How branches influence the performance of your code and what can you do about it?

July 5, 2020Low Level Performance, Performance7 Replies

*We at **Johnny's Software Lab LLC** are experts in performance. If performance is in any way concern in your software project, feel free to contact us.*

This is the third article in the series related to low-level optimizations, the first two being:
1) Make your programs run faster by better using the data cache
2) Make your programs run faster: avoid function calls

OK people, so we've covered the first two topics related to data cache and function call optimizations, next come branches. So, what is so special about branches?

Branches (or jumps)[1] are one of the most common instruction types. Statistically, every fifth instruction is a branch. Branches change the execution flow of the program either conditionally or unconditionally. For the CPU, an effective branch implementation is crucial for good performance.

Before moving on to explain how branches influence CPU performance, a few words need to be said on internal CPU organization.

Table Of Contents

Inside the CPU

Many modern processors of today (but not all of them, especially some processors used in embedded systems) have some or all of the following features:

- *Pipeline*: pipeline allows CPU to execute more than one instruction simultaneously. This happens because the CPU splits the execution of each instruction into several stages, and each instruction is in a different stage of execution. The same principle is employed by a car factory: at any given time there are fifty cars being produced by the factory simultaneously, e.g one car is being painted, an engine is being mounted on another car, lights are being mounted on a third car, etc. Pipelines can be short with a few stages (e.g. three stages) or long with many stages (e.g. twenty stages)
- *Out of order execution*: from the programmer's point of view, the program runs one instruction after another. In the CPU itself things look completely different: CPU doesn't need to execute the instructions in order they appear in memory. During execution, some instructions will be blocked in the CPU waiting for data from memory or waiting for data from other instructions. The CPU can look ahead and execute instructions that come later but are not blocked. When the data for the blocked instructions becomes available, the instructions that were previously not blocked have already finished. This saves CPU cycles.
- *Speculative execution*: the CPU can start executing instruction even though it is not 100% certain they even need to be executed. For example, it will guess the destination of a conditional branch instruction and then start to execute the instructions at the branch destination before it's 100% sure that the branch will be taken. If later on the CPU figures out that the guess (speculation) was wrong, it will cancel the results of speculatively executed instructions and everything will appear as no speculation has been done.
- *Branch prediction*: modern CPUs have special circuitry that for each branch instruction remembers its previous result(s): branch taken or branch not taken. When the same branch instruction is executed next time, the CPU will use this information to guess the destination of the branch and then start speculatively executing instructions at the branch destination. In case the branch predictor was right, this will lead to performance speed up.

All modern processors have pipelines in order to better exploit the CPU's resources. And most of them have branch prediction and speculative execution. As far as out of order execution is concerned, most low-end low-power processor don't have this feature since it consumes a lot of power and increases in speed are not huge. But don't take this too literally, because this information might be outdated in a few years.

You can read more about modern processors in a great article by Jason Robert Carey Patterson: *Modern Microprocessors – a 90 minutes guide*.

Now let's talk about how these CPU features influence branching.

How CPU influences branching?

From the viewpoint of a pipelined CPU branches are expensive operations.

When a branch instruction enters the processor pipeline, the branch destination is not known before it's decoded and its destination calculated. The instruction following a branch instruction can either be 1) an instruction directly following the branch or 2) an instruction at the branch destination.

For processors with pipeline this is a problem. In order to keep the pipeline full and avoid slowdowns, the processor needs to know the branch destination even before the processor has decoded the branch instruction. Depending on how the processor is designed, it can:

1. Pause the pipeline (technical name is *stall the pipeline*) and stop decoding instructions until the branch instruction is decoded and the branch destination is known. Then it can resume loading the pipeline with the correct instructions.
2. Load instructions that immediately follow the branch. In case it turns out later that this was the wrong choice, the processor will need to flush the pipeline and start loading correct instructions from the branch destination.
3. Ask the branch predictor if it should load the instructions that are immediately after the branch or instructions at the destination of the branch. The branch predictor will need to tell the pipeline where is the branch destination as well (otherwise the loading of the new instructions into the pipeline will need to wait until the pipeline resolves the branch destination).

I think processors with 1) are rare nowadays, with exception of some very low-end embedded processors. Just letting the processor do nothing is a waste of its resources so most processors will do 2) instead of 1). Processors with 2) are common in low-end embedded systems and low consumption processors. Processors with 3) are common desktop and laptop CPUs as well as high-performance CPUs.

Branches on CPUs without branch predictor

On CPUs without the branch predictor, the CPU starts loading instructions that come immediately after the branch. In case the branch is taken, the CPU will flush the wrongly loaded instructions. So the branching will be cheaper in case the branch is not taken. Bear in mind that the penalty of making a wrong choice is not normally high since these processors often have a simple design and short pipelines.

Branches on CPUs with branch predictor

If the processor has a branch predictor and speculative execution, the branch will be cheap if the branch predictor is right. In case it wasn't, branching becomes more expensive. This is especially true for CPUs with long pipelines, in that case the CPU will need to flush many instructions in case of misprediction. The exact price of misprediction can very much, but the general rule is: the more expensive the CPU, the higher the price of a branch misprediction.

There are some branches that are simple to predict, and others that are difficult to predict. To illustrate this, imagine an algorithm that loops through an array and finds the maximum element. The condition `if (a[i] < max) max = a[i]` will be false most of the time for an array with random elements. Now imagine a second algorithm that counts number of elements smaller than the array's mean value. The condition `if (a[i] < mean) cnt++` will be very difficult for branch predictor to predict in a random array.

A short note about speculative execution. Speculative execution is a broader term, but in the context of branching it means that speculation (guessing) is done on the condition of the branch. Now it is often that case that the branch condition cannot be evaluated because the CPU is waiting for data or it is waiting for some other instruction to complete. Speculative execution will allow the CPU to execute at least a few instructions that are inside the body of the branch. When the branch condition is eventually evaluated, this work might turn to be useful and the CPU has saved some cycles or useless and the CPU will throw it away.

A view of assembly

Branches in C and C++ consist of a condition that needs to be evaluated and a series of commands that needs to be executed if the condition is fulfilled. On the assembly level, the condition evaluation and branching are normally two instructions. Have a look at the following small example in C:

```
if (p != nullptr) {
    transform(p);
} else {
    invalid++;
}
```

Assembler only has two types of instructions: compare instruction and jump instruction that uses the result of the compare. So the above C++ example roughly corresponds to following pseudoassembler:

```
    p_not_null = (p != nullptr)
    if_not (p_not_null) goto ELSE;
    transform(p);
    goto ENDIF;
ELSE:
    invalid++;
ENDIF:
```

Original C condition `(p != nullptr)` is evaluated and in case it is false the branch to the instructions corresponding to the else branch is performed. Otherwise, we fall through and perform the instructions corresponding to the body of the if branch.

The same behavior could have been achieved slightly differently. We could have fallen through to the instructions corresponding to the else block and jumped to instructions corresponding to the if block. Like this:

```
    p_not_null = (p != nullptr)
    if (p_not_null) goto IF:
    invalid++;
    goto ENDIF;
IF:
    transform(p);
```

```
ENDIF:
```

Most of the time the compiler will generate the first assembly for the original C++ code, but developers can influence this using GCC builtins. We will talk later about how to tell the compiler what kind of code to generate.

Maybe you are asking yourself why did we mention assembly? Well, on some processors falling through can be cheaper than jumping. In that case, telling the compiler how to structure the code can bring better performance.

Branches and Vectorization

Branches influence the performance of your code in more ways than you can think. Let’s talk about vectorization first- (you can find more information about vectorization and branching here). Most modern CPUs have special vector instructions that can process more than one data of the same type. For example, there is an instruction that can load 4 integers from memory, another instruction that can do 4 additions and another one that can store 4 results back to the memory.

Vectorized code can be several times faster than its scalar counterpart. The compilers know this and can often automatically generate vector instruction in a process called *autovectorization*. But there is a limit to automatic vectorization, and this limit is set by branches. Consider the following code:

```
for (int i = 0; i < n; i++) {
    if (a[i] > 0) {
        a[i]++;
    } else {
        a[i]--;
    }
}
```

This loop is difficult for the compiler to vectorize because the type of processing depends on the data: if the value `a[i]` is positive, we do addition; otherwise, we do subtraction. There is no instruction that does addition on positive data and subtraction on negative data. The type of processing differs depending on the data value, and this code is difficult to vectorize.

Bottom line: branches inside hot loops make it difficult or completely prevent compiler autovectorization. Efforts to get rid of the branches inside the hot loop can bring large speed improvements because the compiler if the compiler manages to vectorize the loop because.

Techniques for making your program run faster

Before talking about techniques, let’s define two things. When we say *condition probability*, what we actually mean is what are the chances that the condition is true. There are conditions that are mostly true and there are conditions that are mostly false. There are also conditions that have equal chances of being true or false.

CPUs with branch prediction are quick to figure out which conditions are mostly true or mostly false and you shouldn’t expect any performance regressions there. However, when it comes to conditions that are difficult to predict, branch predictors will be right 50% of the time. These are the conditions where the optimization potential is hidden.

Second thing, we will use a term *computational intensive*, *expensive* or *heavy condition*. This term can actually mean two things: 1) it takes a lot of instruction to calculate it or 2) the data needed to calculate it is not in the cache and therefore a single instruction takes a lot of time to finish. The first is visible by counting instructions, the second isn't but it is also very important. If we access the memory in a random fashion[2], the data will probably not be in the cache and this will cause pipeline stalls and lower performance.

Now moving on to programming tips. Here are several techniques to make your program run faster by rewriting the critical parts of your program. Note however that these tips can also make your programs run slower and this will depend on 1) does your CPU support branch prediction and 2) does your CPU have to wait for data from the memory. Therefore, measure!

Joined conditions – cheap and expensive conditions

Joined conditions are conditions of the type `(cond1 && cond2)` or `(cond1 || cond2)`. According to C and C++ standards, in case of `(cond1 && cond2)`, if `cond1` is false `cond2` will not get evaluated. Similarly, in case of `(cond1 || cond2)`, if `cond1` is true, `cond2` will not get evaluated.

So, in case you have two conditions out of which one is cheap to evaluate and the other is expensive to evaluate, put the cheap one first and expensive one second. That will make sure that the expensive condition doesn't get evaluated needlessly.

Optimize chains of if/else commands

If you have a chain of if/else commands in a critical part of your code, you will need to look at the condition probability and condition computational intensity in order to optimize the chain. For example:

```
if (a > 0) {
    do_something();
} else if (a == 0) {
   do_something_else();
} else {
    do_something_yet_else();
}
```

Now imagine that the probability of `(a < 0 )` is 70%, `(a > 0)` 20% and `(a == 0)` is 10%. In that case it would be most logical to rearrange the above code like this:

```
if (a < 0) {
    do_something_yet_else();
} else if (a > 0) {
   do_something();
} else {
    do_something_else();
}
```

Use lookup tables instead of switch

Lookup tables (LUT) are occasionally handy when it comes to removing branches. Unfortunately, in a switch statement, branches are easy to predict most of the time, so this optimization might turn out not to have any effect. Nevertheless, here it is:

```
switch(day) {
    case MONDAY: return "Monday";
    case TUESDAY: return "Tuesday";
   ...
    case SUNDAY: return "Sunday";
    default: return "";
};
```

The above statement can be implemented using a LUT:

```
if (day < MONDAY || day > SUNDAY) return "";
char* days_to_string = { "Monday", "Tuesday", ... , "Sunday" };
return days_to_string[day - MONDAY];
```

Often the compilers can do this work for you, by replacing a switch with a lookup table. However, there is no guarantee this happens and you would need to take a look at the compiler vectorization report.

There is a GNU language extension called *computed labels* that allows you to implement look-up tables using labels stored in an array. It is very useful to implement parsers. Here is how it looks for our example:

```
    static const void* days[] = { &&monday, &&tuesday, ..., &&sunday };
    goto days[day];
monday:
    return "Monday";
tuesday:
    return "Tuesday";
...
sunday:
    return "Sunday";
```

Move the most common case out of switch

In case you are using switch command and one case seems to be most common, you can move it out of the switch and give it a special treatment. Continuing on the example from the previous section:

```
day get_first_workday() {
     std::chrono::weekday first_workday = read_first_workday();
    if (first_workday == Monday) { return day::Monday; }
    switch(first_workday) {
        case Tuesday: return day::Tueasday;
        ....
    };
}
```

Rewrite joined conditions

As already mentioned, in case of joined conditions, if the first condition has a specific value, the second condition doesn't need to get evaluated at all. How does the compiler do that? Take the following function as an example:

```
if (a[i] > x && a[i] < y) {
    do_something();
}
```

Now assume that `a[i] > x` and `a[i] < y` are cheap to evaluate (all the data is in registers or cache) but difficult to predict. This sequence would translate to following pseudoassembler:

```
    if_not (a[i] > x) goto ENDIF;
    if_not (a[i] < y) goto ENDIF;
    do_something;
    ENDIF
```

What you have here are two difficult to predict branches. If we join two conditions with `&` instead of `&&`, we will:

1. Force the evaluation of both conditions at once: & operator is arithmetic AND operation and it must evaluate both sides.
2. Make the condition easier to predict and thus decrease branch misprediction rate: two completely independent conditions with a probability of 50% will yield one joint condition with a probability of being true 25%.
3. Get rid of one branch: instead of originally two branches we will have one that is easier to predict.

Operator `&` evaluates both conditions and in the generated assembly there will be only one branch instead of two. The same story applies for operator `||` and its twin operator `|`.

Please note: according to C++ standard `bool` type has value 0 for false and any other value for true. The C++ standard guarantees that the result of a logical operation and arithmetic comparison will always be zero or one, but there is no guarantee that all bools will have only these two values. You can normalize bool variable by applying `!!` operator on it.

Suggest to the compiler which branch has a higher probability

GCC and CLANG offer keywords that the programmer can use to tell them which branches have a higher probability. E.g:

```
#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)
if (likely(ptr)) {
    ptr->do_something();
}
```

Normally we use `__builtin_expect` through the macros `likely` and `unlikely` since their syntax is cumbersome to be used everywhere.

When annotated like this, the compiler will rearrange the instructions in if and else branches in order to most optimally use the underlying hardware. Please make sure that the condition probabilities are correct, otherwise you can expect performance degradation.

Use branchless algorithms

Some algorithms which are naturally expressed with branches can be converted to branchless algorithms. For example, a function `abs` bellow uses a trick to calculate the absolute value of a number. Can you guess what trick is?

```
int abs(int a) {
  int const mask =
        a >> sizeof(int) * CHAR_BIT - 1;
    return  = (a + mask) ^ mask;
}
```

There is a whole bunch of branchless algorithms and the list is carefully maintained on site Bit Twiddling Hacks. God bless them!

Use conditional loads instead of branches

Many CPUs have support for conditional move instructions that can be used to remove branches. Here is an example:

```
if (x > y) {
    x++;
}
```

Can be rewritten as:

```
int new_x = x + 1;
x = (x > y) ? new_x : x; // the compiler should recognize this and emit a conditional branch
```

The compiler should recognize that the command on line 2 can be written as a conditional load to the variable `x` and emit conditional move instruction. Unfortunately, the compilers have their own internal logic on when to emit conditional branches which is not always as the developer expects. However, you can use inline assembly to force the conditional load (more on this later).

Please note that the branchless version does more work. The variable `x` is increased regardless if the branch is taken or not. Addition is a cheap operation, but for other expensive operations (like division) this kind of optimizations might be bad for performance.

Go branchless with arithmetic

There is a way to go branchless by cleverly using arithmetic operations. Example of conditional increment:

```
// With branch
if (a > b) {
    x += y;
}
// Branchless
x += -(a > b) & y;
```

In the above example, the expression `-(a > b)` will create a mask that is zero when the condition is false and all 1s when the condition is true.

An example of conditional assignment:

```
// With branch
x = (a > b) ? val_a : val_b;
// Branchless
x = val_a;
x += -(a > b) & (val_b - val_a);
```

All of the above examples use arithmetic to avoid branches. Depending on your CPU's branch misprediction penalty and data cache hit rates this might or might not bring performance increase.

An example of moving index in a circular buffer:

```
// With branch
int get_next_element(int current, int buffer_len) {
    int next = current + 1;
    if (next == buffer_len) {
        return 0;
    }
    return next;
}
// Branchless
int get_next_element_branchless(int current, int buffer_len) {
    int next = current + 1;
    return (next < buffer_len) * next;
}
```

Reorganize your code in order to avoid branching

In case you are writing software that needs to be high-performance, you should definitely have a look at data oriented design principles. Here is one of the recommendations that applies to branches.

Say you have a class called `animation` which can be visible or hidden. Processing a visible animation is quite different from processing a hidden one. There is a list containing animations called `animation_list` and your processing looks something like this:

```
for (const animation& a: animation_list) {
   a.step_a();
   if (a.is_visible()) {
      a.step_av();
   }
   a.step_b();
   if (a.is_visible) {
       a.step_bv();
}
```

The branch predictor can really have a hard time processing the above code unless the animations are sorted according to visibility. There are two approaches to solve this. One is to sort the animations in `animation_list` according to `is_visible()`. The second is to create two lists, `animation_list_visible` and `animation_list_hidden`, and rewrite the code like this:

```
for (const animation& a: animation_list_visible) {
   a.step_a();
   a.step_av();
   a.step_b();
   a.step_bv();
}
for (const animation& a: animation_list_hidden) {
   a.step_a();
   a.step_b();
}
```

All the conditions have disappeared and there are no branch misspredictions.

Remove branches with templates

If a boolean is passed to the function and it is used inside the function as a parameter, you can remove it by passing it as a template parameter. For example:

```
int average(int* array, int len, bool include_negatives) {
    int average = 0;
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (include_negatives) {
            average += array[i];
        } else {
            if (array[i] > 0) {
                average += array[i];
                count++;
            }
        }
    }
    if (include_negatives) {
         return average / len;
    } else {
        return average / count;
    }
}
```

In this function, the condition with `include_negatives` can be evaluated many times. To remove the evaluation, pass the parameter as a template parameter instead of a function parameter.

```
template <bool include_negatives>
int average(int* array, int len) {
    int average = 0;
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (include_negatives) {
            average += array[i];
        } else {
            if (array[i] > 0) {
                average += array[i];
                count++;
            }
        }
    }
    if (include_negatives) {
         return average / len;
    } else {
        return average / count;
    }
}
```

With this implementation the compiler will generate two versions of the function, one with `include_negatives`, one without it (in case there is a call to functions with a different value for this parameter). The branches have completely disappeared, and the code in the unused branches is gone as well.

But you need to call your functions a bit differently now. So you will call it like this:

```
int avg;
bool should_include_negatives = get_should_include_negatives();
if (should_include_negatives) {
    avg = average<true>(array, len);
} else {
    avg = average<false>(array, len);
}
```

This is in fact a compiler optimization called *branch optimization*. If the value of the `include_negatives` is known at compile-time and the compiler decides to inline function average, it will remove the branches and unused code. However, the version with templates guarantees this, which is not the case with the original version.

The compilers can often do this optimization for you. If the compiler can guarantee that the value `include_negatives` doesn't change its value during the loop execution, it can create two versions of the loop: one for the case where its value is true, and another where its value is false. This optimization is called *loop invariant code motion* and you can learn more about it in our post about loop optimizations. Using templates guarantees that this optimization always happens.

A few other tricks to avoid branches

If you are checking an unchangeable condition several times in your code, you might achieve better performance by checking it once and then doing some code copying. So, in the following example, two branches can be replaced with one branch.

```
if (is_visible) {
    hide();
}
process();
if (is_active) {
    display();
}
```

Can be replaced with:

```
if (is_visible) {
    hide();
    process();
    display();
} else {
    process();
}
```

You could also introduce a two element array, one to keep the results when the condition is true, the other to keep results when the condition is false. An example:

```
int larger = 0;
for (int i = 0; i < n; i++) {
    if (a[i] > m) {
        larger++;
    }
}
return larger;
```

Can be replaced with:

```
int result[] = { 0, 0 };
for (int i = 0; i < n; i++) {
    result[a>i]++;
}
return result[1];
```

Experiments

Now let's get to the most interesting part: the experiments. We decided on two experiments, one is related to going through an array and counting elements with certain properties. This is a cache-friendly algorithm since the hardware prefetcher will most likely keep the data flowing through the CPU.

The second algorithm is a classical binary search algorithm we introduced in the article about data cache friendly programming. Due to the nature of the binary search, this algorithm is not cache friendly at all and most of the slowness comes from waiting for the data. We'll keep as a secret for the time being on how cache performance and branching are related.

For testing we are using three chips with three different architectures:

- **AMD A8-4500M quad-core x86-64** processor with 16 kB L1 data cache for each individual core and 2M L2 cache shared by a pair of cores. This is a modern pipelined processor with branch prediction, speculative execution and out-of-order execution. According to technical specifications, the misprediction penalty on this CPU is around 20 cycles.
- **Allwinner sun7i A20 dual-core ARMv7** processor with 32kB L1 data cache for each core and 256kB L2 shared cache. This is a cheap processor intended for embedded devices with branch prediction and speculative execution but no out-of-order execution.
- **Ingenic JZ4780 dual-core MIPS32r2** processor with 32 kB L1 data cache for each core and 512kB L2 shared data cache. This is a simple pipelined processor for embedded devices with a simple branch predictor. According to technical specifications, branch misprediction penalty is around 3 cycles.

Counting example

To demonstrate the impact of branches in your code, we wrote a very small algorithm that counts the number of elements in an array bigger than a given limit. The code is available in our Github repository, just type `make counting` in directory 2020-07-branches.

Here is the most important function:

```
int count_bigger_than_limit_regular(int* array, int n, int limit) {
    int limit_cnt = 0;
    for (int i = 0; i < n; i++) {
        if (array[i] > limit) {
            limit_cnt++;
        }
    }
    return limit_cnt;
}
```

If you were asked to write the algorithm, you would probably come up with something like this.

In order to enable proper testing, we compiled all the functions with optimization level -O0. In all other optimization levels, the compiler would replace the branch with arithmetic and do some heavy loop processing and obscure what we wanted to see.

The price of branch missprediction

Let's first measure how much branch misprediction costs us. The algorithm we just mentioned counts all elements of the array bigger than `limit`. So depending on the values of the array and value of `limit`, we can tune the probability of `(array[i] > limit)` being true in `if (array[i] > limit) { limit_cnt++ }`.

We generated elements of the input array to be uniformly distributed between 0 and length of the array (`arr_len`). Then to test missprediction penalty we set the value of limit to 0 (the condition will always be true), `arr_len / 2` (the condition will be true 50% of the time and difficult to predict) and `arr_len` (the condition will never be true). Here are the results of our measurements:

	Condition always true	Condition unpredictable	Condition false
Runtime (ms)	5533	14176	5478
Instructions	14G	13.5G	13G
Instructions per cycle	1.36	0.50	1.27
Branch misspredictions (%)	0%	32.96%	0%

Array length = 1M, searches 1000 on AMD A8-4500M

The version of the code with the unpredictable condition is three times slower on x86-64. This happens because the pipeline has to be flushed every time the branch is mispredicted.

Here are the runtimes for ARM and MIPS chips:

	Condition always true	Condition unpredictable	Condition always false
ARM	30.59s	32.23s	25.89s
MIPS	37.35s	35.59s	31.55s

Runtimes on MIPS and ARM chips, array length 1M, searches 1000

MIPS chip doesn't have a misprediction penalty according to our measurement (not according to the spec). There was a small penalty on ARM chip, but certainly not as drastic as in case of x86-64 chip.

Can we fix this? Read forward.

Going branchless

Now let's rewrite the condition according to the advice we gave you earlier. Here are three implementations with a rewritten condition:

```
int count_bigger_than_limit_branchless(int* array, int n, int limit) {
    int limit_cnt[] = { 0, 0 };
    for (int i = 0; i < n; i++) {
        limit_cnt[array[i] > limit]++;
    }
    return limit_cnt[1];
}
int count_bigger_than_limit_arithmetic(int* array, int n, int limit) {
    int limit_cnt = 0;
    for (int i = 0; i < n; i++) {
        limit_cnt += (array[i] > limit);
    }
    return limit_cnt;
}
int count_bigger_than_limit_cmove(int* array, int n, int limit) {
    int limit_cnt = 0;
    int new_limit_cnt;
    for (int i = 0; i < n; i++) {
        new_limit_cnt = limit_cnt + 1;
        // The following line is pseudo C++, originally it is written in inline assembly
        limit_cnt = conditional_load_if(array[i] > limit, new_limit_cnt);
    }
    return limit_cnt;
}
```

There are three versions of our code:

- `count_bigger_than_limit_branchless` (later in text *branchless)* internally uses a small two-element array to count both when the element of the array is larger and smaller than the limit.
- count_bigger_than_limit_arithmetic (later in text arithmetic) uses the fact that expression (array[i] > limit) can have only values 0 or 1 and increases the counter by the value of the expression.
- count_bigger_than_limit_cmove (later in text conditional move) calculates the new value and then uses a conditional move to load it if the condition is true. We use inline assembly to make sure the

compiler will emit cmov instructions.

Please note a common thing for all the versions. Inside the branch there is a job that we must do. When we remove the branch, we are still doing the job, but this time we are doing the job even in case the job is not needed. This makes our CPU execute more instructions, but we expect this to be paid off by fewer branch mispredictions and better instructions per cycle ratio.

Going branchless on x86-64 architecture

How our three different strategies to avoid branches show in numbers? Here are the numbers for predictable conditions:

	Regular	Branchless	Arithmetic	Conditional Move
Runtime (ms)	5502	7492	6100	9845
Instructions executed	14G	19G	15G	19G
Instructions per cycle	1.37	1.37	1.33	1.04

Array length = 1M, searches 1000 on AMD A8-4500M for predictable branches

As you can see above, when the branch is predictable the regular implementation is the best. This implementation also has the smallest number of executed instructions and best instructions per cycle ratio[3].

Runtimes for the always false conditions differ little from the runtimes for the always true conditions and this applies to all four implementations. All other numbers are same for all implementations except for regular implementations. In the regular implementation, the instruction per cycle number is lower but so is the number of executed instructions and no speed difference is observed.

What happens when the branch is not predictable? The numbers look different.

	Regular	Branchless	Arithmetic	Conditional Move
Runtime (ms)	14225	7427	6084	9836
Instructions executed	13.5G	19G	15G	19G
Instructions per cycle	0.5	1.38	1.32	1.04

Array length = 1M, searches 1000 on AMD A8-4500M for unpredictable branches

The regular implementation fares much worse. Now it is the slowest implementation. The instructions per cycle number is much worse because the pipeline has to be flushed due to branch mispredictions. For other implementation, the numbers haven't changed almost at all.

One notable thing. If we are compiling this program with `-O3` compilation option, the compiler doesn't emit the branch for the regular implementation. We could see that because the branch misprediction rate was low and the runtime number was very comparable to the number for arithmetic implementation.

Going branchless on ARMv7

In case of ARM chip, the numbers look again different. We don't show the results for conditional move implementation since the author is not familiar with ARM assembler. Here are the numbers:

Condition predictability	Regular	Arithmetic	Branchless
Always true	3.059s	3.385s	4.359s
Unpredictable	3.223s	3.371s	4.360s
Always false	2.589s	3.370s	4.360s

Runtimes on Allwinner sun71 A20 (ARMv7), array length 1M, 100 searches

Here the regular version is the fastest. Arithmetic and branchless versions don't bring any speed improvements, they are actually slower.

Note that the version with the unpredictable condition is the slowest. This demonstrates that this chip has some kind of branch prediction. However, the price of misprediction is low otherwise we would see other implementation to be faster in that case.

Going branchless on MIPS32r2

Here are the same results for MIPS:

Condition predictability	Regular	Arithmetic	Branchless	Cmov
Always true	37.352s	37.333s	41.987s	39.686s
Unpredictable	35.590s	37.353s	42.033s	39.731s
Always false	31.551s	37.396s	42.055s	39.763s

Runtimes on Ingenic JZ4780 (MIPS32r2), array length 1M, 1000 searches

From these numbers, it seems that the MIPS chip doesn't have any branch misprediction since the running times solely depend on the number of executed instructions for regular implementation (contrary to the technical specification). For regular implementation, the less often the condition is true, the faster the program.

Also, branches seem to be relatively cheap since arithmetic implementation and regular implementation have identical performance in case the condition is always true. Other implementations are slower, but not much.

Annotating branches with likely and unlikely

The next thing we wanted to test is does annotating branches with likely and unlikely have any impact on branch performance. We used the same function as previously, but we annotated the critical condition like this if (likely(a[i] > limit) limit_cnt++. We compiled the functions using optimization level 3 since there is no point in testing the behavior of the annotations on non-production optimization levels.

AMD A8-4500M with GCC 7.5 gave a little unexpected results. Here they are:

Condition predictability	Likely	Unlikely	Unannotated
Always true	904ms	1045ms	902ms
Always false	906ms	1050ms	903ms

Runtimes for different condition probabilities and branch annotations, 1M elements in the array, 1000 searches

In the case the condition is marked with `likely` is always faster than the case where the condition is marked with `unlikely`. This is not completely unexpected when you think about it since this CPU has a good branch predictor. The `unlikely` version only introduces additional instructions without need.

On our ARMv7 chip with GCC 6.3 there was absolutely no performance difference if we were using `likely` or unlikely for branch annotation. Compiler did generate different code for both implementations, but the number of cycles and number of instructions for both flavors were roughly the same. Our guess is that this CPU doesn't make branching cheaper if the branch is not taken, that is the reason why we see neither performance increase nor decrease.

There was also no performance difference on our MIPS chip and GCC 4.9. GCC generated identical assembly for both likely and unlikely versions of the function.

Conclusion: As far as likely and unlikely macros are concerned, our investigation shows that they don't help at all on processors with branch predictors. Unfortunately, we didn't have a processor without a branch predictor to test the behavior there as well.

Joint conditions

To test joint condition in the if clause, we modified our code like this:

```
int count_bigger_than_limit_joint_simple(int* array, int n, int limit) {
    int limit_cnt = 0;
    for (int i = 0; i < n/2; i+=2) {
        // The two conditions in this if can be joined with & or &&
        if (array[i] > limit && array[i + 1] > limit) {
            limit_cnt++;
        }
    }
    return limit_cnt;
}
```

Basically it's a very simple modification where both conditions are difficult to predict. The only difference is in line 4: `if (array[i] > limit && array[i + 1] > limit)`. We wanted to test if there is a difference between using the operator `&&` and operator `&` for joining condition. We call the first version *simple* and the second version *arithmetic.*

We compiled the above functions with `-O0` because when we compiled them with `-O3` the arithmetic version was very fast on x86-64 and there were no branch mispredictions. This suggests that the compiler has completely optimized away the branch.

Here are the results for all three architectures in case both conditions are difficult to predict with optimization:

	Joint simple	Joint arithmetic
x86-64	5.18s	3.37s
ARM	12.756s	15.317s
MIPS	13.221s	15.337s

Runtimes for two kinds of condition joining for three different architectures, 1M elements in the array, 1000 searches

The above results show that for the CPUs with branch predictor and high misprediction penalty joint-arithmetic flavor is much faster. But for CPUs with low misprediction penalty the joint-simple flavor is faster simply because it executes fewer instructions.

Binary Search

In order to further test the behavior of branches, we took the binary search algorithm we used to test cache prefetching in the article about data cache friendly programming. The source code is available in our github repository, just type `make binary_search` in directory 2020-07-branches.

Here is the core code that implements binary search:

```
int binary_search(int* array, int number_of_elements, int key) {
    int low = 0, high = number_of_elements-1, mid;
    while(low <= high) {
        mid = (low + high)/2;
        if (array[mid] == key) {
            return mid;
        }
        if(array[mid] < key) {
            low = mid + 1;
        } else {
            high = mid-1;
        }
    }
    return -1;
}
```

The above algorithm is a classical binary search algorithm. We call it further in text *regular* implementation. Note that there is an essential if/else condition on lines 8-12 that determines the flow of the search. The condition `array[mid] < key` is difficult to predict due to the nature of the binary search algorithm. Also, the access to array[mid] is expensive since this data is typically not in the data cache.

We eliminated the branch in two ways, using conditional move and using arithmetic. Here are the two versions:

```
// Conditional move implementation
int new_low = low + 1;
int new_high = high - 1;
bool condition = array[mid] > key;
// The bellow two lines are pseudo C++, the actual code is written in assembler
low = conditional_move_if(new_low, condition);
high = conditional_move_if_not(new_high, condition);
// Arithmetic implementation
int new_low = mid + 1;
int new_high = mid - 1;
int condition = array[mid] < key;
int condition_true_mask = -condition;
int condition_false_mask = -(1 - condition);
low += condition_true_mask & (new_low - low);
high += condition_false_mask & (new_high - high);
```

The conditional move implementation uses the instructions provided by the CPU to conditionally load the prepared values.

The arithmetic implementation uses clever condition manipulation to generate `condition_true_mask` and `condition_false_mask`. Depending on the values of those masks, it will load proper values into variables `low` and `high`.

Binary search algorithm on x86-64

Here are the numbers for x86-64 CPU for the case where the working set is large and doesn’t fit the caches. We tested the version of the algorithms with and without explicit data prefetching using __builtin_prefetch.

	Regular	Arithmetic	Conditional move

No data prefetching	2.919s	3.623s	3.243s
With data prefetching	2.667s	2.748s	2.609s

Runtimes for various binary search implementations, 4M elements in array, 4M searches

The above tables shows something very interesting. The branch in our binary search cannot be predicted well, yet when there is no data prefetching our regular algorithm performs the best. Why? Because branch prediction, speculative execution and out of order execution give the CPU something to do while waiting for data to arrive from the memory. In order not to encumber the text here, we will talk about it a bit later.

Here are the results of the same algorithm when the working set completely fits the L1 cache:

	Regular	Arithmetic	Conditional move
No data prefetching	0.744s	0.681s	0.553s
With data prefetching	0.825s	0.704s	0.618s

Runtimes for various binary search implementations, 4K elements in array, 4M searches

The numbers are different when compared to the previous experiment. When the working set completely fits the L1 data cache, the conditional move version is the fastest by a wide margin, followed by the arithmetic version. The regular version performs badly due to many branch mispredictions.

Prefetching doesn't help in the case of a small working set: those algorithms are slower. All the data is already in the cache and prefetching instructions are just more instructions to execute without any added benefit.

Binary search algorithm on ARM and MIPS

For the ARM and MIPS chips, prefetching algorithms were slower than non-prefetching ones, so we leave those numbers out.

Here are the binary search runtimes for ARM and MIPS chips on an array with 4M elements:

	Regular	Arithmetic	Conditional Move
ARM	10.85s	11.06s	—
MIPS	11.79s	11.80s	11.87s

Runtimes for ARM and MIPS chips, 4M elements in array, 4M searches

On MIPS, all three flavors give roughly the same numbers. On ARM, the regular version is slightly faster than the arithmetic version.

Here are the runtimes for ARM and MIPS chips on array with 10k elements:

	Regular	Arithmetic	Conditional Move
ARM	1.71s	1.79s	—
MIPS	1.42s	1.48s	1.51s

Runtimes for ARM and MIPS chips, 4M elements in array, 4M searches

The size of the working set doesn't change the relative ratio of numbers. On these chips optimizations related to branching don't produce an increase in speed.

Why is the binary search with branches fastest on a large working set on x86-64?

OK, now back to the interesting question. In case x86-64 chip we saw that if the working set is large, the branching version is the fastest. In case the working set is small, the conditional move version is fastest. When we introduce an explicit software prefetching in order to increase the cache hit rate, we see the advantage of the regular version melting away. Why?

Limitations of out-of-order execution

To explain this, bear in mind that the CPU we are talking about is high-end CPU with branch prediction, speculative execution and out-of-order execution. All these things mean that the CPU can execute several instructions in parallel, but there is a limit to how many instructions it can execute at once. This limit is due to two factors:

- There is a limited number of resources in the processor. A typical high-end processor might for example process four simple arithmetic instructions simultaneously, two load instructions, two store instructions or one complex arithmetic instructions. When the instructions finish execution (technical term is instructions *retire*), the resources became available so the processor can process new instructions.
- There is a data dependency between instructions. If input arguments of the current instruction depend on the result of a previous instruction, the current instruction cannot be processed until the previous one has finished. It is stuck in the processor holding resources and preventing other instructions from coming in.

All code has data dependencies, and the code with data dependencies is not necessarily bad. But data dependencies lower the number of instructions that the processor can execute per cycle.

In case of in-order execution CPUs, the pipeline will be stalled on the current instruction if it depends on the previous instruction and the previous instruction hasn't finished. In case of out-of-order execution CPUs, the processor will try to load other instructions that come after the blocked instruction. If these instructions do not depend on the previous instructions, they can be safely executed. This is the way for CPU to utilize the idle resources.

Explaining the performance for binary search with branch

So how does this relate to the performance of our binary search? Here is an important part of our regular implementation in pseudoassembler:

```
    element = load(base_address = array, index = mid)
    if_not (element < key) goto ELSE
    low = mid + 1
    goto ENDIF
ELSE:
    high = mid - 1
ENDIF:
    // These are the instructions at the beginning of the next loop
    mid = low + high
    mid = mid / 2
```

Let's make certain assumptions: operation `element = load(base_address = array, index = mid)` takes 300 cycles to complete if `array[mid]` is not in the data cache, otherwise it takes 3 cycles. The branch condition `element < key` will be predicted correctly 50% of the time (worst case branch prediction). The price of branch misprediction is 15 cycles.

Let's analyze how our code gets executed. The processor needs to wait for 300 cycles for the load on line 1 to execute. Since it has OOE, it starts executing branch on line 2. Instruction on line 2 depends on the result of line 1 and the CPU cannot execute it. However, the CPU performs a guess and starts running instructions on lines 3, 4, 9 and 10. In case the guess is good, it takes 300 cycles to execute all. In the case the guess is bad, it has lost an additional 15 cycles for branch misprediction plus additional time it needs to execute the instructions 6, 9 and 10.

Explaining the performance for binary search with conditional move

What about conditional move implementation? Here it is, again in pseudoassembler:

```
    element = load(base_address = array, index = mid)
    load_low =  element < key
    new_low = mid + 1
    new_high = mid - 1
    low = move_if(condition = load_low, value = new_low)
    high = move_if_not(condition = load_low, value = new_high)
    // These are the instructions at the beginning of the next loop
    mid = low + high
    mid = mid / 2
```

There are no branches here and therefore no branch missprediction penalties. Let assume the same assumptions as for regular implementation (operation `element = load(base_address = array, index = mid)` takes 300 cycles to complete if `array[mid]` is not in the data cache, otherwise it takes 3 cycles).

This code executes as follows: the processor needs to wait 300 cycles for the load on line 1 to execute. Since it has OOE, it starts executing instruction on line 2, but it is blocked waiting for data from 1. It executes instructions on lines 3 and 4. It cannot execute instructions on lines 5 and 6 since they depend on instruction on line 2. Instruction on line 9 is stuck since it depends on instruction on lines 5 and 6. Instruction on line 10 depends on instruction on line 9 and it is also stuck. Since there is no speculation involved here, getting to instruction 10 will take 300 cycles plus some time to execute instructions 2, 5, 6 and 9.

Branches vs conditional move performance comparison

Let's do some simple math now. In case of the binary search with branch prediction the runtime is:

```
MISSPREDICTION_PENALTY = 15 cycles
INSTRUCTIONS_NEEDED_TO_EXECUTE_DUE_MISSPREDICTION = 50 cycles

RUNTIME = (RUNTIME_PREDICTION_CORRECT + RUNTIME_PREDICTION_NOTCORRECT) / 2
RUNTIME_PREDICTION_CORRECT = 300 cycles
RUNTIME_PREDICTION_NOTCORRECT = 300 cycles + MISSPREDICTION_PENALTY +
INSTRUCTIONS_NEEDED_TO_EXECUTE_DUE_MISSPREDICTION = 365 cycles

RUNTIME = 332.5 cycles
```

In case of version with the conditional moves, the runtime is:

```
INSTRUCTIONS_BLOCKED_WAITING_FOR_DATA = 50 cycles

RUNTIME = 300 cycles + INSTRUCTIONS_BLOCKED_WAITING_FOR_DATA = 350 cycles
```

As you can see, the branch prediction version is on average faster by 17.5 cycles in case where we need to wait for 300 cycles for data to arrive from the memory.

The bottom line

Current processors don't speculate on conditional moves, only on branches. Branch speculation allows them to mask some of the penalties incurred by slow memory access. Conditional moves (and other techniques for branch removal) remove the branch misprediction penalty but introduce data dependency penalty. The processor will be blocked more often and can speculatively execute fewer instructions. And in case of a low cache miss rate data dependency penalties can be much more expensive than branch misprediction penalties.

So the conclusion is: branch speculation breaks some of the data dependencies and effectively masks the time CPU needs to wait for data from the memory. If the guess made by the branch predictor is correct, a lot of work will already be done when the data arrives from the memory. This is not the case for code that goes branchless.

Final Word

When I first started writing this article I was expecting a simple and straight-forward article with a short conclusion. Boy was I wrong 🙄 Let's start off by giving thanks.

First bravo for the compiler makers. This experience has shown me that the compilers are masters of making branching fast. They know the timing of every instruction and they can emit the branch that will have good performance for a wide range of branch condition probabilities.

The second bravo goes to the hardware designers of modern processors. In case the branch is predicted correctly, the HW makes branches some of the cheapest instructions. Most of the time branch prediction works well and this makes our programs run smoothly. The programmers can focus on more important things.

And the third bravo goes to hardware designers of modern processors again. Why? Because of out-of-order execution (OOE). What our experiment in binary search example has shown, even when the branch misprediction rate is high, waiting for data and then executing the branch is more expensive than speculatively executing the branch and then flushing the pipeline in case of misprediction.

A general note about branch optimizations

We made a few recommendations here that are universal and that will work every time and on every hardware, such as optimize chains of if/else commands or reorganize your code in order to avoid branching. However, other techniques presented here are more limited and can be recommended only under certain conditions.

To optimize your branches, the first thing you need to understand is that the compilers are doing a good job of optimizing them. Therefore my recommendation is that most of these optimizations are not worth it most of the time. Make your code simple to understand and the compiler will do its best to generate the best possible code, now and in the future.

The second thing which is also important: before optimizing branches you need to make sure your program uses data cache optimally. In the case of many cache misses, branches are actually defenders of the CPU performance. Remove them and you will get bad results. Improve the data cache usage first, and deal with branches later.

The only place where you should focus on your branches is the case of critical code; one or two functions that will run on one particular computer. As our experiments have shown, there are places where switching from branching to branchless code brings more performance but the exact numbers depend on your CPU, data cache utilization, and possibly other factors as well. So careful measurements need to be taken.

I would also recommend that you write the critical parts of your branches using inline assembly since this will guarantee that the code you write will not be spoiled by a compiler upgrade. And of course, it is critical that you test your code for performance regression since these seem to be fragile optimizations.

The future of branching

I tested two cheap processors and both of them have branch predictors. Trying to find a processor without a branch predictor can be a challenge nowadays. In the future we should expect more complex processor designs even in the low-end CPUs. As more and more CPUs adopt out-of-order execution, the branching misprediction penalties will become higher and higher. For a performance aware developer, taking care of branches will become more and more important.

Further Read

Agner's Optimizing Software in C++: chapter 7.5 Booleans, chapter 7.12 Branches and switch statements

CPW: Avoiding Branches

Power and Performance: Software Analysis and Optimization by Jim Kukunas, Chapter 13: Branching

Featured image courtesy of: https://bh-cookbook.github.io/mips-asm/jumps-and-branches-part-6.html