CMPEN 431 Design Space Exploration Report

Introduction

The goal of this project was to optimize a processor design with respect to performance and energy efficiency. The design space included 18 dimensions, each with a varying number of indices. In total, there were 4 * 2 * 2 * 6 * 4 * 2 * 9 * 3 * 9 * 3 * 10 * 4 * 5 * 5 * 7 * 7 * 9 * 6 = 1.4814213e+12 total designs in the design space. Even after filtering impossible designs using the validation function, there were still a large number of total designs in comparison to the exploration limit of 1,000 designs. Thus, a significant part of the process involved reducing the design space to meet these limitations.

Energy

The way I chose to reduce the search space to optimize energy efficiency for each dimension is outlined below:

- Width: I chose to restrict the possible CPU widths to 1-wide, 2-wide, and 4-wide. This was done after inspecting the core leakage power specifications given in the project instructions. 1-wide, 2-wide, and 4-wide processors scaled instruction width and power leakage linearly. In other words, an increase in instruction width corresponded to a linearly proportional increase in power leakage. However, the 8-wide processor would leak 4 times the amount of power a 4-wide processor, making it inefficient.
- Fetch:speed ratio: I chose to allow both a ratio of 1:1 and 2:1. Although more frequent instruction fetches consumed more power, this was more dependent on cache size.
- Scheduling: Although in class we learned that out-of-order scheduling consumes more power due to more complex hardware than in-order scheduling, the specifications given in the project description revealed that for the scope of this project, the power losses due to out of order scheduling were a maximum of 20-50% greater than in-order scheduling for 1-wide processors, and decreased significantly for multi-wide processors. Thus, I allowed both inorder and out of order scheduling.
- RUU size: I allowed for every RUU size except for the lowest one (4) in my design space. There were no explicit power constraints given due to RUU in the project description, so I allowed most sizes, as a larger RUU generally leads to more efficient pipelining.
- LSQ size: I allowed all LSQ sizes.
- Memports: I allowed both 1 and 2 memports. Although additional access to memory costs power, the performance gains of committing more frequently to memory may lead to power savings in the long run.
- L1 D$ ways/sets, L1 I$ ways/sets, L2 ways/sets: A larger cache size generally consumes more power. However, without prior knowledge of the data and program being executed, it is hard to generalize what cache size/associativity is optimal for power consumption. Thus, I allowed for almost every possible combination. Note: for the L1

caches, I chose to exclude the possibility of 32 sets, and for the L2 cache, I chose to exclude the possibility of 256 and 512 sets. This is due to the fact that these values are unlikely to be viable in the first place due to the constraints placed on minimum cache/block size, and in the case that they are viable, they make computing the number of ways significantly more challenging, and they are unlikely to lead to an efficient configuration due to how small they are. Thus, for both programmer sanity, I chose to exclude them.

- TLB sets: chose to fix at 16 sets. An arbitrary choice, but due to the lack of information on the impact of TLB size on power consumption, to reduce the design space, I chose to fix it at a constant value.
- L1 D$ latency, L1 I$ latency, L2 latency: Simply calculated from cache size and cache associativity.
- Branch predictor: Allowed for all possible predictors. Like cache design, without knowing the dataset and instructions ahead of time, it is hard to generalize what will work better, and branch prediction can have a significant impact on performance and efficiency. Thus, I chose to try all viable combinations.

Performance

The way I chose to reduce the search space to optimize performance for each dimension is outlined below:

- Width: I chose to restrict the possible CPU widths to 4-wide, and 8-wide. This was done due to the large throughput gains in comparison to time loss when compared to 1-wide machines. For example, an 8 fold increase in instruction width came at a cost of only 43% increase in time taken, making increasing CPU width the most worthwhile tradeoff.
- Fetch:speed ratio: I chose to allow both a ratio of 1:1 and 2:1. This was done because although fetching instructions faster than committing them may help, it does take a time penalty of fetching instructions more frequently.
- Scheduling: I chose to use out-of-order scheduling exclusively. In written assignment 1, out of order scheduling performed much better in all scenarios than in-order, and this is consistent with what we learned in class.
- RUU size: I allowed for every RUU size except for the lowest two (4, 8) in my design space. In general, from what we learned from the Sohi paper, increasing RUU size generally improves performance across the board by reducing the latency when dealing with branching, cache misses, and stalls.
- LSQ size: I allowed all LSQ sizes except the lowest 2 (4, 8) in my design space. Like RUU size, larger LSQ size generally improves performance by reducing the effects of memory access latency.
- Memports: I allowed both 1 and 2 memports. Although additional access to memory costs time, the performance gains of committing more frequently to memory may lead to improvements in throughput over time.
- L1 D$ ways/sets, L1 I$ ways/sets, L2 ways/sets: A larger cache size generally takes longer to traverse. However, without prior knowledge of the data and program being executed, it is hard to generalize what cache size/associativity is optimal for

performance. Thus, I allowed for almost every possible combination. Note: for the L1 caches, I chose to exclude the possibility of 32 sets, and for the L2 cache, I chose to exclude the possibility of 256 and 512 sets. This is due to the fact that these values are unlikely to be viable in the first place due to the constraints placed on minimum cache/block size, and in the case that they are viable, they make computing the number of ways significantly more challenging, and they are unlikely to lead to an efficient configuration due to how small they are. Thus, for both programmer sanity, I chose to exclude them.

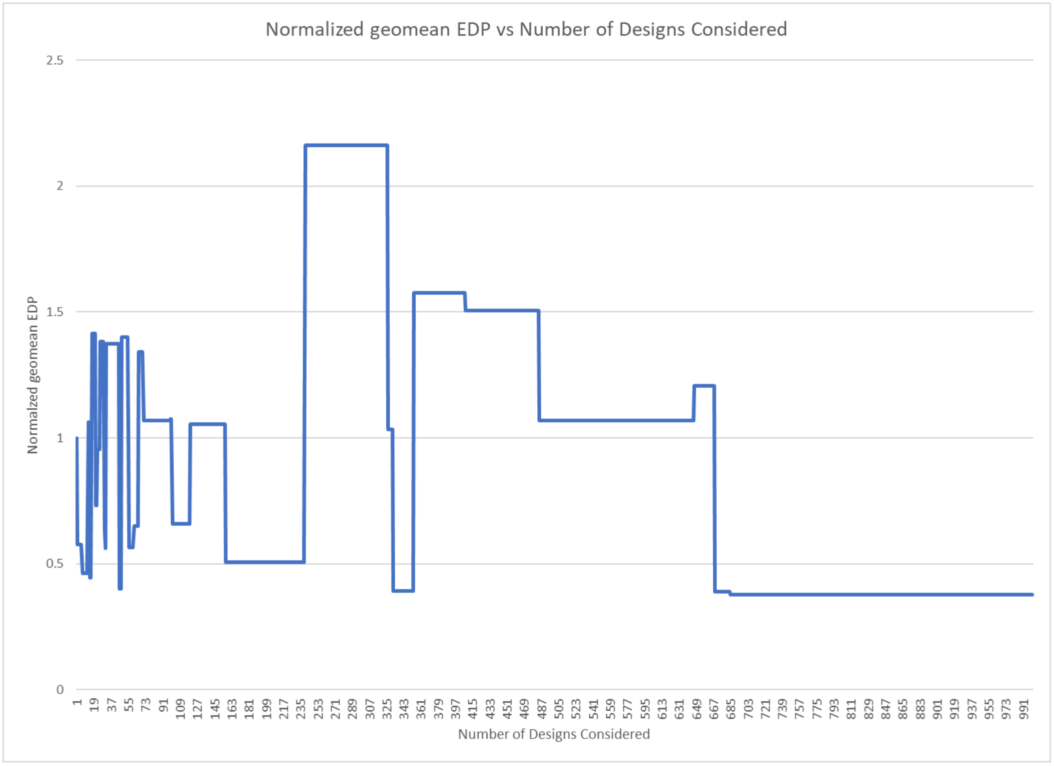
- TLB sets: chose to allow all values except the lowest 2 (4 and 8). Increasing TLB size can improve performance by reducing TLB management overhead (less frequent replacements in the TLB), and the frequency of misses, so I decided to explore a range of values.
- L1 D$ latency, L1 I$ latency, L2 latency: Simply calculated from cache size and cache associativity.
- Branch predictor: Allowed for all possible predictors except for not taken. Like cache design, without knowing the dataset and instructions ahead of time, it is hard to generalize what will work better, and branch prediction can have a significant impact on performance and efficiency. Thus, I chose to try all viable combinations. The reason I chose not to include not taken is because it is almost guaranteed to result in worse performance, as demonstrated by writing assignment 1 results.

Conclusion

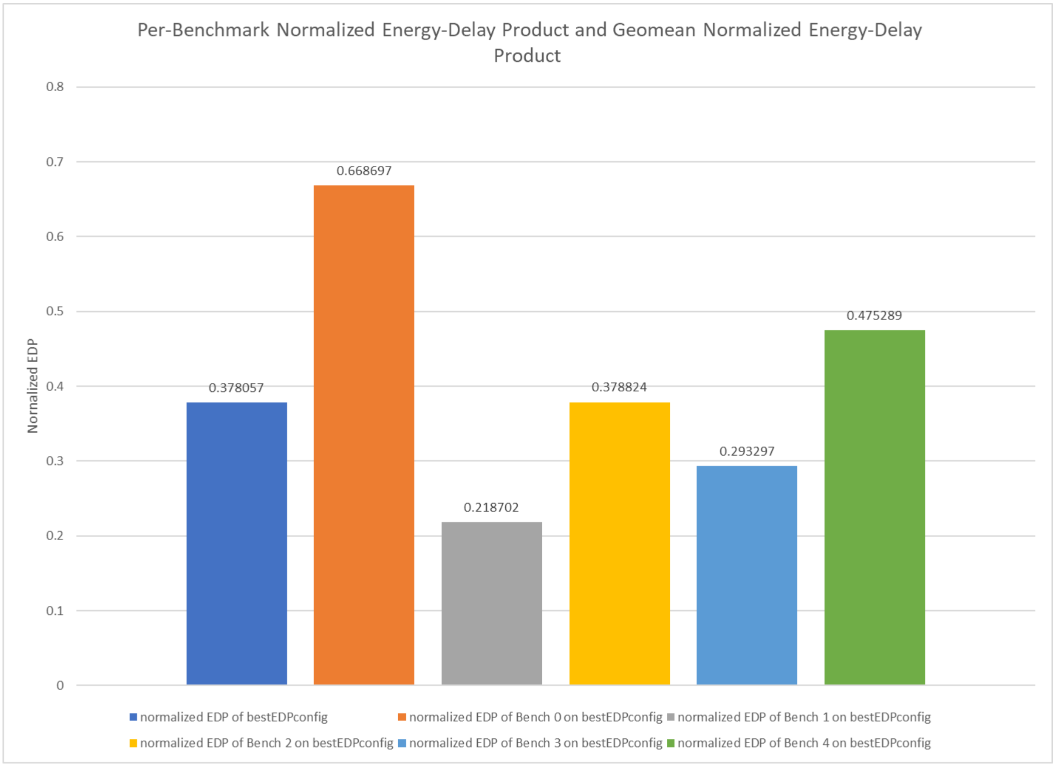
The results of the exploration are attached below (both the optimal configurations found and accompanying graphs). Optimizing for both energy and performance led to designs that were significantly more efficient than the baseline configuration (~63% more efficient for energy, ~84% more efficient for performance). As expected, the energy optimized design had a smaller cache with lower associativity, while the performance optimized cache had a larger cache, and benefited from larger TLBs, RUU's, and LSQ's. Interestingly, the performance optimized design used a bimodal branch prediction method, in comparison to the energy optimized design, which used a combined predictor. Overall, the results aligned with expectations going in. Moving forward, I would experiment more with the energy optimized design to see if a more efficient configuration can be crafted under different constraints.

Energy Configuration and Graphs:

Dimension	Value
Width	4
Fetchspeed	2
Scheduling	Out of order
RUU size	64
LSQ size	32
Memports	2
L1 D$ Sets	256
L1 D$ Ways	1
L1 I$ Sets	256
L1 I$ Ways	1
Unified L2 Sets	2048
Unified L2 Blocksize	64
Unified L2 Ways	1
TLB sets	16
L1 D$ Latency	1
L1 I$ Latency	1
Unified L2 Latency	5
Branch prediction	Combined (Tournament predictor)



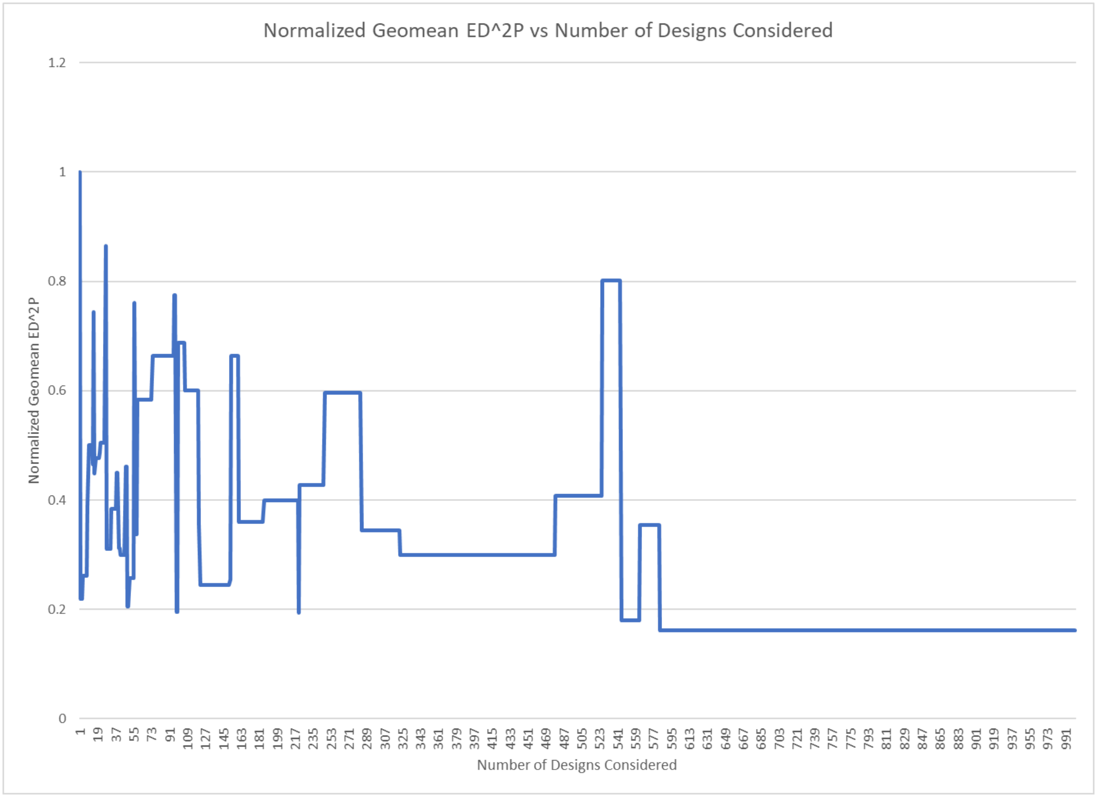
Normalized geomean EDP vs Number of Designs Considered
Normalzed geomean EDP
Number of Designs Considered

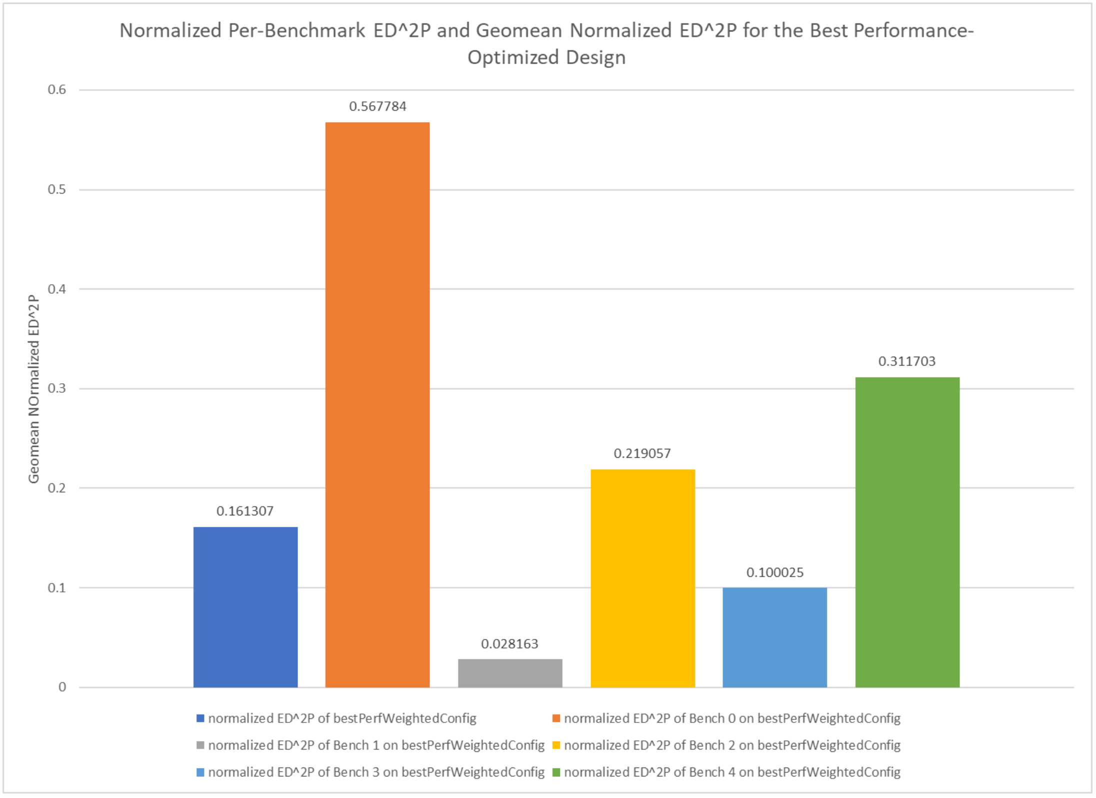


Per-Benchmark Normalized Energy-Delay Product and Geomean Normalized Energy-Delay Product
Normalized EDP
0.378057
0.668697
0.218702
0.378824
0.293297
0.475289
normalized EDP of bestEDPconfig
normalized EDP of Bench 0 on bestEDPconfig
normalized EDP of Bench 1 on bestEDPconfig
normalized EDP of Bench 2 on bestEDPconfig
normalized EDP of Bench 3 on bestEDPconfig
normalized EDP of Bench 4 on bestEDPconfig

Performance Configuration and Graphs:

Dimension	Value
Width	4
Fetchspeed	2
Scheduling	Out of order
RUU size	64
LSQ size	32
Memports	2
L1 D$ Sets	256
L1 D$ Ways	1
L1 I$ Sets	128
L1 I$ Ways	4
Unified L2 Sets	4096
Unified L2 Blocksize	64
Unified L2 Ways	2
TLB sets	64
L1 D$ Latency	1
L1 I$ Latency	4
Unified L2 Latency	8
Branch prediction	Bimodal, 2K entry


Normalized Geomean ED^2P vs Number of Designs Considered
1.2
1
0.8
0.6
0.4
0.2
0
Normalized Geomean ED^2P
Number of Designs Considered


Normalized Per-Benchmark ED^2P and Geomean Normalized ED^2P for the Best Performance-Optimized Design
Geomean NOrmalized ED^2P
0.567784
0.161307
0.028163
0.219057
0.100025
0.311703
normalized ED^2P of bestPerfWeightedConfig
normalized ED^2P of Bench 0 on bestPerfWeightedConfig
normalized ED^2P of Bench 1 on bestPerfWeightedConfig
normalized ED^2P of Bench 2 on bestPerfWeightedConfig
normalized ED^2P of Bench 3 on bestPerfWeightedConfig
normalized ED^2P of Bench 4 on bestPerfWeightedConfig