An Evaluation of the Viability of a Configurable Cache within RISCV

Michael Feldt feltdm@email.arizona.edu
Connor Musick cjmusick@email.arizona.edu
Kristopher Rockowitz rockowitzks@email.arizona.edu
Alex Sisson sisson1@email.arizona.edu
Cole Spinali cspinali@email.arizona.edu

I. Introduction

Optimization of computer architecture relies upon developing systems which intelligently and strategically make use of resources such as power. It follows that within the design process, each component should be evaluated for its respective contribution to performance and subsequent demand for resource. This is best described as the a trade off analysis. It is well known that cache can be a significant drain on system power, consuming as much as forty percent of power in certain scenarios. It is equally true that caches are essential units which bolster performance of the system. With this in mind, a logical notion would be a cache that is able to be configured dynamically to alleviate power consumption woes while also maintaining performance prowess from application to application. Such a cache has been developed at the circuit level by Zhang[1] and thus is believed to be a viable optimization. In this paper, we make attempts to implement a configurable cache in a Reduced Instruction Set Computer Five (RISCV) Instruction Set Architecture (ISA) environment on many repository-sourced cores. Additionally, we experiment with the usage of the Gem5 simulator, which has recently implemented limited functionality with RISCV. While the impact is clearly one of benefit, the way to implementation is still obfuscated by constraints such as completeness of hardware description language design, support for the new ISA with respect to Gem5, and project time limitations. This report serves to document the various trials aimed at implementing a configurable cache utilizing RISCV and preliminary results.

II. Previous Work

Zhang, as well as a few other authors, have explored cache configurability in great detail. Specifically, Zhang developed a circuit level way concatenation method which improved cache performance by facilitating cache way switching through concatenation of bits onto the address, allowing for placement into a variety of user selected or program necessitated cache ways. This method is profound as it implements hardware that saves power by shutting down parts of the cache that are unused at any given time during program execution. As Zhang points out, the prominent differences in the power consumed by direct mapped and set associative caches are found in their tag array access. The multitude of accesses resident to set associativity increase the power usage but potentially reduce the miss rate, providing an interesting trade off to analyze Zhang[1]. It is also worth noting that the hardware overhead, for the exception of a hypothetical tuner, is relatively small as it is designed using computationally and monetarily inexpensive logic gates.

III. Method

Our intent was to implement a configurable cache not at circuit level, but at RTL level using Vivado Verilog as the language proved to be the most accessible. Other languages, like the RISCV native SCALA, and simulators such as Chisel exist and are likely avenues of approach for future exploration. Their closer connection to the new ISA With the explosive popularity of RISCV, a proliferation of GitHub repositories exist with various cores, designed using VHDL, or Verilog. Additionally, with the verbose nature of Vivado, the Xilinx Power Estimator (XPE) is accessible for the latter stages of design in which assessing energy impact would be considered. Additionally, Gem5 supports limited RISCV functionality, providing a less realistic but still accurate model of a core for which to attach and experiment with a configurable cache prototype. The levels of cache design were to be implemented as a series of experimental optimizations, starting at what would have essentially been two merged caches. One cache would be designated direct mapped and the other 2 way set associataive, with the intent to redesign incrementally, eventually graduating to a cache which is able to switch between associativities dynamically. Our plan was to closely follow the research published by Zhang, incorporating way concatenation to decide upon the signal and index into whichever setting the user or program has specified.

IV. Experiments

Our group began by evaluating the lowRISC project and repository as it was free of cost and implemented an untethered RISCV system that could boot linux. We cloned the repository and built the RISCV toolchain, installing all necessary dependencies. Our first experiment was an attempt at running the RISCV benchmarks on the lowRISC booted Linux. These were discovered to be unhelpful as they produced files that contained only the assembly instructions being executed, as well as the data and registers involved in its execution. This did not provide any useful cache performance, energy consumption, or execution time metrics. As this experiment proved unsuccessful, we ventured on in search of a way to

statically assess the performance of modifying cache associativity with RISCV. With uncertainty about how to proceed creation of a cache within the lowRISC implementation we sought to change our experimental focus. The figure below shows the lowRISC L1 instruction cache layout. The relatively simple design is an attractive attribute for the purposes of customization.

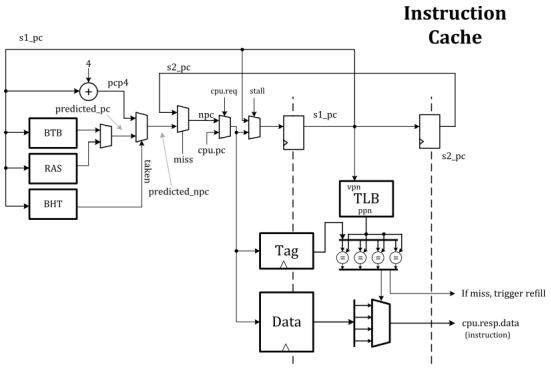

Fig. 1. lowRISC L1 Instruction Cache Layout.

During evaluation of lowRISC, we discovered a wealth of other repositories containing RISCV cores of various quality and completeness with a common factor, VHDL and Verilog files. From our experience with Vivado and the benefit of the power estimation, we began cloning projects and attempting to build them in Vivado to a level of synthesis functionality. It was during this time that we came up with rough drafts of what a configurable cache would look like at the register transfer level. A common variable in most cores dealt with cache ways, a reference to statically setting the associativity of the cache. Our drafts of a configurable cache built off of this by creating a module which accepted that variable with the the intention of spontaneously generating a new cache at some time during program execution. We also discussed implementing a modified tag and index system and using a concatenation theme not unlike way concatenation featured in Zhang[1]. The idea was to make the cache a larger than available size and modify the cache to accept a two bit binary flag representing the three states of 1 way or direct mapped, 2 way, and 4 way. The flag would be interpreted by the logic of the module via case statements. For example, a signal of 00 would indicate the cache was to remain in or switch to a direct mapped 1 way set associativity. Similarly, a signal of 01 or 10 would map to 2 way set associative, and a signal of 11 would set 4 way set associative. The dynamics of this process, as they would certainly not benefit from simple logic gates were to be determined based upon the specifics of the core within each respective repository and optimized accordingly. Below is a figure representing Zhang's configurable cache at the circuit level with way concatenation.

As we discovered, there were tremendous variances between the several core repositories we investigated. The general trend was incompleteness of the hardware description language files, and an inability on behalf of the team to quickly fill

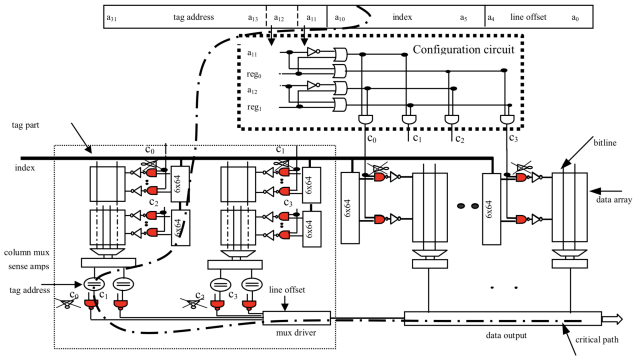

Fig. 2. A configurable cache at circuit level devloped by Zhang.

in the gaps to achieve simulation. We diligently attempted to make the barest of minimum functionality occur with no reportable success. The repositories explored included Ariane, RV12, picorv32, and Potato. Ariane, while robust and well documented had amassed a tremendous amount of files and it was perplexing attempting to sift through and integrate the project files into a working Vivado scenario. RV12 lacked modules that the core imported, and after much searching the modules were unable to be located. Picorv32 lacked sufficient documentation and testbenches to create a working copy of the core. Several other cores were imported into Vivado but lacked cache modules to customize. Potato, the one completely synthesizable implementation we found made use of an interconnect made with Wishbone to take the place of the cache and pertinent documentation led us to believe that decoupling the interconnect and creating caches would violate our time constraints. Below are a few core layouts we explored, they would have made excellent candidates had the documentation and project files work harmoniously in Vivado.

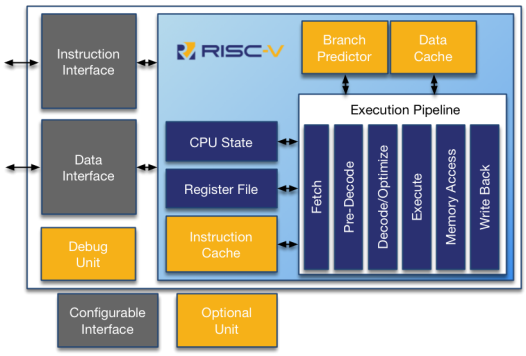

Fig. 3. RV12 core layout.

At the halfway point of our project time frame we voted to abandon the pursuit of a HDL based approach and commit to using the Gem5 simulator. Gem5 recently obtained limited support for RISCV, and possesses extensive tutorials for learning the basics of simulation through an x86 build. The tutorials gave a basic orientation to the simulation process, illustrating the process of creating configuration scripts, simu-

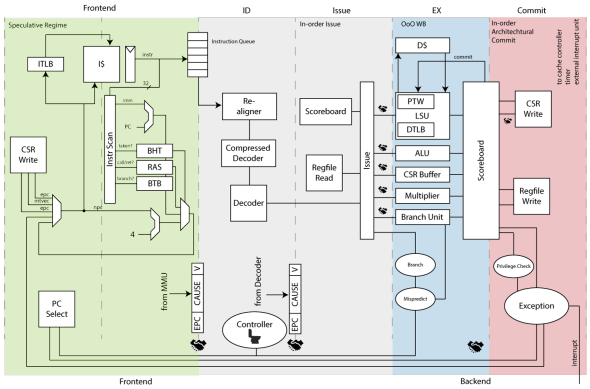

Fig. 4. Ariane core layout.

lation objects, event handling and registration of new objects within Gem5. Immediately, we located the cache C based files of gem5 and began evaluation of the functions. Figure 5 below gives a brief schematic of how the connections function. Within the Base Cache C file, a group of functions exist which

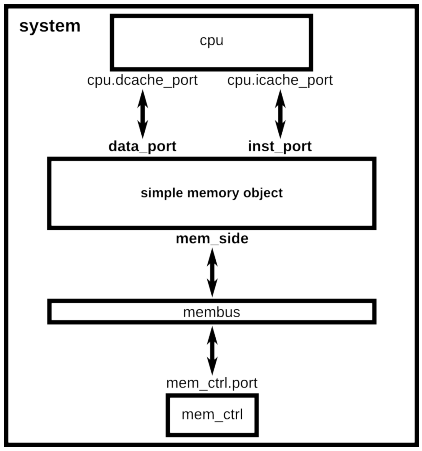

Fig. 5. Gem5 layout of connections for memory objects.

handle message passing between the cpu and memory. A very basic cache which extends this logic was instantiated and connected via crossbar and CPU in the configuration script, mirroring descriptions found in the tutorial. The simulation ultimately failed repeatedly due to an assertion flag detailed in figure 6 below. Probing the error, we found the assertion function call within the header file but no indication of cause for the flag trigger. This same error was recreated on a separate build of Gem5 RISCV. It seemed logical to conclude that the cache logic, an extension of the Base Cache, was somehow incomplete or unable to properly pass the data to the CPU, as all previous simulations with caches were successful. On yet another build, a different error regarding vector port assignment cropped up, presenting a degree of variability with otherwise identical installations and processes. The group sought the advice of Kyle Kuan, as it was indicated that he has extensive knowledge of Gem5. His recommendation was that we statically simulate the cache configurations as a deeper understanding of Gem5 and interconnects would be necessary to produce meaningful results.

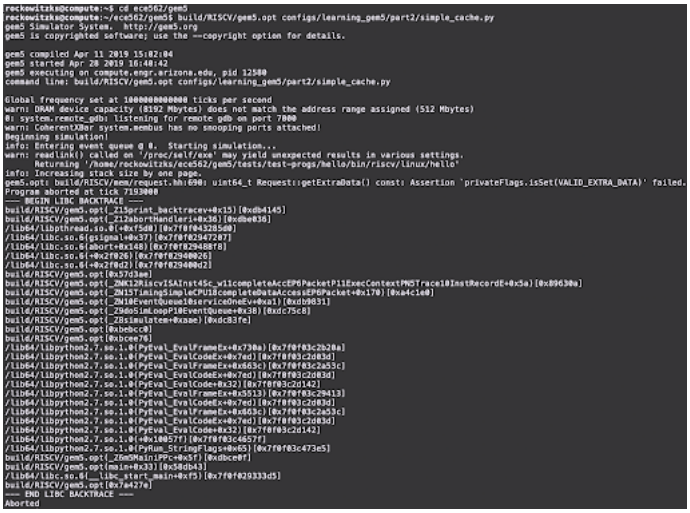

Fig. 6. The error and subsequent back trace produced by implementation of a simple cache object in Gem5.

Our experimental focus eventually became this static simulation, to re-assess the viability and impact of a configurable cache within the RISCV ISA by running benchmarks and documenting the performance of the cache at various associativites.

V. Results

We presumed that a direct mapped cache should be outperformed by set associativity at some level and proceeded with experiments to that effect. As time and again the RISCV benchmarks, compiled properly with the GNU GCC and static flag, served only to create page faults, we devised our own benchmarks. The MergeSort and QuickSort sorting algorithms were created in C, selected for their memory intensive nature. They were then compiled using the RISCV Newlin Cross Compiler Toolchain, then run with the gem5.fast build. The algorithms, although similar in function provide a degree of variability in their cache performance as see in the figure. QuickSort has a locality advantage and is an in-place sort while MergeSort takes a dynamic approach to data. The base expectation for the cache miss rates is predicted to be low based upon the relatively small set of instructions which are repeated, regardless of the set associativity. These benchmarks operate on the common dataset of 1048576 random, 4 byte integers generated at compile time. The values of the integers range from 0 to 65535. The experiments performed included variation of associativity from 1, 2, 4 and 8 ways on each benchmark. Variations on Cache sizes were attempted but generated warnings, dissuading us from future trials. A sample of commands executed is featured in the figures below.

The $MAXINST shell script variable was set to 300000000 for every execution of the benchmarks. This was done for time-saving purposes since a single merge sort run could last 1.5 billion instructions with such a large dataset. Upon

```
$ build/RISCV/gem5.fast --stats-file=mergesort_8.txt --dump-config=mergesort_8.ini
configs/example/se.py -c ~/Documents/MergeSort/MergeSort.riscv
--mem-size=8192MB --caches --l1i_size=32kB --l1i_assoc=8 --l1d_size=32kB
--l1d_assoc=8 --cacheline_size=64 --cpu-clock=1.6GHz -n 1 --maxinsts=$MAXINST
```

Fig. 7. A sample command run on gem5.

```
$ build/RISCV/gem5.fast --stats-file=mergesort_4.txt --dump-config=mergesort_4.ini
configs/example/se.py -c ~/Documents/MergeSort/MergeSort.riscv --mem-size=512MB
--caches --l1i_size=32kB --l1i_assoc=4 --l1d_size=32kB --l1d_assoc=4
--cacheline_size=64 --cpu-clock=1.6GHz -n 1 --cpu-type=TimingSimpleCPU
```

Fig. 8. Another sample command run with TimingSimpleCPU set as the processor.

Benchmark	CPU Type	Associativity	Time(s)	Miss Rate
MergeSort	TimingSimpleCPU	1	3.270675	0.005462
MergeSort	**TimingSimpleCPU**	**2**	**3.166558**	**0.003008**
MergeSort	TimingSimpleCPU	4	3.169825	0.003077
MergeSort	TimingSimpleCPU	8	3.166745	0.003006
QSort	TimingSimpleCPU	1	3.007224	0.002432
QSort	**TimingSimpleCPU**	**2**	**2.990074**	**0.00143**
QSort	TimingSimpleCPU	4	2.990534	0.001443
QSort	TimingSimpleCPU	8	2.99106	0.001457

Fig. 9. The metrics for the static associativity trials.

completion of the full set of benchmarks, simulation data was retrieved from the stats files and compared between each benchmark run. In MergeSort, an increase in associativity generally led to a decrease in miss rate. The associativity that broke the trend was 4-way. This associativity had worse performance than 2-way and 8-way, but better than direct-mapped. 2-way set associative resulted in the lowest miss rate. For the QSort benchmark, 2-way set associative came out on top for miss rate, while 4-way performed slightly worse, and 8-way performed slightly worse than 4-way. Once again, direct mapped was the worst performer for miss rate. The data shows that, at least for these benchmarks, 2-way set associative is the best performer. This implies that increased associativity does not guarantee a lower miss rate, and that it is worthwhile to find the associativity that creates a relatively low miss rate with lower overhead than a higher associativity. Re-running the benchmarks without an instruction limit did change the outcome, likely due to there being many more capacity misses when the simulation is left to run to the end. For MergeSort, 8-way set associative became slightly better than 2-way set associative, but only by a 0.04% lower miss rate. For QSort, 2-way set associative maintained the lowest miss rate, and the ranking of the associativities was the same as when the instruction limit was in place. (This next part refers to the results of the no instruction limit simulations using the timing CPU). Although the miss rate of the 2-way set associative cache was slightly higher than that of the 8-way set associative cache (by 0.06%), the execution time was better with the 2-way set associative cache (by 0.005%). Misses do incur a large penalty, but the overhead of an 8-way set associative cache may have slowed down that system enough to negate the benefits of fewer misses. The static simulation of cache configurability is still incomplete as a variety of other benchmarks would need to be run to establish baselines of comparison. Additionally, given our intention to change configurability primitively by invalidating the old cache and essentially instantiating a new cache with the desired associativity, the system would incur a broad range of misses as penalty for the switch. Those results would have been useful to discuss the tradeoff with such a simplistic model. Results of the trials described are tabled below.

VI. Future Work

Future work will be to meticulously dissect some of the message passing functions at work within Gem5. With sufficient experimental background, we are confident that a configurable cache could successfully be implemented within the simulator. The benefits of such implementation, as previously stated, would be significant. There are likely still other benefits which a HDL version of a configurable cache might provide, including realistic power and energy consumption metrics as Vivado has a robust tool. Future work would be to find or develop a RISCV core completely realized in Vivado, and implement the configurable cache.

VII. Conclusion

From the limited results, we conclude that a 2 way set associative cache performs best for these particular applications. Provided more robust variety of benchmarks, we would expect to see certain classes of applications and programs exhibit preference for a broader range of associativities. This seems logical, as a Direct Mapped cache notoriously suffers from conflict misses and a 2 way set associative cache alleviates the contention without sacrificing much in the way of increased hardware overhead. As power estimation tools such as Mcpat largely rely on have full system mode up and running, which at this time is not supported in Gem5 for RISCV, we do not have the other half of the configurable cache performance analysis. We would expect to see an increase in power consumption as the hardware grows to due the increase in comparisons needed for increasing associativities. Although RISCV support is limited within Gem5, a configurable cache may not be far off. Given more time to analyze the intricacies of the simulator and shore up deficiencies in our understanding of the interface, dynamic configurability like that featured in previous works could be realized and rigorously tested.

References

[1] Zhang, C. (2003). A highly configurable cache architecture for embedded systems - IEEE Conference Publication. [online] Ieeexplore.ieee.org. Available at: https://ieeexplore.ieee.org/document/1206995 [Accessed 10 Feb. 2019].

[2] Rocket core overview, lowRISC. [Online]. Available: https://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core/. [Accessed: 10-Feb-2019].

[3] M. Nltner-Augustin, RISC-V Architecture and Interfaces The RocketChip, ADVANCED SEMINAR COMPUTER ENGINEERING, UNIVERSITY OF HEIDELBERG WT16/17. [Online]. Available: https://ra.ziti.uni-heidelberg.de/cag/images/seminars/ws16/2016-noeltner-augustin-report.pdf. [Accessed: 10-Feb-2019].

[4] A. Badicioiu, Customizing RISC-V core using open source tools, riscv.org. [Online]. Available: https://content.riscv.org/wp-content/uploads/2018/05/4-Designing-a-custom-RISC-V-core-using-Chisel-Alex-Badicioiu-NXP.pdf. [Accessed: 10-Feb-2019].

[5] C. Yarp, CS250 Discussion 2 RISC-V, Rocket, and RoCC, EECS Instructional and Electronics Support. [Online]. Available: http://www-inst.eecs.berkeley.edu/ cs250/sp16/disc/Disc02.pdf. [Accessed: 10-Feb-2019].

[6] A. Gordon-Ross, A Self-Tuning Configurable Cache, An introduction to biometric recognition - IEEE Journals & Magazine. [Online]. Available: https://ieeexplore.ieee.org/document/4261178. [Accessed: 10-Feb-2019].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.