CMPEN 431 Project 1

Kuria Mbatia

Partners: None

Framework

The framework consists of three interconnected components that let's us explore and evaluate different processor designs. First, SimpleScalar simulates each design by running five benchmarks (bzip2, mcf, hmmer, sjeng, and equake) to measure its performance. Then, runprojectsuite.sh handles the configuration parameters (like cache sizes and branch predictors), translating them for SimpleScalar. The cpp code ties everything together - managing the design space exploration, calculating the important metrics (IPC, EDP), and making sure our designs follow realistic architectural constraints. This modular approach helped us systematically test different processor configurations to find optimal designs for both performance and energy efficiency.

Exploration Order

Table 1. Exploration Orders based on PSU ID.

$(PSU\_ID\_Number\_Sum)$ mod 24	1^{st}	2^{nd}	3^{rd}	4^{th}
0	BP	Cache	Core	FPU
1	BP	Cache	FPU	Core
2	BP	Core	Cache	FPU
3	BP	Core	FPU	Cache
4	BP	FPU	Cache	Core
5	BP	FPU	Core	Cache
6	Cache	BP	Core	FPU
7	Cache	BP	FPU	Core
8	Cache	Core	BP	FPU
9	Cache	Core	FPU	BP
10	Cache	FPU	BP	Core
11	Cache	FPU	Core	BP
12	Core	BP	Cache	FPU
13	Core	BP	FPU	Cache
14	Core	Cache	BP	FPU
15	Core	Cache	FPU	BP
16	Core	FPU	BP	Cache
17	Core	FPU	Cache	BP
18	FPU	BP	Cache	Core
19	FPU	BP	Core	Cache
20	FPU	Cache	BP	Core
21	FPU	Cache	Core	BP
22	FPU	Core	BP	Cache
23	FPU	Core	Cache	BP

Design Point

Design Point:

```
Iter # 0 config: 1 1 2 4 1 4 1 1 3 4 0 0 4 3 1 2 2 5 :
found in fileIter # 28 config: 1 1 2 4 1 4 1 1 3 4 0 0 4 3 1 2 2 5 :
found in file
proposedGeoEDP=2.84063e-08, bestEDP=2.84063e-08, proposedGeoTime=0.000149341,
bestTime=0.000149341
*Explanation on each design point is listed below
```

Design Space Exploration (DSE):

Parameter	EDP Value	Perf Value	EDP Why	Perf Why
width	1 (2-way)	1 (2-way)	Using 2-way instead of higher widths saved power while still being fast enough	This provided enough instruction throughput while maintaining a reasonable clock-frequency for an overall better IPC It was basically the sweet spot as 4-way and 8-way would have longer clock-cycles without any additional speed improvement
scheduling	1 (Out-Of-Order)		For Out-Of-Order although this uses more power, it helped finish programs faster which saved energy overall.	This allowed for better instruction reordering to hide latencies and extract more ILP. Basically it improves the speed by running instructions when they were ready

				rather than wait
l1block	2 (32 bytes)		This size worked well for getting nearby data without wasting energy on too large transfers	Captured spatial locality without an excess fetch overhead.
d1sets	4 (512)		Gave enough space for data without being too power-hungry	This was found to be the sufficient amount of capacity to capture working sets of access data and reduce L2 lookups. Basically it's enough sets to avoid thrashing, but not so many that lookups are slow.
dl1assoc	1 (2-way)		Two-way associativity helped avoid cache conflicts without using too much power	This reduced the number of conflict misses while keeping our access time reasonable.
il1sets	4 (512)		Matched our data cache size since instructions need similar space	This was found to be the best overall size to hold the working set instructions while also keeping the number of pipeline stalls at a minimum
il1assoc	1 (2-way)		Same as data cache - good balance of hits vs power use	This was found to be the most balanced hit rate-access latency (i.e. most balanced conflict avoidance w/ quick lookups)
ul2sets	1 (512)		This L2 size caught most of what missed in L1	512 was found to be the minimum size needed to

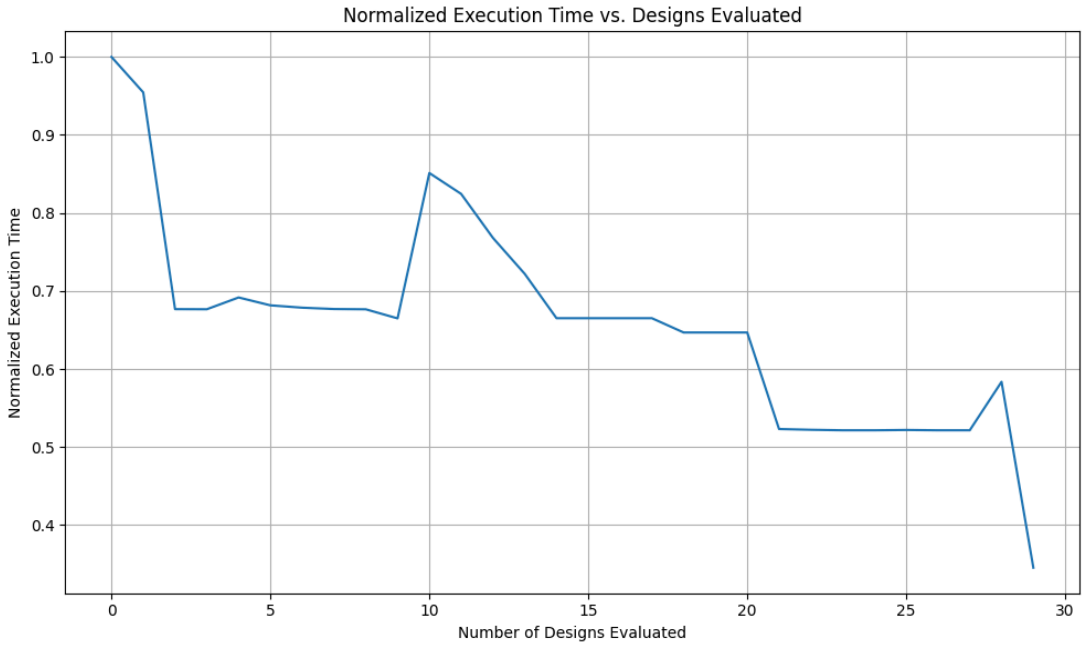
				capture the blocks that were evicted from L1 cache. Basically it caught the most misses without having longer access times associated.
ul2block	3 (128)		Bigger blocks in L2 helped get more data at once	128 bytes or the max L2 block size was used because larger blocks exploit spacial locality (from testing). This just means that with bigger blocks it was much better at getting instructions that were likely to be executed much sooner.
ul2assoc	4 (16-way)		L2 can handle higher associativity since it's accessed less often	16-way L2 associativity was used as L2 is not used as frequently as L1, thus a higher associativity reduced conflict misses for a better hit rate corresponding to better performance
replacepolicy	0 (LRU)		Keeping the most recently used stuff just works better	LRU was fund to be the most effective at keeping the frequently used blocks in cache while discarding everything else.
fpwidth	0 (val = 1)		I assumed that there wouldn't be a need for FP, from testing it seems as though most programs don't need lots of	Seems as though most tests were not floating point heavy (unless my way of searching was completely off)

			floating point, so there was a save on power usage	
branch settings	4 (Choice: -bpred comb -bpred:comb 1024)		Used a tournament style/combined BP approach,reduces mispredictions. It helped avoid wasting energy on wrong pathways by not taking wrong pathways	A combined BP reduced the misprediction and pipeline stalls that occurred, I used a tournament style predictor as through testing it helped reduce the number of pipeline stalls.
ras	3 (RAS size = 8)		Through testing t was found to work for most function calls without wasting space	Size of 8 means that we can handle deeper call chains for better prediction (decent size for tracking function calls/returns)
btb	1 (256 8)		Through testing this was found to be a good middle ground for branch target prediction	256 8, was used as it was a good balance for predicting quickly while avoiding fetch stalls
dl1lat	2 (3 cycles)		This came from the cache size choices selected	Same as the Cache organization choice
il1lat	2 (3 cycles)		il1 lat should be the same as out d1lat cycle cost	Same as dl1lat
ul2sat	5 (10 cycles)		What was noticed through testing was that there was an increase in the cost for going to L2 (obviously). But this just means taht a bigger/more associative L2 means there is a higher lookup time	This is the result of having a larger size and high associativity.

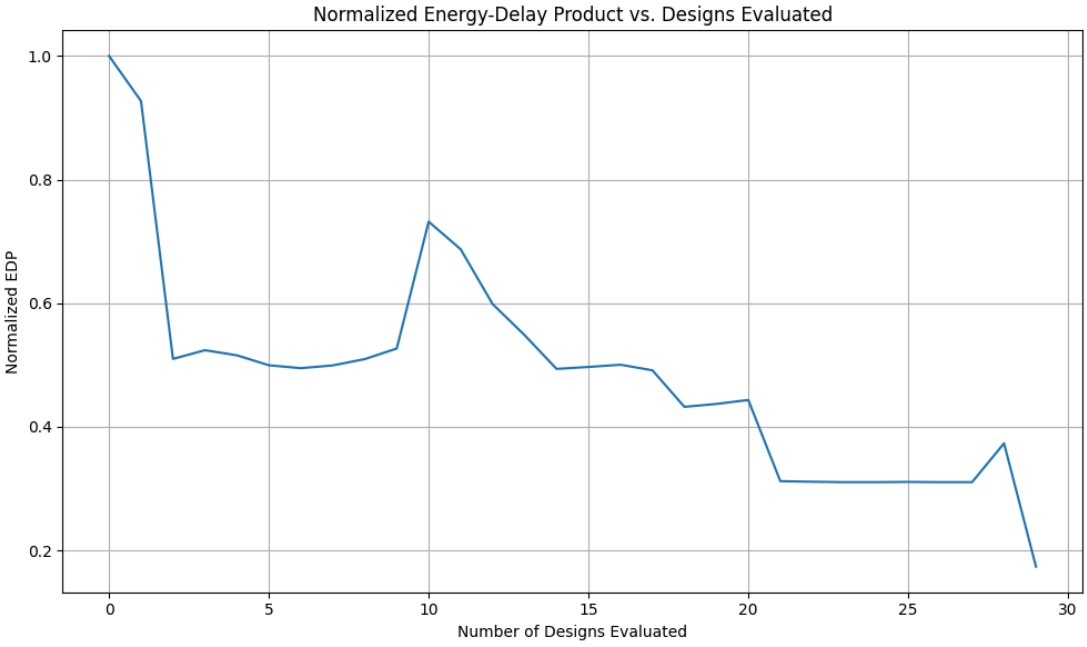
			thus a higher cycle penalty cost.	

Plots

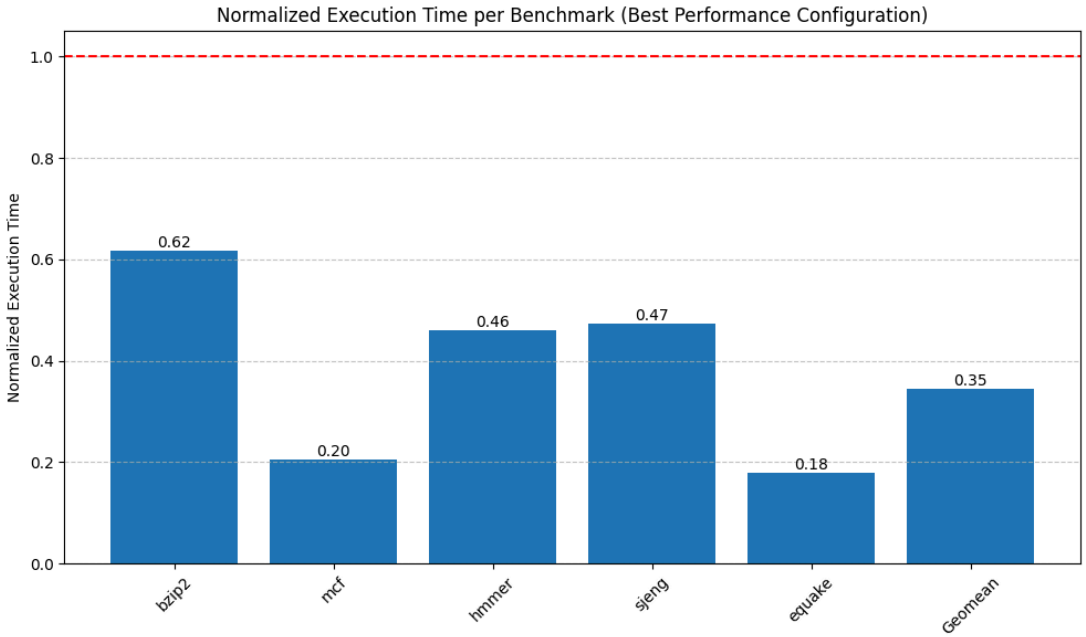
A.



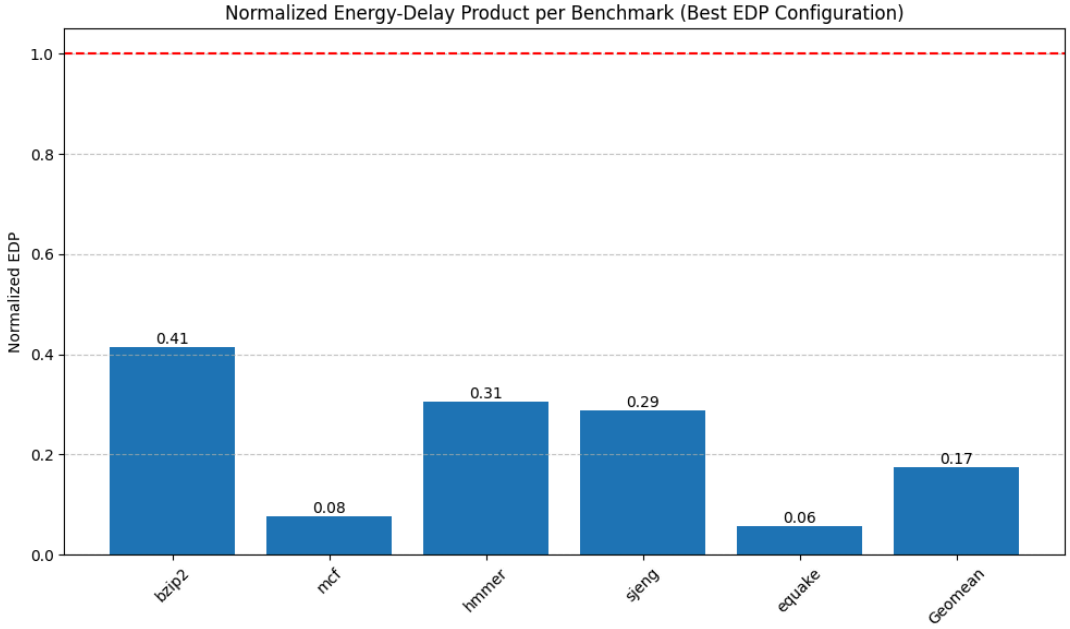
B.



C.


Normalized Execution Time per Benchmark (Best Performance Configuration)
Normalized Execution Time
1.0
0.8
0.6
0.4
0.2
0.0
0.62
0.20
0.46
0.47
0.18
0.35
bzip2
mcf
hmmer
sjeng
equake
Geomean

D.


Normalized Energy-Delay Product per Benchmark (Best EDP Configuration)
Normalized EDP
1.0
0.8
0.6
0.4
0.2
0.0
0.41
0.08
0.31
0.29
0.06
0.17
bzip2
mcf
hmmer
sjeng
equake
Geomean

Heuristics

Heuristics Improvements

Improvements

- A more adaptive exploration based on the results we see rather than just comparing with what is previous.
    - I could keep a history of configurations and their related parameters rather than just the config. This would help see much more broadly what is really affecting what, as we can prioritize exploring dimensions that show larger performance gains for example
- More sophisticated filtering strategy of bad configurations as the current strategy sacrifice broadness/scope of configurations for ones that are more likely to work
- Better Parameter interaction functions to see how different values end up affecting one another
    - Could see how cache size and associativity further relate to each other
    - Could see how core width and the scheduling policy interact with one another
- Trying to Start from multiple areas rather than just one specific point.

Other heuristic strategies that could be much better

1. Backtracking
    a. Remember multiple good configurations instead of just one
        i. Return to a previous good point if the current path isn't improving
    b. Try different exploration directions from good points
2. Segmented Level exploration
    a. With each level the scope of configurations becomes more focus with fore example
        i. Level 1: Broad search of varying valid configurations
        ii. Level 2: Optimization of said good configurations from Level 1
        iii. Level 3: Make small adjustments to the best configurations from Level 2
        iv. Level 4: Tune top 3 and kill the rest (example)
3. Performance prediction would help, as we could just skip configurations that we can predict will perform poorly.
4. Workload-Aware Algorithms
    a. This idea seems like a cheat code in hindsight but if we analyze the benchmark behavior then we can prioritize improving related parameters to obtain better configurations (Source)

Insights

Personal Insights

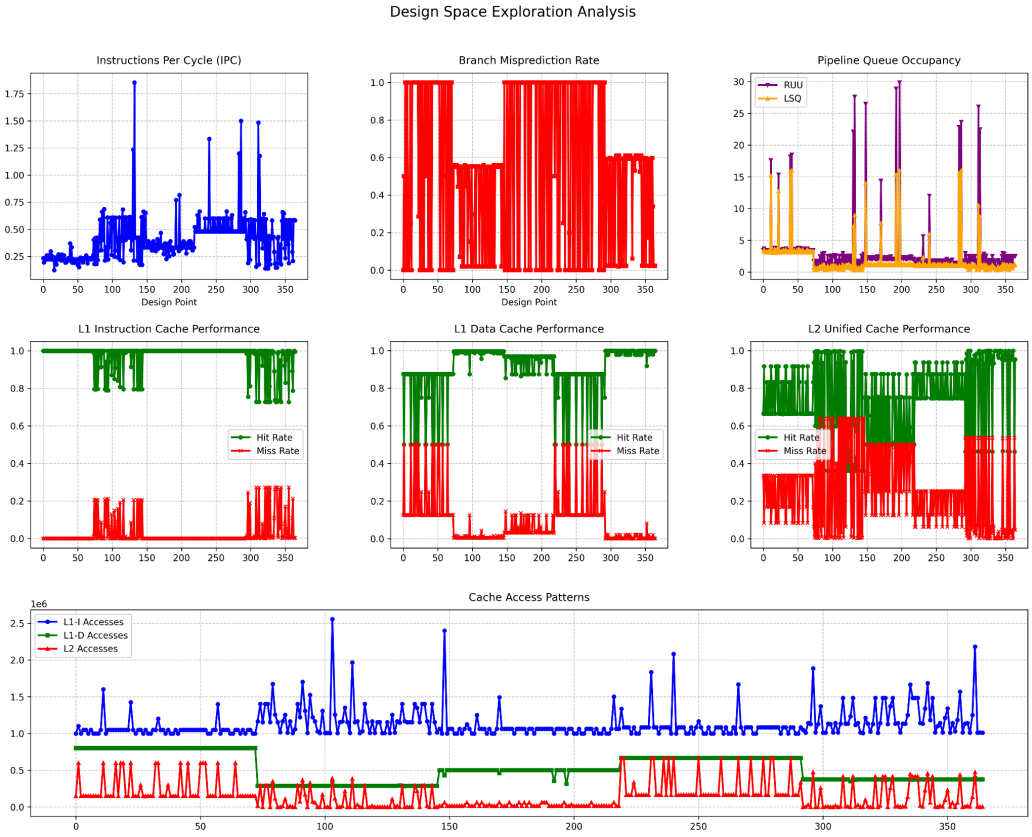
1. Work on items early and often, the more that I thought about how things related the more in depth that I could understand what was occurring under the hood, waiting to do stuff means that you can rush on things and skim over important areas (really just don't procrastinate to do shit)
2. The intricacies of how one small thing can build up and cascade into affecting an entire system, changing or even adjusting one small parameter can drastically affect the end score, results, or value of the entire system we are testing and that is fascinating.
    a. Why is because a tiny value like what should our L1 cache size, if we made it almost as large as L2 I noticed more hits, but if a miss occurred it affected my performance ratio as it related to the ability of finding it in L2, which then led to larger miss times, & so on & so forth. This example shows that something intuitive just can play a part in the entirety.

Technical Insights:

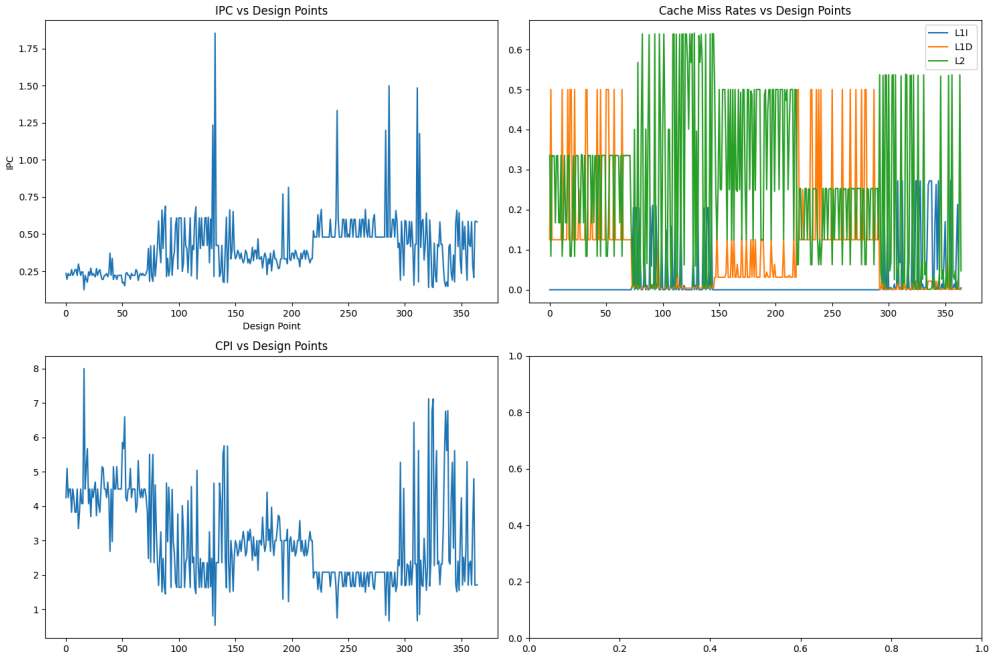
1. Go to office hours (frequently & as soon as possible) just because I made a ton of dumb mistakes such as not making my currentgroupindex & currentdimgroup inside of the generatenextconfigurationproposal function and didn't realize it kept resetting every time a new configuration was generated.
2. Read the project outline just because the first half of the project is really straightforward.

Additional

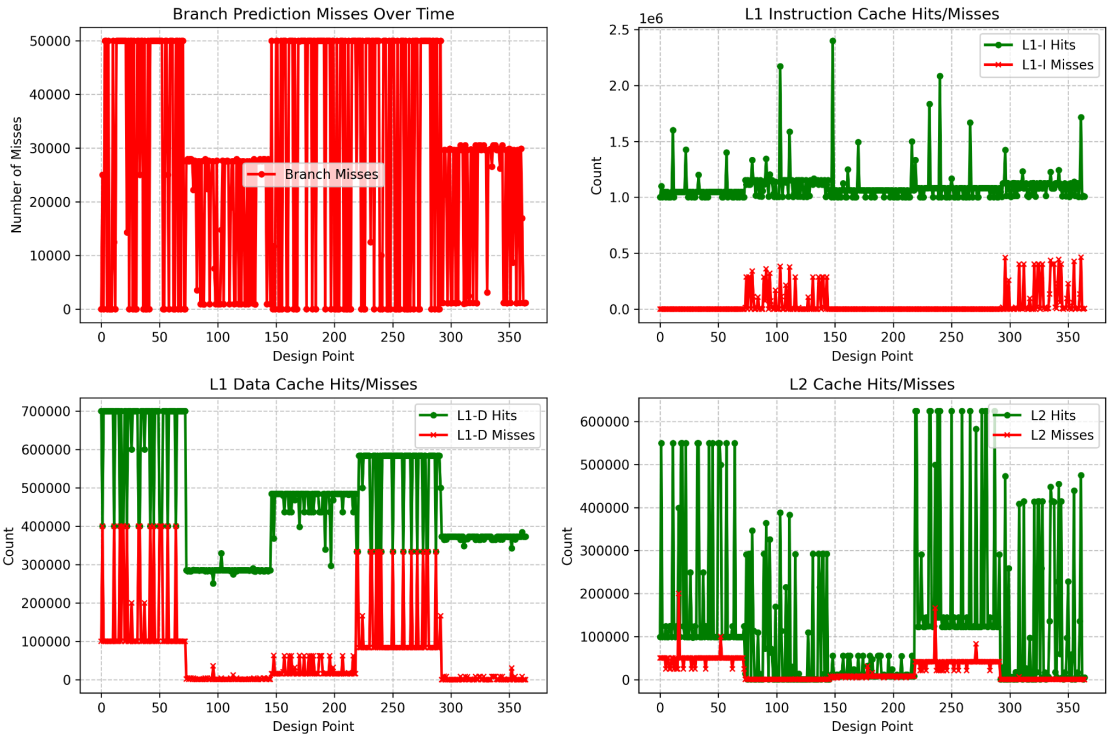
Additional Plots



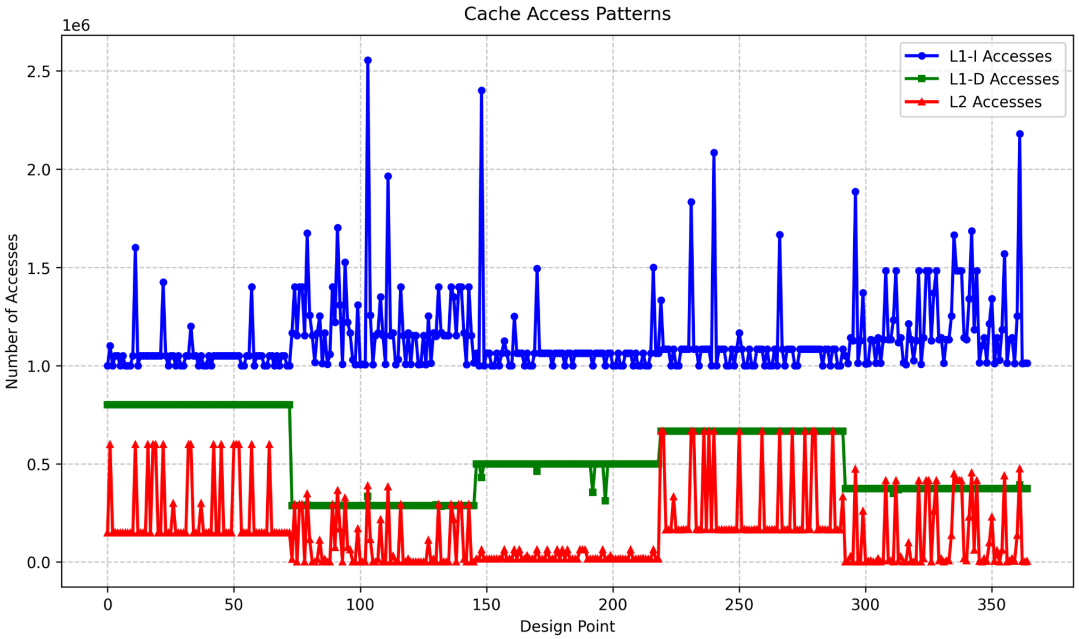
Detailed Design Space Exploration IMG 0.5



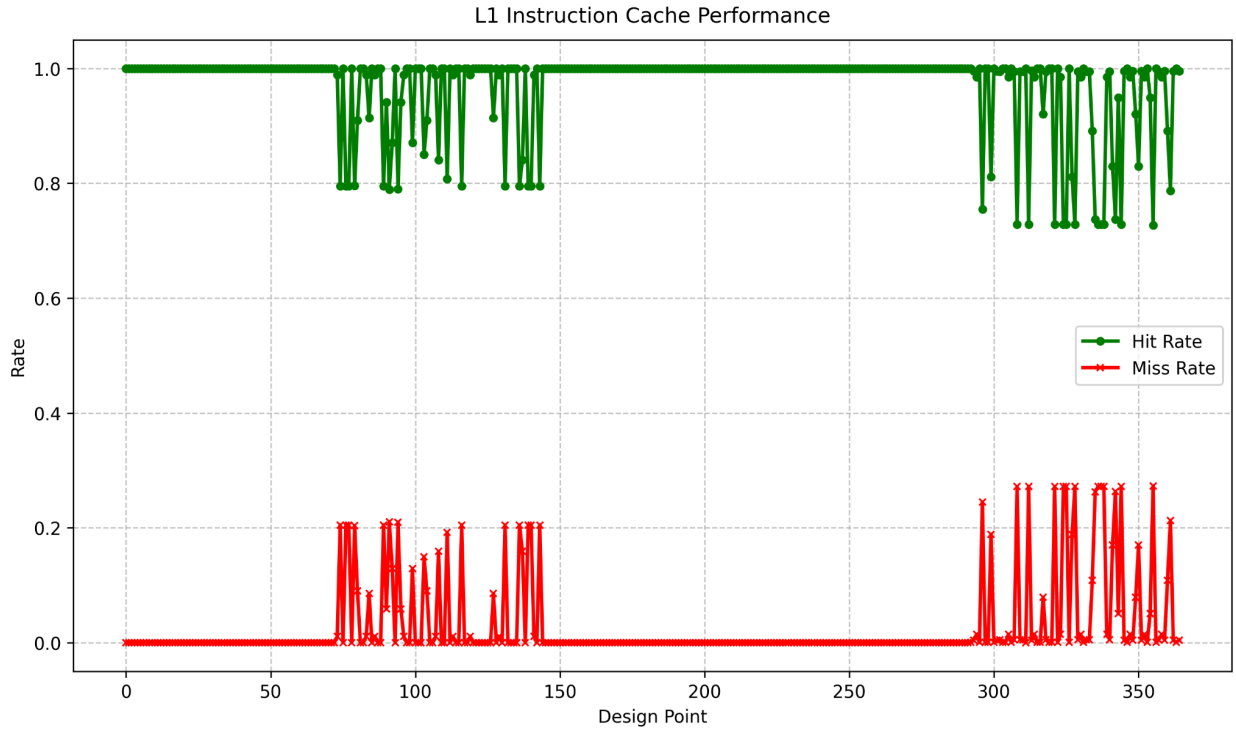
Design Space Exploration IMG 0.6



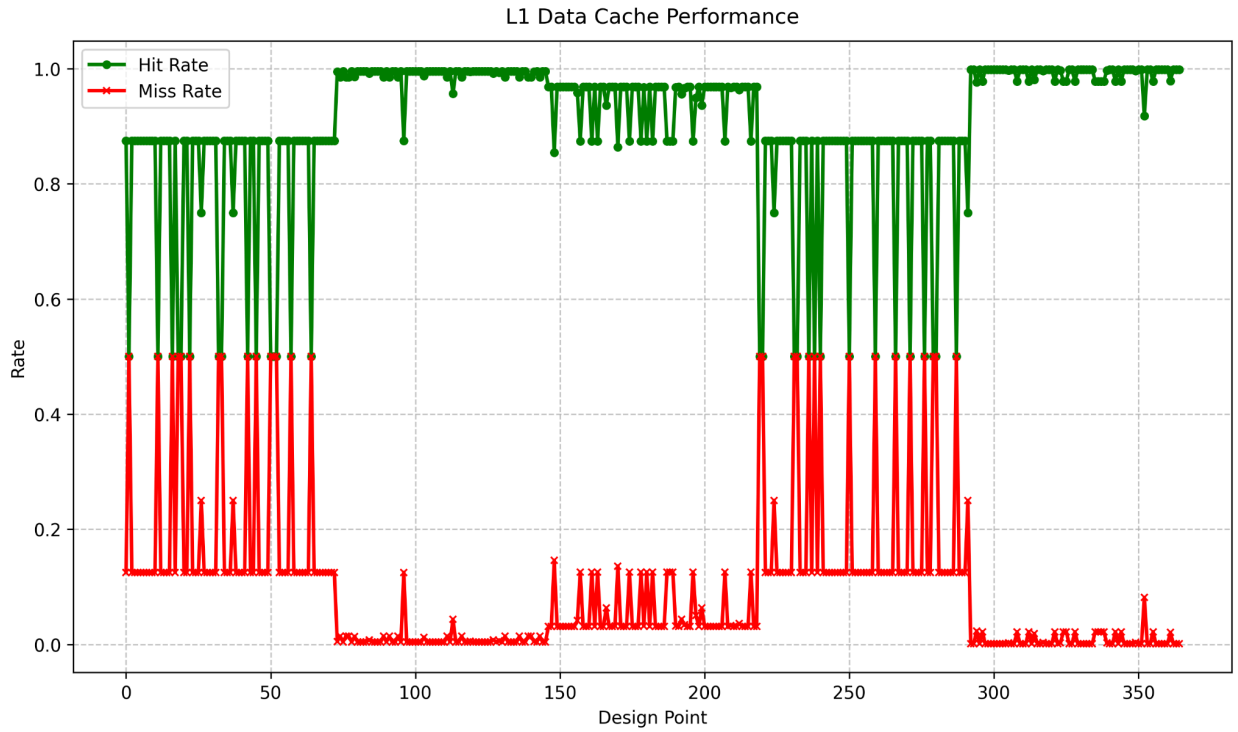
Hit Miss Trends (L1 & L2 Cache) IMG 0.7



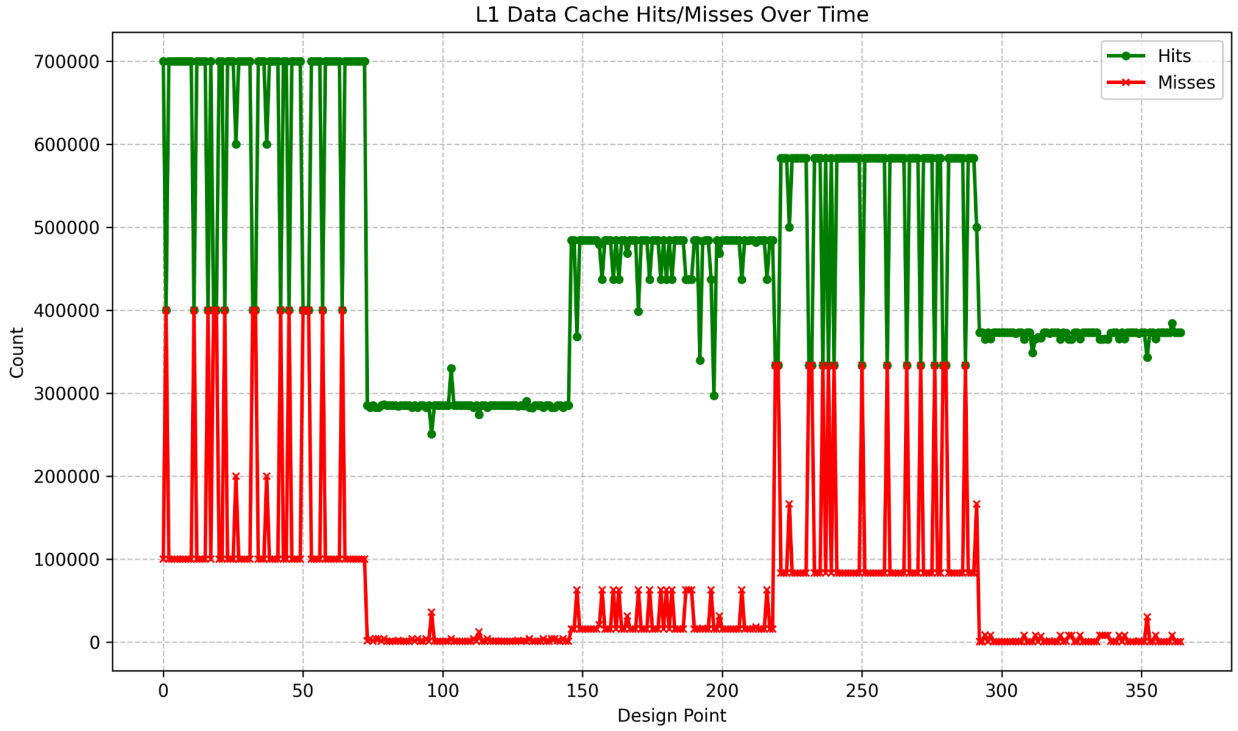
Cache Access Patterns IMG 0.8



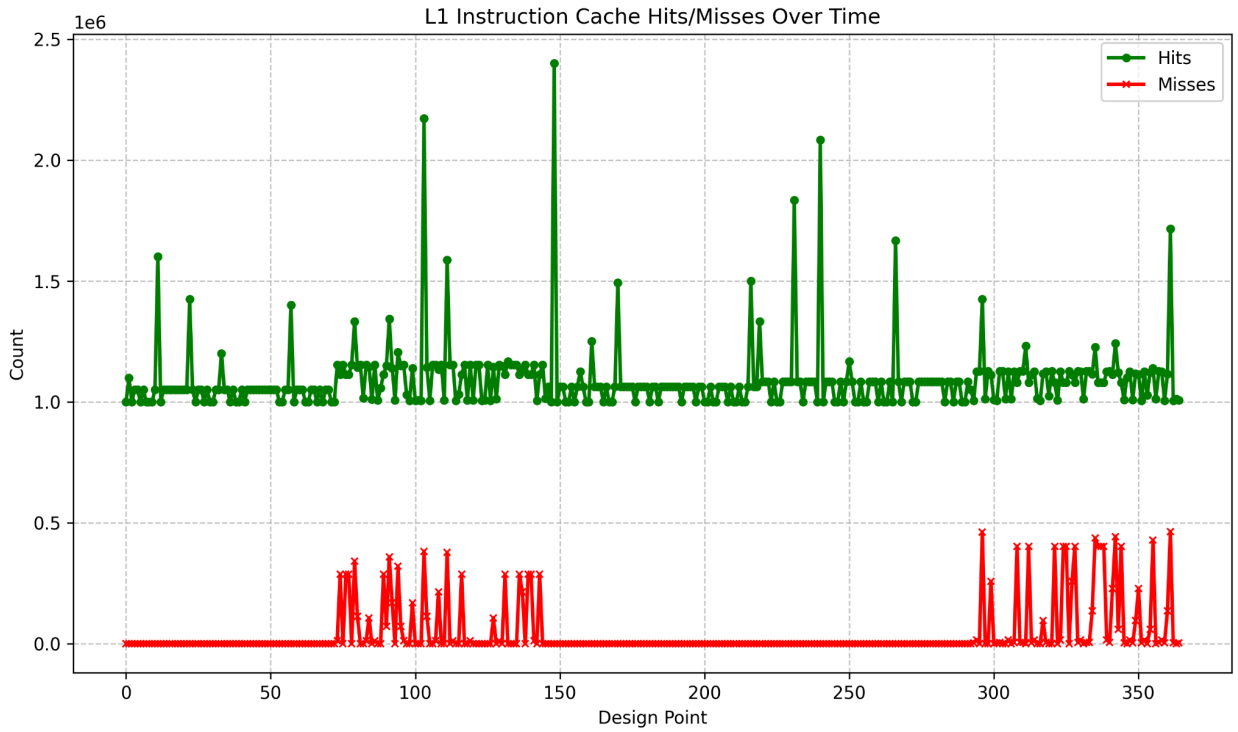
L1 Instruction Cache Performance IMG 0.9



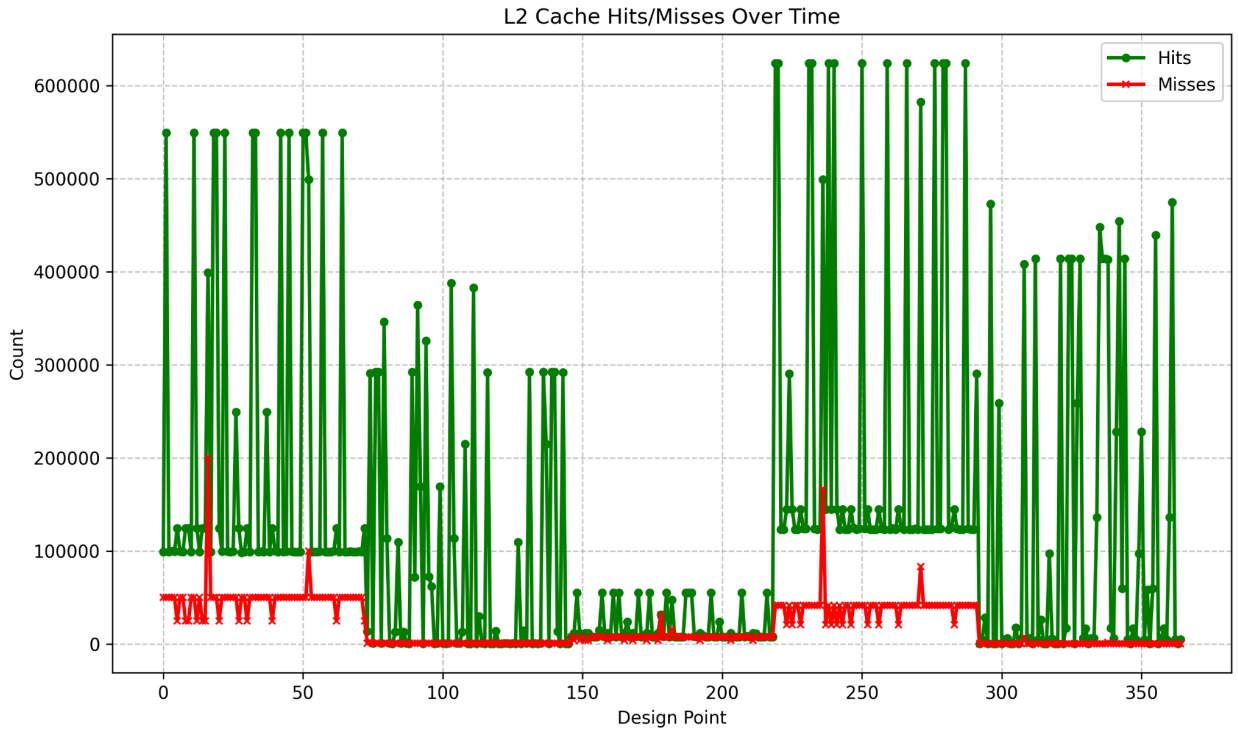
L1 Data Cache Performance IMG 1.0



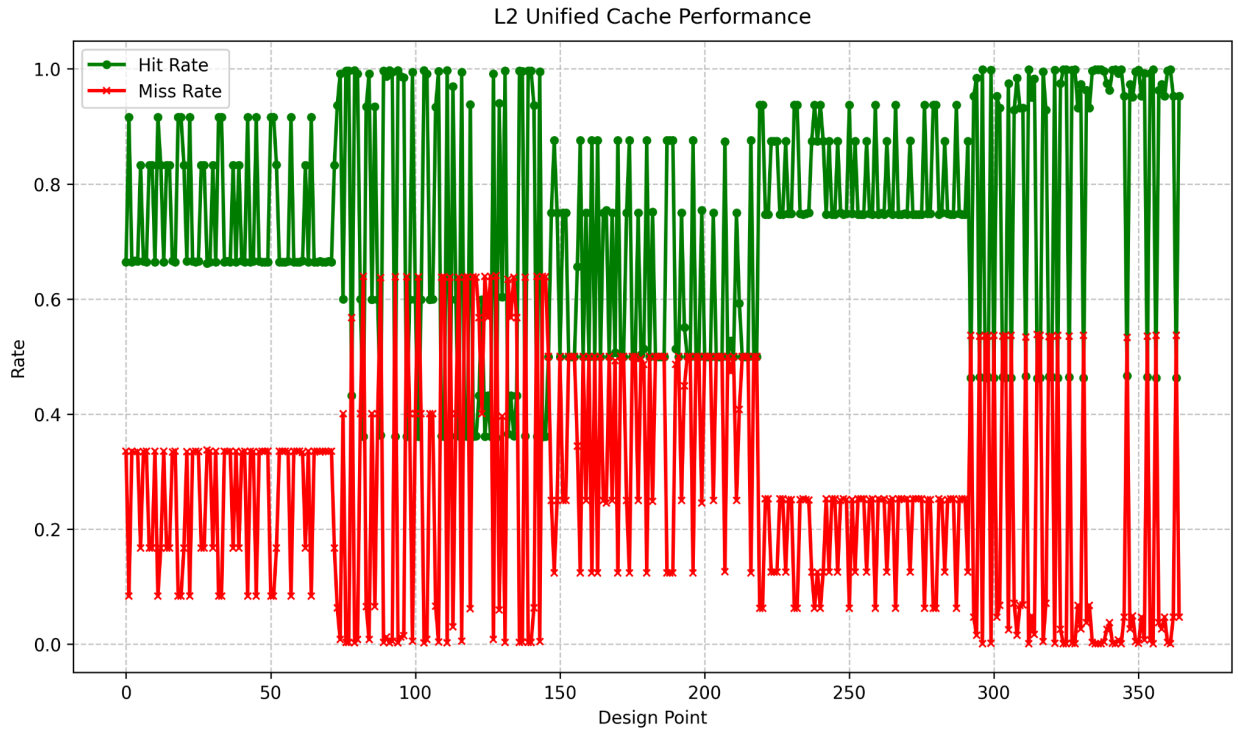
L1 Data Cache Hit/Miss Rate Count IMG 1.1



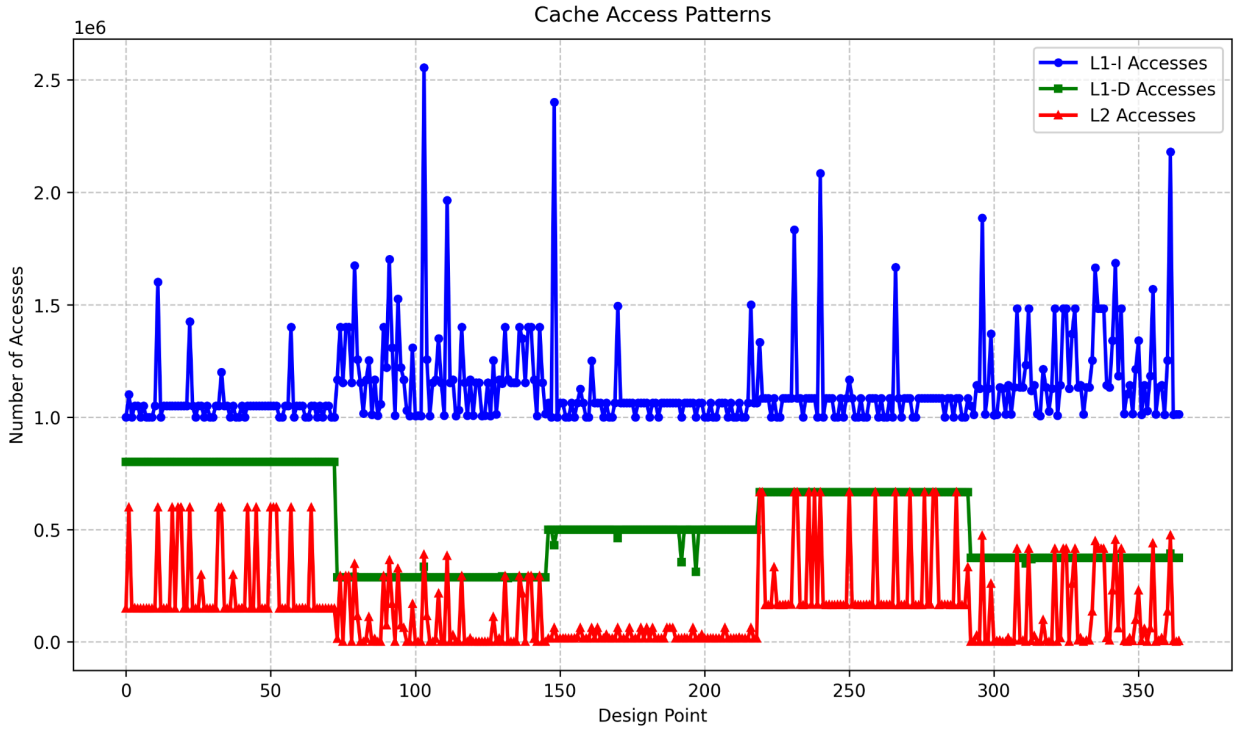
L1 Data Cache Hit/Miss Count IMG 1.2



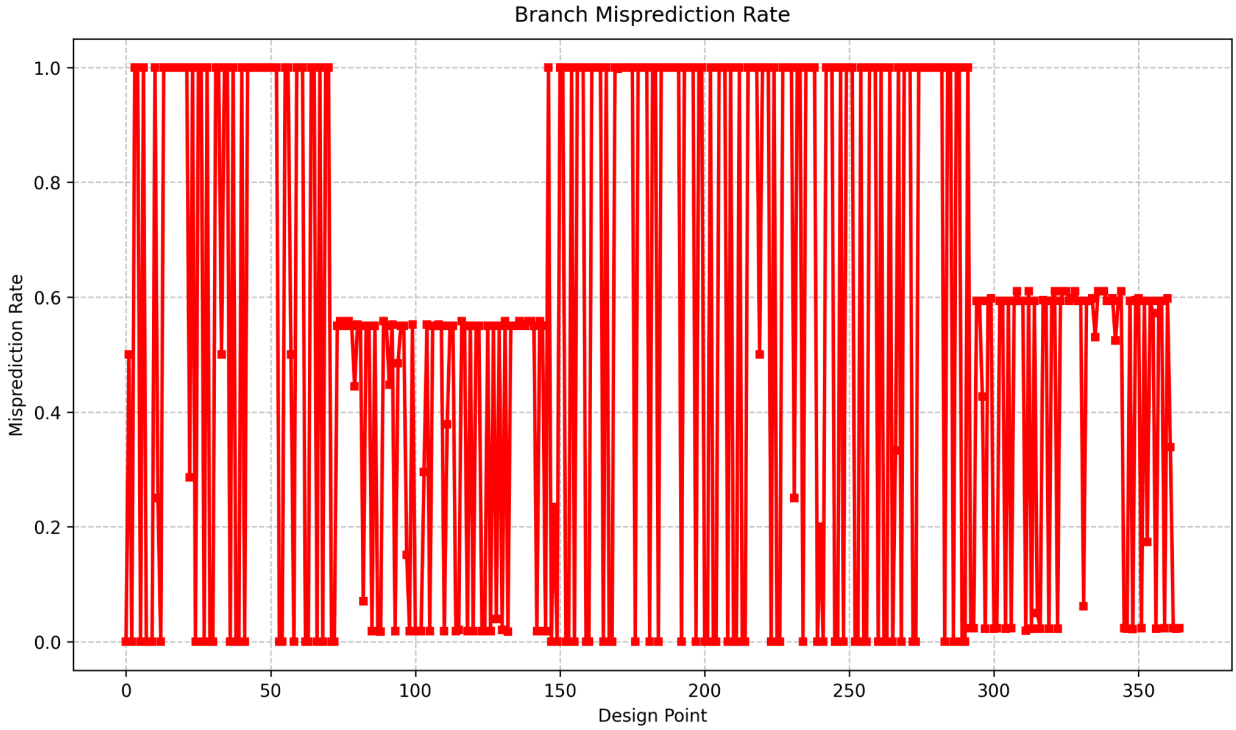
L2 Cache Hits/Miss Count IMG 1.3



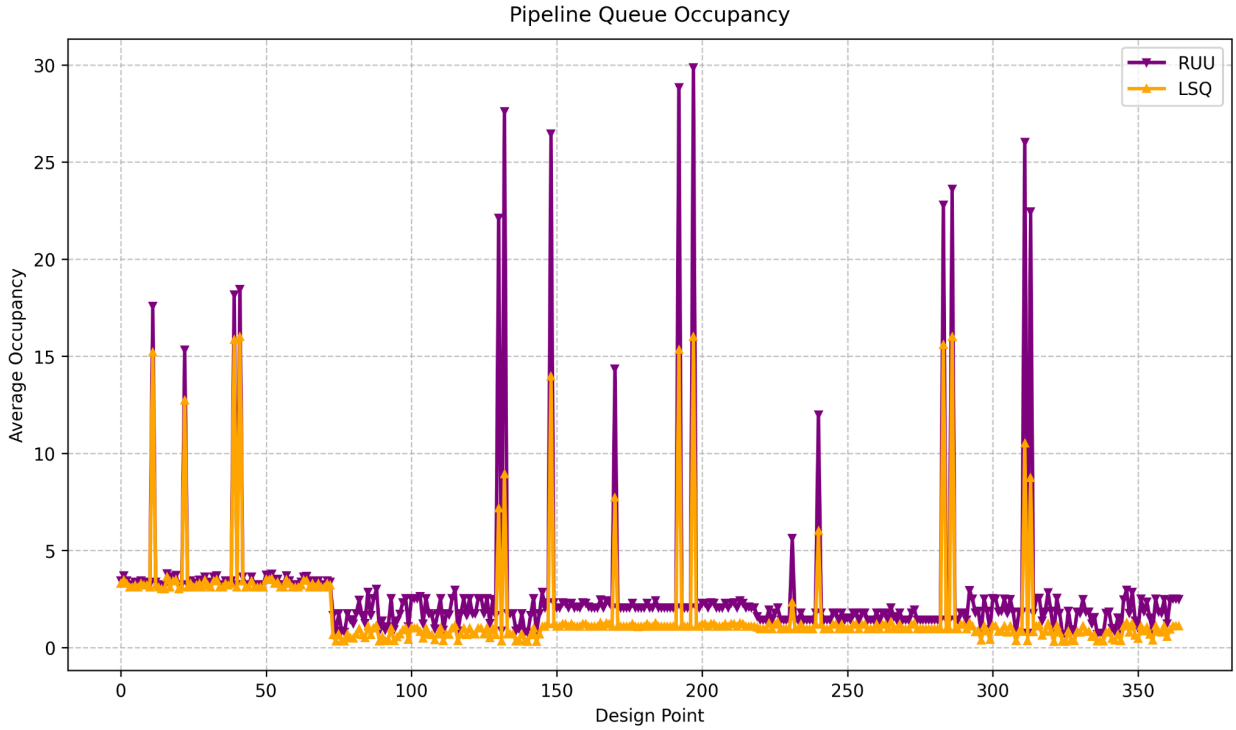
L2 Cache Performance IMG 1.4



Total Cache Access Per Design Point IMG 1.5



Branch Misprediction Rate Per Design Point IMG 1.6



Pipeline Occupancy Per Design Point (RUU vs LSQ) IMG 1.7

Heuristics Explanation

Heuristic Implementation Explanation:

a. Group Based Exploration:
    i. This is implemented from the exploration order we are assigned #20 (FPU→Cache → BP → Core)
b. Exploration Process
    i. Start with the baseline configuration
    ii. For each group (FPU → Cache – BP → Core)
        1. Test each value for the current parameter
        2. Check if the config is valid
        3. Evaluate the performance/energy benchmarks
        4. Keep track if the best seen configuration of said dimension
        5. Move to the next dimension once we've fully explored the current one
c. Dimension ordering & logic sequence
    i. I used the structure of ExplorationState to store information pertaining to where/what is being explored presently (also helps prevent the current exploration index from resetting to 0 when we test a new config)
d. Configuration generation
    i. Key thing for this function is that it tests different variations of the current dimension being explored, keeping only the valid configurations as well as ensuring that we don't repeat some of the same configurations (all while following the group order). One important aspect is that I implemented a tournament predictor for branch prediction to validate that a configuration will work in a related support function, this changed how my branch prediction worked and could be an improvement for a new heuristic on how I explored configurations!
        1. Tests one parameter at a time
        2. Use the previous good results as a guide on finding the next good value
        3. Follows group ordering
        4. Avoid combinations we've tried already
e. Configuration Validation
    i. I followed the rules outlined in the project instructions (sections 8.3) such as L1 & L2 block sizing being a minimum, cache size limitations, L2 inclusivity, and valid latency parameter marking.
        1. Just checks that rules are followed
f. Parameter Generation Logic (same as exploration process):
    i. Start with the baseline configuration we are given (i.e. start with baseline values for unexplored dimensions)
    ii. Try each possible value in the current dimension
    iii. Keep the best configuration seen
    iv. Move to the next dimension when the current dimension is fully explored
    v. Auto-Calculate dependent parameters (i.e. the cache latencies)
    vi. Repeat ii