Final Project Documentation

CSEE4824 Computer Architecture

John Graham
Yipeng Huang

In this report, we will present the software optimizations, hardware configurations, performance, and power analysis of four processor designs. The following section shows the configuration and results of our best hardware design (Design B), with an objective function of 351.4. From there, we will describe the steps we took to eventually achieve this value.

You may find our other configurations and corresponding performance tables (Designs A, C, D) at the end of the report, in the Appendix. We will be referring to those designs throughout the report.

Simulation Results

Design B—Multi Core Final
Configuration:

Core count	2
Core type	Small out-of-order cores
Issue width	1
Frequency	2.1 GHz
IL1 Cache Size	16384
IL1 Associativity	1
IL1 Access Time (ns)	0.326370006889
IL1 Access Time (cc)	1
DL1 Cache Size	32768
DL1 Associativity	8
DL1 Access Time (ns)	1.64704053756
DL1 Access Time (cc)	4
Cache Block Size	128

Performance:

	Power	Time	Speedup over base
simsmall	19.119W	5.277ms	352.8
simmid	18.965W	68.836ms	355.0
simlarge	18.596W	324.883ms	347.0
Objective Function			**351.4**

Software Optimizations

The final project called for two important methods in optimizing the matrix multiplication program: optimizing code and optimizing hardware. Before even running trials, we analyzed the code to determine where the number of computations could be reduced and where code could be made more efficient.

We started by looking at the brute force algorithm that was supplied to us, shown below. Looking at the two lines that are in bold, we determined that there are a lot of extraneous calculations in this brute force algorithm.

```
for(j=0; j<a[0]; j++)
{
	for(k=0; k<a[*b2+1]; k++)
	{
		B[*b2][j][k] = 0;
		for(l=0; l<a[*b2]; l++)
		{
			B[*b2][j][k] +=
			B[*b1][j][l] *
			A[*b2][l][k];
		}
	}
}
```

For every multiplication in the algorithm, this brute force method accesses a point in memory relative to a 3-dimensional array. The second line that is bolded is a good example of why this computation is not optimal. First of all, modifying `B[*b2][j][k]` for every iteration of `l` requires a computation of `*b2`, `j`, and `k` such that the memory address is relative to the `double ***B`. This alone uses several additions and multiplications to compute the address current address in memory. However, it's interesting to note that in the innermost `for` loop, the address of `B[*b2][j][k]` never changes because the loop is iterating over `l`, but the address in memory is being *recalculated* every time. Therefore, this is an extraneous calculation and can be eliminated. An optimized version of the brute force algorithm is displayed below.

```
for(j=0; j<a[0]; j++)
{
	for(k=0; k<a[*b2+1]; k++)
	{
		double tempNum = 0;
		double a1 = B[*b1][j];
		for(l=0; l<a[*b2]; l++)
		{
```

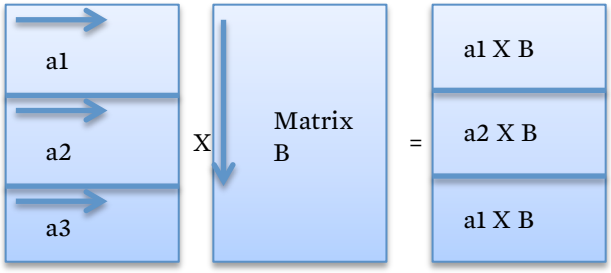
```
                    tempNum += a1[l]
                    * A[*b2][l][k];
                }
                B[*b2][j][k] = tempNum;
            }
        }
```

We added a temporary variable that keeps track of result values in the matrix. This allows for the removal of the calculation in the `for` loop over `l` that caused a lot of unnecessary calculations.

After running the new code with the default "NoL2SinglePower.conf" file, we were able to reduce the number of computations significantly using `tempNum`, which eliminates the 3-dimensional matrix calculations in every iteration of `l`, and plugs it into the matrix at the end of the computations. It reduced the number of instructions from 32,142,898 instructions to 19,204,179, a reduction by **40.3%** of the number of computations (time on a small matrix, for example, reduced from 1861ms to 1107ms). We tested on a default configuration because we could compare the results of code optimization to the default result. Reducing the number of instructions by 40.3% thereby reduces the time by 40.3% in single core architectures. This was a great start to the project, because it nearly reduces the computation times by half and we didn't even improve the hardware yet.

We were able to further reduce the number of computations by creating the variable `a1`, which replaces `B[*b1][j]` and eliminates more computations. The number of computations, for the small matrix configuration, was reduced even further to 16,951,049 instructions, decreasing the number of computations by **48.8%**.

After having optimized the code, we added multithreading capabilities that separated the matrix multiplications pretty evenly between the many threads. As suggested in the homework descriptions, we were able to have multiple cores working on a single matrix multiplication at a time. Each core computes a fraction of the resulting matrix independently, and each core computes the same amount of the resulting matrix and allows for the most optimal parallelization. The following diagram shows how the cores computed the matrix results:



Initial Findings

We began our experiments with initial trials to determine the effects of various configuration parameters. These were not directed experiments; rather, educated trial and error from what we learned in class. Because there were so many different hardware parameters, we attempted to narrow down the number of parameters so that we could experiment more thoroughly. The discoveries we made in this trial-and-error phase were incorporated into all of our final designs: A, B, C, and D discussed in this report. This section lays out what we found during this initial trial phase.

Core count

We tested out our multithreaded software with a multi-core architecture consisting of 1, 2, 3, and 4 small cores. All else being equal, increasing the core count improves performance because it allows for parallelization of instructions. However, each additional core consumed a significant amount of power. We had already noted that high frequency and out-of-order execution are both important for good performance, and maintaining a high core count ruled out the possibility of including these other improvements. Therefore, we chose to proceed with two designs: one dual-core design, and one single-core design.

Core selection

We considered using an architecture consisting of a mix of different cores. However, we found that our software would not be able to utilize the different cores to their full extent, which would require the program to create threads with different responsibilities. Matrix multiplication is not easily split between different roles that can be executed by different threads. Therefore, we chose to proceed with homogenous core architecture for the dual-core design.

Cache size

We noted from `report.pl` printouts that the instruction L1 cache rarely has a miss rate greater than .5%. Therefore, we chose to keep instruction L1 cache at the minimum size, 16384 bytes. We attempted to also keep data L1 cache at a minimum for the sake of conserving power. However, this caused the data L1 miss rate to exceed 1% at times, and caused a noticeable decrease in performance. Therefore, we chose to use a data L1 cache size of 32768 going forward.

L2 design

With the L1 cache miss rates below 1%, there is little benefit from including an L2 cache in the design. A multicore processor potentially benefits from having a shared L2 cache that allows cores to access data that other cores have recently accessed. However, due to the way our program divides up matrices among threads, our cores are rarely looking at data that is from similar locations in the memory. Therefore, we chose to not include any L2 cache in our designs going forward.

EXPERIMENTS

We used these initial findings and theory learned from class to come up with two candidate, high-performance designs—Design A, a base multi core design with an objective function of 159, and Design C, a single core design with an objective function of 132.

Then, we ran experiments with these two candidate cores to find out the effects of parameters we were less familiar with. The improvements were then applied to both of the candidate designs (Design A and C), which eventually morphed into the final designs Design B and Design D, where Design B became our best solution.

In-order vs. out-of-order

Based on what we learned in class, out-of-order issuing made a big difference in performance over in-order issuing. This allows processes to issue instructions before or after they were initially intended, to eliminate stalls. Therefore, we tested the an in-order configuration against an out-of-order configuration (on small matrices and using the default frequency of 100 Mhz) and saw the following results:

	power (W)	simsmall time (ms)
Single_InOrder.conf	1.848	1004.466
Single_OutOrder.conf	3.776	179.301

We were happily surprised to see that the out-of-order issuing decreased the time significantly at the small expense of adding about 2 watts to the power consumption. For such a small amount of additional power, there was a large gain in performance.

For all tests that followed this, we chose to keep out-of-order issuing.

Frequency

Next we looked at the frequencies of the in-order and out-of-order architectures. We learned from Homework 3 that increasing clock frequency is important for improving performance. To determine how in-order and out-of-order architectures respond to changes in frequency, we gradually increased the frequencies of the in-order and out-of-order architectures until the frequency could no longer be increased (for the in-order architecture) or we surpassed the 20W allowance (for the out-of-order architecture).

in-order	power (W)	simsmall time (ms)
900MHz	3.884	111.674
1000MHz	4.117	100.517
1100MHz	4.351	91.388
1300MHz	4.818	77.344
1500MHz	5.288	67.045
2000MHz	6.471	50.309
2500MHz	6.821	46.863

Once we reached a threshold of 2500 MHz, we saw minimal gains by increasing the clock frequency. As the clock frequency increased, the return on decreased time decreased. Now looking at the out-of-order architecture and increasing frequencies, we were only able to reach a maximum frequency of 1300MHz, which became Design C, before it surpassed the 20W maximum. But because instructions could be executed out of order, the overall time was much smaller:

out-of-order	power (w)	simsmall time (ms)
900MHz	14.422	19.969
1000MHz	15.728	17.978
1100MHz	17.035	15.728
1200MHz	18.334	14.989
1300MHz*	19.63	13.841

* *Single_OutOrder_1300MHz.conf (Design C)*

Analyzing strictly the clock frequencies and whether or not the architecture supported out-of-order issuing, we were able bring the objective function to 132.367 with just 1 core. Also, testing on medium and large matrices, we found this configuration to be scalable and always within the 20W budget. (See Design C in the appendix for more configuration details and performance).

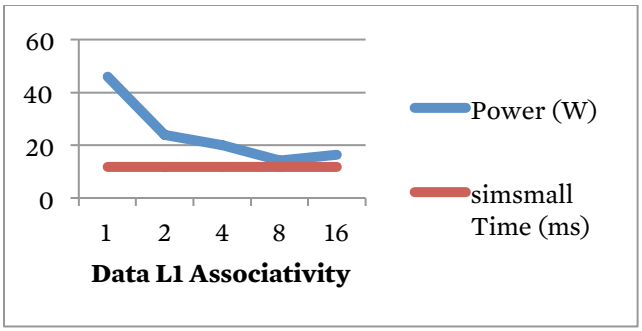
Because frequency has such a straightforward impact on the performance of a chip, in following design stages we chose to take draft designs and increase the frequency until we hit our 20W power budget.

Associativity

When we were running initial tests to arrive at Design A and Design C, we discovered that changing associativity had a large impact on how much power the chip consumed, which in turned influenced how fast we could clock the design. As an additional effect, associativity also influences the L1 miss rates, which in turn influences how frequent the processor has to stall for memory access.

This experiment was done using our dual core Design A, which is shown in the Appendix. We set both instruction L1 and data L1 associativity to 1, and varied the two parameters find out how associativity would influence performance and power consumption:

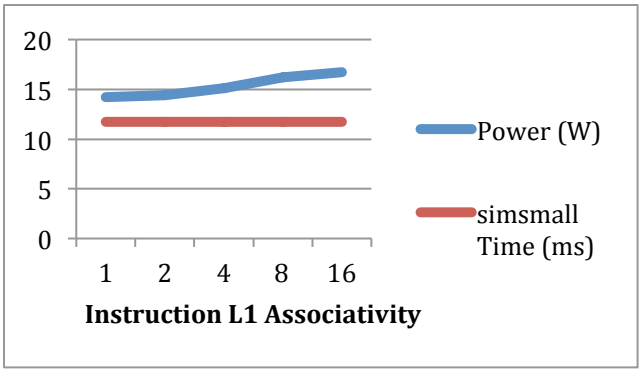
IL1 assoc.	DL1 assoc.	power (W)	simsmall time (ms)
1	1	46.041	11.729
1	2	23.996	11.738
1	4	19.905	11.738
1	8	14.194	11.738
1	16	16.293	11.738



We find that changing data L1 associativity does not significantly change the simulation time, however, there are significant power savings as the associativity approaches 8. We also noted that as the associativity increased, the cache miss rate of the data L1 decreased.

Next, we examine whether or not changing the associativity for instruction L1 influences power and performance.

IL1 assoc.	DL1 assoc.	Power (W)	simsmall time (ms)
1	8	14.194	11.738
2	8	14.442	11.735
4	8	15.111	11.734
8	8	16.182	11.734
16	8	16.721	11.734



In contrast to data L1 associativity, increasing instruction L1 associativity yields no improvement in power consumption. The instruction L1 miss rate was extremely low to begin with, and any potential increase in cache hit-rate was negligible.

Changes to instruction cache have less impact than those on data cache because our program accesses the data cache far more frequently than the instruction cache. Furthermore, because each core is accessing

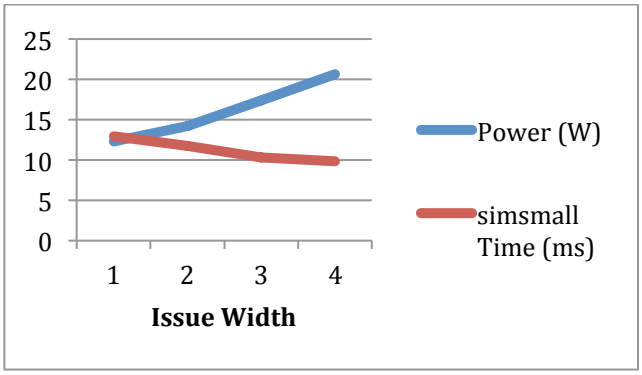
different parts of a matrix in a short time span, it is likely that data recently accessed in cache will still be useful in the near future. Hence, higher data L1 associativity aids performance and power.

Issue width

In our initial tests, we had set the issue width of our processors to 2 because our understanding of ILP told us that increasing the issue width would increase the IPC of the processor. However, we also noted that maintaining a high issue width incurred a large power cost. We decided to investigate whether or not a large issue width is a worthwhile configuration.

For this experiment, we took Design A, and augmented it to have the improved cache associativity we found in the previous section. We set IL1 associativity to 1 and DL1 associativity to 8. We varied the issue width of the processor:

issue width	power (W)	simsmall time (ms)
1	12.276	12.96
2	14.194	11.738
3	17.369	10.301
4	20.685	9.84



Increasing the issue width did increase the IPC, which is reflected in the decrease in simulation time as we increased the issue width. However, this benefit tapered off as we increased issue width from 2 to 3, as other factors such as cache misses started holding back the IPC gains.

The graph shows that increasing the issue width beyond 2 consumes a lot of power. Therefore, we decided that it is not worth the extra power to increase the issue width beyond two for future designs.

It is worth noting that this might only be true for small cores. With only one set of execution units, the small core likely does not benefit from larger issue widths as larger cores would. For example, we used an issue width of 2 to improve design C, which has a mid-sized core.

Cache block size

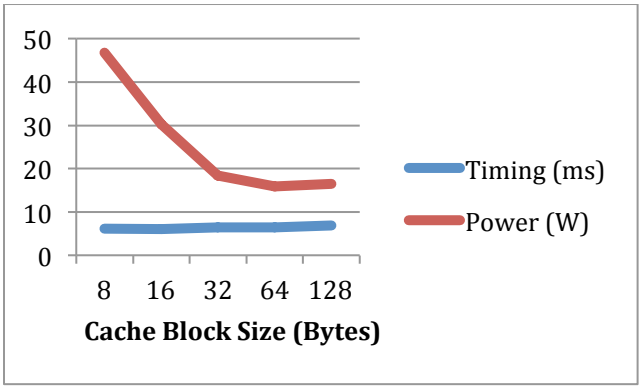
After experimenting with the associativity of the caches, we realized that the cache designs have a much larger impact on performance and power than we initially thought.

While none of the three simulations need large caches to execute with low cache miss rates, the arrangement of the caches themselves has a large influence on the power consumption of the chip.

Again, we took Design A with the augmented IL1 and DL1 associativity, and varied the cache block size:

block size	access time (cc)	timing (ms)	power (W)
8	1	6.085	46.833
16	1	6.048	30.267
32	2	6.476	18.472
64	2	6.466	15.866
128	3	6.831	16.444
256		CACTI fails*	

** A cache block size of 256 decreases the cache index below 32*



We found that the power consumption (red line in the previous graph) improves significantly as the cache block size nears 64 bytes.

This makes sense when taking into consideration how the program uses memory. The processor performs

matrix multiplication on a number that is not already in cache, it loads a block of numbers, which contains the numbers that the processor will need in subsequent cycles. Therefore, increasing the block size decreases the cache miss rate.

Single core testing with multi core updates

With more confirmation that higher frequencies are better, we looked into other combinations of hardware. From the multi-core architecture testing, we discovered that the results were better with a higher issue width.

Furthermore, prior testing revealed that modifying the data cache associativity to 8 (previously 1 or 2) significantly decreased the overall power consumption. With a much less power consumption, we could now incorporate a larger issue width and still use an extremely high clock frequency. We continued testing the single-core architecture with a medium-sized core.

This next configuration became Design C. It has a single core, 3000MHz frequency, issue width of 2, data associativity of 8, and instruction associativity of 1, and we achieved an objective function **259.527**. The configuration and performance analysis of this design, Design C, can be found in the Appendix.

Conclusions

This section discusses any general observations concerning tradeoffs of each of the hardware components available for system configuration.

	Effects on performance	**Power consumption**
In-order vs. out-of-order cores	Out-of-order cores have a significant performance improvement over in-order cores (Tomasulo's)	Does increase power slightly, but time gains outweigh power gains.
Core size	Mid-core showed improvements for single-core designs, but too power hungry for multi-core designs.	Bigger cores use significantly more power
Core count	More cores increases parallelization, but requires more power	It's directly related to the number of cores
Clock speed	Extremely important. Speed gains diminish as clock speed increases to very large numbers.	Increases almost linearly with clock speed.
Issue width	Improves performance noticeably, allows for multiple instructions issued at once	High power consumption, therefore difficult include in multi-core architectures.
L2	Negligible benefits because cores don't have much parallel data access.	Power hungry.
Instruction L1	Minimum L1 is sufficient through empirical testing, cache miss rate is low. Associativity should be 1	Minimal impact.
Data L1	Small L1 is bad because it significantly increases miss rate. Associativity is most efficient at 8	Size has minimal impact on power. Associativity has significant impact on power.

APPENDIX

Design A—Multi Core Base
Configuration:

Core count	2
Core type	Small out-of-order cores
Issue width	2
Frequency	740 MHz
IL1 Cache Size	16384
IL1 Associativity	1
IL1 Access Time (ns)	0.263321980197
IL1 Access Time (cc)	1
DL1 Cache Size	32768
DL1 Associativity	4
DL1 Access Time (ns)	0.464963823447
DL1 Access Time (cc)	1
Cache Block Size	32

Performance:

	Power	**Time**	**Speedup over base**
simsmall	19.905W	11.738ms	158.6
simmid	19.767W	152.73ms	160.0
simlarge	19.761W	702.364ms	160.5
Objective Function			**159.9**

Design C—Single Core
Configuration:

Core count	1
Core type	Mid out-of-order core
Issue width	1
Frequency	1.3 GHz
IL1 Cache Size	32769
IL1 Associativity	2
IL1 Access Time (ns)	0.433806432652
IL1 Access Time (cc)	1
DL1 Cache Size	32768
DL1 Associativity	2
DL1 Access Time (ns)	0. 433806432652
DL1 Access Time (cc)	1
Cache Block Size	32

Performance:

	Power	**Time**	**Speedup over base**
simsmall	19.63W	13.841ms	134.51
simmid	19.155W	184.354ms	132.54
simlarge	18.923W	859.628ms	131.147
Objective Function			**132.376**

Design D—Single Core Final

Configuration

Core count	1
Core type	Mid out-of-order core
Issue width	2
Frequency	3.0 GHz
IL1 Cache Size	16384
IL1 Associativity	1
IL1 Access Time (ns)	0.263321980197
IL1 Access Time (cc)	1
DL1 Cache Size	32768
DL1 Associativity	8
DL1 Access Time (ns)	0.577057051912
DL1 Access Time (cc)	1
Cache Block Size	32

Performance:

	Power	**Time**	**Speedup over base**
simsmall	17.619W	6.889ms	270.25
simmid	16.874W	94.178ms	259.45
simlarge	16.558W	443.42ms	254.246
Objective Function			**259.53**