Optimizing Machine Learning Algorithms for Stock Price Prediction

Ambrin Begam Riaz Ahmed, Palkpoom Pongsuphat, Jeremy Suon, Dayuan Tan
Computer Science and Electronic Engineering
University of Maryland, Baltimore County
Maryland, USA
{ambrinr1, palkpon1, jsuon1, dayuan1}@umbc.edu

I. Introduction

In today's business world, the stock market has become an important tool to measure how well a certain business is performing and pass those profits to its investors. The numerical values also help investors determine whether an investment is a worthwhile endeavor. Therefore, systems that allow them to make predictions are very desirable. One such method is to apply machine learning algorithms to past data sets to make mathematical predictions. However to apply such algorithms in real-time may be difficult due to the time sensitive nature of the stock market. Which is why we propose a few optimizations that may be included in a specialized processor. Our suggestions include optimizations the System Organization and an addition to the ISA that includes a corresponding circuit and its HDL functional equivalent.

II. Characteristics of the Application

For the examination of the prices, we propose the use of the machine learning algorithms: Linear Regression and K-nearest neighbor as our methodology in predicting stock prices based on past data. We chose linear regression amongst other machine learning algorithms due to the simplicity of its implementation as well as the versatility of the algorithm in its use in machine learning in general. K-nearest neighbor was selected due to its capabilities in identifying and relating past trends to future predictions which can better account for the fluctuations in stock prices. By considering more than one algorithm, we were able to make our suggestions to the System Organization and HDL functional modules more robust.

A. Data collected

The data set used was taken from Kaggle and consists of 506 companies along with their history stock prices from February 8th, 2013 to February 7th, 2018. The exact number of data points for an individual company varies. Each company has open price, high price, low price, close price and volume for each day. We will use those history price and volume to predict the price and volume of February 8th, 2018, using both linear regression and K-nearest neighbor.

B. Design influenced by data

For the linear regression algorithm, the variance and covariance is calculated for the previous days to find values for the equation y = b1*x + b0, where x is the day and y is the price. The number of prior days used varies according to the accuracy able to be obtained through the equation formed. B1 represents the slope in the equation whereas b0 is representative of the bias. B1 is calculated by dividing the sum of cross-deviations of y and x by the sum of squared deviations of x. That is, representing the sum of cross-deviations of y and x as SS_XY, and the sum of squared deviations of x as SS_XX with a total size of n..

$$SS\_XY = \sum y * x - n * avg(x) * avg(y) \tag{1}$$

$$SS\_XX = \sum x^2 - n * (avg(x))^2 \tag{2}$$

$$B1 = SS\_XY/SS\_XX, \tag{3}$$

$$B0 = avg(y) - B1 * avg(x). \tag{4}$$

For the K-nearest neighbor algorithm, every ten days prices are treated as a set, named history sets HIS[j], where j is the number of which ten days. Then we will calculate the difference between each continuous two days by subtract the second days price or volume with first days price or volume. So we get nine differences for each HIS[j], named as HIS_DIFF[j]. The last nine days prices and volumes will be arranged as a special set TARGET with the predicted price and volume of February 8th as tenth days price and volume. We will also calculate the differences between each continuous two days for first nine elements of TARGET as what we do for HIS[j]. Let us name those nine differences TARGET_DIFF, which will be matched with history sets HIS_DIFF[j] and the best matched one of history sets will be selected.

We use euclidean distance as match value and the match formula we use is as follows:

$$\begin{aligned} &Match\_Value[j] \\ &= sqrt(\sum_{i=0}^{7}(TARGET\_DIFF[i] \\ &- HIS\_DIFF[j][i])^2). \end{aligned} \tag{5}$$

One of HIS_DIFF[j] which has minimum match_value will be selected, and its corresponding HIS[j] will be recognized as the best matched history set, named HIS[best_matched].

Then we use the tenth days price and volume of the best matched history set, i.e. HIS[best_matched][9], as the predicted price and volume of February 8th. So each time when we calculate a companys predicted price and volume we need to match lots of history sets, leading to majority of execution time is spent on reading data. In order to make us better focus on analysing computation part, only 5% selected randomly of those 506 companies were used in the interest of time.

C. Quantitative Characteristics

For the analysis, both algorithms were implemented in C++ and compiled using g++. Table 1 and Table 2 list some quantitative characteristics of our code collected by Valgrind with cachegrind tool.

TABLE I
QUANTITATIVE CHARACTERISTICS OF THE LINEAR REGRESSION CODE

Total Number of Instructions:	73,058,685,219
Total Number of Load Instructions:	36,217,050,754
Total Number of Store Instructions:	32,867,578,794
Total Number of conditional branches:	12,282,295,005
Total Number of mispredicted conditional branches:	18,584,208
Total Number of indirect branches:	234,979,770
Total Number of mispredicted indirect branches:	11,276,437

TABLE II
QUANTITATIVE CHARACTERISTICS OF THE K-NEAREST NEIGHBOR CODE

Total Number of Instructions:	8,644,674,639
Total Number of Load Instructions:	3,055,248,933
Total Number of Store Instructions:	2,272,661,476
Total Number of conditional branches:	478,626,875
Total Number of mispredicted conditional branches:	8,424,809
Total Number of indirect branches:	223,685,852
Total Number of mispredicted indirect branches:	627,203

III. DESCRIPTION OF THE DESIGN

A. Floating Point Squaring

After looking at the machine learning algorithms, there seems to be a reasonably high occurrence within loops where a floating point number had to be squared. Therefore, an additional instruction could be added to the ISA specifically for floating point squaring with a more optimized circuit inserted in the Floating Point Arithmetic Units to support it. Figure 1 shows a functional block diagram of such a Floating Point Square Unit.

The "Exponent Multiply" doubles the value of the exponent while accounting for the bias used in IEEE-754 Floating Point Format. The 24-Bit Square block squares the mantissa with the implicit '1' being used in the calculations. The most significant bit of the result is used to determine if the exponent needs to be incremented and to determine if a left shift is necessary.

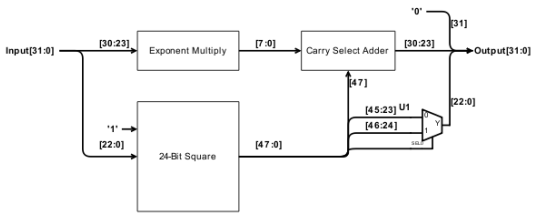

Fig. 1. 32-Bit Floating Point Squaring Circuit.

These corrections are necessary to maintain the placement of the decimal point in keeping with IEEE-754 Floating Point Format. The Carry Select Adder being used instead of a ripple adder is used in the interest of reducing the critical path. A Carry Select Adder basically computes both possible sums, in this case a carry of '0' or a carry of '1', where a multiplexer selects the correct value once the carry is finished calculating. However the critical path of this circuit is in the 24-Bit Square block, with the rest of the circuit being largely the same as a normal Floating Point Multiplier.

After researching improvements that could be done to make a more efficient Binary Squaring Circuit, the following paper was found *An Improved Squaring Circuit for Binary Numbers* by Kabiraj and Rutuparna [1]. Their design applies the ideas of Vedan Mathematics such that the design uses small adds done in parallel with larger multiplier being built hierarchically. However their template is meant to be used for bit-sizes that are powers 2 therefore some modifications were necessary. Using the functional blocks shown in the paper, the circuit shown in Figure 2 was implement and tested. Figures 3 and 4 are taken from [1], they show the template used for making n by n bit multiplier and n-bit square circuits respectively.

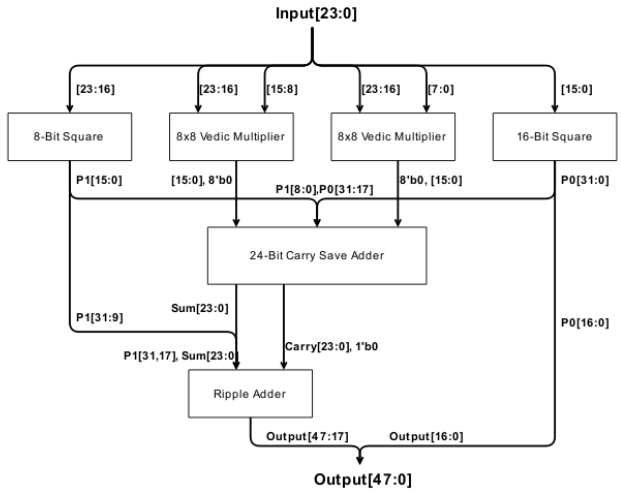

Fig. 2. 32-Bit Floating Point Squaring Circuit.

To analyze the relative speed of this new design, it is assumed that a Wallace Tree design is normally used for the

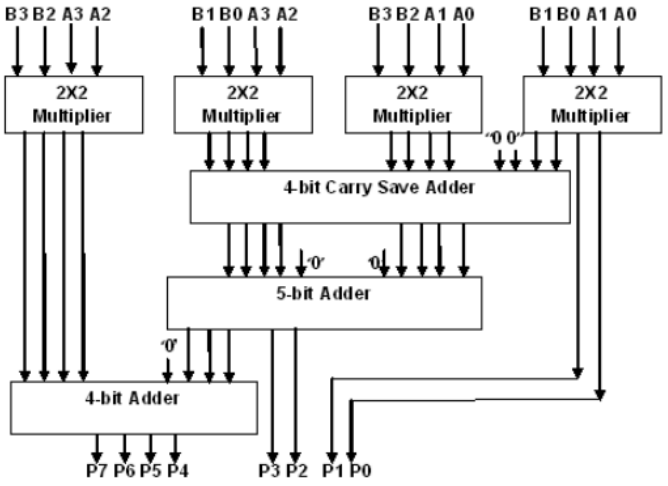

Fig. 3. 32-Bit Floating Point Squaring Circuit.

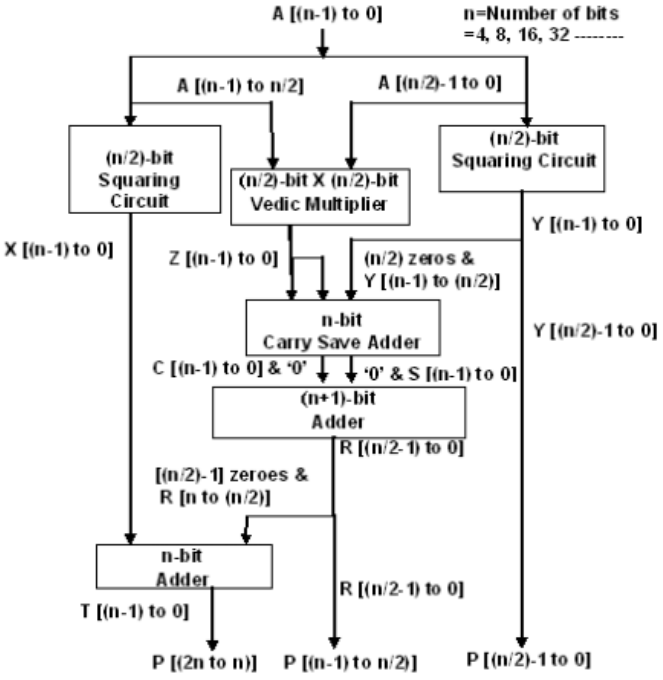

Fig. 4. 32-Bit Floating Point Squaring Circuit.

multiplier. The Wallace Tree design uses stages of Full Adders and Half Adders to reduce the pyramid of digits of a multiply operation and then uses a single long ripple adder to finish the calculation. As opposed to the design proposed in [1] where shorter ripple adds are done in parallel with a single stage of Full Adders are done in between. An example of a Wallace Tree is shown in Figure 5, taken from Wikipedia.

In analyzing the critical path of both designs, a Half Adder is assumed to be one half of a delay unit and a Full Adder is one delay unit. For the conventional 24-Bit Wallace Tree Multiplier, after implementing and testing, the design requires 7 stages of Full Adders followed by a 40-Bit ripple adder. For the new design, because the design is hierarchically the

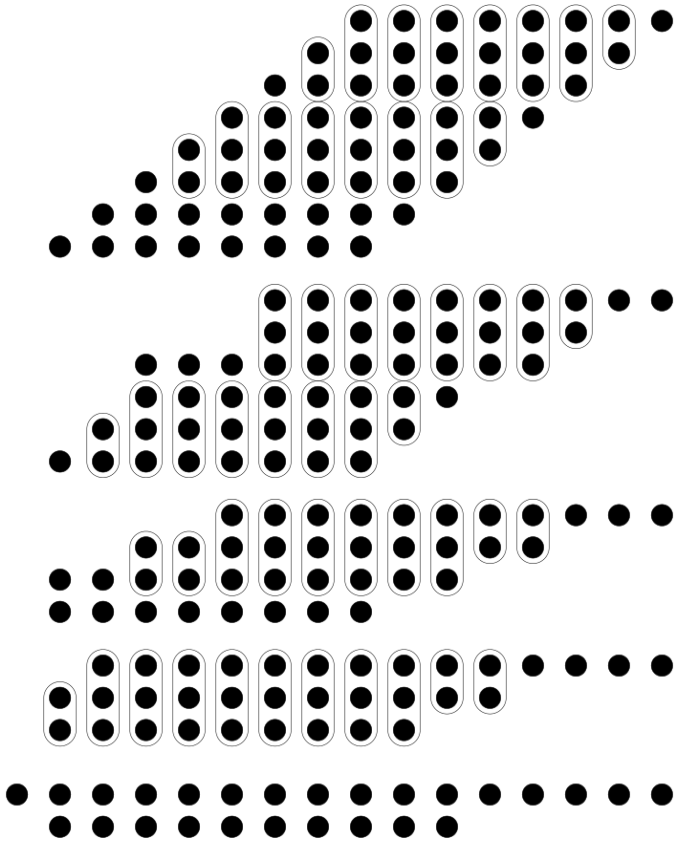

Fig. 5. 32-Bit Floating Point Squaring Circuit.

critical path delays are enumerated in Table 3. The table only includes the delays of the critical path instead of the entire circuits.

TABLE III
24-BIT SQUARE CRITICAL PATH DELAY

24-Bit Square	4 + 46-Bit Ripple
30-Bit Ripple Adder	30-Bit Ripple
1 CSA Stage	1 Unit
8-Bit Vedic Multiplier	3 + 16-Bit Ripple
11-Bit Ripple Adder	11-Bit Ripple
1 CSA Stage	1 Unit
4-Bit Vedic Multiplier	2 + 5-Bit Ripple
5-Bit Ripple Adder	5-Bit Ripple
1 CSA Stage	1 Unit
2-Bit Vedic Multiplier	1 Unit
2 Half Adders	1 Unit

As you can see, the speed-up is highly dependant on the quality of the ripple adder used, with the 2 delays becoming equal when using a ripple adder that has a delay of 0.5 units per bit and the speedup becoming 1.75x as the adders delay approaches 0. The delay of a standard ripple adder is a half adder per bit which is 0.5 units in these calculations. If you use carry select adders to cut delays in half, then the speedup becomes 17 / 15.5 = 1.09677x. There are also additional adder topographies like the carry save adder and its variants that are not considered that should increase the speed up further.

However, the precision of the clock also needs to be considered to determine whether this speed-up is effective or not. In the Intel Nehalem i7, the latency of the mulss xmm instruction is only 4 cycles which requires a speed-up of 1.333x to remove 1 clock cycle. The non-SIMD fmul instruction of the Nehalem i7 is 5 cycles which requires a speed-up 1.2x to remove 1 cycle. In either case, a simple adder with the new squaring circuit will not remove any clock cycles. The delay of the of adders would have to be approximately 0.078125 and 0.1447 delay units for these cases, respectively, to remove 1 cycle.

B. Cache Resizing

Our program for the prediction of stock prices using the KNN machine learning algorithm has three functions: str2flt, str2int and compEuclideanDistFloat. All of these functions include operations on vectors. The valgrind tool is used to profile the code. We found that the std::vector$\langle float \rangle$ function contributes to 67.3% of the total execution time and that this function has the most cache misses. Hence, we focused on reducing the cache misses of our code in order to improve the overall execution time of the code.

We also found that the malloc system calls in the functions contribute to the second most cache misses. Malloc system calls contribute to 12% of the total execution time. Hence, we used cachegrind and the cg_annotate simulation tool to view the current cache strategy used by the processor and determine which cache needs changes. The cg_annotate simulation shows that our code uses three types of cache: I1 cache, D1 cache and LL cache. I1 refers to the L1 instruction cache, D1 refers to the L1 data cache and LL refers to last level of cache.

Definitions of more terms needed in following parts are listed:

- Ir : I cache reads (instructions executed)
- I1mr: I1 cache read misses
- ILmr: LL cache instruction read misses
- D1mr: D1 cache read misses
- DLmr: LL cache data read misses
- D1mw: D1 cache write misses
- DLmw: LL cache data write misses
- L1m: Total L1 cache misses
- LLm : Total LL cache misses

Old Cache:

- I1 cache: 32,786 B, 8-way associativity.
- D1 cache: 32,786 B, 8-way associativity.
- LL cache: 6.29 MB, 12-way associativity.

Clock cycles are calculated in cachegrind using these formulae from Valgrind:

$$L1m = I1mr + D1mr + D1mw. \tag{6}$$

$$LLm = ILmr + DLmr + DLmw. \tag{7}$$

$$Estimatedclockcycles = Ir + 10 * L1m + 100 * LLm. \tag{8}$$

Improved Cache:

- I1 cache: 65,536 B, 8-way associativity.
- D1 cache: 65,536 B, 16-way associativity.
- LL cache: 6.29 MB, 12-way associativity.

Based on Table IV and V we can get

$$\begin{aligned} &Speedup\ of\ Malloc\ function\ calls \\ &= Old\ cycles/New\ cycles \\ &= 1.094. \end{aligned} \tag{9}$$

Comparing total cycles with old cache and new cache for KNN algorithm, some results can be achieved, which are shown in Table VI. Based on Table VI we can get

$$\begin{aligned} Overall\ Speedup &= Old\ cycles/New\ cycles \\ &= 1.03. \end{aligned} \tag{10}$$

Our program for the prediction of stock prices using the linear regression algorithm has three functions: get_b1, get_b0 and output. The valgrind tool is used to profile the code and we found that the output function contributes to 27% of the total execution time and that this function has the most cache misses. Also the malloc system calls in each function have the most cache misses. Hence, we focused on reducing the cache misses of our code in order to improve the overall execution time of the code.

Cachegrind and the cg_annotate simulation tool are used to view the current cache strategy used by the processor and determine which cache needs changes. The default configuration of the cache is viewed using cg_annotate.

Old Cache:

- I1 cache: 32,786 B, 8-way associativity.
- D1 cache: 32,786 B, 8-way associativity.
- LL cache: 6.29 MB, 12-way associativity.

Cache size can be changed while running valgrind by passing new cache size arguments as explained in previous section.

Improved Cache:

- I1 cache: 65,536 B, 8-way associativity.
- D1 cache: 65,536 B, 16-way associativity.
- LL cache: 6.29 MB, 12-way associativity.

Based on Table VII and VIII we can get

$$\begin{aligned} &Speedup\ of\ Malloc\ function\ calls \\ &= Old\ cycles/New\ cycles \\ &= 1.096. \end{aligned} \tag{11}$$

Comparing total cycles with old cache and new cache for Linear Regression algorithm, some results can be achieved, which are shown in Table IX.

$$\begin{aligned} Overall\ Speedup &= Old\ cycles/New\ cycles \\ &= 1.003. \end{aligned} \tag{12}$$

TABLE IV
OLD CACHE FOR KNN

Function	Ir	I1mr	ILmr	D1mr	DLmr	D1mw	DLmw	cycles
Malloc.c_int_free	126,989,280	1,066	3	1,140	0	56	0	127,012,200
Malloc.c:_int_malloc	92,203,772	120,520	62	2,597	5	952	146	93,465,762
Old cycles							Total	220,477,962

TABLE V
IMPROVED CACHE FOR KNN

Function	Ir	I1mr	ILmr	D1mr	DLmr	D1mw	DLmw	cycles
Malloc.c_int_free	109,278,681	43	22	7	0	0	0	109,281,381
Malloc.c:_int_malloc	92,203,772	3,447	62	32	5	173	146	92,261,592
New cycles							Total	201,542,973

TABLE VI
COMPARISON OF OLD CACHE WITH NEW CACHE

KNN Algorithm	Ir	I1mr	ILmr	D1mr	DLmr	D1mw	DLmw	cycles
Old cache (D1- 8 way)	1,673,000,162	5,199,178	2,458	65,530	6,675	8,978	1,553	1,726,805,622
New cache (D1- 16 way)	1,673,000,162	95,131	2,458	9,883	6,675	2,007	1,553	1,675,138,972

TABLE VII
OLD CACHE FOR LINEAR REGRESSION

Function	Ir	I1mr	ILmr	D1mr	DLmr	D1mw	DLmw	cycles
Malloc.c_int_free	317,124,576	12,172	22	1,494	0	6	0	317,263,496
Malloc.c:_int_malloc	250,628,344	44,068	62	8,109	5	3,968	51	251,201,594
Old cycles							Total	622,465,090

C. Branch Prediction

For k-nearest-neighbor, based on Table 2, we can get the total branch misprediction rate is

$$\begin{aligned} &Total\ branch\ misprediction\ rate \\ &= (conditional\ mispredicted\ branches \\ &\quad + \ indirect\ mispredicted\ branches)\ / \\ &\quad (conditionalbranches + indirectbranches) \\ &= (8,424,809 + 627,203)/ \\ &\quad (478,626,875 + 223,685,852) \\ &= 1.3\%. \end{aligned} \tag{13}$$

Even though the rate is small but we still think it is worth to do some improvement for this part. So we investigated each function of our program and figured out branch misprediction in compEuclideanDistFloat function takes 8.86% of all branch misprediction, while other functions which take more than 8.86% are C library functions.

Focusing on compEuclideanDistFloat function, it has two conditional branches, one is a FOR loop and another one is IF branch. For the FOR loop, it calculates the Match_Value[j] by calculating the sum of euclidean distances of TARGET_DIFF and each element of HIS_DIFF[j], shown in (5). Since we have only eight elements in TARGET_DIFF so actually the iteration times of this FOR loop is always eight times. Which also means FOR branch takes 8/9 of all branches in compEuclideanDistFloat function, since IF branch is a conditional branch which will be executed once each time when this function is called.

A loop predictorccan be used at the FOR branch part, even harding code this part by repeating the inside part of formula (1) eight times works. It should be like this:

```
Sum += TARGET_DIFF[0]-HIS_DIFF[j][0]
Sum += TARGET_DIFF[1]-HIS_DIFF[j][1]
Sum += TARGET_DIFF[2]-HIS_DIFF[j][2]
Sum += TARGET_DIFF[3]-HIS_DIFF[j][3]
Sum += TARGET_DIFF[4]-HIS_DIFF[j][4]
Sum += TARGET_DIFF[5]-HIS_DIFF[j][5]
Sum += TARGET_DIFF[6]-HIS_DIFF[j][6]
Sum += TARGET_DIFF[7]-HIS_DIFF[j][7]
Match_Value[j] = sqrt{sum}
```

In this case a branch predictor is not needed at the FOR branch part, leading to its branch misprediction reduce to 0. So we focused on FOR part and obtained the branch amount and midprecited branch amount of this part using same analysing method we used for compEuclideanDistFloat function. The analysing result of this FOR branch part is shown in Table X.

The misprediction penalty of an Intel i7-6700 (Skylake) is 16.5 cycles on average form [2]. With the total penalty for branch misprediction of FOR part in compEuclideanDistFloat function being 12490 * 16.5 = 206,085 cycles.

TABLE VIII
IMPROVED CACHE FOR LINEAR REGRESSION

Function	Ir	I1mr	ILmr	D1mr	DLmr	D1mw	DLmw	cycles
Malloc.c_int_free	317,124,576	23	22	410	0	0	0	317,131,106
Malloc.c:_int_malloc	250,628,344	196	62	782	5	378	51	250,653,704
New cycles							Total	567,784,810

TABLE IX
COMPARISON OF OLD CACHE WITH NEW CACHE FOR LINEAR REGRESSION

KNN Algorithm	Ir	I1mr	ILmr	D1mr	DLmr	D1mw	DLmw	cycles
Old cache (D1- 8 way)	5,744,287,675	871,979	2,180	477,042	6,384	452,498	2,137	5,763,372,965
New cache (D1- 16 way)	5,744,287,675	35,980	2,180	83,822	6,384	73,037	2,137	5,747,286,165

TABLE X
MISPREDICTED BRANCHES IN FOR PART

Branch misprediction penalty:	16.5 cycles on average
Mispredictions in compEuclideanDistFloat FOR Loop:	12,490

From Table 2 and (8) we can get the estimated total cycles is 8,860,652,742.

As we analysed above, the branch misprediction of FOR part in compEuclideanDistFloat function is eliminated, so we can obtain

$$\begin{aligned} &Overall\ SpeedUp \\ &= \frac{time_{old}}{time_{new}} \\ &= \frac{Instruction\ amount * CPI * Cycletime}{Instruction\ amount * CPI * Cycletime} \\ &= \frac{total\ cycles\ amount_{old}}{total\ cycles\ amount_{new}} \\ &= \frac{8,860,652,742}{(8,860,652,742 - 12,490)} \\ &= 1.0000014096. \end{aligned} \tag{14}$$

IV. CONCLUSION

We found that both of these algorithms have significant cache misses and we have proposed changes to cache associativity to improve cache misses. We found that for KNN and linear regression, overall improvement of up to 3.00014096% and 0.3% respectively is possible. While the results of an improved Square function may not necessary yield any improvement, it also should not give any penalties. Therefore the design is worth a certain amount of consideration.

REFERENCES

[1] Kabiraj Sethi and Rutuparna Panda, An Improved Squaring Circuit for Binary Numbers International Journal of Advanced Computer Science and Applications(IJACSA), 3(2), 2012. http://dx.doi.org/10.14569/IJACSA.2012.030220.

[2] Skylake. https://www.7-cpu.com/cpu/Skylake.html.