# Report: An Experimental Study of Sorting and Branch Prediction

Methew Guda - A00381751
Nrimoni Chowdhury - A00371596

---

Sortings are one of the most important and well-studied problems in computer science. Two important features of these sorting algorithms are Caches and branch predictors. There has been a significant amount of research into the cache performance of general purpose sorting algorithms and there also has been little research on their branch prediction properties. This paper, observe and examine the behavior of the branches in all the most common sorting algorithms and also observe the interaction of cache optimization on the predictability of the branches in these algorithms. Through the number of experimental studies following problems found: insertion sort to have the fewest branch mispredictions of any comparison-based sorting algorithm, that bubble and shaker sort operate in a fashion that makes their branches highly unpredictable. It is also found that optimizations to quicksort, for example the choice of pivot, have a strong influence on the predictability of its branches. Finally, when sorting random data two-level adaptive branch predictors are usually no better than simpler bimodal predictors.

**1. Motivation:** The aim of algorithm analysis is to make simplifying assumptions about the cost of instructions on different machines. For example, the RAM model used for establishing asymptotic bounds and Knuth's MIX (a hybrid binary–decimal computer) machine code both make drastically simplifying assumptions about the cost of machine instructions. On modern computers the cost of accessing memory can change dramatically depending on whether the data can be found in the first-level cache, or must be fetched from a lower level of cache or even main memory.

Conditional branch is an another type of instruction, whose cost can change dramatically. Modern pipelined processors depend on branch prediction for much of their performance. The cost of the conditional branch may be small, If the direction of a conditional branch is correctly predicted ahead of time. On the other hand, the processor must flush its pipeline and restart from the correct target of the branch, if the branch is mispredicted. Luckily, branch mispredictions are rare, because the branches in most programs are very predictable. The inner loops of most sorting algorithms consist of comparisons of items to be sorted, so that makes the cost of executing branches is particularly important for sorting. Therefor, the predictability of these comparison branches is critical to the performance of sorting algorithms. In this paper, researchers focused on the behaviour of the branches whose outcome depends on a comparison of keys presented to the sorting algorithm in its input.

**2. Branch Prediction:** Branches are a type of instruction used for defining the flow control of programs. In high-level languages, these flow controls are statements like if, while, and their variants. Branches can be either taken, indicating that the address they provide is the new value for the program counter, or they can be not taken, in which case sequential execution continues as though the branch had not been present.

Branch instructions cause a difficulty for pipelined processors, while the program counter points to a particular instruction by putting a potentially large number of subsequent instructions in a partially completed state. It is not possible to know about the next executed instructions, until it is known whether a branch is taken or not taken. Resulting the processor cannot fill its pipeline. Modern processors attempt to anticipate the outcome of branch instructions by employing branch predictors to keep the utilization of the pipeline at a reasonable level.

There are number of branch predictors: static, semi static and dynamic. Static branch predictor's job is to always predict the same direction for a branch each time it is executed. A usual heuristic is to predict forward branches not taken and backward branches taken. Semi Static branch predictors support a hint bit that allows the compiler to determine the prediction direction of the branch. Dynamic predictors are most commonly used in real processors. Static predictors are, in most cases, trivial to analyze compared to the more realistic case of dynamic predictors. 1-bit predictor is referred as the simplest type of dynamic predictor, that keeps a table recording for each entry, whether a particular branch was taken or not taken. Using the data from the table, it predicts that a branch will go the same way as it went on its previous execution. The 1-bit predictors achieve 77 to 79% accuracy.

A bimodal predictor, which can be also referred as a 2-bit dynamic predictor, operates in the same way as a 1-bit predictor, but each table entry can be thought of as maintaining a counter from 0 to 3. The counter decrements on each taken branch with the exception of when the count is 0 and increments on each not-taken branch with the exception of when the count is 3. The next branch is predicted as taken when counts equal to 0 or 1 and the branch is predicted not taken when counts equal to 2 or 3. The predictor is in the strongly taken state when the counter is 0 and it is important that the branch must be not taken twice before the not-taken direction will be predicted again. The predictor is in the taken state when the counter is 1. Similarly counts of 2 and 3 are referred to as not taken and strongly not taken, respectively. Bimodal predictors achieve 78–89% accuracy.

Finally, some branch predictors attempt to improve accuracy by exploiting correlations in branch outcomes. A two-level adaptive predictor maintains a branch history register to records the outcomes of a number of previous branch instructions. For example, a register contents of 110010 indicates that the six previous branch outcomes were taken, taken, not taken, not taken, taken, and not taken. For every branch, this register is used to index a table of bimodal predictors. The program counter can also be included in the index, either concatenated or XORed with the indexing register. Two-level adaptive predictors are accurate about 93% of the time. But two different branches can often reduce accuracy by mapping to the same table location, because of the size of the predictor, collision tendencies of table branches.

**3. Experimental Setup:** This experimental setup was designed with ten sets of random data containing $2^{22}$ (4194304) keys. Where a particular experiment used only part of the keys in a set, it used the leftmost keys. The results of experiment are averaged over these sets of data. In order to experiment with a variety of cache and branch prediction results the SimpleScalar PISA processor simulator version 3 was used. Sim-cache and Sim-bpred programs were used to generate results for caching and branch prediction characteristics. There were variety of cache configurations were used in the experiments, an 8-KB level-1 data cache, 8-KB level-1 instruction cache and a shared 2MB instruction and data level-2 cache, all with 32-byte cache lines. Power of two sized sets of random keys ranging from $2^{12}$ to $2^{22}$ keys in size were used when the measurements were taken using SimpleScalar. This is with the exception of the quadratic time sorting algorithms, for which the maximum set size used was $2^{16}$ keys.

Filling arrays require time and this can distort the relationship between the results for small set sizes and larger set sizes, for which this time is amortized. So as a result, time was measured and subtracted it from experimented results. But this practice also distorts the results. It removes compulsory cache misses from the results, as long as the data fits in the cache. When the data does not fit in the cache, the keys from the start of the array are faulted, with the effect that capacity misses reoccur, which are not discounted. Bimodal and two-level adaptive predictors were both simulated for branch prediction. The simple bimodal predictor provides a good baseline than the two-level adaptive predictor. Power of two sized tables were used for the branch predictors. Tables containing keys between $2^{11}$ and $2^{14}$ predictors were used, because these are close to the size of the $2^{12}$ entry table of the Pentium 4. SimpleScalar sim-bpred provides total results over all the branches in a program.

Finally, PapiEx were used to access performance counters on a Pentium 4 1.6 GHz, with 1GB of memory. The hardware performance counters allow cycle counts to be determined for the sorting algorithms on a real processor. These cycle counts include the stall cycles resulting from branch mispredictions and cache misses. The results from these tests are averaged over 1024 runs and use the system's random number generator to provide data. The data ranged in size from $2^{12}$ to $2^{22}$ keys in size. The processor we used had an eight-way associative 256-KB level-2 cache with 64-byte cache lines. The separate data and instruction level-1 caches are four-way associative, 8KB in size, and have 32-byte cache lines. The Pentium 4 uses an unspecified form of dynamic branch prediction. A reference manual for Intel 2004 mentions a branch history register, so it is probable that this is some variety of two-level adaptive scheme.

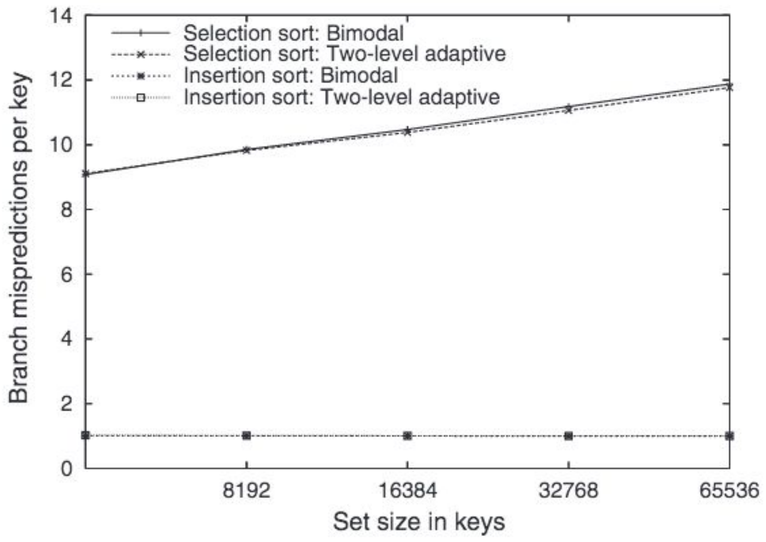
**4. Elementary Sorts:** The analyzing time takes to sort using different sorting algorithms where all  takes O($n^2$) time which means for each list of n it takes n*n time to sort. The biggest advantage of all these algorithm is that they are simple and fast to calculate than the algorithm which takes O(nlogn**)** time for small list of numbers. Below we will discuss about some most popular, simple and fast sorting algorithms whose time complexity is O($n^2$).

**Selection sort** is a very simple time sorting algorithm, it starts by searching for the smallest key in input and then moving into a position and it just keeps repeating until it reaches the correct position for sorting. This part of algorithm runs a loop total of n-1 times on an input.

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

The number of comparisons is approximately $n^2/2$, as you can understand from the algorithm that for each value of n it takes square of n number of comparisons as it passes through the loop starting from 0 to n-1. The number of exchange is exactly n-1 because it swaps all the number till it is sorted except the last number gets sorted automatically, that's the reason there are n-1 number of exchange.

At the end of the inner loop, the comparison branch checks if a[j] is a minimum of a[i...j], where the probability is 1/(j - i + 1). The loop runs for $H_n$ - 1 on first iteration where $H_n = \sum^{n}_{i=1} 1/i$ is the nth harmonic number. As, H ≈ ln n and $\sum^{n}_{i=1} H_i = O(n \log n)$ the comparison branch will be highly predictable. The *min* value will never change if the array is already sorted and the comparison branch will never be used too. On the other side, the array is in reverse order, the value of *min* will keep changing at every step and there will be comparison on every key. The worst case scenario for selection sort would be a partially sorted reverse listed array, thus the value of *min* would change often, but will be unpredictable.



**Graph: (a)**

The graph (a) shows how selection sort behaves to the size of keys vs branch mispredictions per key. As it can be seen from the graph, the selection sort Bimodal and Two-level adaptive predictor have very similar performance. The y-axis (vertical axis) starts at 9 and the number of branch predictions per key increases constantly with an increase in set size in keys.

**Insertion sort** is another simple algorithm, it operates by running the inner loop (n-1) times where n is the list size which means it runs till the second last number as the last number already gets sorted automatically.

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

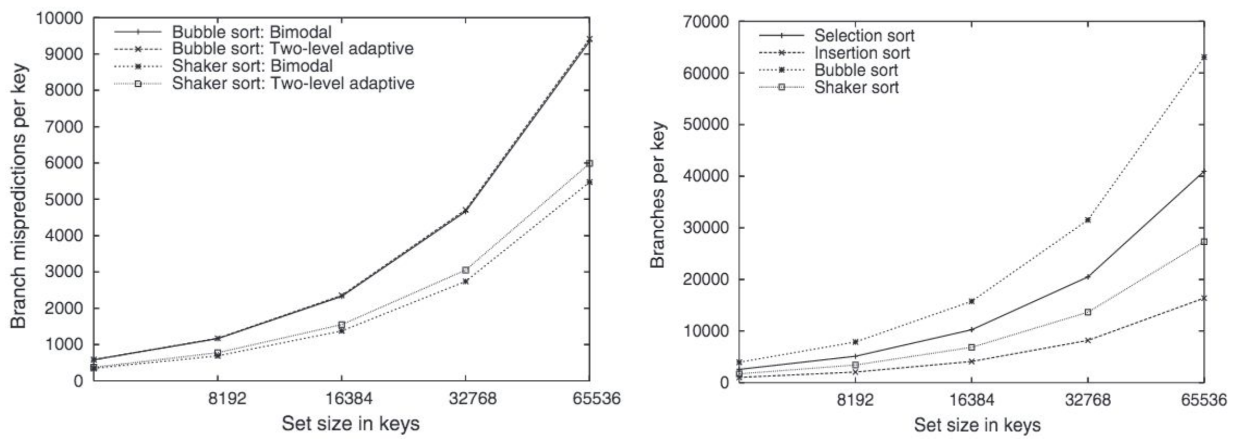
So, when the inner loop is finished running,the array a[0...i - 1] is sorted and then a[i] is placed in the correct position, variable named "*item*" as demonstrated above. The insertion sort has approximate total of $n^2/4$ comparisons and $n^2/4$ assignments. Therefore, insertion sort is faster and takes half the time of selection sort, i.e., $n^2/4 < n^2/2$.

From the graph (a), it can be demonstrated that the insertion sort  Bimodal and Two-level adaptive predictor have exactly same performance. The y-axis (vertical axis), branch misprediction per key starts at 1 and stays the same for all values of set sizes in keys, this is because the loop quits causing this misprediction and the current value is found and put into the right position. In reality, a static predictor will perform same as the two predictors demonstrated above but will produce one misprediction per key.

**Bubble sort** is another simplest and most inefficient sorting algorithm. It calculates by running the loop at most n-1 times on any range of lists a[0...n - 1]. Here, *i* is the outer loop counter count from 0 to n-1 and then when it is complete the largest elements are moved to the right to the final position.

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

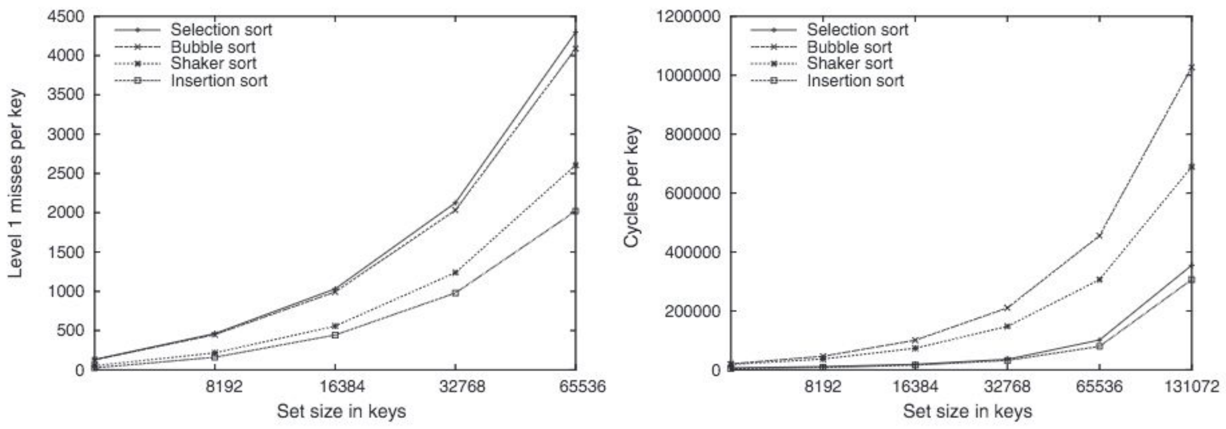
Shaker sort is an another variant of bubble sort which runs on two alternating inner loops. One is like shown above and the other runs by moving smallest items from right to left marking them in the final position. Surprisingly, shaker sort has same worst case scenario but it is more efficient than bubble sort because shaker sort.



**Graph (b)**

The left graph of (b) demonstrates the performance comparison between Bimodal and Two-level adaptive predictor for bubble sort and shaker sort. At first, the bubble sort Bimodal and Two-level adaptive predictor performs exactly the same but the number of branch mispredictions per key increase exponentially to the increase of set size in keys. The shaker sort Bimodal and Two-level adaptive predictor performs really close but the gap in between the two predictors slightly increases in as the data gets larger. Similar to bubble sort, the shaker sort behaves exponentially, which means the branch mispredictions per key increases exponentially to the increase of set size in keys. But the shaker sort, causes less mispredictions because it executes a lot fewer branches than bubble sort shown in right graph of (b).

The branches per key increase exponentially for all the sorting algorithms. The predictability decreases as the outer loop iterates. For bubble sort, as the data gets larger, predictability for bubble sort decreases but when it comes close to get sorted the predictability starts to increase. The partial-sorting bubble sort performs allowing it to potentially converge to fully sorted data in fewer outer loop iterations than selection sort. It can be detected that the data is sorted early if the branch shown in inner loop is never considered and the partial sorting also highly increases the number of branch mispredictions when the bubble sort is run than selection or insertion sort.



**Graph: (c)**

The experiments above clearly explains how important are the branch predictions for these elementary sorting algorithms. The shaker sort has fewer cache misses and lower instruction count than selection sort but it is slower because of the branch mispredictions as shown in the above graphs. Selection sort causes on average fewer than 12 mispredictions per key for a set size of 65536 keys, where bubble and shaker sort cause many thousands. Insertion sort behaves more efficiently than the selection sort causing just a single misprediction per key and with fewest cache miss than other sorting algorithms.

At last, the results show that over these sorting algorithms, two-level adaptive branch predictors and simpler bimodal predictors are same. Despite of both the predictors are similar to each other but surprisingly the two-level adaptive predictor out-performs the bimodal predictor in shaker sort.