Final Project Report

Sam Ferguson

ELE 475

**Abstract**

Here we examine the effects of tiling, cache size and associativity on code performance. With several non-trivial test programs, we simulate hundreds of caching and tiling configurations to discover, refine and discuss trends in performance (specifically cycles/instruction) due to these variables. Although not every combination leads to interesting results, we discuss a set that illustrate the following points.

* As cache sizes go up, the overall performance of code follows a curvature, reaching a peak performance then falling as cache sizes become excessive.
* Even moderate associativity quickly improves performance. However, excessively high or low associativity can quickly stunt execution progress in the caches. We suspect there is a mathematically optimal associativity for any system.
* Excessive size differentials in either size or associativity screw the statistics toward the smaller, lower associative cache
* There is insignificant change in performance with increased tiling regardless of other variables. Overall however, it trends downwards in efficiency for our applications.

In addition, we briefly discuss a probable security vulnerability in both OpenPiton’s, and many private system’s caches: the flush-reload side channel attack. However unsuccessfully implemented here, this attack is also the key to recently famous such as Spectre and Meltdown.

**Goal**

The overall goal of this project is to explore the performance of different caching hierarchies within the OpenPiton open-source multicore infrastructure and additionally attempt to expose some vulnerabilities inherent to the overall architecture. We determine the effect of increasing and decreasing the cache sizes at all levels except L1 (since altering the L1D and L1I sizing can cause conflict with the TLB if set low enough) [3]. Additionally, we wanted to make modifications with the overall scale of the tiling of the core to see if it had any affect. Then, since these factors involved a significant amount of data collection, scripting, and simulation We also wanted to create a similarly interesting custom test. Some of the most notable cyber attacks of the past year involve the exploitation of side channels in computer caches. Both Meltdown and Spectre subvert normal security channels through invalid caching of temporary, out of order, or misspredicted datum [5][6].

**Results**

The process of this project is straightforward. Choose some example code to compile down and simulate. (ideally code which covers a large portion of overall program behaviors). Then, simply, experiment repeatedly. We create a number of testing shell scripts to aid and automate the data collection process.

The following chart shows just one of the trends in simulating three separate executables under the variable cache size, associativity, and tiling. This one demonstrates the first point of our abstract. Although showing a triangle rather than curvature with increasing cache size, the trend is clear. As cache sizes increase, they reach a peak and then experience diminishing returns. Associativity here is held a 4 for both the L1.5 and L2 caches.

Performance in relation to Cache Sizing

|  |  |  |
| --- | --- | --- |
| L1.5 Size Dim = 4096  L2 Size Dim = 32768  Factorial:  CPU Time: 1.650 seconds; Data structure size: 1.3Mb  Cyc= 58408, Sec= 1.650, C/S=35398.8  ExecCyc= 52544.5, Sec= 1.650, EC/S=31845.2  Print:  CPU Time: 1.110 seconds; Data structure size: 1.3Mb  Cyc= 38528, Sec= 1.110, C/S=34709.9  ExecCyc= 32664.5, Sec= 1.110, EC/S=29427.5  Quicksort:  CPU Time: 4.160 seconds; Data structure size: 1.3Mb  Cyc= 58957, Sec= 4.160, C/S=14172.4  ExecCyc= 53093.5, Sec= 4.160, EC/S=12762.9 | L1.5 Size Dim = 8192  L2 Size Dim = 32768  Factorial:  CPU Time: 1.570 seconds; Data structure size: 1.3Mb  Cyc= 58408, Sec= 1.570, C/S=37202.5  ExecCyc= 52544.5, Sec= 1.570, EC/S=33467.8  Print:  CPU Time: 1.100 seconds; Data structure size: 1.3Mb  Cyc= 38528, Sec= 1.100, C/S=35025.5  ExecCyc= 32664.5, Sec= 1.100, EC/S=29695.0  Quicksort:  CPU Time: 3.820 seconds; Data structure size: 1.3Mb  Cyc= 58957, Sec= 3.820, C/S=15433.8  ExecCyc= 53093.5, Sec= 3.820, EC/S=13898.8 | L1.5 Size Dim = 16384  L2 Size Dim = 32768  Factorial:  CPU Time: 1.590 seconds; Data structure size: 1.3Mb  Cyc= 58408, Sec= 1.590, C/S=36734.6  ExecCyc= 52544.5, Sec= 1.590, EC/S=33046.9  Print:  CPU Time: 1.020 seconds; Data structure size: 1.3Mb  Cyc= 38528, Sec= 1.020, C/S=37772.5  ExecCyc= 32664.5, Sec= 1.020, EC/S=32024.0  Quicksort:  CPU Time: 3.810 seconds; Data structure size: 1.3Mb  Cyc= 58957, Sec= 3.810, C/S=15474.3  ExecCyc= 53093.5, Sec= 3.810, EC/S=13935.3 |
| L1.5 Size Dim = 4096  L2 Size Dim = 65536  Factorial:  CPU Time: 1.740 seconds; Data structure size: 1.3Mb  Cyc= 58302, Sec= 1.740, C/S=33506.9  ExecCyc= 52438.5, Sec= 1.740, EC/S=30137.1  Print:  CPU Time: 1.030 seconds; Data structure size: 1.3Mb  Cyc= 38528, Sec= 1.030, C/S=37405.8  ExecCyc= 32664.5, Sec= 1.030, EC/S=31713.1  Quicksort:  CPU Time: 3.790 seconds; Data structure size: 1.3Mb  Cyc= 58897, Sec= 3.790, C/S=15540.1  ExecCyc= 53033.5, Sec= 3.790, EC/S=13993.0 | L1.5 Size Dim = 8192  L2 Size Dim = 65536  Factorial:  CPU Time: 1.610 seconds; Data structure size: 1.3Mb  Cyc= 58302, Sec= 1.610, C/S=36212.4  ExecCyc= 52438.5, Sec= 1.610, EC/S=32570.5  Print:  CPU Time: 1.030 seconds; Data structure size: 1.3Mb  Cyc= 38528, Sec= 1.030, C/S=37405.8  ExecCyc= 32664.5, Sec= 1.030, EC/S=31713.1  Quicksort:  CPU Time: 3.930 seconds; Data structure size: 1.3Mb  Cyc= 58897, Sec= 3.930, C/S=14986.5  ExecCyc= 53033.5, Sec= 3.930, EC/S=13494.5 | L1.5 Size Dim = 16384  L2 Size Dim = 65536  Factorial:  CPU Time: 1.780 seconds; Data structure size: 1.3Mb  Cyc= 58302, Sec= 1.780, C/S=32753.9  ExecCyc= 52438.5, Sec= 1.780, EC/S=29459.8  Print:  CPU Time: 1.130 seconds; Data structure size: 1.3Mb  Cyc= 38528, Sec= 1.130, C/S=34095.6  ExecCyc= 32664.5, Sec= 1.130, EC/S=28906.6  Quicksort:  CPU Time: 3.800 seconds; Data structure size: 1.3Mb  Cyc= 58897, Sec= 3.800, C/S=15499.2  ExecCyc= 53033.5, Sec= 3.800, EC/S=13956.2 |

Additionally, from the following, we see the similar behavior by instead effecting the associativity of the caches rather than their sizes. The caches sizes are held at 8192 bytes and 6556 bytes for the L1.5 and L2 caches respectfully.

Performance in relation Cache Associativity

|  |  |  |  |
| --- | --- | --- | --- |
| L1.5 Ass Dim = 2  L2 Ass Dim = 2  Factorial:  CPU Time: 1.610 seconds; Data structure size: 1.3Mb  Cyc= 58362, Sec= 1.610, C/S=36249.7  ExecCyc= 52498.5, Sec= 1.610, EC/S=32607.8  Print:  CPU Time: 1.030 seconds; Data structure size: 1.3Mb  Cyc= 38454, Sec= 1.030, C/S=37334.0  ExecCyc= 32590.5, Sec= 1.030, EC/S=31641.3  Quicksort:  CPU Time: 3.710 seconds; Data structure size: 1.3Mb  Cyc= 57611, Sec= 3.710, C/S=15528.6  ExecCyc= 51747.5, Sec= 3.710, EC/S=13948.1 | L1.5 Ass Dim = 2  L2 Ass Dim = 4  Factorial:  CPU Time: 1.610 seconds; Data structure size: 1.3Mb  Cyc= 58302, Sec= 1.610, C/S=36212.4  ExecCyc= 52438.5, Sec= 1.610, EC/S=32570.5  Print:  CPU Time: 1.040 seconds; Data structure size: 1.3Mb  Cyc= 38528, Sec= 1.040, C/S=37046.2  ExecCyc= 32664.5, Sec= 1.040, EC/S=31408.2  Quicksort:  CPU Time: 3.590 seconds; Data structure size: 1.3Mb  Cyc= 57217, Sec= 3.590, C/S=15937.9  ExecCyc= 51353.5, Sec= 3.590, EC/S=14304.6 | L1.5 Ass Dim = 2  L2 Ass Dim = 16  Factorial:  CPU Time: 1.600 seconds; Data structure size: 1.3Mb  Cyc= 58508, Sec= 1.600, C/S=36567.5  ExecCyc= 52644.5, Sec= 1.600, EC/S=32902.8  Print:  CPU Time: 1.060 seconds; Data structure size: 1.3Mb  Cyc= 38588, Sec= 1.060, C/S=36403.8  ExecCyc= 32724.5, Sec= 1.060, EC/S=30872.2  Quicksort:  CPU Time: 3.680 seconds; Data structure size: 1.3Mb  Cyc= 57151, Sec= 3.680, C/S=15530.2  ExecCyc= 51287.5, Sec= 3.680, EC/S=13936.8 |  |
| L1.5 Ass Dim = 4  L2 Ass Dim = 2  Factorial:  CPU Time: 1.600 seconds; Data structure size: 1.3Mb  Cyc= 58362, Sec= 1.600, C/S=36476.2  ExecCyc= 52498.5, Sec= 1.600, EC/S=32811.6  Print:  CPU Time: 1.040 seconds; Data structure size: 1.3Mb  Cyc= 38454, Sec= 1.040, C/S=36975.0  ExecCyc= 32590.5, Sec= 1.040, EC/S=31337.0  Quicksort:  CPU Time: 3.930 seconds; Data structure size: 1.3Mb  Cyc= 59323, Sec= 3.930, C/S=15094.9  ExecCyc= 53459.5, Sec= 3.930, EC/S=13602.9 | L1.5 Ass Dim = 4  L2 Ass Dim = 4  Factorial:  CPU Time: 1.590 seconds; Data structure size: 1.3Mb  Cyc= 58302, Sec= 1.590, C/S=36667.9  ExecCyc= 52438.5, Sec= 1.590, EC/S=32980.2  Print:  CPU Time: 1.050 seconds; Data structure size: 1.3Mb  Cyc= 38528, Sec= 1.050, C/S=36693.3  ExecCyc= 32664.5, Sec= 1.050, EC/S=31109.0  Quicksort:  CPU Time: 3.900 seconds; Data structure size: 1.3Mb  Cyc= 58897, Sec= 3.900, C/S=15101.8  ExecCyc= 53033.5, Sec= 3.900, EC/S=13598.3 | L1.5 Ass Dim = 4  L2 Ass Dim = 16  Factorial:  CPU Time: 1.630 seconds; Data structure size: 1.3Mb  Cyc= 58508, Sec= 1.630, C/S=35894.5  ExecCyc= 52644.5, Sec= 1.630, EC/S=32297.2  Print:  CPU Time: 1.040 seconds; Data structure size: 1.3Mb  Cyc= 38588, Sec= 1.040, C/S=37103.8  ExecCyc= 32724.5, Sec= 1.040, EC/S=31465.9  Quicksort:  CPU Time: 3.890 seconds; Data structure size: 1.3Mb  Cyc= 58797, Sec= 3.890, C/S=15114.9  ExecCyc= 52933.5, Sec= 3.890, EC/S=13607.6 |

We could fill 10 pages alone just with these proofs. In the process of revealing these trends 100s of testing loops have been run, analyzed and categorized. However, these two main trends are key. We believe it’s from these two parabular trends we can explain essentially every other data trend brought up in this analysis.

**Discussion**

So why do we observe these trends in the data? Its not difficult to conceptually link these trends with caching fundamentals from class.

Performance goes up with cache size. As there is more space for data, while at the cost of resources, executing code gets quicker as the cache provides access to data that otherwise would’ve been evicted. In theory an infinite sized cache with a high enough associativity will never experience conflict or capacity misses. In such a case, only compulsory misses due to the first references to blocks of memory will slow down execution.

As Caches become more and more associative, they continually lower the odds of having conflict misses. The other hand is true as well, and a direct or nearly direct mapped cache will experience significant conflict missage. However, realistically there is a tradeoff between preventing conflict misses and the need to search the breadth of each way in the cache. As the associativity become excessive, therefore we experience a increase in overhead for using the cache at all.

Additionally, given these trends and the interdependence of the L2 and L1.5 caches, we can explain the skew inherent to corner cases involving low-high sizing and associativity between the caches. Since all data is ultimately held in all caches, any one cache that has significantly narrower capacity to others constitutes a blockage in code performance. Should a higher-level cache be to small, the lower level cache in the hierarchy it wastes resources by being to large or associative while the higher cache doesn’t have the need or capacity to utilize all of its breadth. Alternatively, it a higher cache is far to large or associative, it’s difficult for the required data to filter upwards.

The lack of impact around tile size almost certainly has to do with the underutilization of multiple cores. Although the program examples chosen are not simple assembly or other benchmarks, they also are generally single threaded. In fact, given this, one could expect performance to decrease slightly as the tiling factor increases due to the increased weight of tile networking and cache coherence protocols keeping all cores on the same page.

Unfortunately for the discussion of flush-reload we ran into several consistencies under the cross compiler, and it ultimately never reach a testing stage. We suspect that the caching architecture of OpenPiton is vulnerable, at least on the hardware level, to malicious caching which can then be exposed by side-channel methods. Two likely candidates include the use of incorrect prediction of execution, much like Spectre [5], or even some insidious use of momentary cache incoherencies in the multicore system.

**Comments and Future Steps**

Given additional time and resources we would’ve loved to move out of simulation and actually run similar tests on FPGA implementations of OpenPiton. Running baseline measures on a true silicon implementation would also be a great control. Additionally, We think given the predictable trends of much of my data it would be intriguing to try and mathematically model the parameters of each programs cache usage in accordance with their best preforming cache size and associativity.

Additionally, we regret not getting my custom code sample implementing flush-reload working. We’re sure that with enough time We could’ve found equivalent commands or assembly for key parts of it that were uncompilable in this case. Should we have gotten working, it would also be valuable to look into potential changes to the cache RTL code in order to try and flusher this and similar side channel attacks.

**References**

[1] Sun Microsystem Inc, Santa Clara, CA, 2008, “OpenSPARC T1 Microarchitecture Specification”

[2] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, David Wentzlaff,

Princeton University, Atlanta, Georgia 2016, “OpenPiton: An Open Source Manycore Research Framework”

[3] Wentzlaff Parallel Research Group, Princeton University, “OpenPiton Simulation Manual”, 2017

[4] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Dkaniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. “Meltdown.”

[5] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom,, 2019, “Spectre Attacks: Exploiting Speculative Execution”

### [6] Eugnis/spectre-attack. (2018). GitHub. Retrieved 17 December 2018, from <https://github.com/Eugnis/spectre-attack/blob/master/Source.c>

[7] Spectre example code. (2018). Gist. Retrieved 17 December 2018, from https://gist.github.com/ErikAugust/72