as one other processor with a different CPU architecture to compare with.

For our RISC-V platform we chose the Freedom E310 processor on the HiFive1 board. The Freedom E310 CPU was chosen due to it being currently the only easily available RISC-V chip, it is still a good reference point due to being almost identical to the Berkeley Rocket RISC-V reference design. The Freedom E310 is an implementation of the Berkeley "Rocket" in-order RISC-V reference design and is clocked at 275Mhz.

For our reference platform we chose the Freescale ARM Cortex-M4 processor on the Teensy 3.6 board. We went for the ARM architecture as the reference architecture to compare with due to it being mature, well supported and has good compiler support. The Cortex-M4 was chosen for also being an in-order low-power CPU with a similar feature set. It is clocked at 175Mhz so 100Mhz lower than the Freedom E310 but that should not make much of a difference when we comparing in performance/Mhz. We will overclock the Cortex-M4 processor to compensate slightly for it however.

It is worth mentioning that there is also an out-of-order RISC-V design named BOOM (Berkeley out-of-order Machine)[12]. While out-of-order processors are much faster per Mhz, they come at the cost of being less efficient. Since no out-of-order RISC-V processors are available at the time the processors we have chosen to benchmark are both in-order processors.

4.1.2 Benchmarks

The benchmarks we chose were the popular Dhrystone benchmark in addition to CoreMark. Dhrystone was chosen due to it being a very popular embedded processor benchmark, which makes it easy to compare the results with lots of other processors as well as being multiple scientific analyses on it (Such as this paper written by EEMBC[17]). To compensate for Dhrystones flaws we decided to also run the more modern benchmark CoreMark which addresses multiple of Dhrystones issues.

Dhrystone[18] is a synthetic computing benchmark developed in 1984 intended to be representative of processors integer performance. It has been an industry standard for almost two decades and is therefore a great reference point since it is easy to find Dhrystone performance for most processors which could compared against. It does come with a few shortcomings however, the most severe being that compilers nowadays have optimization techniques that did not exist in the 80's when the benchmarks was written, so comparing recent compilers with an older one on the same processor can give significantly different results. In 1988 they released the 2.0 versions of Dhrystone that tried to fix some of the optimizations by modifying parts of the code so the compiler is unable to optimize those parts. While that solved many of the optizations issues during the time, more have surfaced since so this is still a relevant issue. This issue however has made Dhrystone into a compiler benchmark aswell which in our case is not necessarily bad since that is what we are interested in, however it is still an issue that Dhrystone does not reflect a common real-world computational scenario. Dhrystone is available in multiple languages, but the most commonly used C implementation version 2.1 is what will be used.

CoreMark[19] is a synthetic computing benchmark developed in 2009 by EEMBC (Embedded Microprocessor Benchmark Consortium) intended to become the successor to Dhrystone and therefore the new industry standard. Its strengths is similar to Dhrystone that it's both small, portable and free but has also addressed the ability of modern compilers to optimize away work. The CoreMark version to be used will be 1.0.

4.1.3 Compilers and Optimization

The compilers available with support currently are GCC and LLVM.

GCC has been the most popular compiler for RISC-V due to it having more support and being in active development. The GCC version for RISC-V used in our experiment was a version from the RISC-V foundations github repository based on GCC 6.1 (from 2017-01-24). As of May 2017 RISC-V has upstream support in GCC 7.1, but during the execution of this experiment that was not available.

LLVM/clang on RISC-V on the other hand is still in development and not considered complete, but it works and should stable enough for small and simple benchmarks. The LLVM/clang version for RISC-V used in our experiment was a version from the RISC-V foundations github repository based on LLVM 3.9-SVN (from 2016-06-22). One patch was applied to the LLVM repository to fix a bug where registers were not saved when calling a function, but that patch was available on a branch on the RISC-V foundations own LLVM repository but was for a unknown reason not merged into neither their stable or development branch. It is also worth mentioning that LLVM does not yet support linking so the linking for our LLVM experiments will be done with GCC. Since we will not be using link time optimization (LTO) in any of our tests it should not make a notable difference in performance however.

On ARM we used GCC 4.9.2 due to it being the officially supported version for our Teensy 3.6 board. We tried looking into using newer GCC versions on the Teensy board but quickly realized that it would be more work than expected so we gave up and used 4.9.2. The GCC version was the upstream 4.9.2 so it was a stable release and should contain no performance regressions.

The optimization options tested on both GCC and LLVM will most importantly be -O1, -O2 and -O3 to see the difference in performance but will also include the -O0 and -Os to evaluate how the performance is without optimizations and how performance is with minimal code size. These optimization options are sets of multiple optimization techniques which the compiler can utilize which can also be selected individually, but we chose these default optimization level options to see the performance difference between utilizing no optimization (-O0) and almost all optinizations (-O3). For CoreMark on RISC-V we will also test adding additional flags manually which we will in the rest of this thesis call +extra. In the CoreMark port for the RISC-V Freedom E310 we found that the developers had added a few extra optimization options, so we compared the performance with and without these extra options and saw a significant performance increase. Since we want to find the optimal performance, we decided to include these flags to our tests aswell. These extra optimization options were also tested on Dhrystone without any significant difference. The options are specifically -fno-common, -funroll-loops, -finline-functions, –param max-inline-insns-auto=20, -falign-functions=4, -falign-jumps=4 and -falign-loops=4.

4.2 Execution

In the forthcoming subsections we will describe both of our experiments which are the Dhrystone and CoreMark benchmarks. Each subsection will go into detail on how the experiments will be set up and describe what configurations they will be executed with.

For the results we will be describing both total performance as well as performance per clock. What is interesting in these benchmarks are not which processor scores higher per clock or in total, but how the performance differs depending on compiler optimization settings. This thesis is not written to find out whether RISC-V or ARM is better for microcontrollers, and to evaluate that you would also need to take power draw into account which we do not.

4.2.1 Experiment 1: Dhrystone

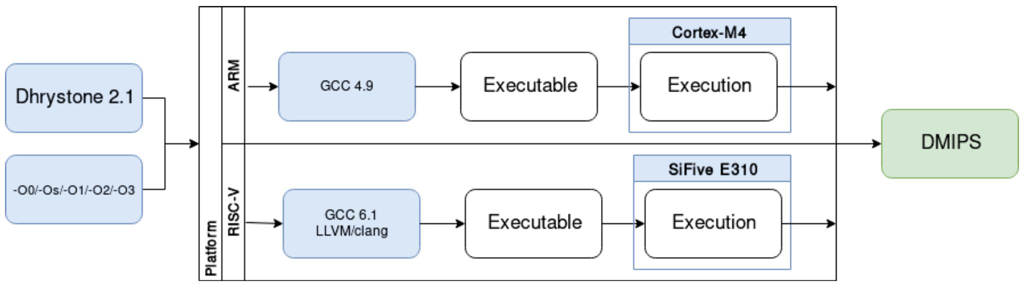

Figure 3: Overview of methodology for Dhrystone experiment

Dhrystone is a benchmark where a lot of performance can be gained from optimization made by compilers, even on the most recent 2.1 version. Hopefully this benchmark will show how just minor optimizations can increase the performance tremendously, and hopefully this will be the case on both architectures and all compilers.

We will run Dhrystone on both ARM and RISC-V in addition to run both GCC and LLVM/clang on RISC-V and only GCC on ARM. We will also compile the benchmark with the -O3, -O2 and -O1 optimization options focused on execution performance in addition to -Os for code size optimization and -O0 for no optimizations.

The score which Dhrystone reports after execution is "Dhrystones per Second" which is how many times the dhrystone tests complete in one second on average. It is common practice however to convert that result to Dhrystone MIPS (DMIPS) which is the score divided by 1757 (this is due to MIPS on a CISC and RISC processor are very different, so the reference is a 1 MIPS VAX11/780 machine which has a Dhrystone score of 1757). On all of our tests we executed 100 000 000 iterations of dhrystone which should be more than enough to get a good average of the performance per second. We will also convert the results to DMIPS/Mhz when comparing RISC-V with ARM.

4.2.2 Experiment 2: CoreMark

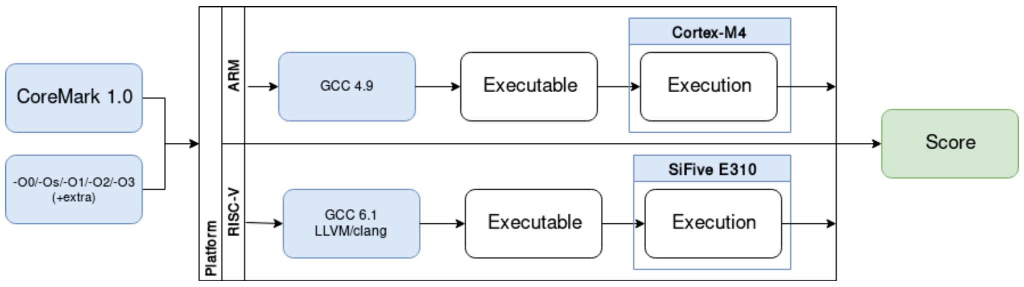

Figure 4: Overview of methodology for CoreMark experiment

CoreMark is a minimal benchmark supposed to replicate common computational workloads, and in comparison to Dhrystone it is less volatile to compiler optimizations. This should give us a decently realistic scores of the performance of programs with a smaller codesize since much of such small programs can fit in the cache.

We will run CoreMark on both ARM and RISC-V in addition to run both GCC and LLVM/clang on RISC-V and only GCC on ARM. The compiler optimization options tested will be the standard -O3, -O2 and -O1 focused on performance, -Os focused on code size and -O0 for no optimizations. In contrast to our Dhrystone experiment we will also execute all of our tests with a set of extra optimization flags both on and off (more information about this in section 4.1.3).

The CoreMark score is the amount of iterations of the tests executed divided by total execution time (iterations/second). On all of our tests we set the amount of CoreMark iterations to 400 000 iterations. We will also convert the results to score/Mhz when comparing RISC-V with ARM.

4.3 Hypothesis

To make a hypothesis we need facts to base our guesses on the performance implications of each independent variable.

Firstly what we know about the compilers for RISC-V is that GCC is much more actively developed than LLVM/clang and should therefore likely perform better in most scenarios, but by how large margin is unclear. On other architectures however the performance difference between GCC and LLVM/clang should not be very large on average, but in specific workloads it can be quite different as shown in the 'Comparison of LLVM and GCC on the ARM Platform'[14] paper. The LLVM/clang compiler is still supposed to be rather stable so it should hopefully not contain any critical bugs and since RISC-V support for GCC is upstreamed and has been thoroughly tested it definitely should not contain any critical bugs.

Secondly regarding our benchmarks and optimization flags we expect to see a bigger difference between higher optimization levels on the Dhrystone benchmark than in CoreMark. On the -O0 optimization level we expect the score between GCC and LLVM/clang to be almost identical on RISC-V.

Thirdly regarding our architecture we expect RISC-V with GCC on the highest optimization level to be competetive with ARM but still perform worse per clock. The reason we expect this is because the RISC-V core design such as the ISA is final and should be competetive but there is still room for improvement on the architecture and probably in compiler optimizations aswell.

In conclusion we expect RISC-V to perform better than ARM due to its higher clockspeed but not in performance per clock. Regarding the compiler comparisons we expect GCC to perform slightly better due to its mature RISC-V support but still expect LLVM to not be far behind. Regarding optimization levels we expect them to be similar between compilers and possibly also between architectures, but if we find any optimization issues this might not be the case.

4.4 Validity Threats

There are a few threats[16] to the results in this thesis which could skew the results slightly, but by knowing them when analyzing the results we can approximate the error margins of the results and still get a fair evaluation.

Firstly the only officially supported compiler for our ARM platform is GCC 4.9 while for RISC-V it is GCC 6.1. From looking at benchmarks of both GCC versions on the same system under the x86_64 architecture [3] the difference will most likely not be more than a few percent however that is not certain. The GCC compiler developers are doing rather extensive regression tests on each major version release and no significant optimization improvements have been made during the two years between those releases, so hopefully this will not become an issue.

Secondly you could argue that the tests are biased due to the ARM processor being overclocked while the RISC-V processor is not. The reason that we overclocked the ARM processor was to make the clock frequency to be closer to that of the RISC-V processor to more easily compare performance

[3] `http://www.phoronix.com/scan.php?page=article&item=gcc-61-debian`
last accessed: 2017-05-15

per clock. It is mentioned in the documentation for the Freedom E310 RISC-V processor that it is possible to under- and overclock it, but we found no documentation on how to. If a method to easily underclock the Freedom E310 was available we would love to try to underclock it to the same frequency as our ARM Cortex-M4 processor.

A third potential threat to our results could be that memory, cache and storage have different speeds on our RISC-V and ARM platforms. Since our benchmarks are rather small in terms of executable size however it is not impossible that the whole program could be contained in the CPU cache. Since the cache is a part of the processor design you could argue that we want that to be a part of the comparisons between the architectures though. We do not expect cache to be making a significant difference to the performance in these tests though since it is unlikely to be the bottleneck in processors with such low clock frequencies.

5 Results

The results aligned with our hypothesis in some areas, but there certainly were some outliers. We expected that GCC would outperform LLVM/clang slightly, however the case was more extreme and LLVM/clang either performed very low or crashed during compilation. A general trait in the results in both benchmarks are that the -O0 and -Os optimization levels was considerably slower on RISC-V than on ARM which was unexpected. In the next subsections we will present the results in-depth and compare them with what was expected from our hypothesis.

5.1 Dhrystone

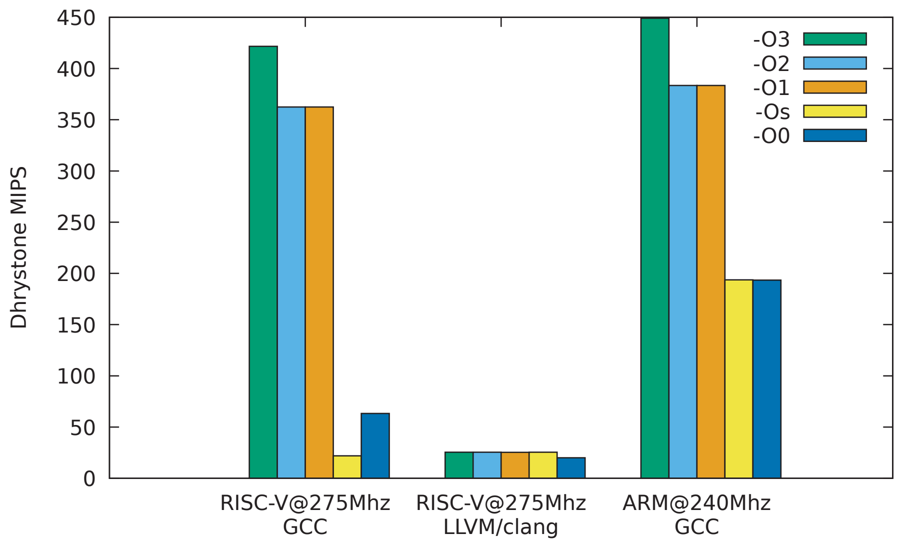

Figure 5: Dhrystone 2.1 Performance

In figure 5 we can see that the Dhrystone performance was overall rather decent on both RISC-V and ARM with GCC. The -O0 and -Os performance on GCC seems to be very low for a unknown reason, but since even the -O1 performance is pretty good this should not be an issue unless you are using the -Os optimization level or are debugging on assembly level under a heavy load. What is odd here however is that -Os is significantly slower than -O0, which definitely points to some issue. Our RISC-V performance on GCC was very close to what was advertised for the Freedom E310 (We got 1.53 DMIPS/Mhz on 275Mhz while it was advertised as 1.6 DMIPS/Mhz on 320Mhz).

The LLVM/clang on the other hand seems to be broken. On all optimization levels above -O0 only performs slightly faster than -O0 on GCC so optimizations barely seem to make a difference. The -O0 level on LLVM is also a third of the performance of -O0 on GCC which in turn is a third of the performance of -O0 on ARM which certainly points to some significant issue in its code generation.

The performance compared to ARM on GCC is very good and matches with our hypothesis that it would be slightly lower per clock but still rather close and competetive. An unexpected similarity was that the performance between -O1 and -O2 was insignificant in Dhrystone, but -O3 gave a larger performance boost on ARM than on RISC-V. It is rather sad to see how badly LLVM/clang performed, but hopefully more development will be done on LLVM/clang for RISC-V in the forthcoming years when RISC-V starts becoming more popular.

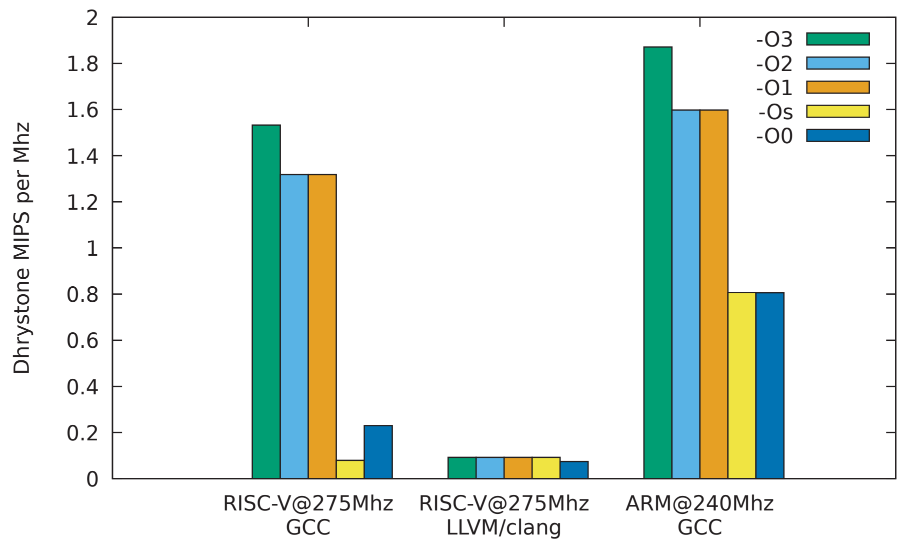

Figure 6: Dhrystone 2.1 Performance per clock

In figure 5 we can see that the performance was 6% faster on our ARM processor than on our RISC-V processor, but when looking at figure 6 which takes clock speed into account the ARM processor was actually 22% faster. What made RISC-V so competetive in total performance in figure 5 was due to its higher clockspeed, and as we see in figure 6 the ARM processor would be approximately 22% faster if they ran on the same clock speed.

5.2 CoreMark

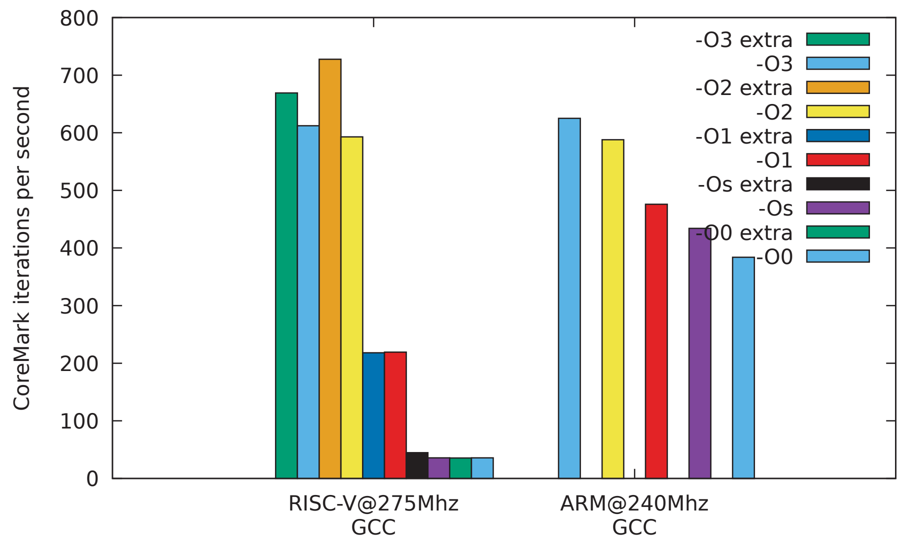

Figure 7: CoreMark Performance

In figure 7 we can see that the performance is rather even between RISC-V and ARM in the CoreMark benchmark. The GCC performance seems to be very good and those extra optimization flags gives a very good performance increase especially in conjunction with -O2. The optimization flags in extra are not in any of the optimization levels, but what is odd is that we do see quite a bit of performance increase with extra on -O2 and -Os but a very small improvement in conjunction with -O3 and no improvement with -O1 and -O0. This has made -O2 extra the fastest optimzation configuration, but why this is the case we do not know.

The state of the LLVM/clang compiler however is rather bad since it did not even compile CoreMark for RISC-V. CoreMark is a rather small program and not overly complex, so we did not expect LLVM/clang to fail to compile it. There are however reports online of patching the LLVM compiler to be able to compile CoreMark for RISC-V which was tested, so hopefully it atleast should not be to hard for this issue soon.

The performance with GCC on RISC-V compared to ARM is great and in fact almost equal on -O2 and -O3 without the extra optimization flags. The lower -O1, -O0 and -Os optimization levels is good on ARM but definitely lacking with GCC on RISC-V though, if it is possible to fix this in the compiler or an issue with the architecture however needs to be investigated.

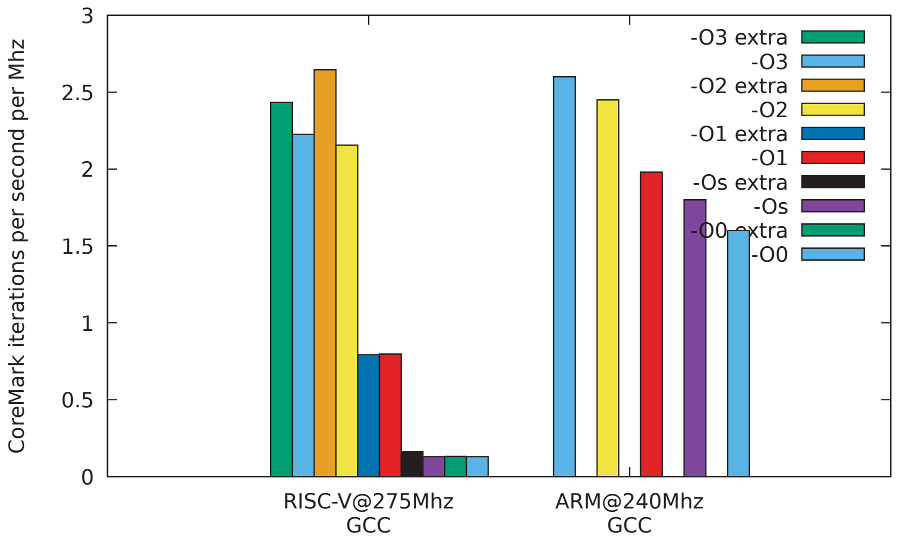

Figure 8: CoreMark Performance per clock

As seen in figure 8 the performance per clock is slightly won by our ARM processor, however the extra flag on RISC-V makes quite an impact so if we count that in RISC-V with -O2 extra is actually faster than ARM with -O3. This is unfair though because we have not tested the ARM processor with extra optimization flags and it is very likely that there are some specialized flags available that could help our ARM processor squeeze out slightly more performance per Mhz in this specific workload. From looking at the paper 'Identifying Compiler Options to Minimize Energy Consunption for Embedded Platforms'[15] we can see proof that performance impact of optimization flags are very specific on both workload and architecture so optimizing our ARM processor to gain slightly more performance to become faster than RISC-V with the extra optimization flags is very likely to be possible. The extra optimization flags are mostly align optimizations such as align-loops, align-jumps and aligh-functions which would very likely make a similar impact on ARM as on RISC-V. The unroll-loops and inline-functions optimizations in extra would also likely improve performance on the ARM processor due to branch prediction being more primitive on these slower and more efficient microprocessors in comparison to faster ARM architectures and CISC processors. Still even without the extra optimization flags RISC-V is competitive in performance per Mhz to ARM, and with some architecture or compiler optimization it might have the ability to beat it in the future.

6 Analysis

6.1 How well does compilers currently optimize for the RISC-V target? (RQ1)

It does perform rather well with GCC on the higher optimization levels such as -O3 and -O2 with GCC, but there seems to be some issues on lower optimization levels. Since performance is the most important in production and the higher optimization levels perform well you could consider the GCC support to be fast enough for production, but when debugging on assembly level or you need faster compilation time during development this could be an issue on computationally demanding programs.

LLVM/clang on the other hand failed to compile our CoreMark benchmark and the optimization flags on Dhrystone did not make any performance impact at all. A lot of work needs to be done on LLVM/clang until it becomes a viable alternative for compiling RISC-V binaries.

6.2 How does compilers optimization flags affect performance on RISC-V? (RQ2)

GCC on RISC-V performed very well on the higher optimization levels, but was significantly slower on lower optimizaion levels. The -O1 optimization flag was 16.6x faster than -O0 in Dhrystone which is not normal for other architectures. But the higher optimization levels such as -O2 and -O3 was very competetive with ARM which was rather impressive for such an young architecture.

LLVM on the other hand had a few issues. With Dhrystone none of the the optimization levels barely made a performance improvement over -O0 so the performance was very slow, especially considering that -O0 seems to be significantly slower on than both GCC och ARM and on RISC-V. With CoreMark it did not compile, making us unable to run any performance tests.

The -O0 and -Os optimization levels on GCC were very low on both benchmarks in addition to -O1 also being rather slow on CoreMark when comparing RISC-V with ARM. Further investigation is needed to find out why this is the case, possibly it is just an issue with the compiler but it could also be an architectual defect. Another odd find was that -Os was slower than -O0 on Dhrystone with GCC, which points to some issue where some optimization must be decreasing execution performance significantly.

The extra flags gave quite a bit of extra performance in CoreMark on RISC-V, it would be interesting to see if these optimization flags could be an general improvement on most workloads or if it is specific to CoreMark.

If the performance increase is not specific to CoreMark and is generally increases rather than decreases performance it could possibly give a rather significant improvement. Regarding the flags in extra we have some which reduce branching (unroll loops and inline functions) and some which align branches, loops, jumps and labels. All the align optimizations are enabled by default in -O2 and -O3, but at what levels the align optimizations are set by default on the RISC-V platform I do not know. So whether the align optimizations in extra are actually doing anything above -O1 is unclear and needs further investigation. That the extra flags which unroll loops and inline functions increases performance could point to that the branch prediction in RISC-V is subpar. On simpler architectures focused on efficiency which microprocessors like those we tested are branch prediction is generally not a priority however. Both the Rocket architecture on our RISC-V processor as well as our Cortex-M4 processor does have simple branch prediction though, so if these will be used in higher performance chips it would possibly need some improvement.

6.3 Does GCC or LLVM have any RISC-V specific performance issues? (RQ3)

Both GCC and LLVM/clang have RISC-V specific performance issues.

GCC performs rather well with the -O2 and -O3 optimization levels but it does have issues on lower optimization levels which is not too critical but it is an issue nonetheless. The -Os optimization level on also performs a third of the performance of -O0 on Dhrystone which is rather troublesome. We do not know the reason for these performance issues so further investigation would be needed to understand whether this is an compiler issue or an architectual issue.

LLVM/clang on the other hand definitely has a long way to go in terms of optimization since it seems to be fundamentally broken by not leveraging any performance increase at all. The current state of the LLVM/clang support for RISC-V is that the maintainer has been inactive for a while now, so to get this working again we firstly would either need a new maintainer or the current maintainer to start working on the project again.

7 Conclusion

Further investigation needs to be done on why the performance on the lower optimization levels on RISC-V is so low to understand how the architecture responds to optimizations and find out if there is room for improvement in the compilers or if it is a side-effect of the core design of the RISC-V architecture. Since the -O2 and -O3 optimization levels already perform well this is not a critical issue but will be an annoyance for projects that depends on -Os or developers who need to debug heavy applications on assembly level with -O0. A lot of work is also needed on the LLVM/clang compiler on RISC-V to become competetive with the performance and stability of GCC. We did not expect LLVM/clang to be on-par with GCC on RISC-V, but we at least expected it to compile our benchmarks and execute at acceptable speeds which it did not.

To conclude this thesis, shortened answers to each of the research questions are:

RQ1: How well does compilers currently optimize for the RISC-V target?
The performance of RISC-V under optimal conditions such as with the GCC compiler with the -O2 or -O3 optimization flags was very competetive with ARM especially for being such an young architecure. The LLVM/clang compilers support for RISC-V was in very bad shape however and in the case of CoreMark crashed during compilation and in Dhrystone the code optimizations barely made a performance impact.

RQ2: How does compilers optimization flags affect performance on RISC-V?
As stated in RQ1, the performance of RISC-V under the -O2 and -O3 optimization flags perform very well on GCC. Where issues arise with GCC on RISC-V is on the -O0 and -Os (and -O1 on CoreMark) optimization levels which performed significantly lower than ARM. On LLVM/clang with Dhrystone the performance was identical on all optimization levels except for -O0, but on all levels it performed less than a third of the performance of GCC even on -O0 which signals that it has some more fundamental issue than just optimization.

RQ3: Does GCC or LLVM have any RISC-V specific performance issues?
As stated in RQ2, both the GCC and LLVM/clang compilers have performance issues. While GCC does have good performance on -O3 and -O2 which are generally the only ones used in production and deployment it is less of an issue, but that -O0 and -Os perfors so badly is still an issue and if fixed could potentially improve performance on -O2 and

-O3 aswell. LLVM/clang has severe performance and stability issues currently which does not make it a viable choice for now.

7.1 Future Work

There is a lot of room for improvement for the compiler infrastructure for RISC-V. In the forthcoming subsections i will present ideas both for additional research as well as development which would help the development of the RISC-V and its compilers.

7.1.1 Research

Investigate why -O0 is slow on RISC-V as well as figure out why -O1 is significantly slower in CoreMark on RISC-V than on ARM. This would be interesting to find out if there is room for improvement or if it is a side-effect of the design of the RISC-V architecture.

Efficiency of RISC-V compared to ARM on processors with the same process node to solely compare the architectures and implementations efficiency.

Benchmark an out-of-order (BOOM) RISC-V processor when one goes into mass production and compare with other modern high performance out-of-order processors such as ARMv8 and x86_64.

7.1.2 Development

An active RISC-V LLVM maintainer to start taking in pull requests as well listing what is and is not supported in addition to what needs to be fixed. The LLVM compiler infrastructure is currently not an attractive alternative on RISC-V and competition to GCC could drive development of both further.

Fix CoreMark not compiling on LLVM/clang would allow us to properly analyze the potential of the LLVM/clang compiler and compare it with GCC.

Fix Dhrystone performance issues on LLVM/clang would allow us to properly analyze the performance of the LLVM/clang compiler and compare it with GCC.

Automated benchmarks for embedded systems on multiple platforms (possibly both hardware and emulator) would make benchmarking of multiple benchmarks and configurations much easier. Making this automatic with preconfigured toolkits, automatic installation of compilers and automatic execution would allow much more rapid analysis of performance under different workloads on each platform so you could put less time setting up the tests and spend more time on analysing them and improving the performance.

8 References

[1] Faculty of Computer Architecture & Engineering
`https://www2.eecs.berkeley.edu/Research/Areas/ARC/`
(Last accessed: 2017/05/05)

[2] RISC I: A Reduced Instruction Set VLSI Computer
David Patterson, Carlo Sequin
ISCA '81 Proceedings of the 8th annual symposium on Computer Architecture, p. 443-457
1981-05-12

[3] The RISC-V Instruction Set Manual
Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic
`https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf`
2014-05-06 (last accessed 2017-05-19)

[4] RISC-V Foundation Members Directory
`https://riscv.org/membership/`
(Last accessed: 2017/05/05)

[5] OpenRISC
`https://openrisc.io/architecture`
(last accessed: 2017-05-23)

[6] Fifty Years of Moore's Law
Chris A. Mack
IEEE Transactions on Semiconductor Manufacturing, Volume 24, No. 2
2011-05

[7] A RISC-V instruction set processor-micro-architecture design and analysis
Aneesh Raveendran, Vinayak B. Patil, David Selvakumar
2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)
2016-01-10

[8] Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed
Andrew S. Waterman
`https://people.eecs.berkeley.edu/~krste/papers/waterman-ms.pdf`
2011-05-13 (last accessed 2017-05-19)

[9] GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator
Jan Gray
2nd International Workshop on Overlay Architectures for FPGAs (OLAF 2016)
2016-05

[10] A RISC-V Vector Processor With Simultaneous-Switching Switched-Capacitor DC–DC Converters in 28 nm FDSOI
Brian Zimmer, Yunsup Lee, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevti, Ben Keller, Steven Bailey, Milovan Blagojevi, Pi-Feng Chiu, Hanh-Phuc Le, Po-Hung Chen, Nicholas Sutardja, Rimas Avizienis, Andrew Waterman, Brian Richards, Philippe Flatresse, Elad Alon, Krste Asanovic, Borivoje Nikolic
IEEE Journal of Solid-State Circuits, Vol 51, No 4
2016-03-01

[11] RISC-V Compiler Performance Part 1: Code Size Comparisons
Graham Markall
`http://www.embecosm.com/2016/05/26/risc-v-compiler-performance-part-1-code-si`
2016-05-26 (last accessed 2017-04-27)

[12] The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor
Christopher Celio, David A. Patterson, Krste Asanović
`https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.pdf`
2015-06-13 (last accessed 2017-04-27)

[13] The Berkeley Out-of-Order Machine: An Open-source Industry-Competetive, Synthesizable, Parameterized RISC-V Processor
Presentation at the 3rd RISC-V Workshop
`https://riscv.org/wp-content/uploads/2016/01/Wed1345-RISCV-Workshop-3-BOOM.pdf`
2016-01-06 (last accessed 2017-04-27)

[14] Comparison of LLVM and GCC on the ARM Platform
Jae-Jin Kim, Seok-Young Lee, Soo-Mook Moon, Suhyun Kim
5th International Conference on Embedded and Multimedia Computing (EMC)
2010-08-11

[15] Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms
James Pallister, Simon J. Hollis and Jeremy Bennett
The Computer Journal, Volume 58 Issue 1
2015

[16] Basics of Software Engineering Experimentation
Natalia Juristo, Ana M. Moreno
ISBN: 9780792379904
2001

[17] Dhrystone Benchmark: History, Analysis, "Scores" and Recommendations
Alan R. Weiss
`http://www.eembc.org/techlit/datasheets/dhrystone_wp.pdf`
2002-10-01 (last accessed 2017-05-09)

[18] Dhrystone: a synthetic systems programming benchmark
Reinhold P. Weicker
Communications of the ACM, Volume 27 Issue 10
1984-10

[19] CoreMark Homepage
`http://www.eembc.org/coremark/about.php`
(last accessed: 2017-05-09)