AN EXPERIMENTAL STUDY OF SORTING AND BRANCH PREDICTION (A detailed yet easy report)

PAUL BIGGAR, NICHOLAS NASH, KEVIN WILLIAMS, and DAVID GREGG

(Trinity College Dublin)

Divye Gupta

B.Sc. Student

Saint Mary's University

Kashif Kashif

B.Sc. Student

Saint Mary's University

IMPORTANT NOTE: *All the definitions are defined within the paragraphs itself. The sequence of the paragraphs written in this report matches the sequence of the paragraphs written in the original paper. Therefore, this report is written in paragraph-by-paragraph format.*

2.0 INTRODUCTION:

Sorting is a process of arranging any randomly ordered set of items into an ordered sequence. In the world of technology, sorting is considered to be a common operation used in developing many applications and hence efficient algorithms are being developed to perform them. Because of these reasons, sorting is considered to be one of the most important and well-studied problems in computing science.

The author describes that any good algorithm is known for maintaining its balance in areas between its efficiency, simplicity, memory use and other factors. However, these so-called good algorithms fail to take into account of the importance of modern computer architecture that could play a major role in increasing performance. The two such features of modern computer architecture include **caches** (*component that stores data, so that future requests for that data could be served faster*) and **branch predictors**. Though there has been a good amount of research in cache performance of general sorting algorithms, for some reason the importance of branch prediction properties is ignored.

This paper includes study of behaviour of the branches in the most commonly used sorted algorithms, the communication of cache optimization on branch predictability. The paper further covers how the insertion sort has fewest branch mispredictions in terms of comparison-based algorithms in relation with 'bubble' and 'shaker' sort, and how the cache behaviour could be improved.

2.1 MOTIVATION:

A classical algorithm analysis on early computers has a high possibility result in precise predictions of running times. However this case is not the same with modern systems and algorithms as they are much more complex.

The author gives a supporting example on how classical analyses of algorithms make simplifying assumptions about the cost of different machine instructions. For instance,
RAM Model (*also known as Real RAM - random access machine, is a computational model that operates real/irrational numbers in a true mathematical manner, whereas standard computers only support approximate computations with floating point arithmetic/exact arithmetic which is restricted to real integers/rational numbers*) used for establishing **asymptotic Bounds** (*a curve representing the limit of a function, where distance between a function and the curve tends to zero*) and **Knuth's MIX Machine** (*imaginary computer invented by Don Knuth in 1960s. Knuth wanted to explain how an imaginary computer and machine language helps avoid distracting the reader with the technicalities one particular system, and the focus on the valid truths will always be independent of any kind of technical evolution*) **code** make extreme simplifying assumptions about the cost machine instructions. In recent findings, researchers have concluded that on modern computers the memory accessing cost can vary considerably depending on **first level cache**(*the session cache caches object within the current session*), lower level cache or even main memory of the machine. This has led to further research in the field of cache-efficient searching and sorting.

In a computer program, instructions are executed in a sequence by default. An exception to this is a **'branch'** or **'jump'** instruction that tells the computer to skip a particular instruction and begin execution of different part of a program. High level language statements such as '*if-else*' are used to express the **conditional branch**.

The author states that the conditional branch can lead to a dramatic variation in its cost of execution. **Modern pipelined processors** (*series of processing stages of computer instructions that are arranged linearly or dynamically to perform a specific function*) are reliant on **branch prediction** for most of their performance properties. If the direction of a conditional branch is predicted correctly, that too ahead of time, it will lead to a very low cost of conditional branch. Whereas if the branch is not predicted correctly, the processor needs to clean the pipeline first and then it requires restarting from the 'actual' correct target of the branch. This misprediction of branch involves a very high cost, which is typically a multiple of the cost of executing a correctly predicted branch. For instance, Intel Pentium 4 processors have a pipelines of up to 31 stages, that means a single branch misprediction can lead to 30 cycles! But luckily, branches in

most of the programs are highly predictable and therefore the sights of branch mispredictions are a bit rare.

The cost of executing branches plays an important factor when it comes to sorting, simply because the inner loops of most sorting algorithms contain total comparisons of items to be sorted. Therefore the performance of sorting algorithms is heavily dependent on the predictability of these comparison branches. The author discusses that this paper further includes the analysis on the behaviour of the branches whose outcome is dependent on a comparison of 'keys' that is included in the sorting algorithm in its input. Since branches associated with controlling simpler aspects of the control flow of algorithms are 'almost perfectly predictable', they are of much less interest.

2.2 BRANCH PREDICTION

As we had already described in the section 2.1 what 'branches' actually mean, the author states the same thing in the first paragraph of this section. The author adds on that the branches can be either **'taken'**, meaning that the address they provide is the new value for the program counter (leading to *dynamic/non-linear pipelining*), or they can be **'not taken'**, meaning that the sequential execution continues as though the branches didn't exist (leading to *linear pipelining*).

In the second paragraph the author tries describe why branch instructions can create a difficult scenario for pipelined processors. In a pipeline processor, while the program counter points to a specific instruction, huge numbers of subsequent instructions are in an incomplete (partially complete) state. Until the processor isn't aware whether the branch is 'taken' or 'not taken', it is impossible to know the next to-be executed instructions. Therefore, since the processor cannot tell what these next instructions are, it cannot fill its pipeline. To overcome this hurdle, modern processors keep the utilization of the pipeline at a considerable level by anticipating the outcome of branch instructions through **branch predictors**.

The author describes the various types of branch predictors, namely, static, semistatic and dynamic. Static branch prediction is the simplest branch prediction technique mainly because it

is *not dependent on the information about the dynamic history of code executing*. It predicts the outcome of branch by solely relying on branch instruction. This kind of branch predictor always predicts the same direction for a branch whenever it is executed. It's heuristic approach involves predicting forward branches as 'not taken' and backward branches as 'taken'. However, *static branch predictors are less common in modern processors*. A semistatic branch predictor involves a '**hint bit**', that helps the compiler to find the prediction direction of the branch. On the other hand, dynamic branch prediction relies on the information about 'taken' or 'not taken' branches collected at run-time to predict the outcome of a branch. As stated earlier, static branch predictors are insignificant when they are compared with the dynamic branch predictors. Due to this reason, the latter is most commonly used in real processors. The simplest type of branch predictor is the 1-bit predictor which has *1 bit counter that records the last outcome of the branch*. It is responsible for keeping a table (at each entry) that categorizes the branches into 'taken' or 'not taken'. For example, if a branch is taken, it will be given a value of 1, and if the branch is not taken it will be assigned a value of 0. Using this table data, it is able to predict whether a branch will go the same way as it went on its previous execution. These kinds of branch predictors have an accuracy range of 77% to 79%.

The **2-bit dynamic predictor** is also referred as a **bimodal predictor**. It operates similarly as a **1-bit predictor**, the only difference is that instead of having a 1-bit counter, a bimodal predictor has a counter from 0 to 3. The author describe a pretty straightforward approach of the counters - counter decrements on each 'taken' branch except when the counter is 0 and increments each 'not taken' branch except when the counter is 3. Counters 0 and 1 symbolize that the next predicted branch is taken, and counter 0 in particular is '**strongly taken**'. Similarly counters 2 and 3 symbolize as not taken branches with 2 reflecting it as '**strongly not taken'**. This dynamic predictor is one of the most commonly used predictors as it combines affordable costs and good accuracy predictions (78% - 89%).

In this paragraph, the author explains how branch predictors can also manipulate correlations in branch outcomes to improve accuracy. A two-level adaptive predictor records the outcomes of previous branch instructions in it's history register. The author explains this by giving an example of register contents of 110010. The 6 previous branch outcomes in this case are taken-taken-not taken-not taken-taken-not taken. Similarly for 101101, it's

taken-not taken-taken-taken-not taken-taken (Note: 1 symbolizes 'taken' and 0 symbolizes 'not taken'). This history register is used to index a bimodal predictor table. The program counter can also be used in the manipulation to increase the overall accuracy (either concatenating or XORing it with indexing register). Due to the flexibility in its manipulation, two-level adaptive branch predictors have an accuracy of about 93%.

In the end of this section, the author describes due to size of the predictor, the table branches can collide; two different branches can point to same table entry which reduces the overall accuracy. The author doesn't explain his point pretty well here. *He could have mentioned how often do branches map the same table location? What happens after that? How much is the accuracy reduced? How can it be prevented?*

3. EXPERIMENTAL SETUP

Here the author uses ten sets of random data (*provided by Haahr [2006], if interested checkout random.org. It's a web service for random numbers developed by Haahr*) that has 2^{22} or 4194304 keys. The author states that whenever a particular experiment used only part of the keys in a chunk, it always used the leftmost keys. He doesn't describe why is this the case? Why not the rightmost keys?

For experimenting with various cache-and branch-prediction results, SimpleScalar PISA processor simulator version 3 [Austin et al. 2001] was used. A SimpleScalar is an open source computer architecture simulator developed by Todd Austin. In simple terms, it is used to show that 'Machine A is better than Machine B without building either of the machines. To know more about it's basic functionality check References [10], and for a tutorial see [11]. Cache simulator (referred as sim-cache) was used to generate results for caching characteristics and sim-bred (branch predictor simulator) was used to generate results for branch prediction characteristics.

The author in this paragraph describes the used cache configurations and then states that his group also experimented with direct-mapped and fully associative caches. These two are cache mapping techniques. Direct-mapped caches have 'good' hit ratio and 'best' searching speed,

where as fully associative caches have 'best' hit ratio and 'moderate' searching speed. If you want to learn more about these cache-mapping techniques, please refer to References [12].

All the SimpleScalar measurements, keys ranging from 2^{12} to 2^{22} keys were used which did not include quadratic sorting algorithms which had maximum set size of 2^{16} keys. This was deliberately done because of the fact that these algorithms take a lot of time to sort large inputs.

Filling arrays is time consuming as it comes at a price; it has the possibility to give a false relationship between the results for small set sizes, and larger set sizes. Due to this reason, it's associated time is amortized (reduced). Therefore, his research group took a measure of this time and later subtracted it from their net result. The surprising thing to note was that this particular practice also gave a false information about the results. It removes the inevitable cache misses as long as the data fits in the cache. When the data does not fit anymore, the starting keys of the array are faulted due to which capacity misses reoccur and these misses are not discounted.

In this paragraph, the author explains more about the configuration of predictors used in their collective research. Both, bimodal (predicting 'taken' or 'not taken' branches) and two-level predictors were used and the branch predictors used the power of two sized tables. Since Pentium 4 had an entry table of 2^{12} predictors [Hinton et al. 2001], they used predictors that were closer to that number (2^{11} and 2^{14} predictors). Also, the SimpleScalar sim-bpred simulator provides results with respect to all branches in a program. So, the author along with his team added their own simulations of branch predictors to the programs in order to examine the results of specific branches in a program. For the same, they averaged the results of each branch over ten turns of random data (provided by Haahr [2006], see the opening paragraph of Experimental Setup) containing 2^{22} (4194304) keys.

Lastly they used PapiEx on a Pentium 4, that had 1.6 GHz speed and 1 GB memory. 'PapiEx' is a PAPI based program which is used to measure hardware performance events (or counters) of an application using command line. To know more about PapiEx, see References [13]. The

hardware performance counters permits running times, also called cycle counts for sorting algorithms (selection sort, insertion sort, bubble sort) on a real processor. These cycle counts also included the stall cycles (not running) resulting from mispredicted branches and cache misses. The results from these tests were then averaged over 1024 runs and was used with random number generator for the resulting data. Again, the data fell within the range of 2^{12} to 2^{22} keys. Eight-way associative 256-KB level-2 cache with 64 byte cache lines was used. An eight way associative has each set containing 8 slots (an N-way associative cache has each set containing 8 slots, where N is a power of 2). The separate data and instruction level-2 caches are four way associative (therefore each set containing 4 slots), size of 8KB and 32 byte cache lines. In a four way associative cache, the memory cache will have 2048 blocks containing 4 lines each.

4. ELEMENTARY SORTS

Here, the branch predictions of the following sorting algorithms - selection sort, insertion sort and bubble sort are analyzed, where each sorting algorithm is operated in O(n^2) time (explained why in each section below). The author states that the 'simplicity' of these algorithms (they way they are designed and operated) plays an important factor that makes them stand out (in terms of performance) against the other sorting algorithms having O(nlog n) time complexity for small inputs.

4.1 Selection Sort

Selection sort is one of the simple quadratic time (i.e. O(n^2)) sorting algorithm. It involves sequentially (therefore linearly) searching for the smallest key (element in the list) in its provided list, and then moving it into it's correct position. The algorithm continues to repeatedly search for the smallest element and then it swap to its appropriate position, till all the elements in the list are all finally sorted. Consider the inner loop of selection sort given below:

```
min = i;
for (j = i + 1; j < n; j++)
    if (a[j] < a [min]) min = j;
swap (a[i], a[min]);
```

This inner loop of selection sort, runs a total of n-1 times on a [0, 1, … , n-1]. Since the author hasn't given the outer-loop of his selection sort pseudocode, here is the new code that contains both outer and inner loop (according to what the author is trying explain)

```
for (i = 0; i <= n-1; i++) // therefore running n-1 times
{
    min = i;
    for (j = i + 1; j < n; j++)
        if (a[j] < a [min]) min = j;
    swap (a[i], a[min]);
 }
```

The author states that it is easy to observe that the selection sort performs approximately of $\frac{n^2}{2}$ **comparisons** and **n-1 changes**. He could have given a brief explanation as to why is the case. Here is the analysis of selection sort that determines the total number of comparisons and exchanges involved.

Value of 'i' (outer loop)	Total number of comparisons
0	n-1 *(1st element comparing itself to the rest of the elements*)
1	n-2 (*Since, 1st element has already compared*)
…	…
n-1	1

Total number of comparisons = (n-1) + (n-2) + … + 3 + 2 + 1 = $\frac{n(n-1)}{2}$

Therefore, total number of comparisons = $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$ **comparisons**

Size of the list	Number of Exchanges ('swaps')
1	0
2	1
…	…
N	N-1

Therefore the list of size N has **N - 1 swaps**

On the j^{th} iteration of the inner loop (∵ `for (j = i + 1; j < n; j++)`) , the comparison branch is 'taken' if a[j] is the minimum of a[i..j] that has a probability of $\frac{1}{j-i+1}$. The expected number of times the branch is 'taken' on the first time is given by $H_n - 1$, where $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the nth harmonic number. The n'th harmonic number is about as big as ln n, since the the sum is approximated by $\int_1^n \frac{1}{x} dx$ that is ln(n) (See References[17]). Because of this reason, $\sum_{i=1}^{n} H_i$ = O(n log(n)). This makes the comparison branch highly predictable. The author then explains unique scenarios branch predictability with respect to the ordering of the elements in the array list. If the array is already sorted, then value of min would never change (min will always be equal to the first element which is the smallest element in the list) in the inner loop. This would therefore imply that the comparison branch is 'never taken' and hence will be almost perfectly predictable (always never taken). On the other hand if the array is sorted in the reverse (decrease) order, the value of min will change at each element. The comparison branch will always be 'taken' at every element, and therefore again the branch will be almost perfectly predictable (always taken). Frequent changes (with no pattern) with respect to branch prediction, are the main indicators of poor performance. Therefore the worst case of selection sort would involving having an array which is already, partially sorted in reverse order. As explained above, that will lead to frequent changes of min (changes at each element), but not predictably so (since it's *partially* sorted; no fixed pattern).

4.2 Insertion Sort

Insertion sort also has a quadratic worst time case. Consider the provided pseudocode below:

```
item = a[i];
while(item < a[i - 1])
{
    a[i] = a[i - 1];
    i--;
}
a[i] = item;
```

The inner loop (∵ `while(item < a[i - 1])`) iterates to a total of n-1 times. On the ith iteration, a[0, 1, ..., i-1] is sorted (∵ `a[i] = a[i - 1];`) that places a[i] in its appropriate position. The author states briefly that the average case scenario of insertion sort involves a total of $\frac{n^2}{4}$ **comparisons** and $\frac{n^2}{4}$ **assignments** . Here is the detailed explanation of it:

new pseudocode (taken from algorithm analysis class notes):

```
for i = 2 to N do
    newElement = list[ i ]
    location = i - 1
    while (location ≥ 1) and (list[ location ] > newElement) do
        list[ location + 1 ] = list[ location ]
        location = location - 1
    end while
    list[ location + 1 ] = newElement
end for
```

Final location of new element	Number of comparisons	Number of assignments
i	1	0 + 2 = 2
i-1	2	1 + 2 = 3
...	...	...
2	i-1	(i - 2) + 2 = i
1	i-1	(i - 1) + 2 = i + 1

The ith element could end up in i places, therefore the average number of assignments is

$$= \frac{(2 + 3 + \ldots + i + i + 1)}{i} = \frac{1}{i}\left(\sum_{j=1}^{i+1} j - 1\right) = \frac{(i+1)(i+2)}{2i} - \frac{1}{i}$$

Simplifying this, we would get,

$$\frac{i^2 + 3i}{2i} = \frac{i}{2} + \frac{3}{2}$$

Hence, average cost of ith insertion $= \frac{i}{2} + \frac{3}{2}$

∴ Total average $= \sum_{i=2}^{n} \frac{i}{2} + \frac{3}{2} = \frac{1}{2}\sum_{i=1}^{n} i + \frac{3}{2}\sum_{i=1}^{n} i - 2$

$= \frac{n(n+1)}{4} + \frac{3n}{2} - 2 = \frac{n^2}{4} + \frac{7n}{4} - 2$

$\approx \frac{n^2}{4}$ **comparisons**

Therefore insertion sort performs almost one half as many branches as selection sort (∴ selection sort $\rightarrow \frac{n^2}{2}$, insertion sort $\rightarrow \frac{n^2}{4}$). Additionally insertion sort has a property that it generally causes a single branch misprediction per key. When the appropriate location for the current item as been assigned, the inner while loop exit (`while(item < a[i - 1])`) is responsible for this branch misprediction.

4.3 Bubble Sort

Bubble sort is one of the other simple sorting algorithms, although the author claims it to be comparatively inefficient. Below is the backing of his reason:

```
for(j = 0; j < n - i - 1; j++)
{
    if(a[j + 1] < a[j])
         swap(a[j + 1], a[j]);
}
```

Consider the code provided by the author above. This loop is actually an inner loop that runs from 0 to n - i -1. Since he hasn't provided the whole code (outer loop), here is the full version of the code according to what he means.

```
for(i = 0; i < n -1; i++)
{
    for(j = 0; j < n - i - 1; j++)
    {
        if(a[j + 1] < a[j])
            swap(a[j + 1], a[j]);
    }
}
```

Therefore the outer-loop iteration (i) counter and begins at 0 and counts towards n - 1.

The author briefly states that unlike the selection sort, bubble sort uses information given by the number of comparisons. It would have been better if he would have described this point in a bit detail, as just the statement itself is isn't that direct. However, in terms of our own understanding the author tries to focus on the point that in selection sort the comparisons (if statement) just indexes the value of minimum (swapping happens outside the if statement), where as in the Bubble Sort, the number of comparisons lets you swap the number to it's right position (inside the if statement itself).

Shaker sort is variation build up on bubble sort. it contains two inner loops that are alternated. One of the inner loop is same as the bubble sort. The other inner loop is responsible for scanning elements from right to left moving small elements to the left.

Below is the code that tries to explain the same: (References [19])

```
for (int i = 0; i < array.length/2; i++) { //the parent inner loop
    for (int j = i; j < array.length - i - 1; j++) { //scan elements from left to right
        if (array[j] < array[j+1]) {
            int tmp = array[j];
            array[j] = array[j+1];
             array[j+1] = tmp;
            swapped = true;
         }
     }


    for (int j = array.length - 2 - i; j > i; j--) {  //scan elements from right to left
        if (array[j] > array[j-1]) {
            int tmp = array[j];
            array[j] = array[j-1];
            array[j-1] = tmp;
            swapped = true;
        }
    }
}
```

Let's examine the worst case time for bubble sort -

Consider a situation when the initial list is ordered in the decreasing order, the bubble sort algorithm does not terminate early.

Pass	Number of comparisons
1	N-1
2	N-2

...	...
N-1	1
N	0

$\therefore W(N) = \sum_{i=N-1}^{1} i = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2}$

Since, the shaker sort uses the same for loop that is used in the bubble sort, both have the same worst case time. However, in practice, shaker sort is generally more efficient.

Figure 1b displays the analysis of the number of branch mispredictions per key for shaker sort and bubble sort. When the data is large, bubble sort produces as many as 10,000 branch mispredictions per key, whereas shaker sort produces relatively fewer branches and fewer branch mispredictions (Figure 2b). However, the rate of branch misprediction is same.

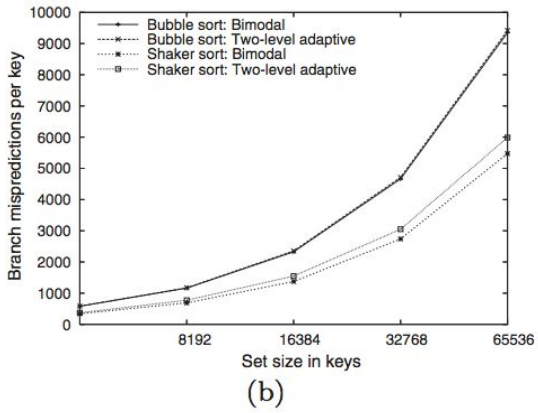

Figure 1b

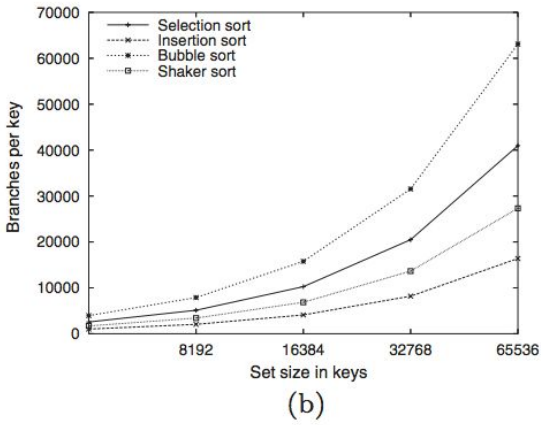

Figure 2b

The author states that at the initial stage the branches are quite easy to predict. It's the outer-loop that makes things difficult. When the outer loop iterates, the predictability reduces rapidly. The predictability improves again, when the data is close to get fully sorted. The partial-sorting bubble sort, with fewer outer loop iterations, converges the data into fully sorted data. This can be observed if the branch in the inner loop (`for(j = 0; j < n - i - 1; j++)`), is never taken for a whole iteration. However, with this outcome, bubble sort gets slower

if compared with selection or insertion sort since partial sorting increases the number of branch mispredictions the bubble sort incurs.

4.4 Remarks

The author concludes here by stating that all the above mentioned results demonstrate the importance of branch prediction for the elementary sorting algorithms. As depicted in the figure 4(a), the shaker sort algorithm has fewer cache misses compared with our selection sort algorithm but because the branch mispredictions it is slower. It is important to note how the author is trying to describe his conclusions through different graph figures. For instance, Figure 2a shows that shaker sort algorithm has lower instruction count than selection sort.

Insertion sort on the other hand performs better than the selection sort since it only cause almost a single branch misprediction per key. In addition, insertion sort has the fewest cache misses, therefore it is highly recommended as an elementary sorting algorithm. Though these cases are a bit different with bubble sort. The partial sorting algorithm of the bubble sort causes difficulties in branch prediction and therefore is not highly recommended as an elementary sorting algorithm.

Finally, these results conclude that the two-level adaptive branch predictors don't outplay the simple bimodal predictors. In particular, the bimodal predictor performs better than a two-level adaptive predictor for shaker sort. This is quite interesting to note since two-level branch predictors are similar to/more accurate than bimodal predictors for almost other types of branches.

REFERENCES:

1) Wikipedia: The free encyclopedia. (2015). “Sorting”
https://en.wikipedia.org/wiki/Sorting

2) Sedgewick, R., & Flajolet, P. (2013). “Analysis of Algorithms”
http://aofa.cs.princeton.edu/10analysis/

3) Wikipedia: The free encyclopedia. (2015). “Real Ram”
https://en.wikipedia.org/wiki/Real_RAM

4) Black, P. (2015). “Asymptotic Bound”
https://xlinux.nist.gov/dads/HTML/asymptoticBound.html

5) Wikipedia: The free encyclopedia. (2015). “MIX”
https://en.wikipedia.org/wiki/MIX

6) Tryfonas, G. (2011). “A Simulator for Knuth's MIX Computer”
http://www.codeproject.com/Articles/152527/A-Simulator-for-Knuth-s-MIX-Computer

7) Wikipedia: The free encyclopedia. (2015). “Branch (computer science)”
https://en.wikipedia.org/wiki/Branch_(computer_science)

8) Wikipedia: The free encyclopedia. (2015). “Pipeline (computing)”
https://en.wikipedia.org/wiki/Pipeline_(computing)

9) Wikipedia: The free encyclopedia. (2015). “Branch predictor - Static prediction”
https://en.wikipedia.org/wiki/Branch_predictor#Static_prediction

10) Lee, B. “Dynamic Branch Prediction”
http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/#l2

11) Introduction to SimpleScalar
http://www.ecs.umass.edu/ece/koren/architecture/Simplescalar/SimpleScalar_introduction.htm

12) Austin, T., & Burger, D. “SimpleScalar Tutorial”
http://www.simplescalar.com/docs/simple_tutorial_v2.pdf

13) The PC Guide. (2001).“Comparison of Cache Mapping Techniques” http://www.pcguide.com/ref/mbsys/cache/funcComparison-c.html

14) Mucci, P. “papiex - Command line/library utility to measure hardware performance counters with PAPI” http://icl.cs.utk.edu/~mucci/papiex/

15) University of Maryland. “Set Associative Cache” https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/set.html

16) Torres, G. (2007). “How the Memory Cache Works” http://www.hardwaresecrets.com/how-the-cache-memory-works/8/

17) Wikipedia: The free encyclopedia. (2015). “Harmonic Number” https://en.wikipedia.org/wiki/Harmonic_number

18) Wikipedia: The free encyclopedia. (2015). “Harmonic Number - Calculation” https://en.wikipedia.org/wiki/Harmonic_number#Calculation

19) McConnel, J. “Insertion Sort Average Case Analysis” http://moodle.cs.smu.ca/pluginfile.php/784/mod_resource/content/4/InsertionSort_AssignmentAnalysis.pdf

20) Algoritmy.net. “Shaker Sort” http://www.programming-algorithms.net/article/40270/Shaker-sort