MICROPROCESSOR OPTIMIZATION FOR GO BENCHMARK

LAB 3

Yousuke Z. Matsui, 1st Lt, USAF
David R. Crow, 2nd Lt, USAF
Nick C. Echeverry, 2nd Lt, USAF
Joseph A. Vagedes, 2nd Lt, USAF
Dan F. Koranek, Civ, USAF

DEPARTMENT OF THE AIR FORCE AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

Abstract

Optimizing a system's configuration proves to be a difficult balancing act of maximizing performance and simultaneously meeting budgetary constraints. We present a method to eliminate suboptimal configurations of a processor for three different area constraints by analyzing the simulated cycles per instruction (CPI) to measure performance during execution of the Go benchmark. We identify the optimal configuration under each area constraint, and we explore the potential for further speedup by altering the cache configuration. We show that larger processor sizes are the best predictor for improved performance with this benchmark, but these results are application-dependent. Additionally, we show that cache configuration for the Go benchmark only provides trivial improvement. Still, we predict that cache configuration will be made more significant by larger data sets and different benchmarks.

Table of Contents

Introduction

Because processor material strongly correlates with processor cost, one of the primary constraints to consider during microprocessor design is the total area required for the processor's components and connections. Design choices often result in changes to the required space on the die, but it is not guaranteed that any increase in die space will increase the processor's speed.

This lab addresses the problem of determining the best combination of available options in a processor design to maximize its speed, given maximum on-chip areas of 2600, 4000, and 5000 units^2. Because a processor's optimal configuration is dependent upon its intended application, the optimization process will be shown for one specific use: the Go benchmark, executed in the SimpleScalar PISA simulator. Eight different parameters were varied to identify these optimal configurations. In addition, this lab addresses the problem of identifying the best cache configuration for a given processor design. Variables under test include cache size, block size, and associativity.

It might be possible to exhaustively examine the possible combinations of the available design options given the project time constraints. Unfortunately, searching the space of all options does not scale well – especially if more options than already exist in this project are considered. To address this, the authors make choices to minimize the number of simulations required to arrive at the optimal processor configuration for each size constraint.

Prior to experimentation, it is assumed from previous labs that increasing the number of any resource will not lower the speedup, but it also may not increase the speedup. Additionally, it seems almost certain that choosing a 2-level branch predictor or out-of-order issue will increase speedup, but it is an open question how much area will be required for using either to be feasible.

It is also known from previous labs that increasing cache size and associativity may increase access time, reducing speedup in some applications. However, this is dependent upon the application because some benchmarks may have higher locality than others. One useful product of this research will be observations of what the best cache parameters mean about the locality of the target benchmark.

Background

Although a microprocessor contains many hardware components, this experiment focuses on the machine's width, the level-one data cache, the number of available memory ports, and the number of arithmetic logic units (ALU). Additionally, we vary the branch prediction strategy and the instruction issue schemes to analyze the impact on processor performance. We now discuss each component in detail.

The machine width, which determines how many instructions can be executed at once, consists of four separate components: the fetch, decode, issue, and commit widths. In this experiment, changing the machine width to v is akin to individually changing each of these four separate widths to v. As an example, with a machine of width two, we can fetch two instructions, decode an additional two instructions, issue two more instructions to the ALUs, and commit two final instructions – all in the same clock cycle.

The cache is a highly-customizable piece of hardware separate from the CPU; it is responsible for storing data that are not currently being used by the processor. The CPU views the cache, main memory and disk (and any other storage devices) as one large memory space and will read and store in this space. The cache itself is vital to processor performance because it is the fastest and closest memory available to the CPU; in other words, the cache is directly responsible for how fast the CPU can read from/write to memory.

Each memory port allows the processor to either write or read data to/from memory in a given cycle. Individual memory ports are capable of doing both, but no port can read and write at the same time. Thus, increasing the number of available memory ports increases the number of reads and writes the CPU can execute at any given time.

Arithmetic logic unit is a general term used to represent those units that perform mathematical functions like addition, multiplication, and negation. For this experiment, we vary the number of integer ALUs, integer multipliers, floating point ALUs, and floating point multipliers. As the names suggest, these particular units are responsible for the various mathematical operations on integers and floating point numbers.

The branch prediction strategy determines how the hardware handles branch instructions. There are many different branch predictors, but, for the scope of this experiment, we only consider the branch-taken, branch-not-taken, and 2-level strategies. As the name suggests, a branch-taken predictor will always predict the branch is taken and will begin fetching the instructions at the new program counter (PC). Branch-not-taken predicts the opposite, thus fetching the next sequential instruction. The 2-level predictor is a more complex predictor that uses a pattern history table to store previous branch outcomes. This predictor uses that table when making a future prediction.

A processor can issue instructions either in-order or out-of-order. In-order execution behaves exactly as it sounds: the processor fetches and issues instructions in the compiler-created order. If there exists a hazard between instructions that cannot be resolved, the processor and its entire pipeline stall. Out-of-order issue, on the other hand, fetches instructions from memory in order, but the order of issue can be changed dynamically to prevent stalls. This means that, if the processor will have a stall between two instructions with an unresolvable hazard, it will attempt to find an instruction that does not cause a hazard – and that maintains program correctness – to fill the gap(s).

Methodology

The first observation for maximizing performance within the given constraints is that a brute-force search must iterate over 18,432 different combinations. Because each simulation takes a few minutes to run on a modern laptop, brute-force testing is impractical, even after eliminating combinations outside the area limitations. Instead, we must identify the features that give the best improvement for the area they utilize and then prioritize these features. To measure this performance improvement, we record the CPI of each simulation – this directly relates to the execution time.

When we have little-to-no information about the optimal configurations, branch prediction can be ignored to simplify the narrow-down process. The benchmark is constant, so either the branch-taken or branch-not-taken predictor will exhibit better performance, and it will do so consistently across all configurations. Additionally, it is expected that a 2-level predictor will outperform both static prediction schemes across all configurations. We thus hold the branch prediction strategy constant while varying individual hardware components to determine whether or not each component significantly improves performance. After sufficiently reducing the search space for our configurations, we perform a brute-force search to evaluate CPI values between closely performing configurations.

The first simulation configurations were chosen specifically to narrow the scope of the simulations required for the project. These simulations selected one resource at a time and iterated over its possible values; we then recorded any change in CPI. This helped to identify resources that made the largest impact on CPI. In these exploratory simulations, different processor features were chosen as the variable under test and all of the remaining features were minimized or off; branch prediction was arbitrarily chosen to be taken. Each simulation was performed for both in-order and out-of-order issue because it seemed possible that some elements might only benefit performance when instructions are issued out-of-order. As expected, out-of-order issue resulted in a lower CPI than in-order issue.

From these initial simulations, it was clear that increasing the amount of hardware did not necessarily result in performance benefits. The remaining challenge was to then determine the optimal width and numbers of functional units and memory ports. Under these conditions, we quickly realized that only increasing the width strictly reduces CPI (for both in-order and out-of-order issue).

The next set of simulations utilized the above results and fine-tuned the search to answer the following question: for a higher width, what additional resources can be added to decrease CPI? Like the previous tests, we independently vary each resource and keep the remaining components at the minimum possible value. During these simulations, for both in-order and out-of-order issue, we found that configurations of width w and n of some component are typically suboptimal when $n > w$. Furthermore, we found that the integer ALU is the only component that can improve performance for the Go benchmark.

We follow this method once more by now fixing the number of integer ALUs at a higher number and independently changing the remaining resources. It is found that, for out-of-order issue only, increasing the memory ports will reduce the CPI, but only to the point that the number of memory ports is equal to the number of integer ALUs.

By employing this method one last time, we find that no additional resources will independently reduce CPI. For all previous tests, we operated under the assumption that the resources do not hold two-way dependence with each other. In other words, we assume that, because individual increases in two different resources don't independently contribute to performance, increasing both of them in the same simulation won't improve performance either. We verify our assumption by simulating Go with the maximum number of all units and observing the change in CPI; by running this simulation once, we observed that increasing all other functional units does not affect CPI. Thus, the assumption holds. Practically speaking, it is firmly concluded that the number of integer multipliers, FP ALUs, and FP multipliers can be fixed at the minimum value of 1 with no harm to performance.

The optimal static branch predictor (i.e., branch-taken vs. branch-not-taken) can be decided very easily. For a single configuration, simulating each prediction scheme shows that branch-taken yields better CPI than not branch-not-taken (for this benchmark).

The final simulations evaluate every width w and every number of integer ALUs n such that $n \leq w$. Additionally, these tests are executed for both taken and 2-level branch prediction and for both in-order and out-of-order issue. Finally, we vary the number of memory ports when using out-of-order instruction issue such that the number of memory ports is no greater than the number of integer ALUs. In total, then, we have 42 remaining simulations; this can be completed in a few hours.

To extract the optimal configurations, we group the simulations and their results by cost. The groups have costs ranging from [0 units^2, 2600 units^2], (2600 units^2, 4000 units^2], and (4000 units^2, 5000 units^2] as specified by the assignment instructions. In each category, the configuration with the lowest CPI is the optimum.

Figure 1 below illustrates our search for the optimal configurations. Specifically, it provides an abstract overview of our decision-making process.

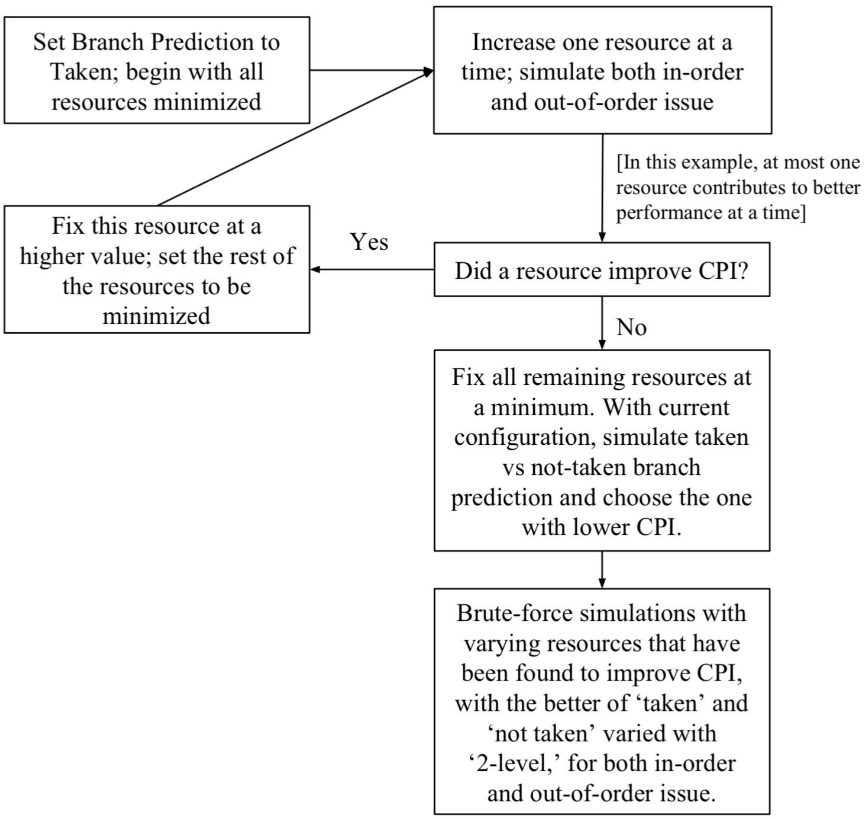
Set Branch Prediction to Taken; begin with all resources minimized
Increase one resource at a time; simulate both in-order and out-of-order issue
[In this example, at most one resource contributes to better performance at a time]
Did a resource improve CPI?
Yes
Fix this resource at a higher value; set the rest of the resources to be minimized
No
Fix all remaining resources at a minimum. With current configuration, simulate taken vs not-taken branch prediction and choose the one with lower CPI.
Brute-force simulations with varying resources that have been found to improve CPI, with the better of 'taken' and 'not taken' varied with '2-level,' for both in-order and out-of-order issue.

Figure 1: Decision-making process – It is specific to this application, as generally more than one resource can be modified at a time to increase performance.

This methodology works on the basis of eliminating possible configurations when they have been observed to offer no benefit in performance. We know we do not eliminate potential optimal configurations because of two principles. First, if a resource does not increase CPI for a specific configuration, it exclusively increases area cost and thus is worse than useless. Second, even if a resource does not increase CPI for a specific configuration, we know said resource will be verified later for different configurations. This ensures that resource dependencies are taken into account and that we do not blindly remove an entire subsection of configurations. Thus, we can eliminate wide swaths of potential configurations without also eliminating the solutions.

To actually conduct the simulations, we utilize the SimpleScalar toolset to simulate a processor with each configuration. Additionally, we use several bash scripts to improve efficiency. Specifically, we use `go.sh` to run SimpleScalar with a specific configuration, and we use `go_repeater.sh` as a template for our `go_repeater_test_suite_n.sh` files, where `n` is the number of the test suite. The `repeater` files simply run `go.sh` several times, one for each input configuration. Finally, we use `go_cache.sh` to simulate a specific cache configuration, and we use `go_cache_repeater.sh` to simulate the various cache configurations under test.

Analysis

Task A: Identifying the Optimal Processor Configuration

Using the methodology above, we obtain many test results, including those that were used to narrow down our search. As described previously, we verified that these tests include the optimal configurations for various size-limiting constraints. To extract results, we simply group tests by their costs and sort by CPI to identify the highest performing configuration for each desired processor size.

Below, we show our five highest-performing configurations under each size constraint. In these figures, we calculate speedup respective to a baseline processor of width one with one of each hardware unit, a branch-not-taken predictor, and in-order instruction issue. Of course, the speedup equation is given by the textbook as

$$Speedup = \frac{execution\ time\ for\ the\ entire\ task\ without\ using\ the\ enhancement}{execution\ time\ for\ the\ entire\ task\ using\ the\ enhancement\ when\ possible}$$

Speedup is thus calculated by dividing the baseline processor's CPI by the CPI of the processor in question. Here, *weighted speedup* refers to the calculated speedup divided by the configuration's total area cost. This calculation indicates performance per unit area for each configuration. The weighted speedups do not determine the optimal configuration given some set of configurations, but we did use these values to guide our search for the optimum. The five best configurations with respect to CPI under an area of 2600 units^2 are shown below. The optimal configuration is the first entry in Figure 2.

Width	Int ALU	Int Multiplier	Memory Port	FP ALU	FP Multiplier	Prediction	Instruction Issue	CPI	Speedup	Cost	Weighted Speedup
4	1	1	1	1	1	2-level	in-order	1.6972	1.3517	2550	0.5301
2	2	1	1	1	1	2-level	in-order	1.7862	1.2843	2150	0.5974
2	2	1	2	1	1	2-level	in-order	1.7862	1.2843	2450	0.5242
2	3	1	1	1	1	2-level	in-order	1.7862	1.2843	2250	0.5708
2	4	1	1	1	1	2-level	in-order	1.7862	1.2843	2350	0.5465

Figure 2: Our top five configurations with costs under 2600 units^2 of area – Better performance is indicated by lower CPI, or equivalently, higher speedup. For convenience, we've highlighted the optimal configuration.

For the 2600-units^2-or-smaller configuration design, out-of-order execution is not feasible from a space constraint. Additionally, with the exception of the integer ALU, none of the arithmetic resources provide any increased performance. Even though increasing the number of integer ALUs (up to and including the value for machine width) was previously found to improve performance, we found during our final simulations that implementing an improved branch prediction policy and employing a larger machine width provides greater performance benefits. Therefore, the largest CPI speedup for the 2600 units^2 configuration was achieved by increasing the machine width and implementing the 2-level branch prediction scheme.

The top-performing configurations under an area cost of 4000 and 5000 are respectively shown below.

Width	Int ALU	Int Multiplier	Memory Port	FP ALU	FP Multiplier	Prediction	Instruction Issue	CPI	Speedup	Cost	Weighted Speedup
2	2	1	1	1	1	2-level	out-of-order	1.3893	1.6513	3570	0.4625
2	3	1	1	1	1	2-level	out-of-order	1.3893	1.6513	3740	0.4415
2	4	1	1	1	1	2-level	out-of-order	1.3893	1.6513	3910	0.4223
4	4	1	1	1	1	taken	out-of-order	1.5117	1.5176	3995	0.3799
4	4	1	1	1	1	taken	out-of-order	1.5117	1.5176	3995	0.3799

Figure 3: Our top five configurations with costs under 4000 units^2 of area – Better performance is indicated by lower CPI, or equivalently, higher speedup. For convenience, we've highlighted the optimal configuration.

Width	Int ALU	Int Multiplier	Memory Port	FP ALU	FP Multiplier	Prediction	Instruction Issue	CPI	Speedup	Cost	Weighted Speedup
4	4	1	1	1	1	2-level	out-of-order	1.2202	1.8801	4760	0.3950
4	3	1	1	1	1	2-level	out-of-order	1.2226	1.8764	4590	0.4088
4	2	1	1	1	1	2-level	out-of-order	1.2435	1.8449	4420	0.4174
4	1	1	1	1	1	2-level	out-of-order	1.4550	1.5767	4250	0.3710
4	1	1	1	1	1	2-level	out-of-order	1.4550	1.5767	4250	0.3710

Figure 4: Our top five configurations with costs under 5000 units^2 of area – Better performance is indicated by lower CPI, or equivalently, higher speedup. For convenience, we've highlighted the optimal configuration.

It is not surprising that we can achieve much higher performance with a larger budget; however, it is insightful to notice that the weighted speedup often decreases with costlier configurations. In other words, the rate at which speedup improves decreases as we continue to add resources, so it becomes increasingly more expensive to enhance performance. Specifically, note that the weighted speedup for the optimal configurations significantly decreases (from 53% faster than the baseline to 46% faster to 40% faster) as the processor size constraint increases. Clearly, then, a tradeoff exists: one must ask whether absolute performance or cost-efficiency is more important. This knowledge affects which configuration is the optimum at a given price-point, but the optimal configuration also depends heavily on the intended application. If area cost is of no issue and the highest performance is required, then the best possible configuration likely requires more than 5000 units^2. In most cases, however, area is constrained on a chip and the optimization problem will extend beyond the processor. As will be seen in Task B, altering cache parameters directly affects performance as well. If a processor configuration were to restrict area that could be used for a better cache configuration, the overall performance is not likely to be optimal.

Task B: Identifying the Optimal Cache Configuration

After identifying the optimal processor configuration no larger than 2600 units^2, we evaluate further potential improvement by varying the level-one data cache configuration. Specifically, we vary the cache size, the block size, and the associativity. Figure 5 below illustrates the miss rate and CPI for each cache configuration.

Cache Size (KB)	Block Size (B)	Associativity	Replacement Policy	Miss Rate	CPI
16	8	1	LRU	0.0300	1.7114
16	8	2	LRU	0.0154	1.7017
16	8	4	LRU	0.0104	1.6977
16	16	1	LRU	0.0351	1.7152
16	16	2	LRU	0.0175	1.7036
16	16	4	LRU	0.0088	1.6966
16	32	1	LRU	0.0458	1.7228
16	32	2	LRU	0.0257	1.7100
16	32	4	LRU	0.0097	1.6972
64	8	1	LRU	0.0088	1.6896
64	8	2	LRU	0.0022	1.6828
64	8	4	LRU	0.0011	1.6797
64	16	1	LRU	0.0102	1.6907
64	16	2	LRU	0.0019	1.6827
64	16	4	LRU	0.0009	1.6799
64	32	1	LRU	0.0129	1.6927
64	32	2	LRU	0.0019	1.6834
64	32	4	LRU	0.0008	1.6808

Figure 5: Miss rate and CPI for various L1 data cache configurations

For every combination of block size and associativity, even the least performant 64 KB cache outperforms every 16 KB cache with respect to CPI. Additionally, every 64 KB cache configuration outperforms its otherwise-identical 16 KB configuration, and most outperform *all* 16 KB configurations. Note, however, that the actual hit time and miss penalty are constant in SimpleScalar and thus do not change when varying cache parameters. Logically, though, the hit time and miss penalty for a 16 KB cache would not be the same as for a 64 KB cache. We turn, then, to the average memory access time formula:

$$\textit{average memory access time} = \textit{access time} + \textit{miss rate} * \textit{miss penalty}$$

We are told that the 16 KB caches have an access time of one cycle, and that the 64 KB caches have an access time of two cycles. The unknown variable in the above equation is thus *miss penalty*. For small miss penalties, the difference in access time dwarfs the difference in miss rate, so 16 KB caches are better. For large miss penalties, though, the miss rate is more consequential, and thus optimal configurations must utilize the larger cache.

Still, none of the cache modifications were able to improve the performance of the 2600 units^2 processor to be better than the larger (4000 and 5000 units^2) processors. The significant performance gap is mainly caused by the CPI improvement gained by implementing out-of-order issue on the larger processors. Due to space constraints, out-of-order issue simply cannot optimally be implemented for a 2600 units^2 design. The simulations indicate that, for the Go benchmark input data model, processor resources have a more substantial impact on performance than cache configurations. Therefore, even when given more freedom to vary cache configurations, the 2600 units^2 processor will not outperform the larger processors purely through cache parameter manipulation.

When testing benchmarks with relatively small data sets, the cache is able to exhibit high performance due to an application that grants both spatial and temporal locality. As the size of the dataset increases relative to the cache size, we expect that cache performance will dwindle due to capacity and compulsory misses. That is, a smaller fraction of data will be able to fit in the cache, and it is less likely that the data exhibits spatial locality. In an application such as this, cache configuration becomes much more significant, and it is expected that cache size and set-associativity will have greater impacts on the measured CPI.

In making design decisions solely on the performance of the Go benchmark, however, our results show that we can gain much more performance by increasing resources in the processor (though this takes up more area) as opposed to optimizing the cache configuration. In other applications, however, cache configuration will likely play a larger role in performance, and blindly increasing processor size may decrease the potential speedup that an improved cache can offer.

Conclusion

We found that varying the machine width, branch prediction strategy, and instruction issue provides the largest performance improvement. While it was determined that integer ALU units did provide some additional speedup, the improvement was limited by the machine width of the processor. For the 2600-units^2-or-less configurations, the size constraint resulted in us utilizing a larger machine width and improved branch prediction strategy because out-of-order issuance was too costly. Since out-of-order instruction issue is often faster, and the larger processor configurations have significantly more area available, all of our fastest medium and large configurations utilize out-of-order issue.

Varying the cache parameters was also shown to have performance improvements, but not to the extent that one should prioritize cache configuration over more hardware (at least for this benchmark). Specifically, we found that caches of size 64 KB have a much lower miss rate than their similarly-configured 16 KB caches. Because the access time for large caches is longer, though, a processor designer must consider the miss penalty when configuring the L1 data cache. When the cache configuration does not significantly impact cost (this is unlikely, but we operate under this assumption here), one should always evaluate the best cache parameters before selecting the optimal configuration – without optimizing the cache, the *optimal* processor configuration may in fact be suboptimal.

Lessons Learned

Throughout this project, we each gained an appreciation for processor design and testing. Additionally, we grew more familiar with technical writing, large group projects, and timeliness. We now present our individual lessons learned.

Yousuke Matsui

- Optimizing a processor for performance and size is very application-specific.
- Clearly-defined evaluation criteria assist in the trade-off decisions that occur during design and experimentation.
- When varying cache parameters, the configuration with the lowest CPI does not necessarily imply that it is also the configuration with the lowest miss rate.

David Crow

- At the hardware level, an application's operations can be heavily skewed towards a specific mathematical unit. In this case, for example, we only see improvement when increasing the number of integer ALUs.
- Without additional hardware units, out-of-order instruction execution offers no notable performance improvement.
- Bash scripts allow one to easily iterate over and parallelize program inputs.

Nicholas Echeverry

- Certain components bottleneck performance alone, and speedup will stay constant until these are changed.
    - Extra components can be entirely useless if some other resource is bottlenecking performance.
- Issue order and branch prediction show extremely large impacts on the CPI when compared to additional components.
- Depending on the application, a more sophisticated cache may not be necessary and it'd benefit to allocate more area to the processor.

Joey Vagedes

- Eliminating redundant test cases early is vital to a timely experiment. We were able to eliminate half of the test case pool by determining branch-taken was always more efficient than branch-not-taken.
- Splitting test cases between more members of the group is much more efficient.

Dan Koranek

- There are few performance increases that we get for free. Including any processor feature is always a tradeoff between different costs.
- Few performance increases apply to every benchmark. Either choose a wide range of benchmarks to capture all types of performance, or choose the benchmarks that allow you to best measure the intended use of a processor.
- Spend time at the beginning of a group project intelligently splitting up tasks, and do your best to stay abreast of the progress of other group members.