Extended Essay in Computer Science

Investigating the inter-relationship between programs and branch predictors.

Research question:

To what extent do branch mispredictions affect the performance of well-known comparison sorts?

Session: May 2024

Personal code: kth862

Word count: 3499

Declaration: I declare that this work is my own work and is the final version. I have acknowledged each use of the words of another person, whether written, oral or visual.

Contents

1 Introduction

1.1 Background

Computer programs consist of instructions belonging to instruction sets. These in turn, alongside other features like registers and data types, are defined in instruction set architectures (ISA's). [1] Examples of such architectures include ARM or x86-64. [2] [3] Instruction sets include various types of instructions, which are responsible for managing memory, performing arithmetic operations, controlling the flow of execution as well as any other operations exposed by the hardware to the software. [1]

Most, if not all computer programmers are aware of the existence of these instructions. Nonetheless, only a small fraction of them actually works on such a low level of abstraction, mostly thanks to the existence of programming languages. The tools that connect programming languages with the underlying machine code are compilers and interpreters. They efficiently translate the human-legible source code to machine-compatible instructions. [4]

Although programming languages enable considerable amounts of flexibility and provide clever optimisations, what ultimately decides about performance is not the software or the ISA, but the hardware. [5] [6] More precisely, the way that the hardware is designed. Computers for a long time have worked in pretty much the same way: they execute one instruction after another. [7] The speed of execution can of course be increased by making use of faster clock speeds, but only so much until it becomes inefficient. That's why modern CPU's are designed to parallelise sequential instructions. That may sound counter-intuitive, but fundamentally, it's what the Out-of-Order (OoO) execution paradigm tries to achieve. Instead of waiting until an instruction finishes, modern processors will often already be processing several instructions ahead: fetching them from memory to the internal cache, decoding them to the internal micro-operations that have to be performed, and executing them before the previous instruction even finishes. This allows to use clock cycles that would otherwise be wasted waiting, by speculatively doing work that may, or may not be used. [8]

1.2 Branch prediction

Branches are instructions that are meant to control what the processor executes. They instruct the processor to jump to an address in memory, and execute the instruction that it finds there. There are two types of branches, unconditional and conditional. As their names suggest, unconditional branch instructions, also known as jump instructions or return instructions are always **taken**, that is that the execution unit will always jump to the memory address that the instructions point to. Conditional branches on the other hand, will only be taken if a given condition is met. If it isn't, the branch is **not taken**, meaning that the instruction immediately after the conditional branch instruction will be executed. [9] [7]

Due to the fact that branch instructions may be as much as 20% of all instructions executed by the central processing unit (CPU), it would be very efficient if they could also be executed speculatively. [10] First of all, execution units can speculate on the **direction** of the branch, i.e. whether it is taken or not. Another thing that they can speculate on is also the address to which they should jump to, as sometimes these addresses are not known directly, but are instead calculated or looked up. [9]

Doing these things is the job of the **branch predictor**.

1.3 Branch predictors

Branch predictors are the physical circuits on the dye of the processor that are responsible for performing branch prediction. These circuits are usually used in early stages of the processor pipeline, like the fetch stage. This means that all the later stages heavily depend on the accuracy of its predictions, and they operate under the assumption that its predictions are correct.

Although different branch predictors have been designed over the last years, they generally fall in one of two categories: static or dynamic predictors. Static predictors employ a simple strategy: they always do the same thing. They don't have any way of storing data or performing look-ups. [9] Examples of static predictor strategies include:

- **always not taken** - predicting that conditional jumps are never taken, instead always fetching the instruction directly after the jump instruction and passing it on further to the pipeline;
- **always taken** - predicting that conditional jumps are always taken, which is a more difficult strategy than the previous one, as the predictor now also has to decide where exactly the jump will lead, potentially having to request instructions that have not yet been cached or having to predict the address of the jump;
- **backwards taken, forwards not taken** - predicting that jumps backwards, meaning to addresses before the address of the jump instruction, are always taken, and jumps forwards, meaning to addresses after the address of the jump instruction are never taken. This strategy assumes that backwards branches are loops, and thus should be taken.

Although these branch predictors may provide satisfactory results that are better than just simply stalling the pipeline, most modern architectures provide more complex, dynamic predictors. One of the simplest additions to a static predictor that will improve its effectiveness is a way of storing data with historical information. Such an addition is known as a branch history table (BHT), and enables the following strategies:

- **one bit prediction** (last time prediction) - storing a single bit in the BHT, indexed by the lower bits of the address of the instruction, corresponding to whether the branch was previously taken, or not taken. If the outcome of the branch is the same as the prediction, then the value remains the same, otherwise it is changed.

- **two bit prediction** - storing two bits in the BHT, similar to the one-bit predictor. If the outcome of the branch is the same as the prediction, it stores a 1 representing a weakly taken branch. If a weakly taken branch is repeatedly taken, the value is updated to a strongly taken (11). If it's repeatedly not taken, the value is updated to strongly not taken (00). This approach is similar to the one bit predictor, but allows for more detailed tracking of patterns.

More complex predictors like the **tournament predictor** from the Alpha 21264 microprocessor, can combine local branch history tables with global branch history tables, and multiple predictors, from which a predictor chooses a predictor. [7] There are of course much more complex predictors, some of which are probably trade secrets which are not only far outside the scopes of this investigation, but also outside of reach for most researchers.

1.4 Sorting algorithms

Since the dawn of computing, sorting algorithms have had a notorious reputation in computer science due to their complexity, and various trade-offs that they offered. Some of them have been designed to have small memory footprints, while others were meant to be just as fast as possible. Various tools have been created to quantify the theoretical performance of algorithms, like the venerable big O notation, but they are not capable of showing how an algorithm will perform on actual hardware, on an actual ISA, with its caches and speculative execution.

As branch prediction is such a significant part of a processor pipeline, I would like to investigate what effect on programs it actually has, and there are no better candidates to be tested on than the lab rats of computer science - sorting algorithms. Thus I will be investigating my research question:

To what extent do branch mispredictions affect the performance of well-known comparison sorts?

2 Methodology

In order to isolate the effect that the branch predictor has on the selected sorting algorithms I will be using a simulator, more precisely the **gem5** simulator. Using a simulator over real hardware provides a few advantages, the first ones being *repeatability* and *reproducibility* - the simulations can be repeated as many times as needed, and they can also be reproduced on other hardware yielding the same results, provided the same configuration is used. Another major advantage is *transparency* - as the simulator is open source I can easily understand how it works. The open nature of the software also provides *extensibility* which enables modifying or extending its behaviour. The biggest downside of using gem5 is that it currently only supports simulating an OoO pipeline similar to the one found in the Alpha 21264 microprocesor, which was launched in 1998. [11]

Using the simulator enables me to test the performance of the algorithms on two different systems, with only one difference between them - the branch predictor. This allows me to compare how different predictors will affect the performance of the programs, without changing anything else.

To maximise the difference in the effectiveness of predictors, I wanted to use a static predictor that would provide poor effectiveness, as well as a considerably more accurate dynamic predictor that would outperform the static predictor. The simulator features a variety of predictors, however all of them are dynamic. This meant that I had to extend the functionality of the simulator by implementing a static predictor.

The primary reason behind implementing the static always-taken predictor in gem5 was to have a branch predictor with a considerably lower effectiveness, meaning one that's worse at predicting if branches are taken or not taken. According to various sources, a tournament branch predictor is accurate in 90% [7] of its predictions, while a static (always taken) predictor is around 60-70% accurate. [7] [9] By intentionally causing branch mispredictions, I aimed to show how an OoO pipeline that heavily depends on its branch predictor will be affected by using a worse predictor.

2.1 gem5 configuration

The simulator was built on Ubuntu 22.04 LTS for the x86 ISA with standard optimisations using the command `scons build build/X86/gem5.opt -j 8`.

In order to show the extent of the effects that branch misprediction has on programs, I also wanted to make use of a branch prediction that would perform quite poorly. As gem5 does not include such a predictor I had to implement it myself. I decided that a static-always taken predictor would be a good predictor to compare with the already-present tournament predictor.

gem5 makes use of the SCons software construction tool, which enabled me to easily incorporate my own `StubBP` class that extended the base `BPredUnit` class. The method of the class responsible for returning a prediction to a given branch is `lookup()`, and for a static predictor it does not require any computation and can simply return `true`.

2.2 Sorting algorithms

As the simulations require x86 implementations of sorting algorithms, I implemented the following five algorithms in C, based on the pseudocode provided in Thomas H. Cormen's *"Introduction to Algorithms"* [12]:

- Quicksort;
- Merge sort;
- Insertion sort;
- Heapsort;
- Bubblesort.

Additionally, I implemented Shellsort, based on based on the description in Donald E. Knuth's book *"The Art of Computer Programming"*. [13] . The reference implementations of the sorting algorithms are available in the Appendices. All algorithms were compiled using gcc version 11.4.0 with `-Wall -Wextra -O2` flags.

2.3 Simulations

I ran simulations on array sizes from 0 to 1000 in 100 array key increments. The integers in the arrays were generated using python's builtin `random.randint()`, and the same arrays were provided to different algorithms for a given element count. Then, each available sorting algorithm was ran twice, using the `se.py` configuration included with gem5, which configured the type of processor and caches as well as one of the two branch predictors' that were tested, either TournamentBP or StubBP. Below is the code from the script responsible for running batches of these simulations:

```
subprocess.call([
    GEM5_PATH.joinpath('build/X86/gem5.opt'),
    '--outdir', tmpdir_path,
    GEM5_PATH.joinpath('configs/example/se.py'),
    '--cpu-type=X86O3CPU',
    '--caches',
    '--cmd', algorithm,
    '--options', f'{len(array)} {" ".join(str(i) for i in array)}',
    '--bp-type', predictor
])
```

The statistics generated during the runs were aggregated into a single file, which was later split into smaller spreadsheets used to generate charts using gnuplot. All charts in this document also include lines, which connect the statistics generated by the simulations and are meant to be interpreted as visual aids only.

3 Results and discussion

3.1 Mean results

I will start by analysing the average results from all algorithms, as this gives a good overview of what can be expected to happen to individual sorting algorithms.

3.1.1 Branch mispredictions

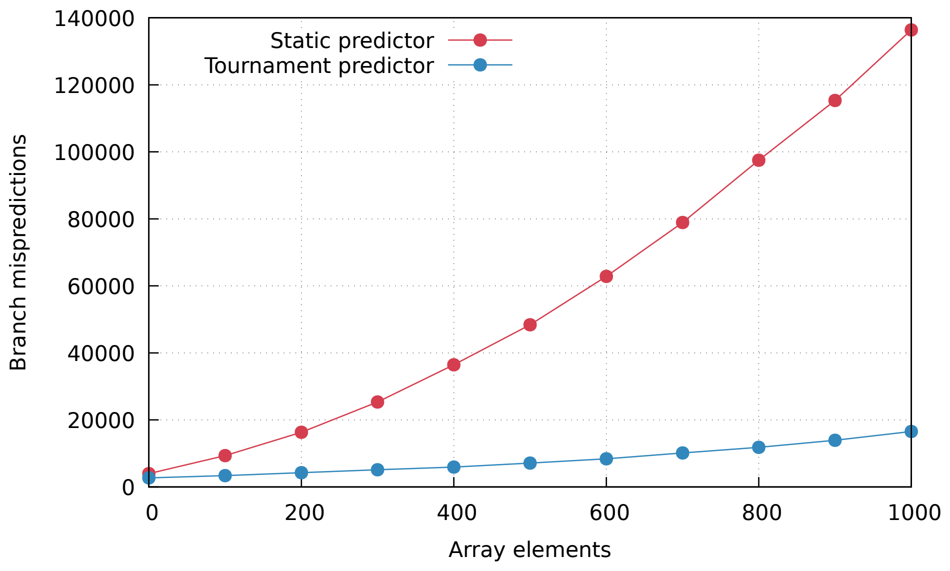

Figure 1: caption

Figure 1 illustrates the rise in the number of branch mispredictions caused by the static predictor. For larger array sizes, the always-taken predictor causes more than 7x more mispredictions than the tournament predictor. This difference in the number of mispredictions should show how the effectiveness of each of the predictors' affected the overall performance of the rest of the CPU.

The results in this figure are averages from different sorting algorithms, and provide an overview of the increase in branch mispredictions. Some algorithms were affected more than others, which I will discuss in a later section.

3.1.2 Execution time

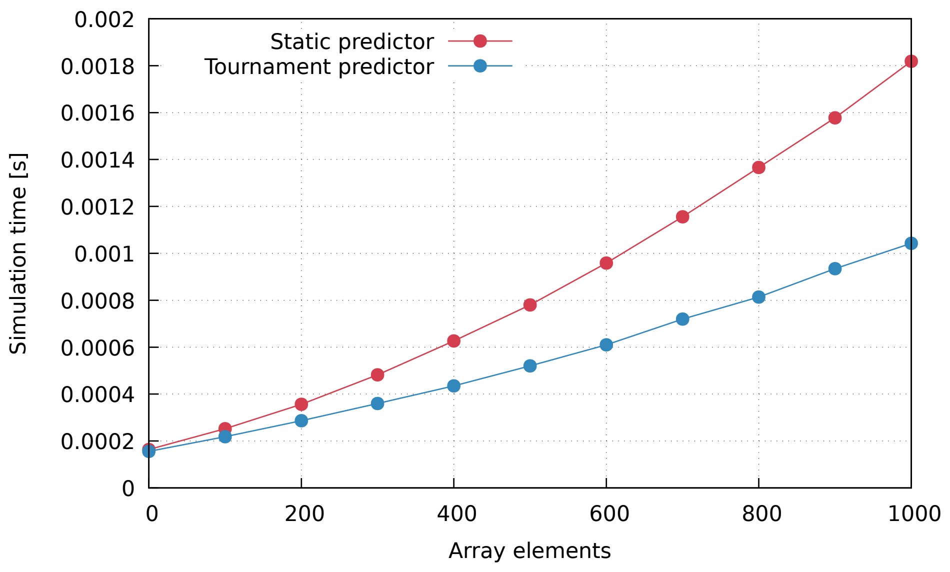

Figure 2: Mean simulation time for both branch predictors.

Figure 2 shows the number or simulated seconds. This statistic is calculated by gem5 by dividing the number of internal simulation ticks by the simulation frequency. The simulation frequency is a constant equal to 1×10^{12} for all runs, which points to the number of simulation ticks being increased. This is true for all the runs making use of the static branch predictor.

Considering that all of the sorting algorithms were supplied with exactly the same randomly-generated array (for a given array size), the degradation in performance cannot be attributed to the sorting algorithms themselves. This is also confirmed by other collected statistics, like the number of instructions simulated and the number of instructions committed being the same for algorithms ran on simulations using different branch predictors.

In summary, the 1.8 time increase in execution time for larger array sizes shown in Figure 2 has not been affected by any external factors, and is directly caused by the increase in branch mispredictions. This is a massive performance degradation, that every algorithm suffered from.

3.1.3 Number of cycles

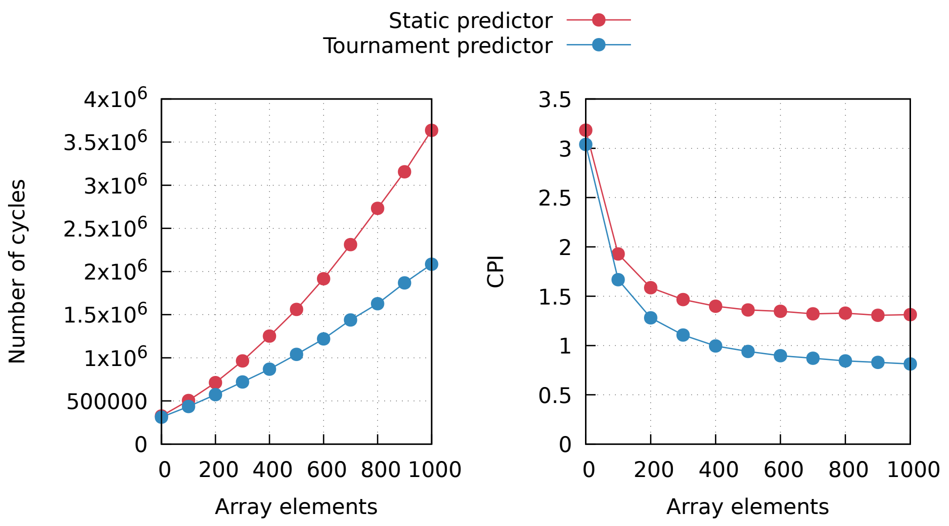

Figure 3: Number of cycles performed by the CPU.

Figure 3 illustrates how the simulations using the static branch predictor took more CPU cycles to complete. The difference between the number of cycles reaches a staggering 1,550,000 cycles for a 1000 element array.

Taking into account the fact that the number of instructions simulated was the same for both predictors, it would seem that the CPU needed more cycles to execute the same instruction. The cycles per instruction (CPI) chart on Figure 3 shows exactly how many more cycles on average the processor needed to execute an instruction.

The branch predictor is not directly responsible for wasting CPU cycles, as it only needs a single cycle of the fetch stage to return a prediction for a branch. This means that although the branch predictor itself did not contribute to an increase in the cycles its predictions did.

Although *what* using a less efficient predictor results in is well established by this statistic, it does not alone point to *why* the number of cycles is increased. The later statistics include some possible explanations.

3.1.4 Squash cycles

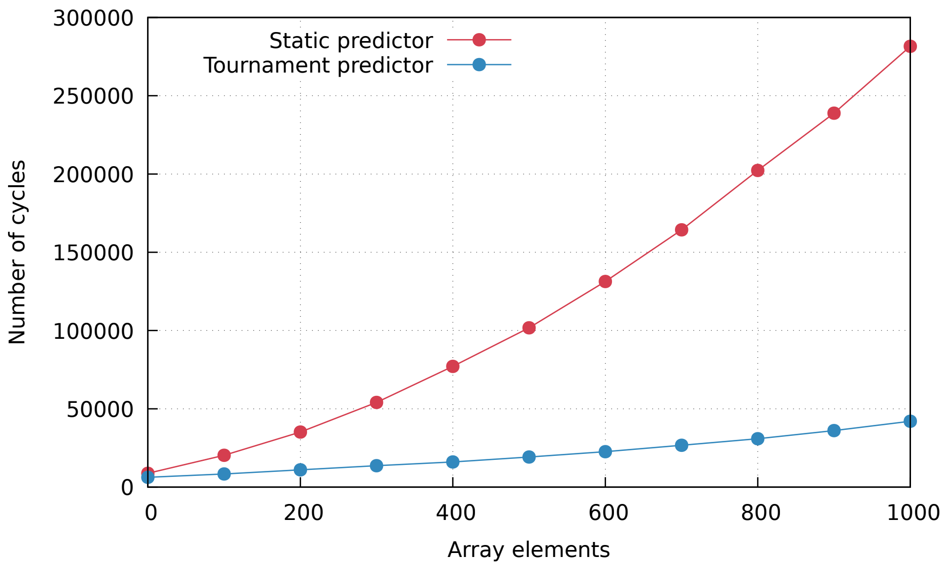

Figure 4: Cycles spent on squashing instructions.

Figure 4 shows the number of cycles that the fetch stage of the CPU spent squashing instructions from the pipeline. After a branch misprediction the CPU needs an entire additional cycle to perform this operation. Although the increase in cycles is significant, it is not the main cause of the performance decrease, as in the case of the 1000 key array it only accounts for 230,000 cycles out of the additional 1,550,000 cycles that the CPU performed.

3.1.5 Additional pipeline cycles

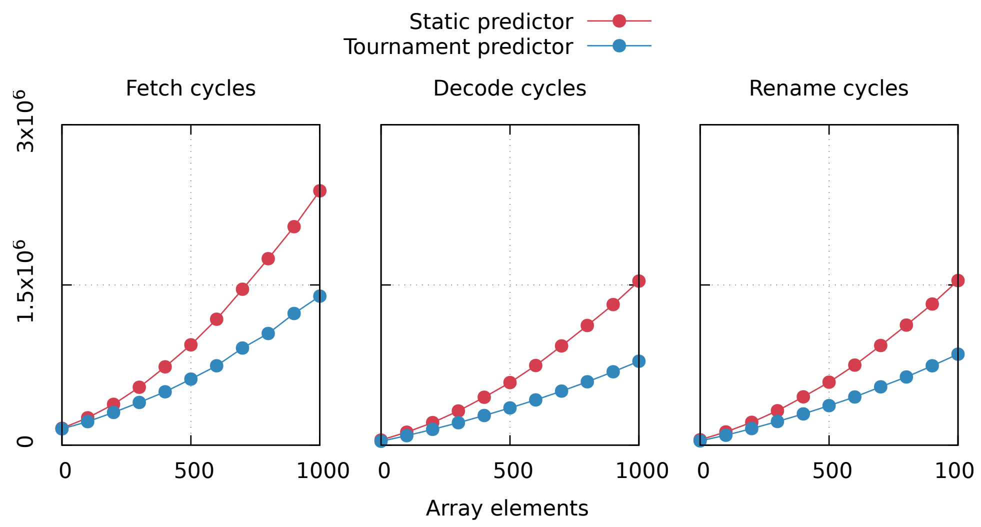

Figure 5: Running cycles performed by three example stages of the pipeline.

Figures 5 show the increase in cycles that the select stages performed. The increase in the number of cycles can be attributed to the pipeline being empty after instructions are squashed after a branch misprediction, as well as to the stages performing unnecessary operations on instructions that were later squashed, therefore discarding their work.

In the case of the fetch stage of the pipeline, a considerable 990,000 cycle increase can be observed, for an array size of 1000 elements. This makes up 60% of the cycle difference between the simulations making use of the two different predictors.

3.1.6 Instruction cache stall cycles

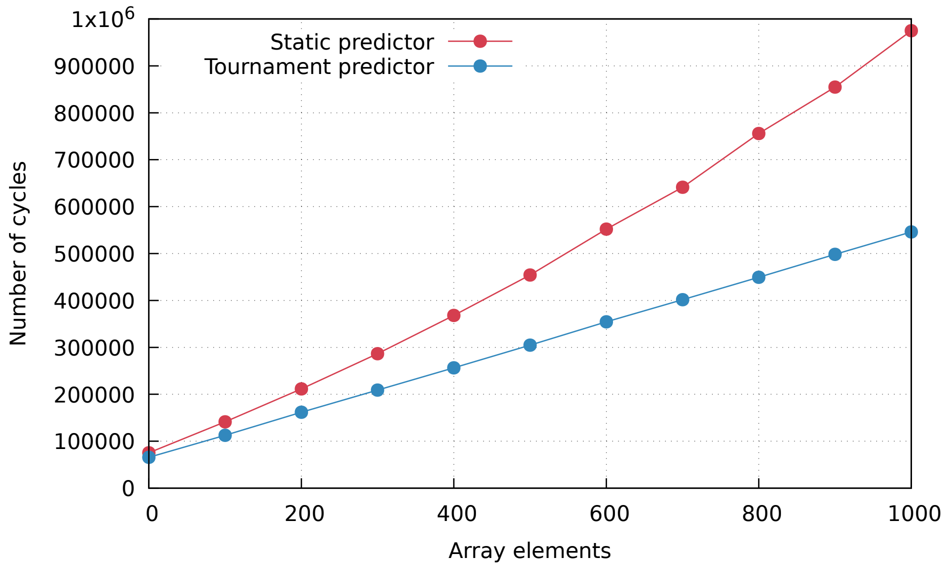

Figure 6: Cycles stalled by instruction cache misses.

Figure 6 illustrates the increase in the number of cycles spent by fetch on waiting for instructions to be fetched from primary memory and into the instruction cache. This increase can be attributed to the instructions that were actually required by the processor being replaced in the cache by instructions that the predictor incorrectly predicted to be required.

The increase is equal to 430,000 cycles for an array with 1000 elements, and is a significant contributor to the difference of cycles performed by the simulations using the two different predictors.

3.1.7 Total additional cycles

Previous sections have showed which operations performed by the pipeline contributed the most to the difference between the simulations making use of the static and tournament branch predictors.

For a 1000 key array, the fetch stage of the pipeline spent an additional 230,000 cycles on squashing instructions, a 990,000 cycle increase spent on actually fetching instructions, and a further 430,000 cycles spent on waiting for the instruction cache. These cycles add up to 1,650,000 cycles, which is more than the additional cycles that the CPU performed. This seems to be a minor discrepancy caused by the statistics module in the simulator itself, as the number of cycles executed by a single stage cannot be larger than the number of cycles performed by the CPU itself.

Although this discrepancy does affect the statistics based around the cycle count itself, it does not effect in any way the performance of the system, the branch mispredictions, or any other functional part of the processor. It simply means that the generated statistics are best compared to themselves for different predictors, as opposed to other statistics.

3.2 Detailed results

3.2.1 Bubble sort

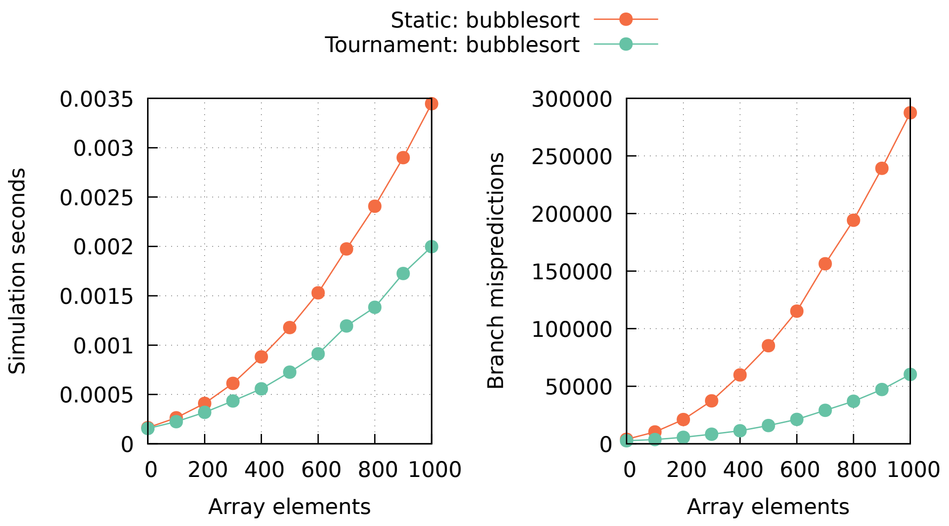

Figure 7: The simulation time and branch mispredictions for bubble sort.

Bubble sort is known for being a particularly slow comparison sort due to its $O(n^2)$ worst-case and average complexity, as opposed to the other more commonly used algorithms included in this investigation. As seen in Figure 7, the performance of bubble sort has decreased significantly, almost by 180% for a 1000 element array. Bubble sort was also one of the algorithms most significantly affected by the predictor, having over six times the number of branch mispredictions with a static predictor as opposed to the tournament predictor.

3.2.2 Insertion sort

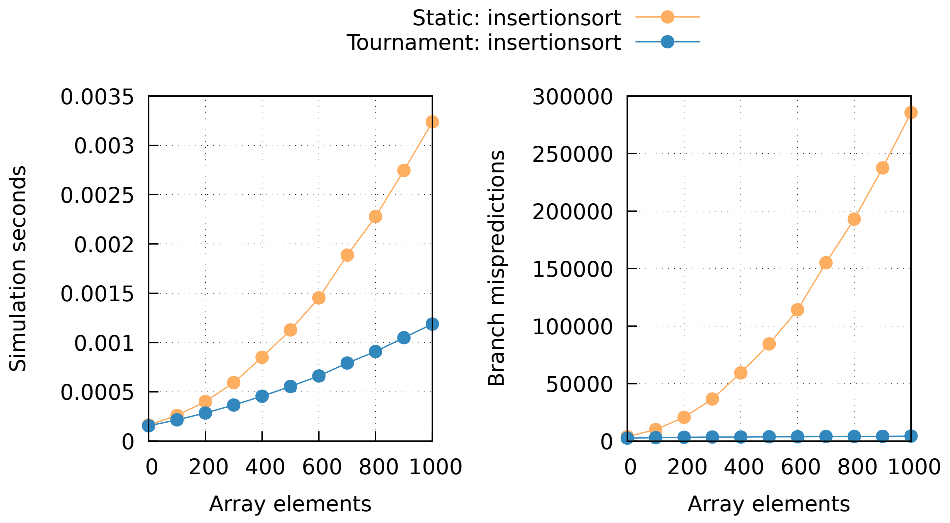

Figure 8: The simulation time and branch mispredictions for insertion sort.

Although insertion sort is also an algorithm with an average complexity of $O(n^2)$, when ran on a simulation using a tournament branch predictor it performed on par with other $O(n \log n)$ sorting algorithms (for the investigated array sizes). When ran on a simulation using the static predictor, the algorithms performance was drastically degraded, to the point where it performed similarly to bubble sort. This is the most drastic effect that the change of branch predictor had on a sorting algorithm, and it illustrates how much it matters, especially for simple sorting algorithms.

3.2.3 Heap sort, Quick sort, Merge sort, and Shell sort

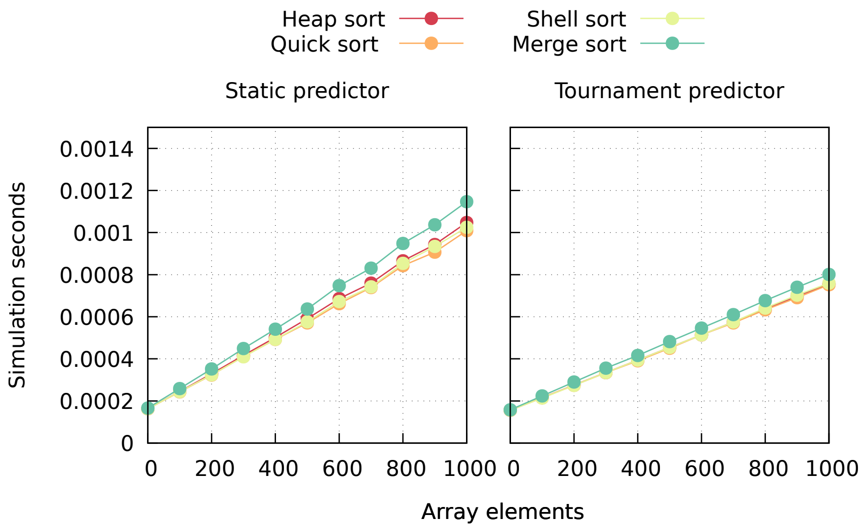

Figure 9: The simulation time and branch mispredictions for heap sort, quick sort, shell sort, and merge sort.

The four algorithms are all algorithms with $O(n \log n)$ worst case and average performance, which is why they are often chosen by programmers who require efficient sorting algorithms. All four of the algorithms saw a 30% performance decrease for a 1000 element array when used in conjunction with a static predictor, but they were not nearly as affected as the other two sorts with $O(n^2)$ complexities. This shows that branch mispredictions affect all sorting algorithms, but the ones that perform more operations and are already slower will experience more significant performance hits.

4 Conclusions

4.1 Limitations of this investigation

Certain effects of using a simulator instead of real hardware have to be taken into consideration.

The first effect I would like to discuss, is that modern processors have considerably more complex and longer pipelines than the Alpha 21264 modelled in the simulation. The processor is very dependant on the branch predictor (as these results have shown), but we cannot know to what extend this is reflected by modern hardware. It could be argued that due to a lower occurrence of branch mispredictions associated with having better predictors the effects of the mispredictions are mostly negated. On the other hand it could also be argued, that the considerably longer pipelines are much more vulnerable to mispredictions, potentially undercutting the work performed by the predictor.

The second factor that should be taken into consideration are the discrepancies in the simulator that make it impossible to precisely say which mechanisms were affected by the branch mispredictions the most, and how much they contributed to the slowdown of execution. Nonetheless, from the data available it can be said that the great majority of additional processing time was spent on work that simply did not have to be performed by the processor, and then on work that had to be performed to undo all of the unnecessary work including all of the overheads associated with it. There are minor issues with precision with this approach, it has to be stated that although the exact details of what consequences mispredictions may be not as clear as I would like, the overall results and the extent to which the predictions affected the execution of the algorithms tested have not been affected by these details.

Apart from internal sources of inaccuracies, there is always the possibility of external inaccuracies. Such inaccuracies may have an effect on the readers understanding of the topic, but they in no way influenced the investigation, or the simulation itself.

4.2 Closing statements

To answer the question: **To what extent do branch mispredictions affect the performance of well-known comparison sorts?**

As this investigation has shown, the performance of the tested sorting algorithms was undoubtedly greatly affected by branch mispredictions, even though the degree of impact varies among different algorithms.

This also implies that programs in general are also considerably impacted by mispredictions. Compilers and/or interpreters should attempt to optimise programs to minimise the number of branch instructions, and make necessary branches as predictive as possible.

References

[1] Y. Li, "CS232 Lecture Notes." `https://cs.colby.edu/courses/F20/cs232/notes/9.ISA(I).pdf`, 2019. Lecture notes, [Online; accessed 12-October-2023].

[2] Wikipedia contributors, "X86 — Wikipedia, The Free Encyclopedia." `https://en.wikipedia.org/w/index.php?title=X86&oldid=1174022952`, 2023. [Online; accessed 12-October-2023].

[3] Wikipedia contributors, "ARM architecture family — Wikipedia, The Free Encyclopedia." `https://en.wikipedia.org/w/index.php?title=ARM_architecture_family&oldid=1178831661`, 2023. [Online; accessed 12-October-2023].

[4] D. A. Watt, "Compilers and Interpreters." `https://www.dcs.gla.ac.uk/~daw/teaching/PL3/Slides/03.Compilers-and-interpreters.pdf`, 2012. Lecture notes, [Online; accessed 30-October-2023].

[5] M. M. K. Martin, "CIS 371 (Spring 2009): Digital Systems Organization and Design." `https://acg.cis.upenn.edu/milom/cis371-Spring09/lectures/01_isa.pdf`, 2009. Lecture notes, [Online; accessed 12-October-2023].

[6] N. Beckmann, "Instruction Set Architecture." `https://www.cs.cmu.edu/afs/cs/academic/class/15740-f18/www/lectures/02-isa.pdf`, 2018. Lecture notes, [Online; accessed 12-October-2023].

[7] N. Beckmann, "Branch Prediction." `https://www.cs.cmu.edu/afs/cs/academic/class/15740-f18/www/lectures/16-branch-prediction.pdf`, 2018. Lecture notes, [Online; accessed 12-October-2023].

[8] N. Beckmann, "Out-of-Order Executio & Dynamic Scheduling." `https://www.cs.cmu.edu/afs/cs/academic/class/15740-f18/www/lectures/15-ooo.pdf`, 2018. Lecture notes, [Online; accessed 12-October-2023].

[9] O. Mutlu, "Branch Prediction." `https://course.ece.cmu.edu/~ece740/`

f13/lib/exe/fetch.php?media=onur-740-fall13-module7.4.1-branch-prediction.pdf, 2013. Lecture notes, [Online; accessed 12-October-2023].

[10] P. Biggar, N. Nash, K. Williams, and D. Gregg, "An experimental study of sorting and branch prediction," *ACM J. Exp. Algorithmics*, vol. 12, jun 2008.

[11] Wikipedia contributors, "Alpha 21264 — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Alpha_21264&oldid=1165444030, 2023. [Online; accessed 11-October-2023].

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, Massachusetts London, England: The MIT Press, fourth ed., 2022. c© 2022 Massachusetts Institute of Technology.

[13] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3. Reading, Massachusetts: Addison-Wesley, second ed., 1998.

Appendices

```c
#include <stdio.h>
#include <stdlib.h>

void bubbleSort(int A[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = n - 1; j > i; j--) {
            if (A[j] < A[j - 1]) {

                int temp = A[j];
                A[j] = A[j - 1];
                A[j - 1] = temp;
            }
        }
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <num_elements> <element1> <element2> ...\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int *arr = (int *)malloc(n * sizeof(int));

    if (argc != n + 2) {
        printf("Error: Incorrect number of elements provided.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = atoi(argv[i + 2]);
    }

    printf("Before sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    bubbleSort(arr, n);

    printf("After sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Listing 1: Bubblesort implementation

```c
#include <stdio.h>
#include <stdlib.h>

void quicksort(int A[], int p, int r);
int partition(int A[], int p, int r);

void quicksort(int A[], int p, int r) {
    if (p < r) {
        int q = partition(A, p, r);
        quicksort(A, p, q - 1);
        quicksort(A, q + 1, r);
    }
}

int partition(int A[], int p, int r) {
    int x = A[r];
    int i = p - 1;

    for (int j = p; j <= r - 1; j++) {
        if (A[j] <= x) {
            i++;
            int temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

    int temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;

    return i + 1;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <num_elements> <element1> <element2> ...\
n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int *arr = (int *)malloc(n * sizeof(int));

    if (argc != n + 2) {
        printf("Error: Incorrect number of elements provided.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = atoi(argv[i + 2]);
    }

    printf("Before sorting: ");
```

```c
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    quicksort(arr, 0, n - 1);

    printf("After sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Listing 2: Quicksort implementation

```c
#include <stdio.h>
#include <stdlib.h>

void merge(int A[], int p, int q, int r);
void mergeSort(int A[], int p, int r);

void merge(int A[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = A[p + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = A[q + 1 + j];
    }

    int i = 0, j = 0, k = p;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        A[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        A[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int A[], int p, int r) {
    if (p < r) {
        int q = (p + r) / 2;
        mergeSort(A, p, q);
        mergeSort(A, q + 1, r);
        merge(A, p, q, r);
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <num_elements> <element1> <element2> ...\
n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int *arr = (int *)malloc(n * sizeof(int));

    if (argc != n + 2) {
        printf("Error: Incorrect number of elements provided.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = atoi(argv[i + 2]);
    }

    printf("Before sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    mergeSort(arr, 0, n - 1);

    printf("After sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Listing 3: Mergesort implementation

```c
#include <stdio.h>
#include <stdlib.h>

void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <num_elements> <element1> <element2> ...\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int *arr = (int *)malloc(n * sizeof(int));

    if (argc != n + 2) {
        printf("Error: Incorrect number of elements provided.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = atoi(argv[i + 2]);
    }

    printf("Before sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    shellSort(arr, n);

    printf("After sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Listing 4: Shellsort implementation

```c
#include <stdio.h>
#include <stdlib.h>

void insertionSort(int A[], int n) {
    for (int i = 1; i < n; i++) {
        int key = A[i];
        int j = i - 1;

        while (j >= 0 && A[j] > key) {
            A[j + 1] = A[j];
            j--;
        }

        A[j + 1] = key;
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <num_elements> <element1> <element2> ...\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int *arr = (int *)malloc(n * sizeof(int));

    if (argc != n + 2) {
        printf("Error: Incorrect number of elements provided.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = atoi(argv[i + 2]);
    }

    printf("Before sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    insertionSort(arr, n);

    printf("After sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Listing 5: Insertion sort implementation

```c
#include <stdio.h>
#include <stdlib.h>

void maxHeapify(int A[], int i, int heapSize);
void buildMaxHeap(int A[], int n);
void heapSort(int A[], int n);

void maxHeapify(int A[], int i, int heapSize) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < heapSize && A[left] > A[largest]) {
        largest = left;
    }

    if (right < heapSize && A[right] > A[largest]) {
        largest = right;
    }

    if (largest != i) {
        int temp = A[i];
        A[i] = A[largest];
        A[largest] = temp;
        maxHeapify(A, largest, heapSize);
    }
}

void buildMaxHeap(int A[], int n) {
    int heapSize = n;

    for (int i = n / 2 - 1; i >= 0; i--) {
        maxHeapify(A, i, heapSize);
    }
}

```

```c
void heapSort(int A[], int n) {
    buildMaxHeap(A, n);

    for (int i = n - 1; i >= 1; i--) {
        int temp = A[0];
        A[0] = A[i];
        A[i] = temp;

        maxHeapify(A, 0, i);
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <num_elements> <element1> <element2> ...\
  n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int *arr = (int *)malloc(n * sizeof(int));

    if (argc != n + 2) {
        printf("Error: Incorrect number of elements provided.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = atoi(argv[i + 2]);
    }

    printf("Before sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    heapSort(arr, n);

    printf("After sorting: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Listing 6: Heapsort implementation