PENN STATE UNIVERSITY, DEPARTMENT OF ELECTRICAL ENGINEERING & COMPUTER SCIENCE

CMPEN 431 - Final Project

Quang Nguyen & Richard Heidorn

December 15, 2015

Best IPC and Execution Time for **mcf** and **milc**:

	IPC	Execution Time (μs)
Base mcf	0.3186	784.682988073
Base milc	0.4234	590.45819556
Best mcf	2.5297	192.710598095
Best milc	1.5165	263.765248928

Best execution time **mcf** issue width and data path type:
Dynamic, 8-wide

Best execution time **milc** issue width and data path type:
Dynamic, 4-wide

Overall Best Execution Time Geometric Means (GM):

	Geometric Mean (μs)
Best Integer GM	263.024742993
Best Floating Point GM	249.152599564

Best execution time GM Integer issue width and data path type:
Dynamic, 4-wide

Best execution time GM Floating Point issue width and data path type:
Dynamic, 4-wide

CONTENTS

1 INTRODUCTION

Designing a computer architecture is far from an exact science. A computer's performance is determined by many variables, ranging from the design and efficiency of the processor to the operating system and programs that run on top of the hardware. To name only a few, a computer's performance is determined by the functional units its processor contains, the length of its busses, the size and number of transistors, the organization of its caches, the speed and reliability of its hard drive, and ultimately the efficiency of its software.

By and large, computer architectures have become exponentially faster, smaller, more efficient, more durable, and more powerful. However, the number of factors that predict the performance of programs with the hardware is so great, it is nearly impossible to predict the most efficient computer designs for any given purpose. After all, the design that is optimal for one application might be sluggish for another. Even the same program, given a different set of data to compute, could perform better on different architectures.

For this project, we've investigated many different computer architectures using the Simple Scalar architecture simulator and how they affect the overall geometric means for four integer benchmarks and two floating point benchmarks provided by the SPEC performance benchmark package. The four integer benchmarks - bzip2, hmmer, mcf, sjeng - and two floating point benchmarks - milc, equake - were used to evaluate the overall performances of each tested architecture, and to evaluate how certain changes to the architectures would impact the programs' performance.

1.1 Testing Methodology

Simple Scalar provides a number of parameters that can be modified to emulate any modern computer architecture design. While the number and purpose of the parameters is easy to understand, the combinations of different parameters is incredibly large and difficult to comprehend. Thankfully, the project has been defined to limit the number of architectures that could be tested, but the sum total of all combinations is far greater than can be reasonably tested in a few weeks' time. Therefore, a brute force approach to determining the best design is impractical at best.

However, due to the number of variables outside of the architecture - the algorithms, access patterns, and behavior of the programs tested - it is not simply good enough to make educated guesses based on our understanding of processor evolution. Instead, a combination approach is required. Scientific reasoning is necessary to isolate the parameters and restrict the range of values which could benefit the performance of a given design. Once the parameters are defined and the range of reasonable values determined, a series of tests need to be run to experimentally verify our predictions and also to determine the best design decisions for a given architecture.

Our testing methods combine scientific analysis and experimental verification, both to determine which variables to test and which tests to run for all static and dynamic issue machines. Following each suite of tests, we identified and analyzed those designs which worked best for each machine and issue width. We've run dozens of test suites which isolated different components of the processor's functional units, branch prediction, instruction issuing, cache design, and the TLB. Each of these tests suites contained dozens to hundreds of individual configurations, all of which provided experimental information that were used to advance our designs to find the best performing systems. These tests were not comprehensive in any way, but we believe that we've isolated strong designs as a result of our methodology.

1.2 Calculating the Geometric Mean

The geometric means were calculated using the program execution times as its input. Since performance is typically defined as the inverse of the execution time, a smaller geometric mean translates to greater overall performance. The geometric mean is defined as:

$$\begin{aligned}\text{Geometric Mean} &= \sqrt[n]{\prod_{i=1}^{n} \text{Execution Time}} \\ &= \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Instruction Count}_i \times \text{Clock Cycle}_i}{\text{Instructions Per Cycle}_i}}\end{aligned}$$

2 Experiments

The majority of experiments were carried out by changing a few related parameters and determining how changing the variables affected the geometric means for all machine issue widths. By default, most of the tests were performed for both static and dynamic machines, with issue widths 1, 2 and 4 for static, and 2, 4, and 8 for dynamic. Near the end of our experimentation, we found that dynamic far outperformed static. To allow us to continue to test with higher accuracy, the static tests were eventually dropped in favor of the dynamic tests.

The order of our initial tests were mostly arbitrary and experimental, but we've found that there is a logical sequence for how testing could be performed. After reviewing our process, the ideal experimentation order would have been to isolate the independent variables that allow us to optimize the performance of all issue widths, to set these variables to their best performing values, and then to test the remaining inter-dependent variables. We present the order of experimentation as we've proceeded, but we focus on the experiments that yielded the most useful results.

2.1 Identifying Independent Variables

The first round of experimentation involved testing different, interrelated parameters, and determining how well they affected the geometric means independent of the other Simple Scalar parameters. For our tests, we began by first testing the ideal number of ALUs and Multipliers, Fetch Speed, Memory Width, and the Register Update Unit & Load / Store Queue. The results for the ALU's / Multipliers and the Fetch Speed were mixed at best, and were ultimately inconclusive to test so early on. However, the results for the Memory Width and the Register Update Unit showed positive trends that we could use to build on.

2.1.1 Best of the Register Update Unit & Load / Store Queue

For the register update unit (RUU) and load / store queue (LSQ), we found that the best performing designs for dynamic issue machines of 2-wide, 4-wide and 8-wide performed best when the number of RUUs or LSQs were maxed out for each issue width:

Issue Width	RUU	LSQ
2-wide	16	8
4-wide	32	16
8-wide	64	32

These results stood across the board, which allowed us to progress with the remainder of our testing by optimizing the performance of all issue widths.

2.1.2 Best of the Memory Width

We tested the combination of each data path and issue width against memory widths of 8 and 16 bytes. Our results were not surprising, as each machine with a memory width of 16 bytes outperformed their 8-byte counterparts. For the remainder of our experiments, we set the optimal memory width to 16 bytes.

2.2 Branch Prediction: Finding an Ideal Method

The branch predictors were tested heavily following the initial evaluation of the RUU, LSQ and memory width. We made the assumption that not-taken and taken will not perform as well as bimodal, two-level or the combination branch predictors, and this was verified after the original tests against the three best-performing branch predictors.

2.2.1 Assumptions about the L1 Cache

For the three tested branch predictors, we analyzed not only the performance of the different branch prediction methods, but we also performed extensive research into the L1 data and instruction caches and the unified L2 cache. In order for the branch prediction tests to be carried out in a timely manner, we made a decision to match the L1 instruction cache to the L1 data cache, while keeping them separated. While this may cause less-than-ideal combinations of both the data and instruction cache, the number of combinations was too great to reasonably analyze. Further, it is common to see instruction and data caches with matching specifications, which aided our decision to simplify the testing process.

While we realize that the more variables being tested, the more complex that the result analysis becomes, we were comfortable altering both the branch prediction type and altering the L1 and L2 caches. The same exact combinations of cache values were tested across all three branch predictors, which gave us a large number of results to compare the performance of each type of branch predictor. Ultimately, we saw that the combination method won out - not comfortably, but consistently enough that allowed us to expand our testing of the combination branch predictor. Of the hundreds of tests we ran, Figures 2.1 through 2.2 show the number of tests ran that returned a geometric mean at or below 1000 for both static and dynamic issue width systems. These figures also include expanded evaluation of the combination method, which is why the number of tests run for combination vastly exceeds bimodal and 2-level.

2.3 TLB Investigation

We expanded our search of independent variables by exploring the affect of various sizes of the TLB on the performance of the different machines. The results were not very surprising,

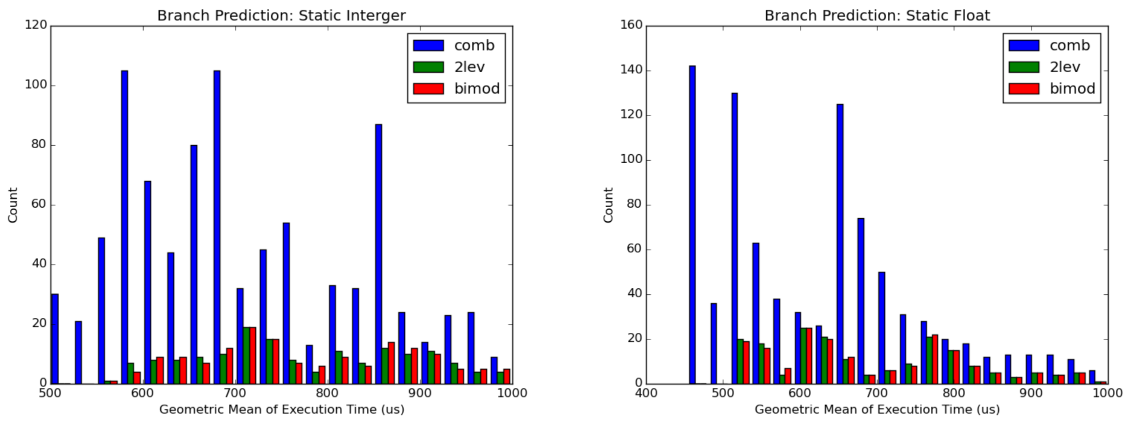

Figure 2.1: A subset of the total tests run for three branch predictor types (bimodal, 2-level, combination) for static issue machines.

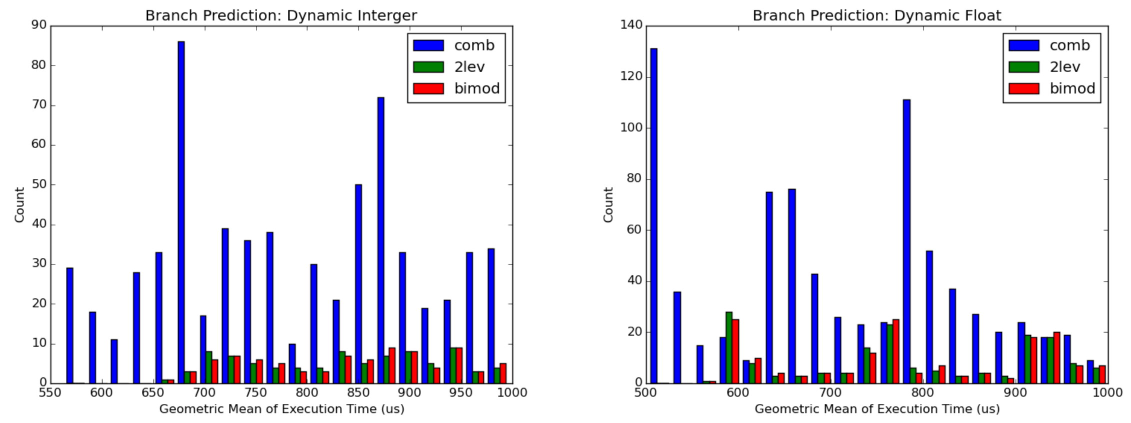

Figure 2.2: A subset of the total tests run for three branch predictor types (bimodal, 2-level, combination) for dynamic issue machines.

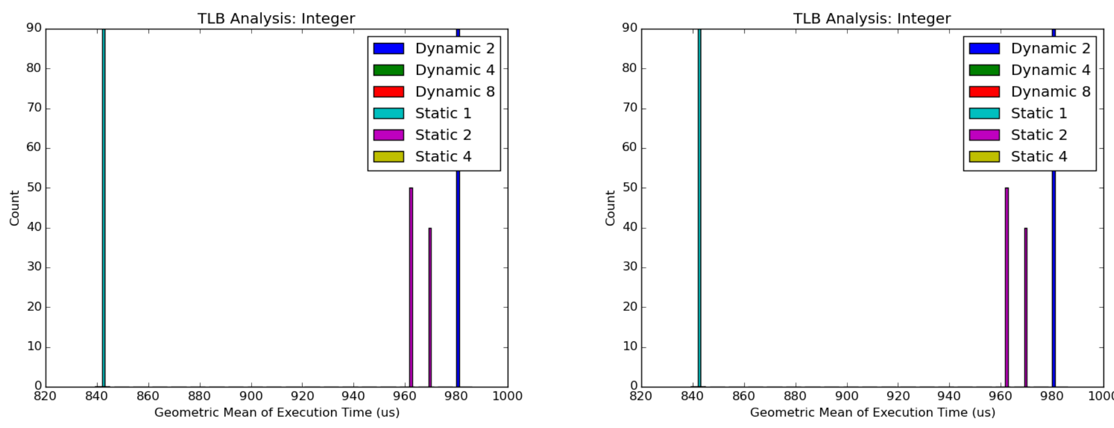

Figure 2.3: Evaluation of the TLB

as the use of the TLB is minimal compared to the time spent in the cache or performing operations. Ultimately, the effects of changing the different sizes of the associativity and the number of sets of the TLB had little impact on the final outcome of the geometric means for each of the machines. We've presented our findings for each machine and issue width in Figure 2.3, for those machines / widths that managed a geometric mean below 1000.

2.4 Revisiting the Baseline with Optimal Values

Following the TLB, we performed validation experiments, by re-running the baseline configurations with our new "optimal" assumptions and constraints that were discovered from the earlier tests. We started to see how the dynamic data paths were catching up or exceeding the static data paths. Next, we divided the work up to explore how the branch predictor fared with varying RAS and BTB values, and simultaneously performed a complex and time-consuming, but automated, series of tests on the L1 cache and L2 cache values.

2.5 Expanding on Branch Prediction, the RAS and BTB Values

For the branch prediction additional values, we explored an increased Return Address Stack and Branch Target Buffer with varying values on the branch predictor. Overall, we found that a RAS of a power of two, usually the value 8, performed ideally. The BTB was ideally set at 1024 for both floating point and integer geometric means. The associativity was interchangeable, but our best values came from 2-way associativity. Figures 2.4 show the wide variety in the geometric means for the given tests.

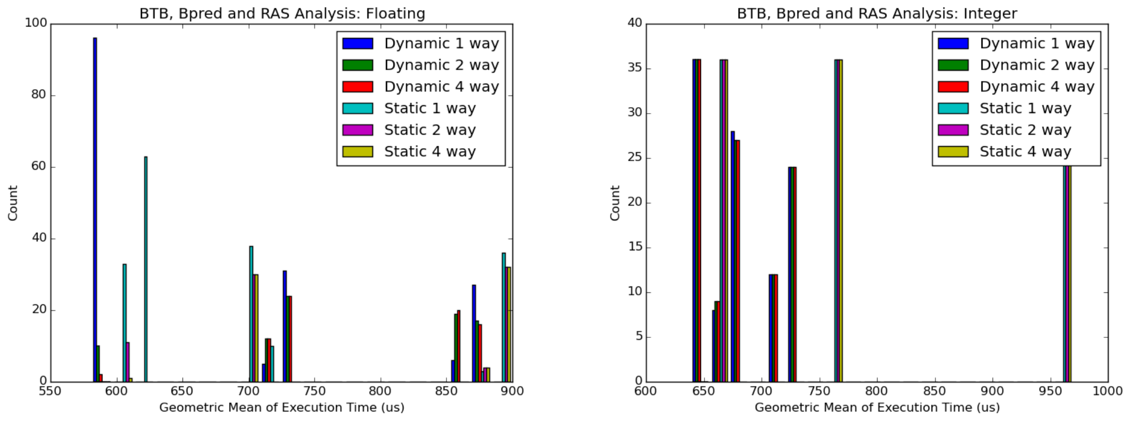

Figure 2.4: A subset of the geometric means calculated by testing the Branch Prediction BTB and RAS for all static and dynamic issue widths. **NOTE:** The term "1-way", "2-way" and "4-way" in the images refers to the BTB associativity.

2.6 Exploring the Cache - L1 and L2

The cache tests were, by far, the most extensive performed. We've tested all permutations of L1 cache associativity, set sizes, and block sizes against variable instruction fetch queue sizes, following the restrictions laid out in the project instructions. For the L1 cache tests, the instruction and data caches were set to identical cache sizes for simplicity - an assumption that pervades to the final tests. For this first set of experiments on the L1 cache, we examined the effects on all static and dynamic data path / issue width combinations, and the L2 cache was set to an arbitrarily large, constant cache that would permit experimentation on the L1 cache.

Following the above, we identified (falsely, at the time) that dynamic 2-issue was performing best, which allowed us to streamline our testing for only one of the data path / issue width pairs. Using the dynamic 2-wide data path, we carried out further experimentation on combinations of the ideal L1 cache values with combinations of the L2 unified cache for all pairs of association, block sizes and set sizes that fit within the project constraints. While we were incorrect in assuming that dynamic 2-wide would perform the best, we did find that our results regarding the L1 and L2 cache combinations were valid across all data path and issue width combinations.

Among our results, we found that a smaller associativity performed better, with larger number of sets and block sizes. The instruction fetch queue size was best set at either 4 or 8, so our L1 cache block sizes were either 32 or 64. The associativity was almost always 1 or 2 in our best-performing machines, and the number of sets were mostly maxed out.

Following all of the above tests, we then took our findings and, for good measure, reran the tests against all static and dynamic issue machines. Fortunately, this allowed us to find that the dynamic 4-wide and 8-wide overtook the performance of all other machines by a large

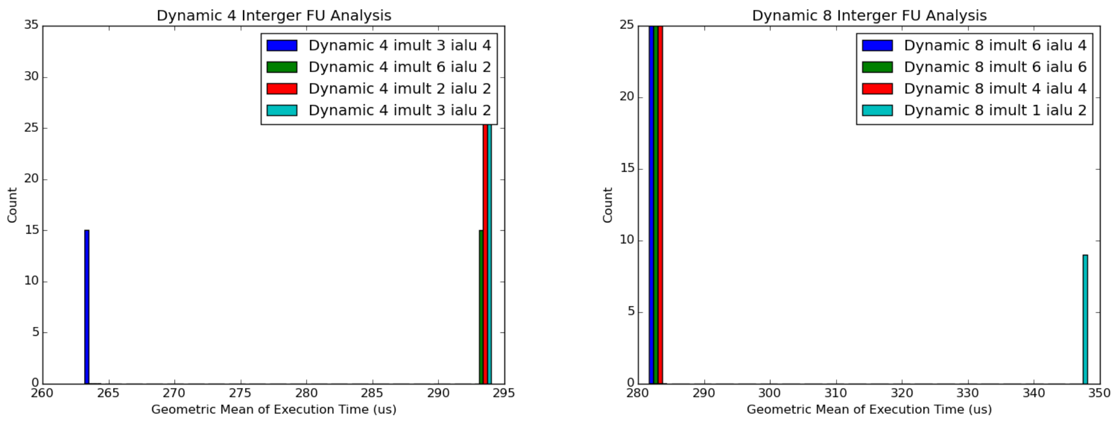

Figure 2.5: Best performing functional units for dynamic 4-wide and 8-wide integer benchmarks.

margin. For our final set of tests, we ran our tests against these two machines, and ran final "sanity check" tests against the other machines to confirm our results.

2.7 Return to the Functional Units

We mentioned earlier how we tested the various combinations of functional units, with inconclusive results. We realized after expanding on the RUU, LSQ, and the caches that the functional units would not be fully optimized unless they had the hardware in place to support them for each issue width. Now with the remaining values set to optimal ranges, we could return to the functional units and determine our best values. Thankfully, we found that certain patterns of combinations of the integer and floating point ALUs and Multipliers were better performing than the others, but with no hard-and-fast rules about which these values needed to be - just as long as the sum total of the Integer ALU and Multiplier or the sum total of the Floating Point ALU and Multiplier were above certain values, and that we had a minimum of ALUs that depended on the size of the issue width. Figures 2.5 and 2.6 show the best performing geometric means for the dynamic 4-wide and 8-wide machines that performed best.

3 Conclusion

When we started out, we decided that we wanted to approach the testing procedure both methodically and holistically. That is, we wanted to predict which values would perform best but then test our assumptions thoroughly. We began our testing by isolating the variables that showed predictable trends in the performance, such as setting the Register Update Unit, Memory Width, and Load / Store Queue for the various data paths and issue

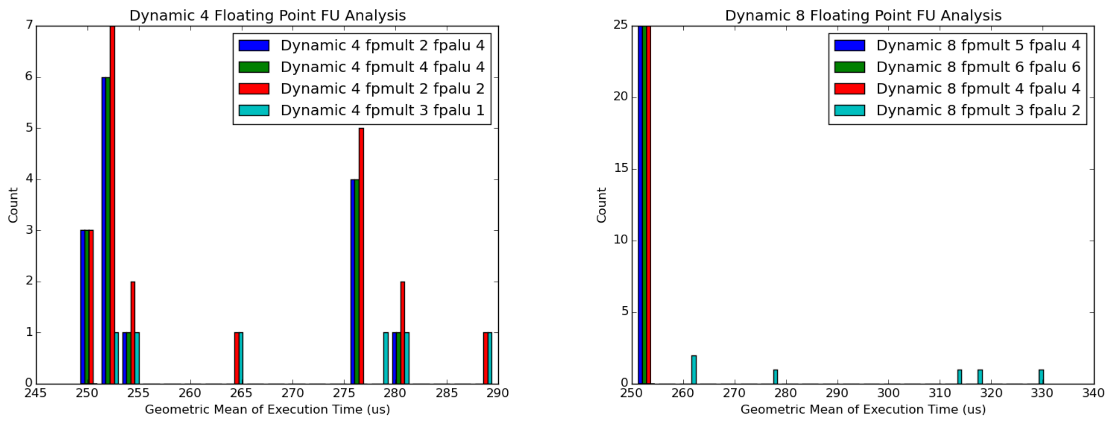

Figure 2.6: Best performing functional units for dynamic 4-wide and 8-wide floating point benchmarks.

widths. Following this, we were able to start expanding our testing into the branch prediction methods to identify the combination brand prediction as our best performing method. Simultaneously, we were able to analyze the Translation Look-aside Buffer and determine that changes to the TLB offered minimal improvements on the overall performance.

After our initial tests, we revisited the baseline and set all of our assumptions for the optimal values. Then, we explored further values of the branch predictors: the Return Address Stack and Branch Target Buffer. This allowed us to optimize our branch predictors. At the same time, we explored the different levels of cache and performed a very thorough investigation into the ideal L1 and L2 caches for our Integer and Floating Point benchmarks. Lastly, with all of our optimized values in place, we revisited the functional units and found the best combinations of functional unit pairs that produced our best systems.

Our best machines turned out to be dynamic 4-wide and 8-wide machines, with performances that exceed the baseline by roughly 3 times the original geometric means. While we expected that the best machine would likely be dynamic 4 or 8-wide, it is nice to see that our expectations are backed with tests and hundreds of benchmark comparisons. It is worth noting that benchmarks are not an absolute indication of the all-around performance of a given machine, but these benchmarks offer an insight into machine designs that are optimal for these given tasks. With thorough investigation and numerous tests, we've found ideal systems that match our expectations and demonstrate the efficacy of dynamic issue machines.

Merry Christmas! Please give us an A.