

Why is processing a sorted array faster than processing an unsorted array?

Asked 12 years, 6 months ago Modified 5 days ago Viewed 1.9m times



In this C++ code, sorting the data (*before* the timed region) makes the primary loop ~6x faster:

27385



```
#include <algorithm>
#include <ctime>
#include <iostream>

int main()
{
    // Generate data
    const unsigned arraySize = 32768;
    int data[arraySize];

    for (unsigned c = 0; c < arraySize; ++c)
        data[c] = std::rand() % 256;

    // !!! With this, the next loop runs faster.
    std::sort(data, data + arraySize);

    // Test
    clock_t start = clock();
    long long sum = 0;
    for (unsigned i = 0; i < 100000; ++i)
    {
        for (unsigned c = 0; c < arraySize; ++c)
        {
            // Primary loop.
            if (data[c] >= 128)
                sum += data[c];
        }
    }

    double elapsedTime = static_cast<double>(clock()-start) / CLOCKS_PER_SEC;

    std::cout << elapsedTime << '\n';
    std::cout << "sum = " << sum << '\n';
}
```

- Without `std::sort(data, data + arraySize);`, the code runs in 11.54 seconds.
- With the sorted data, the code runs in 1.93 seconds.

(Sorting itself takes more time than this one pass over the array, so it's not actually worth doing if we needed to calculate this for an unknown array.)

Initially, I thought this might be just a language or compiler anomaly, so I tried Java:

```

import java.util.Arrays;
import java.util.Random;

public class Main
{
    public static void main(String[] args)
    {
        // Generate data
        int arraySize = 32768;
        int data[] = new int[arraySize];

        Random rnd = new Random(0);
        for (int c = 0; c < arraySize; ++c)
            data[c] = rnd.nextInt() % 256;

        // !!! With this, the next loop runs faster
        Arrays.sort(data);

        // Test
        long start = System.nanoTime();
        long sum = 0;
        for (int i = 0; i < 100000; ++i)
        {
            for (int c = 0; c < arraySize; ++c)
            {
                // Primary loop.
                if (data[c] >= 128)
                    sum += data[c];
            }
        }

        System.out.println((System.nanoTime() - start) / 1000000000.0);
        System.out.println("sum = " + sum);
    }
}

```

With a similar but less extreme result.

My first thought was that sorting brings the data into the [cache](#), but that's silly because the array was just generated.

- What is going on?
- Why is processing a sorted array faster than processing an unsorted array?

The code is summing up some independent terms, so the order should not matter.

Related / follow-up Q&As about the same effect with different/later compilers and options:

- [Why is processing an unsorted array the same speed as processing a sorted array with modern x86-64 clang?](#) - **modern C++ compilers auto-vectorize the loop**, especially when SSE4.1 or AVX2 is available. This avoids any data-dependent branching so performance isn't data-dependent.

- [gcc optimization flag -O3 makes code slower than -O2](#) - branchless scalar with `cmov` can result in a longer dependency chain (especially when GCC chooses poorly), creating a latency bottleneck that makes it slower than branchy asm for the sorted case.

[java](#)[c++](#)[performance](#)[cpu-architecture](#)[branch-prediction](#)[Share](#)[Improve this question](#)[Follow](#)

edited Oct 17 at 14:30



Peter Cordes

361k ● 49 ● 699 ● 958

asked Jun 27, 2012 at 13:51



GManNickG

503k ● 53 ● 502 ● 549

- 143 Another observation is that you don't need to sort the array, but you just need to partition it with the value 128. Sorting is $n \cdot \log(n)$, whereas partitioning is just linear. Basically it is just one run of the quick sort partitioning step with the pivot chosen to be 128. Unfortunately in C++ there is just `nth_element` function, which partition by position, not by value.
– [Šimon Hrabec](#) May 11, 2018 at 12:45
- 59 @screwnut here's an experiment which would show that partitioning is sufficient: create an unsorted but partitioned array with otherwise random contents. Measure time. Sort it. Measure time again. The two measurements should be basically indistinguishable. (Experiment 2: create a random array. Measure time. Partition it. Measure time again. You should see the same speed-up as sorting. You could roll the two experiments into one.)
– [Jonas Kölker](#) Oct 5, 2020 at 8:26
- 54 Btw. on Apple M1 the code runs in 17 sec unsorted, and in 7 sec sorted, so the branch prediction penalty isn't that bad on risc architecture. – [Piotr Czapla](#) Mar 31, 2021 at 9:07
- 47 @RomanYavorskyi: It depends on the compiler. If they make branchless asm for this specific test (e.g. as part of vectorizing with SIMD like in [Why is processing an unsorted array the same speed as processing a sorted array with modern x86-64 clang?](#), or just with scalar `cmov` ([gcc optimization flag -O3 makes code slower than -O2](#)), then sorted or not doesn't matter. But unpredictable branches are still a very real thing when it's not as simple as counting, so it would be insane to delete this question. – [Peter Cordes](#) Apr 15, 2021 at 6:31
- 30 @screwnut: To be fair, though, it still wouldn't be worth partitioning first, because partitioning requires conditional copying or swapping based on the same `array[i] > 128` compare. (Unless you're going to be counting multiple times, and you want to partition the bulk of an array so it's still fast, only mispredicting in an unpartitioned part after some appends or modifications). If you can get the compiler to do it, ideally just vectorize with SIMD, or at least use branchless scalar if the data is unpredictable. (See previous comment for links.)
– [Peter Cordes](#) Apr 15, 2021 at 6:46

25 Answers

Sorted by: Highest score (default)

You are a victim of [branch prediction](#) fail.

35119 What is Branch Prediction?



Consider a railroad junction:



+2200



[Image](#) by Mecanismo, via Wikimedia Commons. Used under the [CC-BY-SA 3.0](#) license.

Now for the sake of argument, suppose this is back in the 1800s - before long-distance or radio communication.

You are a blind operator of a junction and you hear a train coming. You have no idea which way it is supposed to go. You stop the train to ask the driver which direction they want. And then you set the switch appropriately.

Trains are heavy and have a lot of inertia, so they take forever to start up and slow down.

Is there a better way? You guess which direction the train will go!

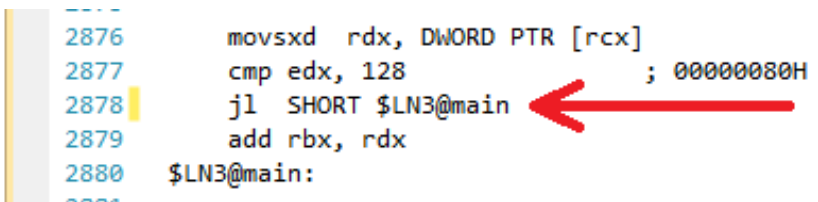
- If you guessed right, it continues on.
- If you guessed wrong, the driver will stop, back up, and yell at you to flip the switch. Then it can restart down the other path.

If you guess right every time, the train will never have to stop.

If you guess wrong too often, the train will spend a lot of time stopping, backing up, and restarting.

Consider an if-statement: At the processor level, it is a branch instruction:

```
if (data[c] >= 128)
    sum += data[c];
```



```
2876 movsxd rdx, DWORD PTR [rcx]
2877 cmp edx, 128 ; 00000080H
2878 jl SHORT $LN3@main
2879 add rbx, rdx
2880 $LN3@main:
```

You are a processor and you see a branch. You have no idea which way it will go. What do you do? You halt execution and wait until the previous instructions are complete. Then you continue down the correct path.

Modern processors are complicated and have long pipelines. This means they take forever to "warm up" and "slow down".

Is there a better way? You guess which direction the branch will go!

- If you guessed right, you continue executing.
- If you guessed wrong, you need to flush the pipeline and roll back to the branch. Then you can restart down the other path.

If you guess right every time, the execution will never have to stop.

If you guess wrong too often, you spend a lot of time stalling, rolling back, and restarting.

This is branch prediction. I admit it's not the best analogy since the train could just signal the direction with a flag. But in computers, the processor doesn't know which direction a branch will go until the last moment.

How would you strategically guess to minimize the number of times that the train must back up and go down the other path? You look at the past history! If the train goes left 99% of the time, then you guess left. If it alternates, then you alternate your guesses. If it goes one way every three times, you guess the same...

In other words, you try to identify a pattern and follow it. This is more or less how branch predictors work.

Most applications have well-behaved branches. Therefore, modern branch predictors will typically achieve >90% hit rates. But when faced with unpredictable branches with no recognizable patterns, branch predictors are virtually useless.

Further reading: ["Branch predictor" article on Wikipedia](#).

As hinted from above, the culprit is this if-statement:

```
if (data[c] >= 128)
    sum += data[c];
```

Notice that the data is evenly distributed between 0 and 255. When the data is sorted, roughly the first half of the iterations will not enter the if-statement. After that, they will all enter the if-statement.

This is very friendly to the branch predictor since the branch consecutively goes the same direction many times. Even a simple saturating counter will correctly predict the branch except for the few iterations after it switches direction.

Quick visualization:

```
T = branch taken
N = branch not taken

data[] = 0, 1, 2, 3, 4, ... 126, 127, 128, 129, 130, ... 250, 251, 252, ...
branch = N N N N N ... N N T T T ... T T T ...

        = NNNNNNNNNNNN ... NNNNNNTTTTTTTTTT ... TTTTTTTTTT (easy to predict)
```

However, when the data is completely random, the branch predictor is rendered useless, because it can't predict random data. Thus there will probably be around 50% misprediction (no better than random guessing).

```
data[] = 226, 185, 125, 158, 198, 144, 217, 79, 202, 118, 14, 150, 177, 182, ...
branch = T, T, N, T, T, T, T, N, T, N, N, T, T, T ...

        = TTNTTTNTNNTTT ... (completely random - impossible to predict)
```

What can be done?

If the compiler isn't able to optimize the branch into a conditional move, you can try some hacks if you are willing to sacrifice readability for performance.

Replace:

```
if (data[c] >= 128)
    sum += data[c];
```

with:

```
int t = (data[c] - 128) >> 31;
sum += ~t & data[c];
```


This eliminates the branch and replaces it with some bitwise operations.

(Note that this hack is not strictly equivalent to the original if-statement. But in this case, it's valid for all the input values of `data[]`.)

Benchmarks: Core i7 920 @ 3.5 GHz

C++ - Visual Studio 2010 - x64 Release

Scenario	Time (seconds)
Branching - Random data	11.777
Branching - Sorted data	2.352
Branchless - Random data	2.564
Branchless - Sorted data	2.587

Java - NetBeans 7.1.1 JDK 7 - x64

Scenario	Time (seconds)
Branching - Random data	10.93293813
Branching - Sorted data	5.643797077
Branchless - Random data	3.113581453
Branchless - Sorted data	3.186068823

Observations:

- **With the Branch:** There is a huge difference between the sorted and unsorted data.
- **With the Hack:** There is no difference between sorted and unsorted data.
- In the C++ case, the hack is actually a tad slower than with the branch when the data is sorted.

A general rule of thumb is to avoid data-dependent branching in critical loops (such as in this example).

Update:

- GCC 4.6.1 with `-O3` or `-ftree-vectorize` on x64 is able to generate a conditional move, so there is no difference between the sorted and unsorted data

- both are fast. This is called "if-conversion" (to branchless) and is necessary for vectorization but also sometimes good for scalar.

(Or somewhat fast: for the already-sorted case, `cmov` can be slower especially if GCC puts it on the critical path instead of just `add`, especially on Intel before Broadwell where `cmov` has 2-cycle latency: [gcc optimization flag -O3 makes code slower than -O2](#))

- VC++ 2010 is unable to generate conditional moves for this branch even under `/Ox`.
- [Intel C++ Compiler](#) (ICC) 11 does something miraculous. It [interchanges the two loops](#), thereby hoisting the unpredictable branch to the outer loop. Not only is it immune to the mispredictions, it's also twice as fast as whatever VC++ and GCC can generate! In other words, ICC took advantage of the test-loop to defeat the benchmark...
- If you give the Intel compiler the branchless code, it just outright vectorizes it... and is just as fast as with the branch (with the loop interchange).
- Clang also vectorizes the `if()` version, as will GCC 5 and later with `-O3`, even though it takes quite a few instructions to sign-extend to the 64-bit sum on x86 without SSE4 or AVX2. (`-march=x86-64-v2` or `v3`). See [Why is processing an unsorted array the same speed as processing a sorted array with modern x86-64 clang?](#)

This goes to show that even mature modern compilers can vary wildly in their ability to optimize code...

Share

Improve this answer

Follow

edited Mar 4 at 17:37



Peter Cordes

361k ● 49 ● 699 ● 958

answered Jun 27, 2012 at 13:56



Mysticial

471k ● 46 ● 339 ● 336

60 wait a second, doesnt shifting negative values to the right yield implementation-defined values? `int t = (data[c] - 128) >> 31; sum += ~t & data[c];` – [Matias Chara](#) Jul 12, 2020 at 23:52

84 Incidentally branch prediction failure can also be [exploited by a program to obtain crypto keys being used by another program](#) on the same CPU core. – [mins](#) Oct 16, 2020 at 14:04 ✎

55 @Mycotina, I'm no expert, but what I understand is: the processor needs multiple steps to execute a single instruction (fetching, decoding, etc) -- this is called "instruction pipelining" -- so, as an optimization, it will fetch multiple instructions at once and "warm up" the next instructions while executing the current one. If the wrong branch is chosen, the instructions being "warmed up" in the pipeline must be discarded, so that the instructions on the right branch can be put into the pipeline instead. – [Raphael](#) Jan 5, 2021 at 15:51

31 @Mycotina It's easier to understand when you think of the instruction pipeline cache as tracks, the train (with cars) as the instructions, and the indicator of whether you go left or right by some dude at the END of the train; not the beginning. By the time you see him to know you've guessed right, not only is it too late to switch things, the pipeline ahead is already

populated, but in the wrong direction. If you guessed wrong the predicted pipeline needs to be thrown out (derail the train; drag it back before the switch house, put it back on the tracks, and send it the other way). – [WhozCraig](#) Jan 8, 2021 at 2:03

- 24 @C.Binair Primarily it's runtime, i.e. processor predicts branches while executing the code. The processor also remembers previous results and use that to predict next jump. However, compiler can provide some initial hints for branch prediction while compiling - search for "likely" and "unlikely" attributes. So you could say the answer is kinda both, but runtime is when it actually happens. – [Tom](#) Mar 10, 2021 at 14:47



Branch prediction.

4726



With a sorted array, the condition `data[c] >= 128` is first `false` for a streak of values, then becomes `true` for all later values. That's easy to predict. With an unsorted array, you pay for the branching cost.



Share Improve this answer

edited Jun 20, 2020 at 9:12

answered Jun 27, 2012 at 13:54



Follow



Community Bot

1 • 1



[Daniel Fischer](#)

184k • 17 • 317 • 434

- 170 Does branch prediction work better on sorted arrays vs. arrays with different patterns? For example, for the array `--> { 10, 5, 20, 10, 40, 20, ... }` the next element in the array from the pattern is 80. Would this kind of array be sped up by branch prediction in which the next element is 80 here if the pattern is followed? Or does it usually only help with sorted arrays? – [Adam Freeman](#) Sep 23, 2014 at 18:58

- 217 So basically everything I conventionally learned about big-O is out of the window? Better to incur a sorting cost than a branching cost? – [Agrim Pathak](#) Oct 30, 2014 at 7:51

- 207 @AgrimPathak That depends. For not too large input, an algorithm with higher complexity is faster than an algorithm with lower complexity when the constants are smaller for the algorithm with higher complexity. Where the break-even point is can be hard to predict. Also, [compare this](#), locality is important. Big-O is important, but it is not the sole criterion for performance. – [Daniel Fischer](#) Oct 30, 2014 at 10:14

- 112 When does branch prediction takes place? When does language will know that array is sorted? I'm thinking of situation of array that looks like: `[1,2,3,4,5,...998,999,1000, 3, 10001, 10002]` ? will this obscure 3 increase running time? Will it be as long as unsorted array? – [Filip Bartuzi](#) Nov 9, 2014 at 13:37

- 113 @FilipBartuzi Branch prediction takes place in the processor, below the language level (but the language may offer ways to tell the compiler what's likely, so the compiler can emit code suited to that). In your example, the out-of-order 3 will lead to a branch-misprediction (for appropriate conditions, where 3 gives a different result than 1000), and thus processing that array will likely take a couple dozen or hundred nanoseconds longer than a sorted array would, hardly ever noticeable. What costs time is i high rate of mispredictions, one misprediction per 1000 isn't much. – [Daniel Fischer](#) Nov 9, 2014 at 13:49



The reason why performance improves drastically when the data is sorted is that the branch prediction penalty is removed, as explained beautifully in [Mysticial's answer](#).

3841



Now, if we look at the code

```
if (data[c] >= 128)
    sum += data[c];
```



+150



we can find that the meaning of this particular `if... else...` branch is to add something when a condition is satisfied. This type of branch can be easily transformed into a **conditional move** statement, which would be compiled into a conditional move instruction: `cmovl`, in an `x86` system. The branch and thus the potential branch prediction penalty is removed.

In `c`, thus `c++`, the statement, which would compile directly (without any optimization) into the conditional move instruction in `x86`, is the ternary operator `... ? ... : ...`. So we rewrite the above statement into an equivalent one:

```
sum += data[c] >= 128 ? data[c] : 0;
```

While maintaining readability, we can check the speedup factor.

On an Intel [Core i7-2600K](#) @ 3.4 GHz and Visual Studio 2010 Release Mode, the benchmark is:

x86

Scenario	Time (seconds)
Branching - Random data	8.885
Branching - Sorted data	1.528
Branchless - Random data	3.716
Branchless - Sorted data	3.71

x64

Scenario	Time (seconds)
Branching - Random data	11.302
Branching - Sorted data	1.830
Branchless - Random data	2.736

Scenario	Time (seconds)
Branchless - Sorted data	2.737

The result is robust in multiple tests. We get a great speedup when the branch result is unpredictable, but we suffer a little bit when it is predictable. In fact, when using a conditional move, the performance is the same regardless of the data pattern.

Now let's look more closely by investigating the `x86` assembly they generate. For simplicity, we use two functions `max1` and `max2`.

`max1` uses the conditional branch `if... else ...`:

```
int max1(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}
```

`max2` uses the ternary operator `... ? ... : ...`:

```
int max2(int a, int b) {
    return a > b ? a : b;
}
```

On an x86-64 machine, `gcc -s` generates the assembly below.

```
:max1
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %eax
    cmpl    -8(%rbp), %eax
    jle     .L2
    movl    -4(%rbp), %eax
    movl    %eax, -12(%rbp)
    jmp     .L4
.L2:
    movl    -8(%rbp), %eax
    movl    %eax, -12(%rbp)
.L4:
    movl    -12(%rbp), %eax
    leave
    ret

:max2
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %eax
    cmpl    %eax, -8(%rbp)
    cmovge  -8(%rbp), %eax
    leave
    ret
```

`max2` uses much less code due to the usage of instruction `cmovge`. But the real gain is that `max2` does not involve branch jumps, `jmp`, which would have a significant performance penalty if the predicted result is not right.

So why does a conditional move perform better?

In a typical `x86` processor, the execution of an instruction is divided into several stages. Roughly, we have different hardware to deal with different stages. So we do not have to wait for one instruction to finish to start a new one. This is called [pipelining](#).

In a branch case, the following instruction is determined by the preceding one, so we cannot do pipelining. We have to either wait or predict.

In a conditional move case, the execution of conditional move instruction is divided into several stages, but the earlier stages like `Fetch` and `Decode` do not depend on the result of the previous instruction; only the latter stages need the result. Thus, we wait a fraction of one instruction's execution time. This is why the conditional move version is slower than the branch when the prediction is easy.

The book [Computer Systems: A Programmer's Perspective, second edition](#) explains this in detail. You can check Section 3.6.6 for *Conditional Move Instructions*, entire Chapter 4 for *Processor Architecture*, and Section 5.11.2 for special treatment for *Branch Prediction and Misprediction Penalties*.

Sometimes, some modern compilers can optimize our code to assembly with better performance, and sometimes some compilers can't (the code in question is using Visual Studio's native compiler). Knowing the performance difference between a branch and a conditional move when unpredictable can help us write code with better performance when the scenario gets so complex that the compiler can not optimize them automatically.

Share Improve this answer

Follow

edited Jul 3, 2022 at 15:12



Mihir Ajmera

127 ● 1 ● 9

answered Jun 28, 2012 at 2:14



WiSaGaN

48k ● 10 ● 58 ● 90

13 [stackoverflow.com/questions/9745389/...](https://stackoverflow.com/questions/9745389/) – Linus Fernandes Dec 3, 2020 at 5:32

4 You forgot to enable optimization; it's not useful to benchmark debug builds that [store/reload everything to the stack](#). Use `gcc -O2 -fno-tree-vectorize -S` if you want efficient scalar asm that hopefully uses `cmov`. (`-O3` would probably auto-vectorize, so would `-O2` with GCC12 or later.) See also [gcc optimization flag -O3 makes code slower than -O2](#) (for the sorted case, when it uses `cmov` poorly for the `if`). – Peter Cordes Jul 3, 2022 at 21:05 ✎

I think calling the ternary operator "branchless" is the wrong idea here. You are relying on a language optimization feature which just so happens to be used in one of the two functions, even when they are semantically equivalent. – ed__ Nov 13 at 15:38



If you are curious about even more optimizations that can be done to this code, consider this:

2653



Starting with the original loop:



```
for (unsigned i = 0; i < 100000; ++i)
{
    for (unsigned j = 0; j < arraySize; ++j)
    {
        if (data[j] >= 128)
            sum += data[j];
    }
}
```

With loop interchange, we can safely change this loop to:

```
for (unsigned j = 0; j < arraySize; ++j)
{
    for (unsigned i = 0; i < 100000; ++i)
    {
        if (data[j] >= 128)
            sum += data[j];
    }
}
```

Then, you can see that the `if` conditional is constant throughout the execution of the `i` loop, so you can hoist the `if` out:

```
for (unsigned j = 0; j < arraySize; ++j)
{
    if (data[j] >= 128)
    {
        for (unsigned i = 0; i < 100000; ++i)
        {
            sum += data[j];
        }
    }
}
```

Then, you see that the inner loop can be collapsed into one single expression, assuming the floating point model allows it (`/fp:fast` is thrown, for example)

```
for (unsigned j = 0; j < arraySize; ++j)
{
    if (data[j] >= 128)
    {
        sum += data[j] * 100000;
    }
}
```

That one is 100,000 times faster than before.



28 This is delightful, but for anyone who might not understand the joke: the purpose of the `for (unsigned i = 0; i < 100000; ++i)` in the original code is to run the code being benchmarked many times so as to be able to more accurately measure the time taken to execute it—measurements that would otherwise be compromised by limited timer resolution and/or jitter caused by other processes (including the OS) unpredictably preempting this one. It is true that the benchmarked code does take much, *much* less time to execute if you run it only once! – [cpcallen](#) Oct 11, 2022 at 13:13

2 @NameSurname - your edit didn't actually change the syntax highlighting; Java/C++ syntax highlighting was already the default because of the tags on the question, and Java vs. C++ highlighting don't differ in how they highlight this so it didn't matter which it picked. Please don't clutter up the edit queue (or spend your time on) edits that don't actually change anything. Also, in some of your other recent edit suggestions, don't use `code formatting` for names of programs, only things that are actual shell commands or source-code names of functions or variables. It's bad for screen readers – [Peter Cordes](#) Jan 2 at 21:37



2198



No doubt some of us would be interested in ways of identifying code that is problematic for the CPU's branch-predictor. The Valgrind tool `cachegrind` has a branch-predictor simulator, enabled by using the `--branch-sim=yes` flag. Running it over the examples in this question, with the number of outer loops reduced to 10000 and compiled with `g++`, gives these results:

Sorted:

```
==32551== Branches:          656,645,130 ( 656,609,208 cond + 35,922 ind)
==32551== Mispredicts:       169,556 ( 169,095 cond + 461 ind)
==32551== Mispred rate:      0.0% ( 0.0% + 1.2% )
```

Unsorted:

```
==32555== Branches:          655,996,082 ( 655,960,160 cond + 35,922 ind)
==32555== Mispredicts:       164,073,152 ( 164,072,692 cond + 460 ind)
==32555== Mispred rate:      25.0% ( 25.0% + 1.2% )
```

Drilling down into the line-by-line output produced by `cg_annotate` we see for the loop in question:

Sorted:

```
      Bc      Bcm Bi Bim
10,001      4  0  0    for (unsigned i = 0; i < 10000; ++i)
.           .  .  .    {
.           .  .  .    // primary loop
```


327,690,000	10,016	0	0		for (unsigned c = 0; c < arraySize; ++c)
.	.	.	.		{
327,680,000	10,006	0	0		if (data[c] >= 128)
0	0	0	0		sum += data[c];
.	.	.	.		}
.	.	.	.		}

Unsorted:

	Bc	Bcm	Bi	Bim	
	10,001	4	0	0	for (unsigned i = 0; i < 10000; ++i)
	{
	// primary loop
327,690,000	10,038	0	0		for (unsigned c = 0; c < arraySize;
++c)					{
.	if (data[c] >= 128)
327,680,000	164,050,007	0	0		sum += data[c];
0	0	0	0		}
.	}
.	}

This lets you easily identify the problematic line - in the unsorted version the `if (data[c] >= 128)` line is causing 164,050,007 mispredicted conditional branches (`Bcm`) under cachegrind's branch-predictor model, whereas it's only causing 10,006 in the sorted version.

Alternatively, on Linux you can use the performance counters subsystem to accomplish the same task, but with native performance using CPU counters.

```
perf stat ./sumtest_sorted
```

Sorted:

Performance counter stats for './sumtest_sorted':

11808.095776	task-clock	#	0.998 CPUs utilized
1,062	context-switches	#	0.090 K/sec
14	CPU-migrations	#	0.001 K/sec
337	page-faults	#	0.029 K/sec
26,487,882,764	cycles	#	2.243 GHz
41,025,654,322	instructions	#	1.55 insns per cycle
6,558,871,379	branches	#	555.455 M/sec
567,204	branch-misses	#	0.01% of all branches

11.827228330 seconds time elapsed

Unsorted:

Performance counter stats for './sumtest_unsorted':

28877.954344	task-clock	#	0.998 CPUs utilized
--------------	------------	---	---------------------

```

2,584 context-switches      # 0.089 K/sec
18 CPU-migrations          # 0.001 K/sec
335 page-faults            # 0.012 K/sec
65,076,127,595 cycles      # 2.253 GHz
41,032,528,741 instructions # 0.63 insns per cycle
6,560,579,013 branches     # 227.183 M/sec
1,646,394,749 branch-misses # 25.10% of all branches

28.935500947 seconds time elapsed

```

It can also do source code annotation with dissassembly.

```

perf record -e branch-misses ./sumtest_unsorted
perf annotate -d sumtest_unsorted

```

```

Percent | Source code & Disassembly of sumtest_unsorted
-----|-----
...
      :      sum += data[c];
0.00 :      400a1a:      mov     -0x14(%rbp),%eax
39.97 :      400a1d:      mov     %eax,%eax
5.31 :      400a1f:      mov     -0x20040(%rbp,%rax,4),%eax
4.60 :      400a26:      cltq
0.00 :      400a28:      add     %rax,-0x30(%rbp)
...

```

See [the performance tutorial](#) for more details.

Share Improve this answer

edited Oct 18, 2012 at 19:20

answered Oct 12, 2012 at 5:53

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133



caf

239k ● 41 ● 335 ● 474

104 This is scary, in the unsorted list, there should be 50% chance of hitting the add. Somehow the branch prediction only has a 25% miss rate, how can it do better than 50% miss?

– TallBrianL Dec 9, 2013 at 4:00

174 @tall.b.lo: The 25% is of all branches - there are *two* branches in the loop, one for `data[c] >= 128` (which has a 50% miss rate as you suggest) and one for the loop condition `c < arraySize` which has ~0% miss rate. – caf Dec 9, 2013 at 4:29

6 Note that benchmarking / profiling un-optimized ("debug mode") code is normally a bad idea. With optimization, that version without branch misses would be faster by a much larger margin, not stalling on store/reload latency for local variables. The actual branch mispredict rate for the critical branch should be about the same, though (assuming there is one: modern compilers can [vectorize this](#) or otherwise make branchless asm). Loop unrolling could change the overall miss rate by running a predictable loop branch less often. – Peter Cordes Apr 21, 2022 at 5:31



I just read up on this question and its answers, and I feel an answer is missing.

1623

A common way to eliminate branch prediction that I've found to work particularly good in managed languages is a table lookup instead of using a branch (although I haven't tested it in this case).



This approach works in general if:



1. it's a small table and is likely to be cached in the processor, and
2. you are running things in a quite tight loop and/or the processor can preload the data.

Background and why

From a processor perspective, your memory is slow. To compensate for the difference in speed, a couple of caches are built into your processor (L1/L2 cache). So imagine that you're doing your nice calculations and figure out that you need a piece of memory. The processor will get its 'load' operation and loads the piece of memory into cache -- and then uses the cache to do the rest of the calculations. Because memory is relatively slow, this 'load' will slow down your program.

Like branch prediction, this was optimized in the Pentium processors: the processor predicts that it needs to load a piece of data and attempts to load that into the cache before the operation actually hits the cache. As we've already seen, branch prediction sometimes goes horribly wrong -- in the worst case scenario you need to go back and actually wait for a memory load, which will take forever (**in other words: failing branch prediction is bad, a memory load after a branch prediction fail is just horrible!**).

Fortunately for us, if the memory access pattern is predictable, the processor will load it in its fast cache and all is well.

The first thing we need to know is what is *small*? While smaller is generally better, a rule of thumb is to stick to lookup tables that are ≤ 4096 bytes in size. As an upper limit: if your lookup table is larger than 64K it's probably worth reconsidering.

Constructing a table

So we've figured out that we can create a small table. Next thing to do is get a lookup function in place. Lookup functions are usually small functions that use a couple of basic integer operations (and, or, xor, shift, add, remove and perhaps multiply). You want to have your input translated by the lookup function to some kind of 'unique key' in your table, which then simply gives you the answer of all the work you wanted it to do.

In this case: ≥ 128 means we can keep the value, < 128 means we get rid of it. The easiest way to do that is by using an 'AND': if we keep it, we AND it with 7FFFFFFF; if we want to get rid of it, we AND it with 0. Notice also that 128 is a power of 2 -- so we

can go ahead and make a table of 32768/128 integers and fill it with one zero and a lot of 7FFFFFFF's.

Managed languages

You might wonder why this works well in managed languages. After all, managed languages check the boundaries of the arrays with a branch to ensure you don't mess up...

Well, not exactly... :-)

There has been quite some work on eliminating this branch for managed languages. For example:

```
for (int i = 0; i < array.Length; ++i)
{
    // Use array[i]
}
```

In this case, it's obvious to the compiler that the boundary condition will never be hit. At least the Microsoft JIT compiler (but I expect Java does similar things) will notice this and remove the check altogether. WOW, that means no branch. Similarly, it will deal with other obvious cases.

If you run into trouble with lookups in managed languages -- the key is to add a `& 0x[something]FFF` to your lookup function to make the boundary check predictable -- and watch it going faster.

The result of this case

```
// Generate data
int arraySize = 32768;
int[] data = new int[arraySize];

Random random = new Random(0);
for (int c = 0; c < arraySize; ++c)
{
    data[c] = random.Next(256);
}

/*To keep the spirit of the code intact, I'll make a separate lookup table
(I assume we cannot modify 'data' or the number of loops)*/

int[] lookup = new int[256];

for (int c = 0; c < 256; ++c)
{
    lookup[c] = (c >= 128) ? c : 0;
}

// Test
DateTime startTime = System.DateTime.Now;
```

```

long sum = 0;

for (int i = 0; i < 100000; ++i)
{
    // Primary loop
    for (int j = 0; j < arraySize; ++j)
    {
        /* Here you basically want to use simple operations - so no
        random branches, but things like &, |, *, -, +, etc. are fine. */
        sum += lookup[data[j]];
    }
}

DateTime endTime = System.DateTime.Now;
Console.WriteLine(endTime - startTime);
Console.WriteLine("sum = " + sum);
Console.ReadLine();

```

Share Improve this answer

edited Jan 16, 2019 at 4:47

answered Apr 24, 2013 at 6:26

Follow



Palec

13.5k ● 8 ● 74 ● 142



atlaste

31.1k ● 3 ● 61 ● 90



As data is distributed between 0 and 255 when the array is sorted, around the first half of the iterations will not enter the `if`-statement (the `if` statement is shared below).

1442



```

if (data[c] >= 128)
    sum += data[c];

```



The question is: What makes the above statement not execute in certain cases as in case of sorted data? Here comes the "branch predictor". A branch predictor is a digital circuit that tries to guess which way a branch (e.g. an `if-then-else` structure) will go before this is known for sure. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving high effective performance!

Let's do some bench marking to understand it better

The performance of an `if`-statement depends on whether its condition has a predictable pattern. If the condition is always true or always false, the branch prediction logic in the processor will pick up the pattern. On the other hand, if the pattern is unpredictable, the `if`-statement will be much more expensive.

Let's measure the performance of this loop with different conditions:

```

for (int i = 0; i < max; i++)
    if (condition)
        sum++;

```

Here are the timings of the loop with different true-false patterns:

Condition	Pattern	Time (ms)
(i & 0x80000000) == 0	T repeated	322
(i & 0xffffffff) == 0	F repeated	276
(i & 1) == 0	TF alternating	760
(i & 3) == 0	TFFFTFFF...	513
(i & 2) == 0	TTFFTTFF...	1675
(i & 4) == 0	TTTTFFFFTTTTFFFF...	1275
(i & 8) == 0	8T 8F 8T 8F ...	752
(i & 16) == 0	16T 16F 16T 16F ...	490

A “**bad**” true-false pattern can make an `if`-statement up to six times slower than a “**good**” pattern! Of course, which pattern is good and which is bad depends on the exact instructions generated by the compiler and on the specific processor.

So there is no doubt about the impact of branch prediction on performance!

Share Improve this answer

edited Feb 27, 2019 at 10:58

answered Feb 15, 2013 at 7:24

Follow



Neuron

5,763 ● 5 ● 43 ● 62



Saqlain

17.9k ● 4 ● 30 ● 33

-
- 53 @MooingDuck 'Cause it won't make a difference - that value can be anything, but it still will be in the bounds of these thresholds. So why show a random value when you already know the limits? Although I agree that you could show one for the sake of completeness, and 'just for the heck of it'. – [cst1992](#) Mar 28, 2016 at 12:58
-
- 54 @cst1992: Right now his slowest timing is TTFFTTFFTTFF, which seems, to my human eye, quite predictable. Random is inherently unpredictable, so it's entirely possible it would be slower still, and thus outside the limits shown here. OTOH, it could be that TTFFTTFF perfectly hits the pathological case. Can't tell, since he didn't show the timings for random. – [Mooing Duck](#) Mar 28, 2016 at 18:27 ✎
-
- 48 @MooingDuck To a human eye, "TTFFTTFFTTFF" is a predictable sequence, but what we are talking about here is the behavior of the branch predictor built into a CPU. The branch predictor is not AI-level pattern recognition; it's very simple. When you just alternate branches it doesn't predict well. In most code, branches go the same way almost all the time; consider a loop that executes a thousand times. The branch at the end of the loop goes back to the start of the loop 999 times, and then the thousandth time does something different. A very simple branch predictor works well, usually. – [steveha](#) Jul 20, 2016 at 21:07
-
- 45 @steveha: I think you're making assumptions about how the CPU branch predictor works, and I disagree with that methodology. I don't know how advanced that branch predictor is, but I seem to think it's far more advanced than you do. You're probably right, but measurements would definitely be good. – [Mooing Duck](#) Jul 20, 2016 at 21:10
-
- 28 @steveha: The Two-level adaptive predictor could lock onto the TTFFTTFF pattern with no issue whatsoever. "Variants of this prediction method are used in most modern

microprocessors". Local branch prediction and Global branch prediction are based on a two level adaptive predictor, they can as well. "Global branch prediction is used in AMD processors, and in Intel Pentium M, Core, Core 2, and Silvermont-based Atom processors" Also add Agree predictor, Hybrid predictor, Prediction of indirect jumps, to that list. Loop predictor wont lock on, but hits 75%. That leaves only 2 that can't lock on – [Mooing Duck](#) Jul 26, 2016 at 4:33



One way to avoid branch prediction errors is to build a lookup table, and index it using the data. Stefan de Bruijn discussed that in his answer.

1378



But in this case, we know values are in the range [0, 255] and we only care about values ≥ 128 . That means we can easily extract a single bit that will tell us whether we want a value or not: by shifting the data to the right 7 bits, we are left with a 0 bit or a 1 bit, and we only want to add the value when we have a 1 bit. Let's call this bit the "decision bit".



By using the 0/1 value of the decision bit as an index into an array, we can make code that will be equally fast whether the data is sorted or not sorted. Our code will always add a value, but when the decision bit is 0, we will add the value somewhere we don't care about. Here's the code:

```
// Test
clock_t start = clock();
long long a[] = {0, 0};
long long sum;

for (unsigned i = 0; i < 100000; ++i)
{
    // Primary loop
    for (unsigned c = 0; c < arraySize; ++c)
    {
        int j = (data[c] >> 7);
        a[j] += data[c];
    }
}

double elapsedTime = static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
sum = a[1];
```

This code wastes half of the adds but never has a branch prediction failure. It's tremendously faster on random data than the version with an actual if statement.

But in my testing, an explicit lookup table was slightly faster than this, probably because indexing into a lookup table was slightly faster than bit shifting. This shows how my code sets up and uses the lookup table (unimaginatively called `lut` for "LookUp Table" in the code). Here's the C++ code:

```
// Declare and then fill in the lookup table
int lut[256];
for (unsigned c = 0; c < 256; ++c)
    lut[c] = (c >= 128) ? c : 0;
```

```
// Use the lookup table after it is built
for (unsigned i = 0; i < 100000; ++i)
{
    // Primary loop
    for (unsigned c = 0; c < arraySize; ++c)
    {
        sum += lut[data[c]];
    }
}
```

In this case, the lookup table was only 256 bytes, so it fits nicely in a cache and all was fast. This technique wouldn't work well if the data was 24-bit values and we only wanted half of them... the lookup table would be far too big to be practical. On the other hand, we can combine the two techniques shown above: first shift the bits over, then index a lookup table. For a 24-bit value that we only want the top half value, we could potentially shift the data right by 12 bits, and be left with a 12-bit value for a table index. A 12-bit table index implies a table of 4096 values, which might be practical.

The technique of indexing into an array, instead of using an `if` statement, can be used for deciding which pointer to use. I saw a library that implemented binary trees, and instead of having two named pointers (`pLeft` and `pRight` or whatever) had a length-2 array of pointers and used the "decision bit" technique to decide which one to follow. For example, instead of:

```
if (x < node->value)
    node = node->pLeft;
else
    node = node->pRight;
```

this library would do something like:

```
i = (x < node->value);
node = node->link[i];
```

Here's a link to this code: [Red Black Trees](#), *Eternally Confuzzled*

Share Improve this answer

Follow

edited Jan 27, 2021 at 10:03



[jakubde](#)

31 ● 1 ● 2 ● 4

answered Jul 22, 2013 at 8:29



[steveha](#)

76.6k ● 21 ● 93 ● 118

- 54 Right, you can also just use the bit directly and multiply (`data[c]>>7` - which is discussed somewhere here as well); I intentionally left this solution out, but of course you are correct. Just a small note: The rule of thumb for lookup tables is that if it fits in 4KB (because of caching), it'll work - preferably make the table as small as possible. For managed languages I'd push that to 64KB, for low-level languages like C++ and C, I'd probably reconsider (that's just my experience). Since `sizeof(int) = 4`, I'd try to stick to max 10 bits. – [atlste](#) Jul 29, 2013 at 12:05

- 41 I think indexing with the 0/1 value will probably be faster than an integer multiply, but I guess if performance is really critical you should profile it. I agree that small lookup tables are essential to avoid cache pressure, but clearly if you have a bigger cache you can get away with a bigger lookup table, so 4KB is more a rule of thumb than a hard rule. I think you meant `sizeof(int) == 4` ? That would be true for 32-bit. My two-year-old cell phone has a 32KB L1 cache, so even a 4K lookup table might work, especially if the lookup values were a byte instead of an int. – [steveha](#) Jul 29, 2013 at 22:02
-
- 33 Possibly I'm missing something but in your `j` equals 0 or 1 method why don't you just multiply your value by `j` before adding it rather than using the array indexing (possibly should be multiplied by `1-j` rather than `j`) – [Richard Tingle](#) Mar 4, 2014 at 15:38
-
- 21 @steveha Multiplication should be faster, I tried looking it up in the Intel books, but couldn't find it... either way, benchmarking also gives me that result here. – [atlaste](#) Mar 18, 2014 at 8:45
-
- 25 @steveha P.S.: another possible answer would be `int c = data[j]; sum += c & -(c >> 7);` which requires no multiplications at all. – [atlaste](#) Mar 18, 2014 at 8:52
-



In the sorted case, you can do better than relying on successful branch prediction or any branchless comparison trick: completely remove the branch.

1236



Indeed, the array is partitioned in a contiguous zone with `data < 128` and another with `data >= 128`. So you should find the partition point with a [dichotomic search](#) (using `Lg(arraySize) = 15` comparisons), then do a straight accumulation from that point.



Something like (unchecked)



```
int i= 0, j, k= arraySize;
while (i < k)
{
    j= (i + k) >> 1;
    if (data[j] >= 128)
        k= j;
    else
        i= j;
}
sum= 0;
for (; i < arraySize; i++)
    sum+= data[i];
```

or, slightly more obfuscated

```
int i, k, j= (i + k) >> 1;
for (i= 0, k= arraySize; i < k; (data[j] >= 128 ? k : i)= j)
    j= (i + k) >> 1;
for (sum= 0; i < arraySize; i++)
    sum+= data[i];
```

A yet faster approach, that gives an **approximate** solution for both sorted or unsorted is: `sum= 3137536;` (assuming a truly uniform distribution, 16384 samples with expected

value 191.5) :-)

Share Improve this answer

edited May 11, 2019 at 11:31

answered Jul 24, 2013 at 7:57

Follow



Konard

2,964 ● 35 ● 26



user1196549

42 `sum= 3137536` - clever. That's kinda obviously not the point of the question. The question is clearly about explaining surprising performance characteristics. I'm inclined to say that the addition of doing `std::partition` instead of `std::sort` is valuable. Though the actual question extends to more than just the synthetic benchmark given. – [sehe](#) Jul 24, 2013 at 16:31

24 @DeadMG: this is indeed not the standard dichotomic search for a given key, but a search for the partitioning index; it requires a single compare per iteration. But don't rely on this code, I have not checked it. If you are interested in a guaranteed correct implementation, let me know. – user1196549 Jul 24, 2013 at 20:37

Binary search for the cut-point is unnecessary, just start summing from the end you want to keep, stopping when you get to a value you don't want. (Unless HW prefetching works *much* worse when looping downwards instead of upwards). Of course, actually taking time to sort is not a useful part of an algorithm that starts with unsorted data. A histogram might be, if you want to answer multiple queries for different cut-points, other than 128, from the same data. (Since the small value-range means there are many duplicates in this smallish array.) – [Peter Cordes](#) Apr 21, 2022 at 10:32



The above behavior is happening because of Branch prediction.

1039



To understand branch prediction one must first understand an **Instruction Pipeline**.



The the steps of running an instruction can be overlapped with the sequence of steps of running the previous and next instruction, so that different steps can be executed concurrently in parallel. This technique is known as instruction pipelining and is used to increase throughput in modern processors. To understand this better please see this [example on Wikipedia](#).

Generally, modern processors have quite long (and wide) pipelines, so many instruction can be in flight. See [Modern Microprocessors A 90-Minute Guide!](#) which starts by introducing basic in-order pipelining and goes from there.

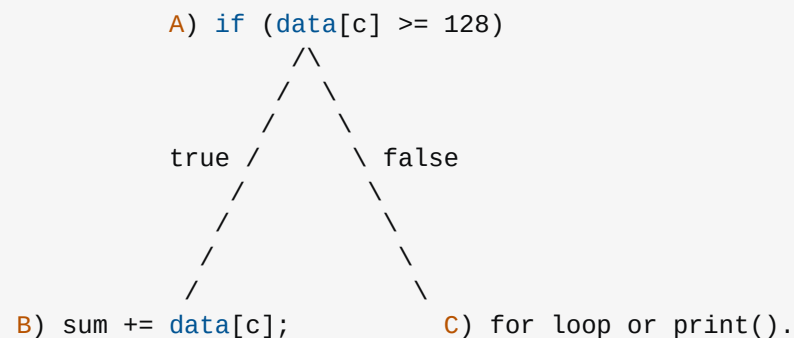
But for ease **let's consider a simple in-order pipeline with these 4 steps only**. (Like a [classic 5-stage RISC](#), but omitting a separate MEM stage.)

1. IF -- Fetch the instruction from memory
2. ID -- Decode the instruction
3. EX -- Execute the instruction
4. WB -- Write back to CPU register

4-stage pipeline in general for 2 instructions.

Instr. No.	Pipeline Stage				
1	IF	ID	EX	WB	
2		IF	ID	EX	WB
clock cycle	1	2	3	4	5

Moving back to the above question let's consider the following instructions:



Without branch prediction, the following would occur:

To execute instruction B or instruction C the processor will have to wait (*stall*) till the instruction A leaves the EX stage in the pipeline, as the decision to go to instruction B or instruction C depends on the result of instruction A. (i.e. where to fetch from next.) So the pipeline will look like this:

Without prediction: when `if` condition is true:

Instr. No.	Pipeline Stage						
1	IF	ID	EX	WB			
2				IF	ID	EX	WB
clock cycle	1	2	3	4	5	6	7

Without prediction: When `if` condition is false:

Instr. No.	Pipeline Stage						
1	IF	ID	EX	WB			
3				IF	ID	EX	WB
clock cycle	1	2	3	4	5	6	7

As a result of waiting for the result of instruction A, the total CPU cycles spent in the above case (without branch prediction; for both true and false) is 7.

So what is branch prediction?

Branch predictor will try to guess which way a branch (an if-then-else structure) will go before this is known for sure. It will not wait for the instruction A to reach the EX stage of the pipeline, but it will guess the decision and go to that instruction (B or C in case of our example).

In case of a correct guess, the pipeline looks something like this:

Instr. No.	Pipeline Stage				
1	IF	ID	EX	WB	
2		IF	ID	EX	WB
clock cycle	1	2	3	4	5

If it is later detected that the guess was wrong then the partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay. The time that is wasted in case of a branch misprediction is equal to the number of stages in the pipeline from the fetch stage to the execute stage. Modern microprocessors tend to have quite long pipelines so that the misprediction delay is between 10 and 20 clock cycles. The longer the pipeline the greater the need for a good [branch predictor](#).

In the OP's code, the first time when the conditional, the branch predictor does not have any information to base up prediction, so the first time it will randomly choose the next instruction. (Or fall back to *static* prediction, typically forward not-taken, backward taken). Later in the for loop, it can base the prediction on the history. For an array sorted in ascending order, there are three possibilities:

1. All the elements are less than 128
2. All the elements are greater than 128
3. Some starting new elements are less than 128 and later it become greater than 128

Let us assume that the predictor will always assume the true branch on the first run.

So in the first case, it will always take the true branch since historically all its predictions are correct. In the 2nd case, initially it will predict wrong, but after a few iterations, it will predict correctly. In the 3rd case, it will initially predict correctly till the elements are less than 128. After which it will fail for some time and the correct itself when it sees branch prediction failure in history.

In all these cases the failure will be too less in number and as a result, only a few times it will need to discard the partially executed instructions and start over with the correct branch, resulting in fewer CPU cycles.

But in case of a random unsorted array, the prediction will need to discard the partially executed instructions and start over with the correct branch most of the time and result in more CPU cycles compared to the sorted array.

Further reading:

- [Modern Microprocessors A 90-Minute Guide!](#)

- [Dan Luu's article on branch prediction](#) (which covers older branch predictors, not modern IT-TAGE or Perceptron)
- https://en.wikipedia.org/wiki/Branch_predictor
- [Branch Prediction and the Performance of Interpreters - Don't Trust Folklore](#) - 2015 paper showing how well Intel's Haswell does at predicting the indirect branch of a Python interpreter's main loop (historically problematic due to a non-simple pattern), vs. earlier CPUs which didn't use IT-TAGE. (They don't help with this fully random case, though. Still 50% mispredict rate for the if inside the loop on a Skylake CPU when the source is compiled to branch asm.)
- [Static branch prediction on newer Intel processors](#) - what CPUs actually do when running a branch instruction that doesn't have a dynamic prediction available. Historically, forward not-taken (like an `if` or `break`), backward taken (like a loop) has been used because it's better than nothing. Laying out code so the fast path / common case minimizes taken branches is good for I-cache density as well as static prediction, so compilers already do that. (That's the [real effect](#) of `likely` / `unlikely` hints in C source, not actually hinting the hardware branch prediction in most CPU, except maybe via static prediction.)

Share Improve this answer

Follow

edited Apr 22, 2022 at 1:31



Peter Cordes

361k ● 49 ● 699 ● 958

answered Jul 3, 2015 at 15:35



Harsh Sharma

11.2k ● 2 ● 19 ● 29

-
- 9 how are two instructions executed together? is this done with separate cpu cores or is pipeline instruction is integrated in single cpu core? – [M.kazem Akhgary](#) Oct 11, 2017 at 14:49
-
- 7 @M.kazemAkhgary It's all inside one logical core. If you're interested, this is nicely described for example in [Intel Software Developer Manual](#) – [Sergey.quixoticaxis.Ivanov](#) Nov 3, 2017 at 7:45
-



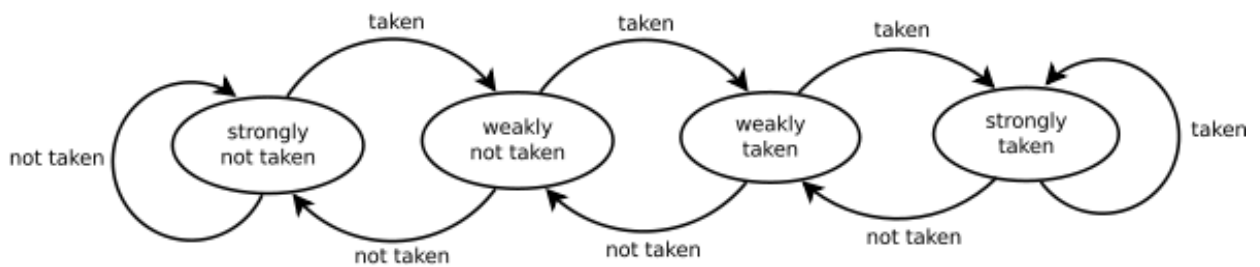
912



An official answer would be from

1. [Intel - Avoiding the Cost of Branch Misprediction](#)
2. [Intel - Branch and Loop Reorganization to Prevent Mispredicts](#)
3. [Scientific papers - branch prediction computer architecture](#)
4. Books: J.L. Hennessy, D.A. Patterson: Computer architecture: a quantitative approach
5. Articles in scientific publications: T.Y. Yeh, Y.N. Patt made a lot of these on branch predictions.

You can also see from this lovely [diagram](#) why the branch predictor gets confused.



Each element in the original code is a random value

```
data[c] = std::rand() % 256;
```

so the predictor will change sides as the `std::rand()` blow.

On the other hand, once it's sorted, the predictor will first move into a state of strongly not taken and when the values change to the high value the predictor will in three runs through change all the way from strongly not taken to strongly taken.

Share Improve this answer
Follow

edited Jan 31, 2017 at 11:39



greatwolf

20.8k ● 13 ● 73 ● 105

answered Oct 11, 2015 at 21:05



Surt

16.1k ● 3 ● 26 ● 40



874



+100



In the same line (I think this was not highlighted by any answer) it's good to mention that sometimes (specially in software where the performance matters—like in the Linux kernel) you can find some if statements like the following:

```
if (likely( everything_is_ok ))
{
    /* Do something */
}
```

or similarly:

```
if (unlikely(very_improbable_condition))
{
    /* Do something */
}
```

Both `likely()` and `unlikely()` are in fact macros that are defined by using something like the GCC's `__builtin_expect` to help the compiler insert prediction code to favour the condition taking into account the information provided by the user. GCC supports other builtins that could change the behavior of the running program or emit low level instructions like clearing the cache, etc. See [this documentation](#) that goes through the available GCC's builtins.

Normally this kind of optimizations are mainly found in hard-real time applications or embedded systems where execution time matters and it's critical. For example, if you are checking for some error condition that only happens 1/10000000 times, then why not inform the compiler about this? This way, by default, the branch prediction would assume that the condition is false.

Share Improve this answer

edited Oct 28, 2016 at 10:28

answered Sep 23, 2015 at 14:57

Follow



Stacked

7,308 ● 7 ● 62 ● 76



rkachach

17.3k ● 8 ● 47 ● 68

I agree with that. – Karol Jun 30 at 1:13



847



+25



Frequently used Boolean operations in C++ produce many branches in the compiled program. If these branches are inside loops and are hard to predict they can slow down execution significantly. Boolean variables are stored as 8-bit integers with the value `0` for `false` and `1` for `true`.

Boolean variables are overdetermined in the sense that all operators that have Boolean variables as input check if the inputs have any other value than `0` or `1`, but operators that have Booleans as output can produce no other value than `0` or `1`. This makes operations with Boolean variables as input less efficient than necessary. Consider example:

```
bool a, b, c, d;
c = a && b;
d = a || b;
```

This is typically implemented by the compiler in the following way:

```
bool a, b, c, d;
if (a != 0) {
    if (b != 0) {
        c = 1;
    }
    else {
        goto CFALSE;
    }
}
else {
    CFALSE:
    c = 0;
}
if (a == 0) {
    if (b == 0) {
        d = 0;
    }
    else {
        goto DTRUE;
    }
}
```

```

}
else {
    DTRUE:
    d = 1;
}

```

This code is far from optimal. The branches may take a long time in case of mispredictions. The Boolean operations can be made much more efficient if it is known with certainty that the operands have no other values than `0` and `1`. The reason why the compiler does not make such an assumption is that the variables might have other values if they are uninitialized or come from unknown sources. The above code can be optimized if `a` and `b` has been initialized to valid values or if they come from operators that produce Boolean output. The optimized code looks like this:

```

char a = 0, b = 1, c, d;
c = a & b;
d = a | b;

```

`char` is used instead of `bool` in order to make it possible to use the bitwise operators (`&` and `|`) instead of the Boolean operators (`&&` and `||`). The bitwise operators are single instructions that take only one clock cycle. The OR operator (`|`) works even if `a` and `b` have other values than `0` or `1`. The AND operator (`&`) and the EXCLUSIVE OR operator (`^`) may give inconsistent results if the operands have other values than `0` and `1`.

`~` can not be used for NOT. Instead, you can make a Boolean NOT on a variable which is known to be `0` or `1` by XOR'ing it with `1`:

```

bool a, b;
b = !a;

```

can be optimized to:

```

char a = 0, b;
b = a ^ 1;

```

`a && b` cannot be replaced with `a & b` if `b` is an expression that should not be evaluated if `a` is `false` (`&&` will not evaluate `b`, `&` will). Likewise, `a || b` can not be replaced with `a | b` if `b` is an expression that should not be evaluated if `a` is `true`.

Using bitwise operators is more advantageous if the operands are variables than if the operands are comparisons:

```

bool a; double x, y, z;
a = x > y && z < 5.0;

```

is optimal in most cases (unless you expect the `&&` expression to generate many branch mispredictions).

Share Improve this answer

Follow

edited May 30, 2019 at 16:34



Sujal Patel

2,532 ● 2 ● 20 ● 40

answered Oct 10, 2015 at 0:30



Maciej

9,605 ● 2 ● 17 ● 18



That's for sure!...

509



Branch prediction makes the logic run slower, because of the switching which happens in your code! It's like you are going a straight street or a street with a lot of turnings, for sure the straight one is going to be done quicker!...



If the array is sorted, your condition is false at the first step: `data[c] >= 128`, then becomes a true value for the whole way to the end of the street. That's how you get to the end of the logic faster. On the other hand, using an unsorted array, you need a lot of turning and processing which make your code run slower for sure...

Look at the image I created for you below. Which street is going to be finished faster?



Which street gonna be done faster?

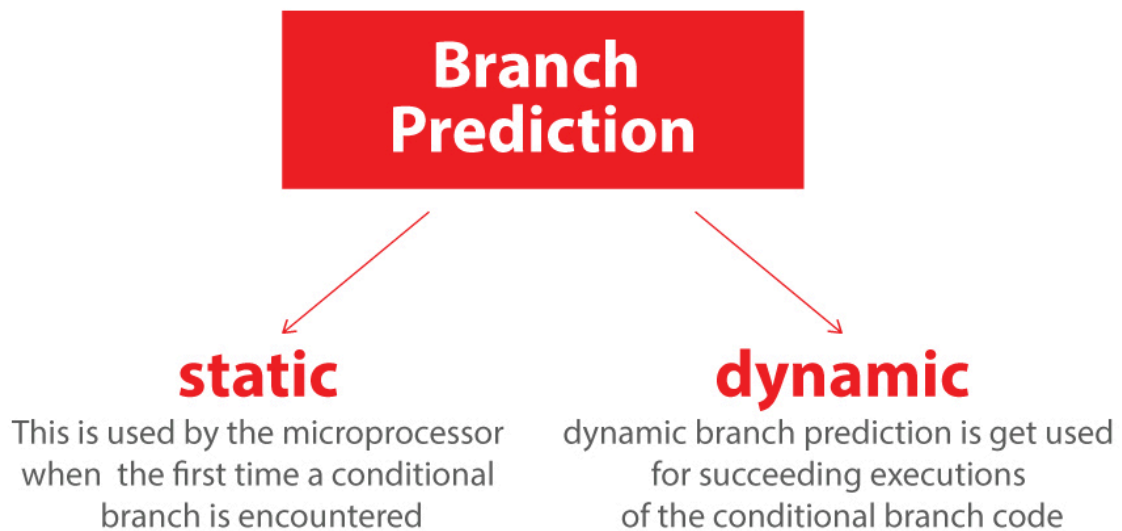


So programmatically, **branch prediction** causes the process to be slower...

Also at the end, it's good to know we have two kinds of branch predictions that each is going to affect your code differently:

1. Static

2. Dynamic



See also [this document from Intel](#), which says:

Static branch prediction is used by the microprocessor the first time a conditional branch is encountered, and dynamic branch prediction is used for succeeding executions of the conditional branch code.

In order to effectively write your code to take advantage of these rules, when writing **if-else** or **switch** statements, check the most common cases first and work progressively down to the least common. Loops do not necessarily require any special ordering of code for static branch prediction, as only the condition of the loop iterator is normally used.

Share Improve this answer
Follow

edited May 10, 2023 at 17:27



miken32

42.7k ● 16 ● 121 ● 171

answered Jun 18, 2017 at 11:40



Alireza

105k ● 27 ● 277 ● 173



This question has already been answered excellently many times over. Still I'd like to draw the group's attention to yet another interesting analysis.

459



Recently this example (modified very slightly) was also used as a way to demonstrate how a piece of code can be profiled within the program itself on Windows. Along the way, the author also shows how to use the results to determine where the code is spending most of its time in both the sorted & unsorted case. Finally the piece also shows how to use a little known feature of the HAL (Hardware Abstraction Layer) to determine just how much branch misprediction is happening in the unsorted case.



The link is here: [A Demonstration of Self-Profiling](#)



- 8 That is a very interesting article (in fact, I have just read all of it), but how does it answer the question? – [Peter Mortensen](#) Mar 16, 2018 at 12:47
- 7 @PeterMortensen I am a bit flummoxed by your question. For example here is one relevant line from that piece: `when the input is unsorted, all the rest of the loop takes substantial time. But with sorted input, the processor is somehow able to spend not just less time in the body of the loop, meaning the buckets at offsets 0x18 and 0x1C, but vanishingly little time on the mechanism of looping.` Author is trying to discuss profiling in the context of code posted here and in the process trying to explain why the sorted case is so much more faster. – [ForeverLearning](#) Mar 16, 2018 at 15:37



418



As what has already been mentioned by others, what behind the mystery is [Branch Predictor](#).

I'm not trying to add something but explaining the concept in another way. There is a concise introduction on the wiki which contains text and diagram. I do like the explanation below which uses a diagram to elaborate the Branch Predictor intuitively.

In computer architecture, a branch predictor is a digital circuit that tries to guess which way a branch (e.g. an if-then-else structure) will go before this is known for sure. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving high effective performance in many modern pipelined microprocessor architectures such as x86.

Two-way branching is usually implemented with a conditional jump instruction. A conditional jump can either be "not taken" and continue execution with the first branch of code which follows immediately after the conditional jump, or it can be "taken" and jump to a different place in program memory where the second branch of code is stored. It is not known for certain whether a conditional jump will be taken or not taken until the condition has been calculated and the conditional jump has passed the execution stage in the instruction pipeline (see fig. 1).

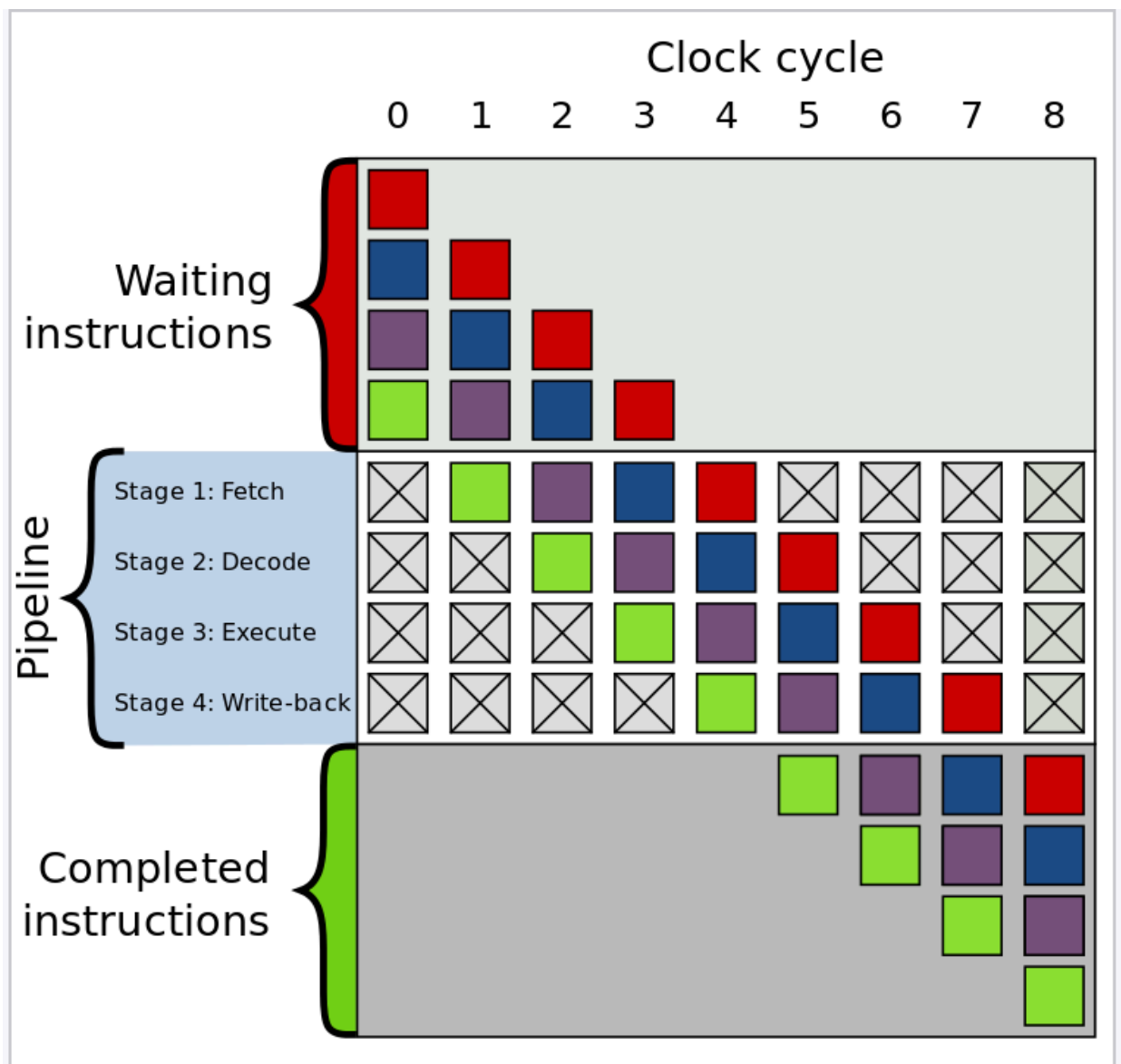


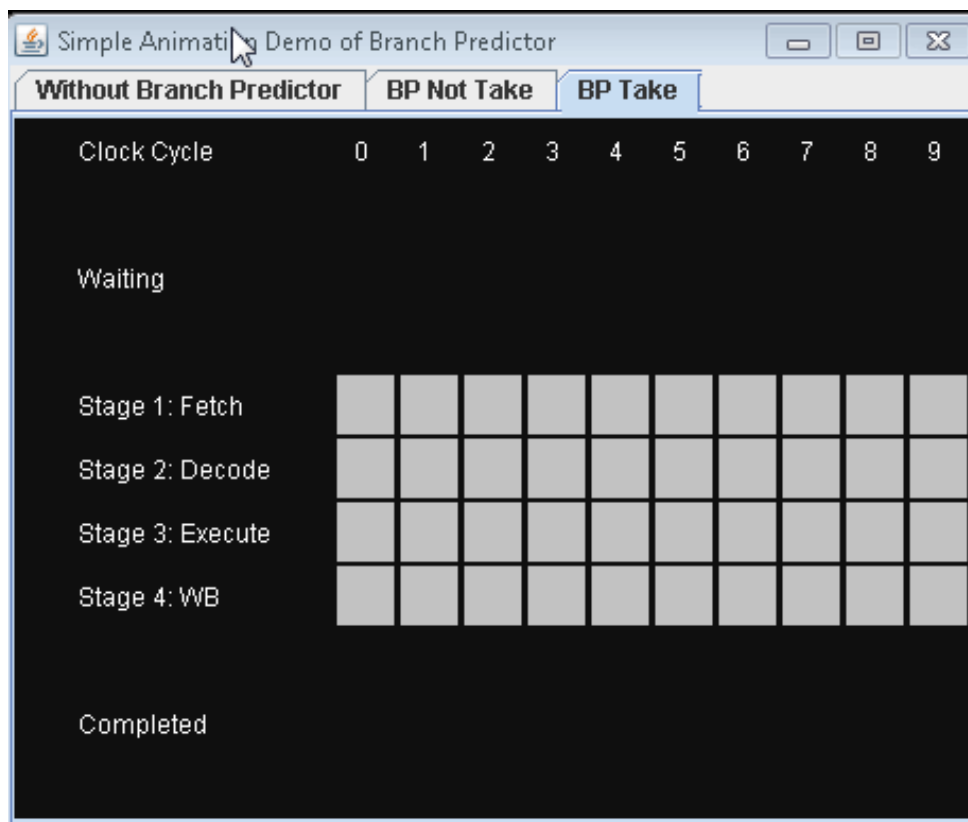
Figure 1: Example of 4-stage pipeline. The colored boxes represent instructions independent of each other

Based on the described scenario, I have written an animation demo to show how instructions are executed in a pipeline in different situations.

1. Without the Branch Predictor.

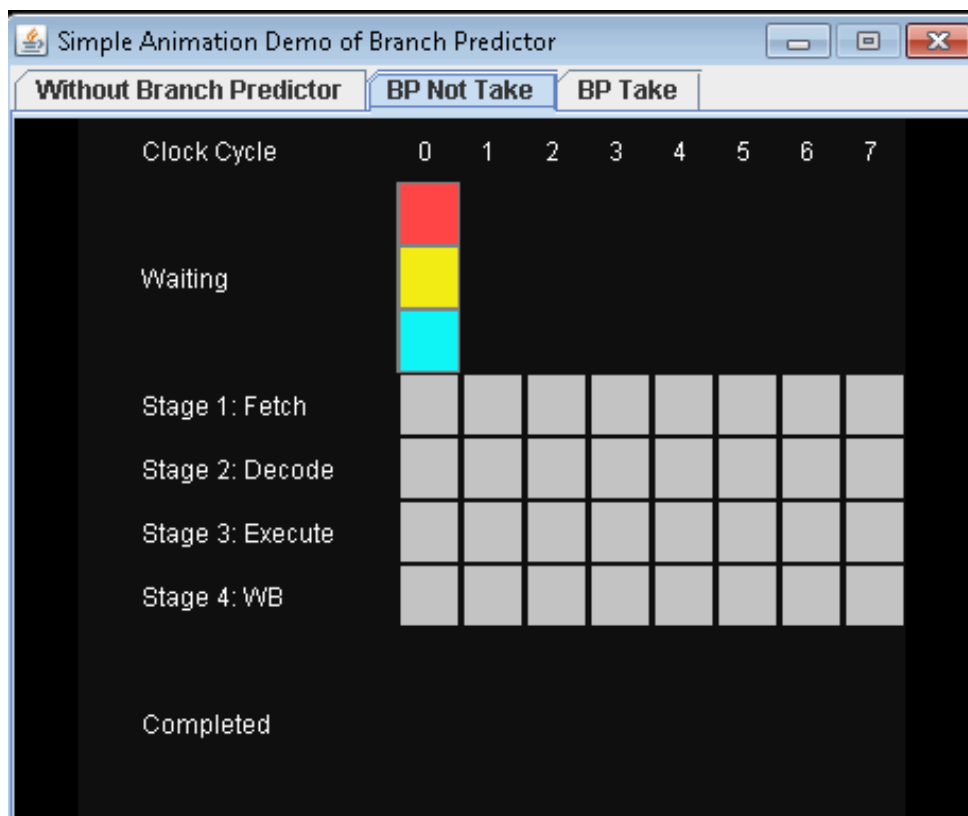
Without branch prediction, the processor would have to wait until the conditional jump instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline.

The example contains three instructions and the first one is a conditional jump instruction. The latter two instructions can go into the pipeline until the conditional jump instruction is executed.



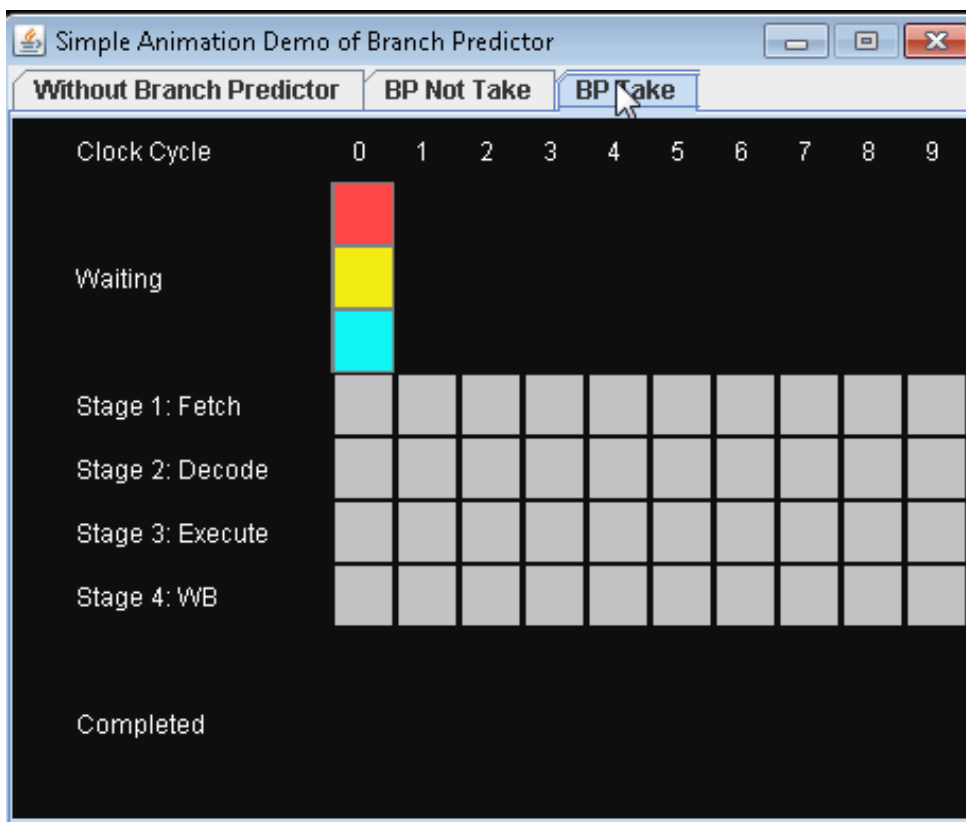
It will take 9 clock cycles for 3 instructions to be completed.

2. Use Branch Predictor and don't take a conditional jump. Let's assume that the predict is **not** taking the conditional jump.



It will take 7 clock cycles for 3 instructions to be completed.

3. Use Branch Predictor and take a conditional jump. Let's assume that the predict is **not** taking the conditional jump.



It will take 9 clock cycles for 3 instructions to be completed.

The time that is wasted in case of a branch misprediction is equal to the number of stages in the pipeline from the fetch stage to the execute stage. Modern microprocessors tend to have quite long pipelines so that the misprediction delay is between 10 and 20 clock cycles. As a result, making a pipeline longer increases the need for a more advanced branch predictor.

As you can see, it seems we don't have a reason not to use Branch Predictor.

It's quite a simple demo that clarifies the very basic part of Branch Predictor. If those gifs are annoying, please feel free to remove them from the answer and visitors can also get the live demo source code from [BranchPredictorDemo](#)

Share Improve this answer

edited Feb 10, 2020 at 2:05

answered Nov 6, 2017 at 16:15

Follow



Eugene

11k ● 6 ● 55 ● 69

11 Almost as good as the Intel marketing animations, and they were obsessed not just with branch prediction but out of order execution, both strategies being "speculative". Reading ahead in memory and storage (sequential pre-fetch to buffer) is also speculative. It all adds up.
– [mckenzm](#) Jul 29, 2019 at 5:14

9 @mckenzm: out-of-order speculative exec makes branch prediction even more valuable; as well as hiding fetch/decode bubbles, branch prediction + speculative exec removes control dependencies from critical path latency. Code inside or after an `if()` block can execute *before* the branch condition is known. Or for a search loop like `strlen` or `memchr`, iterations can overlap. If you had to wait for the match-or-not result to be known before running

any of the next iteration, you'd bottleneck on cache load + ALU latency instead of throughput.
– [Peter Cordes](#) Feb 10, 2020 at 4:03

7 Did you make the example app in JavaFX? – [Justin Meskan](#) Jul 1, 2020 at 0:19

5 @HannaMcquaig No, it's made by Swing. The code is available at github.com/Eugene-Mark/branch-predictor-demo. – [Eugene](#) Jul 1, 2020 at 1:14

Branch-prediction gain!

320

It is important to understand that branch misprediction doesn't slow down programs. The cost of a missed prediction is just as if branch prediction didn't exist and you waited for the evaluation of the expression to decide what code to run (further explanation in the next paragraph).

```
if (expression)
{
    // Run 1
} else {
    // Run 2
}
```

Whenever there's an `if-else` \ `switch` statement, the expression has to be evaluated to determine which block should be executed. In the assembly code generated by the compiler, conditional [branch](#) instructions are inserted.

A branch instruction can cause a computer to begin executing a different instruction sequence and thus deviate from its default behavior of executing instructions in order (i.e. if the expression is false, the program skips the code of the `if` block) depending on some condition, which is the expression evaluation in our case.

That being said, the compiler tries to predict the outcome prior to it being actually evaluated. It will fetch instructions from the `if` block, and if the expression turns out to be true, then wonderful! We gained the time it took to evaluate it and made progress in the code; if not then we are running the wrong code, the pipeline is flushed, and the correct block is run.

Visualization:

Let's say you need to pick route 1 or route 2. Waiting for your partner to check the map, you have stopped at `##` and waited, or you could just pick route1 and if you were lucky (route 1 is the correct route), then great you didn't have to wait for your partner to check the map (you saved the time it would have taken him to check the map), otherwise you will just turn back.

While flushing pipelines is super fast, nowadays taking this gamble is worth it. Predicting sorted data or a data that changes slowly is always easier and better than predicting fast changes.



Share Improve this answer

edited Jun 20, 2020 at 9:12

answered Aug 4, 2017 at 10:07

Follow



Community Bot

1 • 1



Tony Tannous

14.8k • 11 • 56 • 90

- 3 While flushing pipelines is super fast Not really. It's fast compared to a cache miss all the way to DRAM, but on a modern high-performance x86 (like Intel Sandybridge-family) it's about a dozen cycles. Although fast recovery does allow it to avoid waiting for all older independent instructions to reach retirement before starting recovery, you still lose a lot of front-end cycles on a mispredict. [What exactly happens when a skylake CPU mispredicts a branch?](#). (And each cycle can be about 4 instructions of work.) Bad for high-throughput code. – Peter Cordes Feb 10, 2020 at 4:07



279



On ARM, there is no branch needed, because every instruction has a 4-bit condition field, which tests (at zero cost) any of [16 different different conditions](#) that may arise in the Processor Status Register, and if the condition on an instruction is false, the instruction is skipped. This eliminates the need for short branches, and there would be no branch prediction hit for this algorithm. **Therefore, the sorted version of this algorithm would run slower than the unsorted version on ARM, because of the extra overhead of sorting.**

The inner loop for this algorithm would look something like the following in ARM assembly language:

```
MOV R0, #0    // R0 = sum = 0
MOV R1, #0    // R1 = c = 0
ADR R2, data  // R2 = addr of data array (put this instruction outside outer
loop)
.inner_loop  // Inner loop branch label
    LDRB R3, [R2, R1]    // R3 = data[c]
    CMP R3, #128         // compare R3 to 128
    ADDGE R0, R0, R3     // if R3 >= 128, then sum += data[c] -- no branch
needed!
    ADD R1, R1, #1       // c++
    CMP R1, #arraySize  // compare c to arraySize
    BLT inner_loop      // Branch to inner_loop if c < arraySize
```

But this is actually part of a bigger picture:

`CMP` opcodes always update the status bits in the Processor Status Register (PSR), because that is their purpose, but most other instructions do not touch the PSR unless you add an optional `s` suffix to the instruction, specifying that the PSR should be updated based on the result of the instruction. **Just like the 4-bit condition suffix, being able to execute instructions without affecting the PSR is a mechanism that reduces the need for branches on ARM, and also facilitates out of order dispatch at the hardware level**, because after performing some operation X that updates the status bits, subsequently (or in parallel) you can do a bunch of other work that explicitly should not affect (or be affected by) the status bits, then you can test the state of the status bits set earlier by X.

The condition testing field and the optional "set status bit" field can be combined, for example:

- `ADD R1, R2, R3` performs `R1 = R2 + R3` without updating any status bits.
- `ADDGE R1, R2, R3` performs the same operation only if a previous instruction that affected the status bits resulted in a Greater than or Equal condition.
- `ADD S R1, R2, R3` performs the addition and then updates the `N`, `Z`, `C` and `V` flags in the Processor Status Register based on whether the result was Negative, Zero, Carried (for unsigned addition), or oVerflowed (for signed addition).
- `ADDSGE R1, R2, R3` performs the addition only if the `GE` test is true, and then subsequently updates the status bits based on the result of the addition.

Most processor architectures do not have this ability to specify whether or not the status bits should be updated for a given operation, which can necessitate writing additional code to save and later restore status bits, or may require additional branches, or may limit the processor's out of order execution efficiency: one of the side effects of most CPU instruction set architectures forcibly updating status bits after most instructions is that it is much harder to tease apart which instructions can be run in parallel without interfering with each other. Updating status bits has side effects, therefore has a linearizing effect on code. **ARM's ability to mix and match branch-free condition testing on any instruction with the option to either update or not update the status bits after any instruction is extremely powerful, for both assembly language programmers and compilers, and produces very efficient code.**

When you don't have to branch, you can avoid the time cost of flushing the pipeline for what would otherwise be short branches, and you can avoid the design complexity of many forms of speculative evaluation. The performance impact of the initial naive implementations of the mitigations for many recently discovered processor vulnerabilities (Spectre etc.) shows you just how much the performance of modern processors depends upon complex speculative evaluation logic. With a short pipeline and the dramatically reduced need for branching, ARM just doesn't need to rely on speculative evaluation as much as CISC processors. (Of course high-end ARM implementations do include speculative evaluation, but it's a smaller part of the performance story.)

If you have ever wondered why ARM has been so phenomenally successful, the brilliant effectiveness and interplay of these two mechanisms (combined with another mechanism that lets you "barrel shift" left or right one of the two arguments of any arithmetic operator or offset memory access operator at zero additional cost) are a big part of the story, because they are some of the greatest sources of the ARM architecture's efficiency. The brilliance of the original designers of the ARM ISA back in 1983, Steve Furber and Roger (now Sophie) Wilson, cannot be overstated.

Share Improve this answer

edited Oct 20, 2020 at 22:33

answered Dec 22, 2017 at 13:13

Follow



Luke Hutchison

9,140 ● 3 ● 53 ● 50

-
- 5 The other innovation in ARM is the addition of the S instruction suffix, also optional on (almost) all instructions, which if absent, prevents instructions from changing status bits (with the exception of the CMP instruction, whose job is to set status bits, so it doesn't need the S suffix). This allows you to avoid CMP instructions in many cases, as long as the comparison is with zero or similar (eg. SUBS R0, R0, #1 will set the Z (Zero) bit when R0 reaches zero). Conditionals and the S suffix incur zero overhead. It's quite a beautiful ISA. – [Luke Hutchison](#) May 15, 2018 at 17:06
-
- 5 Not adding the S suffix allows you to have several conditional instructions in a row without worrying that one of them might change the status bits, which might otherwise have the side effect of skipping the rest of the conditional instructions. – [Luke Hutchison](#) May 15, 2018 at 17:08
-
- 3 Note that the OP is *not* including the time to sort in their measurement. It's probably an overall loss to sort first before running a branch x86 loop, too, even though the non-sorted case makes the loop run a lot slower. But sorting a big array requires a *lot* of work. – [Peter Cordes](#) Feb 21, 2020 at 11:34
-
- 4 BTW, you could save an instruction in the loop by indexing relative to the end of the array. Before the loop, set up `R2 = data + arraySize`, then start with `R1 = -arraySize`. The bottom of the loop becomes `adds r1, r1, #1 / bnz inner_loop`. Compilers don't use this optimization for some reason :/ But anyway, predicated execution of the add is not fundamentally different in this case from what you can do with branchless code on other ISAs, like x86 `cmov`. Although it's not as nice: [gcc optimization flag -O3 makes code slower than -O2](#) – [Peter Cordes](#) Feb 21, 2020 at 11:36
-
- 4 (ARM predicated execution truly NOPs the instruction, so you can even use it on loads or stores that would fault, unlike x86 `cmov` with a memory source operand. Most ISAs, including AArch64, only have ALU select operations. So ARM predication can be powerful, and usable more efficiently than branchless code on most ISAs.) – [Peter Cordes](#) Feb 21, 2020 at 11:40
-



231



Besides the fact that the branch prediction may slow you down, a sorted array has another advantage:

You can have a stop condition instead of just checking the value, this way you only loop over the relevant data, and ignore the rest.

The branch prediction will miss only once.

```
// sort backwards (higher values first), may be in some other part of the code
std::sort(data, data + arraySize, std::greater<int>());

for (unsigned c = 0; c < arraySize; ++c) {
    if (data[c] < 128) {
        break;
    }
    sum += data[c];
}
```

Share Improve this answer

edited Mar 5, 2019 at 9:58

answered Nov 23, 2017 at 14:28

Follow



Yochai Timmer

49.1k ● 24 ● 154 ● 187

-
- 4 Right, but the setup cost of sorting the array is $O(N \log N)$, so breaking early doesn't help you if the only reason you are sorting the array is to be able to break early. If, however, you have other reasons to pre-sort the array, then yes, this is valuable. – [Luke Hutchison](#) Nov 6, 2018 at 12:28
-
- 2 Depends how many times you sort the data compared to how many times you loop on it. The sort in this example is just an example, it doesn't have to be just before the loop
– [Yochai Timmer](#) Feb 27, 2019 at 12:23 ✎
-
- 4 Yes, that's exactly the point I made in my first comment :-). You say "The branch prediction will miss only once." But you are not counting the $O(N \log N)$ branch prediction misses inside the sort algorithm, which is actually greater than the $O(N)$ branch prediction misses in the unsorted case. So you would need to use the entirety of the sorted data $O(\log N)$ times to break even (probably actually closer to $O(10 \log N)$, depending on the sort algorithm, e.g. for quicksort, due to cache misses -- mergesort is more cache-coherent, so you would need closer to $O(2 \log N)$ usages to break even.) – [Luke Hutchison](#) Feb 28, 2019 at 12:28
-
- 2 One significant optimization though would be to do only "half a quicksort", sorting only items less than the target pivot value of 127 (assuming everything less than *or equal to* the pivot is sorted after the pivot). Once you reach the pivot, sum the elements before the pivot. This would run in $O(N)$ startup time rather than $O(N \log N)$, although there will still be a lot of branch prediction misses, probably of the order of $O(5 N)$ based on the numbers I gave before, since it's half a quicksort. – [Luke Hutchison](#) Feb 28, 2019 at 12:34

Came here looking for this exact answer. Early abort is the main reason to sort. Limit the search space entirely. In my testing it was 2x faster again by being able to stop looking much earlier in the loop. – [Jason Short](#) Jul 5, 2022 at 18:23



Sorted arrays are processed faster than an unsorted array, due to a phenomena called branch prediction.

216



The branch predictor is a digital circuit (in computer architecture) trying to predict which way a branch will go, improving the flow in the instruction pipeline. The circuit/computer predicts the next step and executes it.

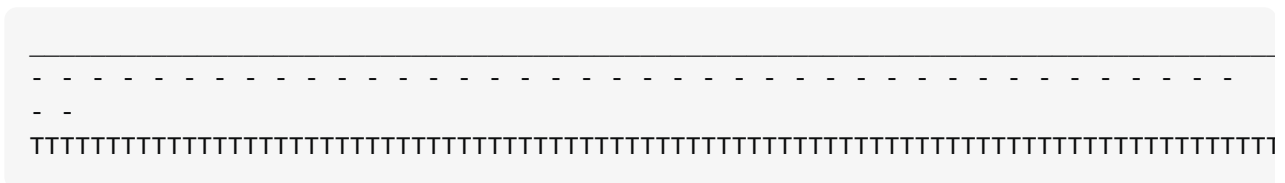


Making a wrong prediction leads to going back to the previous step, and executing with another prediction. Assuming the prediction is correct, the code will continue to the next step. A wrong prediction results in repeating the same step, until a correct prediction occurs.

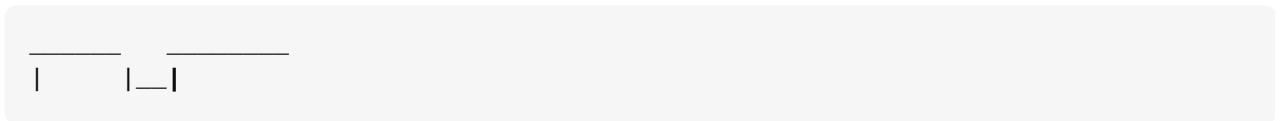
The answer to your question is very simple.

In an unsorted array, the computer makes multiple predictions, leading to an increased chance of errors. Whereas, in a sorted array, the computer makes fewer predictions, reducing the chance of errors. Making more predictions requires more time.

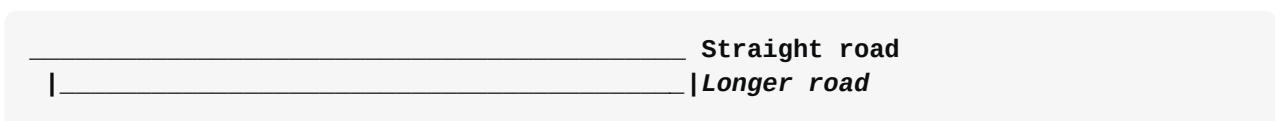
Sorted Array: Straight Road



Unsorted Array: Curved Road



Branch prediction: Guessing/predicting which road is straight and following it without checking



Although both the roads reach the same destination, the straight road is shorter, and the other is longer. If then you choose the other by mistake, there is no turning back, and so you will waste some extra time if you choose the longer road. This is similar to what happens in the computer, and I hope this helped you understand better.

Also I want to cite [@Simon_Weaver](#) from the comments:

It doesn't make fewer predictions - it makes fewer incorrect predictions. It still has to predict for each time through the loop...



193



I tried the same code with MATLAB 2011b with my MacBook Pro (Intel i7, 64 bit, 2.4 GHz) for the following MATLAB code:

```
% Processing time with Sorted data vs unsorted data
%=====
% Generate data
arraySize = 32768
sum = 0;
% Generate random integer data from range 0 to 255
data = randi(256, arraySize, 1);

%Sort the data
data1= sort(data); % data1= data  when no sorting done

%Start a stopwatch timer to measure the execution time
tic;

for i=1:100000

    for j=1:arraySize

        if data1(j)>=128
            sum=sum + data1(j);
        end
    end
end

toc;

ExeTimeWithSorting = toc - tic;
```

The results for the above MATLAB code are as follows:

```
a: Elapsed time (without sorting) = 3479.880861 seconds.
b: Elapsed time (with sorting ) = 2377.873098 seconds.
```

The results of the C code as in @GManNickG I get:

```
a: Elapsed time (without sorting) = 19.8761 sec.
b: Elapsed time (with sorting ) = 7.37778 sec.
```

Based on this, it looks MATLAB is almost *175 times* slower than the C implementation without sorting and *350 times* slower with sorting. In other words, the effect (of branch prediction) is *1.46x* for MATLAB implementation and *2.7x* for the C implementation.

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133



Shan

5,222 ● 13 ● 46 ● 59

-
- 10 Just for the sake of completeness, this is probably not how you'd implement that in Matlab. I bet it'd be much faster if done after vectorizing the problem. – [ysap](#) May 17, 2013 at 19:53
-
- 3 Matlab does automatic parallelization / vectorization in many situations but the issue here is to check the effect of branch prediction. Matlab is not immune in anyway! – [Shan](#) May 17, 2013 at 23:50 ✎
-
- 3 Does matlab use native numbers or a mat lab specific implementation (infinite amount of digits or so?) – [Thorbjørn Ravn Andersen](#) Aug 24, 2013 at 16:34
-



125



The assumption by other answers that one needs to sort the data is not correct.

The following code does not sort the entire array, but only 200-element segments of it, and thereby runs the fastest.

Sorting only k-element sections completes the pre-processing in linear time, $O(n)$, rather than the $O(n \cdot \log(n))$ time needed to sort the entire array.

```
#include <algorithm>
#include <ctime>
#include <iostream>

int main() {
    int data[32768]; const int l = sizeof data / sizeof data[0];

    for (unsigned c = 0; c < l; ++c)
        data[c] = std::rand() % 256;

    // sort 200-element segments, not the whole array
    for (unsigned c = 0; c + 200 <= l; c += 200)
        std::sort(&data[c], &data[c + 200]);

    clock_t start = clock();
    long long sum = 0;

    for (unsigned i = 0; i < 100000; ++i) {
        for (unsigned c = 0; c < sizeof data / sizeof(int); ++c) {
            if (data[c] >= 128)
                sum += data[c];
        }
    }

    std::cout << static_cast<double>(clock() - start) / CLOCKS_PER_SEC <<
    std::endl;
    std::cout << "sum = " << sum << std::endl;
}
```

This also "proves" that it has nothing to do with any algorithmic issue such as sort order, and it is indeed branch prediction.



- 9 I don't really see how this proves anything? The only thing you have shown is that "not doing all the work of sorting the whole array takes less time than sorting the whole array". Your claim that this "also runs fastest" is very architecture-dependent. See my answer about how this works on ARM. PS you could make your code faster on non-ARM architectures by putting the summation inside the 200-element block loop, sorting in reverse, and then using Yochai Timmer's suggestion of breaking once you get an out-of range value. That way each 200-element block summation can be terminated early. – [Luke Hutchison](#) Feb 28, 2019 at 12:18 ✎
- 2 If you just want to implement the algorithm efficiently over unsorted data, you would do that operation branchlessly (and with SIMD, e.g. with x86 `pcmpgtb` to find elements with their high bit set, then AND to zero smaller elements). Spending any time actually sorting chunks would be slower. A branchless version would have data-independent performance, also proving that the cost came from branch misprediction. Or just use performance counters to observe that directly, like Skylake `int_misc.clear_resteer_cycles` or `int_misc.recovery_cycles` to count front-end idle cycles from mispredicts – [Peter Cordes](#) Dec 13, 2019 at 14:44 ✎
- 3 Both comments above seem to ignore the general algorithmic issues and complexity, in favor of advocating specialized hardware with special machine instructions. I find the first one particularly petty in that it blithely dismisses the important general insights in this answer in blind favor of specialized machine instructions. – [user2297550](#) Apr 3, 2020 at 6:44
- 2 Also note that specialized hardware instructions do not help if the computation within the `if` is more complicated than a simple addition, which is quite likely in the general case. Therefore, this answer is unique in offering a general solution that is still $O(n)$ – [user2297550](#) Jun 19, 2021 at 17:50
- 2 Just for the record, since our previous comments have been nuked, I don't think there's anything to gain in *overall* performance by spending time (partially) sorting, unless you're artificially repeating the loop over the array like in this microbenchmark. Then yes, this piecewise sort gets close to the benefit of a full sort (e.g. 2.4s for this vs. 1.7s for a full sort on Skylake, vs. 10.9s for no sort before doing 100k passes. If you use `g++ -O3 -Wa, -mbranches-within-32B-boundaries` to make branchy asm in the first place instead of a normal build). – [Peter Cordes](#) Apr 22, 2022 at 1:45

[Bjarne Stroustrup's Answer](#) to this question:

That sounds like an interview question. Is it true? How would you know? It is a bad idea to answer questions about efficiency without first doing some measurements, so it is important to know how to measure.

So, I tried with a vector of a million integers and got:

Already sorted	32995 milliseconds
Shuffled	125944 milliseconds
Already sorted	18610 milliseconds
Shuffled	133304 milliseconds



115



Already sorted	17942 milliseconds
Shuffled	107858 milliseconds

I ran that a few times to be sure. Yes, the phenomenon is real. My key code was:

```
void run(vector<int>& v, const string& label)
{
    auto t0 = system_clock::now();
    sort(v.begin(), v.end());
    auto t1 = system_clock::now();
    cout << label
          << duration_cast<microseconds>(t1 - t0).count()
          << " milliseconds\n";
}

void tst()
{
    vector<int> v(1'000'000);
    iota(v.begin(), v.end(), 0);
    run(v, "already sorted ");
    std::shuffle(v.begin(), v.end(), std::mt19937{ std::random_device{}() });
    run(v, "shuffled ");
}
```

At least the phenomenon is real with this compiler, standard library, and optimizer settings. Different implementations can and do give different answers. In fact, someone did do a more systematic study (a quick web search will find it) and most implementations show that effect.

One reason is branch prediction: the key operation in the sort algorithm is `"if(v[i] < pivot)] ..."` or equivalent. For a sorted sequence that test is always true whereas, for a random sequence, the branch chosen varies randomly.

Another reason is that when the vector is already sorted, we never need to move elements to their correct position. The effect of these little details is the factor of five or six that we saw.

Quicksort (and sorting in general) is a complex study that has attracted some of the greatest minds of computer science. A good sort function is a result of both choosing a good algorithm and paying attention to hardware performance in its implementation.

If you want to write efficient code, you need to know a bit about machine architecture.

Share Improve this answer

edited Jan 10, 2021 at 16:02

community wiki

Follow

2 revs, 2 users 75%
Selcuk

-
- 2 This seems to be missing the point of the question, and is answering whether sorting itself is faster with already-sorted arrays. This is less surprising because as this answer points out, there's less work to be done (with most sort algorithms other than merge-sort), on top of the



This question is rooted in *branch prediction models* on CPUs. I'd recommend reading this paper:

107



[Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache](#) (But real CPUs these days still don't make multiple taken branch-predictions per clock cycle, except for Haswell and later [effectively unrolling tiny loops in its loop buffer](#). Modern CPUs can predict multiple branches not-taken to make use of their fetches in large contiguous blocks.)

When you have sorted elements, branch prediction easily predicts correctly except right at the boundary, letting instructions flow through the CPU pipeline efficiently, without having to rewind and take the correct path on mispredictions.

Share Improve this answer

edited Nov 14 at 15:48

answered Oct 23, 2019 at 21:35

Follow




[Peter Cordes](#)

361k ● 49 ● 699 ● 958



[hatirlatici](#)

1,685 ● 2 ● 13 ● 26

- 5 The instructions stay hot in the CPU's L1 instruction cache regardless of mispredicts. The problem is fetching them into the *pipeline* in the right order, before the immediately-previous instructions have decoded and finished executing. – [Peter Cordes](#) Dec 13, 2019 at 14:29
- 3 Also, in a simple CPU that has an "instruction register", it definitely always needs to read each instruction into the IR as part of executing it. The last paragraph of this answer is very distorted from how CPUs really work. Some CPUs with a loop buffer may be able to lock a sequence of instructions down into a loop to avoid even re-fetching from L1i cache, as long as they continue to execute the same way, but that's usually minor (e.g. in Intel Skylake the microcode update disabling the LSD didn't hurt much), just getting a little bit more value out of correct branch prediction. – [Peter Cordes](#) Mar 18, 2022 at 14:38
- 2 The paper gives a general idea of how it handles fetching coordinated data as instruction from the $O(n)$ perspective, also it was written in the early 90s so none of the cutting-edge memory/register designs did not exist at that time. The modern CPU cache designs and algorithms can be found in multiple papers for a benchmark, one of them might be ieeexplore.ieee.org/document/1027060?arnumber=1027060 – [hatirlatici](#) Mar 21, 2022 at 22:04 
- 2 I'm not talking about what the linked paper says, I'm talking about the sentences in your actual answer, specifically the final paragraph. (The paper in your answer was published in 1993, and mentions superscalar CPUs and future directions of cpu architecture, so out-of-order exec was on the horizon, and it's definitely assuming parallel fetch and decode of multiple instructions. In fact that's the whole point of their proposal; seeing through multiple branches per clock cycle in a wider design, fetching them from L1i cache into the pipeline. Current CPUs still don't do that.) – [Peter Cordes](#) Mar 22, 2022 at 0:24

An answer for quick and simple understanding (read the others for more details)



This concept is called **branch prediction**

34



Branch prediction is an optimization technique that predicts the path the code will take before it is known with certainty. This is important because during the code execution, the machine prefetches several code statements and stores them in the pipeline.



The problem arises in conditional branching, where there are two possible paths or parts of the code that can be executed.



When the prediction was true, the optimization technique worked out.

When the prediction was false, to explain it in a simple way, the code statement stored in the pipeline gets proved wrong and the actual code has to be completely reloaded, which takes up a lot of time.

As common sense suggests, predictions of something sorted are way more accurate than predictions of something unsorted.

branch prediction visualisation:

sorted



unsorted



Share Improve this answer

edited Mar 18, 2022 at 5:43

answered Nov 5, 2021 at 10:05

Follow



Günter Zöchbauer

656k ● 230 ● 2.1k ● 1.6k



Geek26

499 ● 6 ● 11

- 6 The should be a change near the middle of the sorted train-track / path of execution, as the branch inside the loop is taken for the first ~half, not-taken for the last ~half of the elements. (Or vice versa.) Also, what do the 5 different levels in the unsorted case mean? It's a 2-way branch.
– Peter Cordes Nov 5, 2021 at 10:50



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.