

+ 22

The Hidden Performance Price of Virtual Functions

IVICA BOGOSAVLJEVIC





About me

- Ivica Bogosavljevic - application performance specialist
- Professional focus is C/C++ application performance improvement:
 - Better algorithms
 - Better exploiting the underlying hardware
 - Better usage of the standard library
 - Better usage of programming language
 - Better usage of the operating system.
- Work as a an external expert
 - If your software is slow, I can help you make it faster
- Writer for software performance blog: Johny's Software Lab - link in the footer
 - For all the people interested in software performance





Introduction

- Virtual functions in C++
 - Enable flexibility
 - The basic component of OOP
- Virtual functions are slower than regular functions
- The performance price of virtual functions depends on several factors
 - Here we explain what are those factors



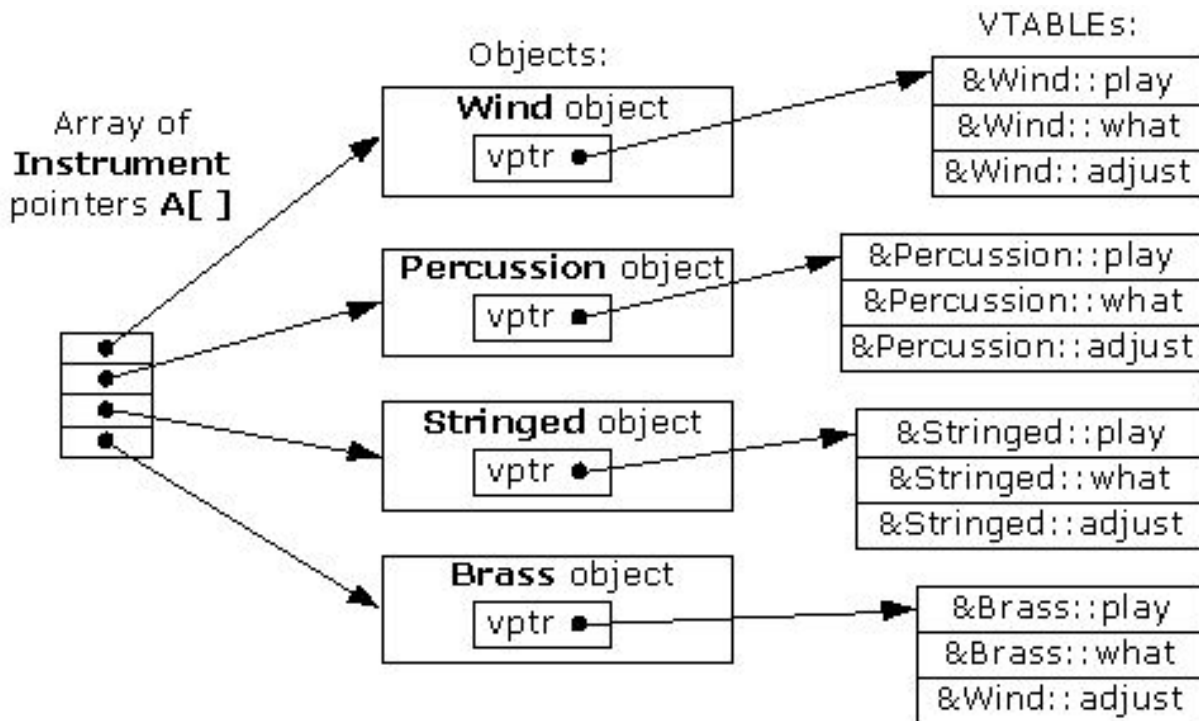


How virtual functions work?

- C++ standard doesn't mandate implementation of virtual functions
- Most compilers, however, implement virtual functions in a similar manner

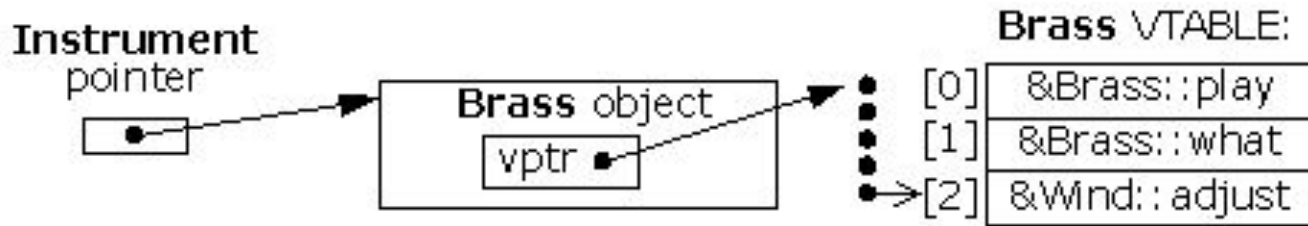


How virtual functions work - virtual tables





How virtual functions work - function calls





Initial Analysis

- Virtual functions are more expensive than non-virtual functions
 - The virtual function's address is not known at compile time
 - The program needs to look up the virtual function's address at runtime
 - Virtual function's address lookup is done through virtual table pointer



Initial Analysis - experiment

- A vector of 20 million objects of the same type
- 20 million calls to the virtual function vs 20 million calls to the non-virtual function

	Virtual function call	Non-virtual function call
Short and fast function	153 ms	126 ms
Long and slow function	32.090 ms	31.848 ms



Initial Analysis - conclusion

- The results don't look that bad
- There is a noticeable overhead for small function (18%).
- For the large function, the overhead is negligible
- But is this all there is to virtual functions?





Vector of pointers

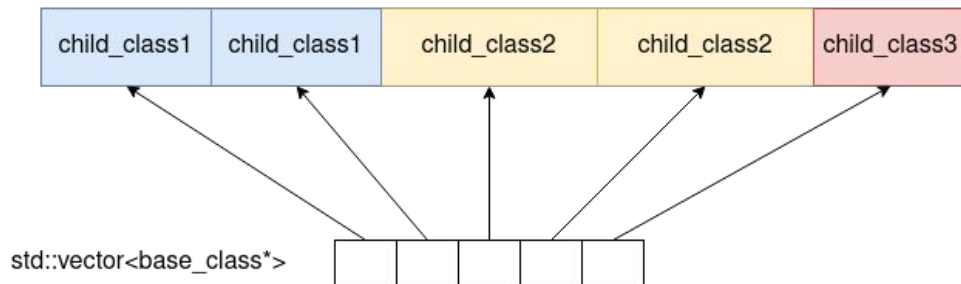
- To activate virtual function mechanism, you need to access the object through a pointer or a reference
 - Objects need to be allocated on the heap (using new, malloc or smart pointers)
- Accessing objects on the heap can be very slow



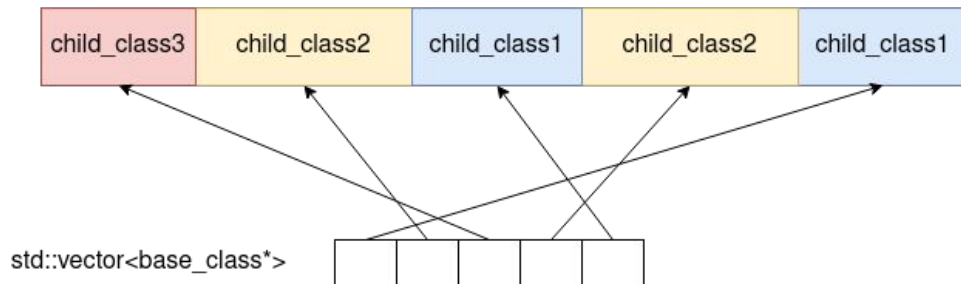


Vector of pointers

Optimal layout



Non-optimal layout





Vector of pointers

- Accessing objects on the heap can be very slow
 - The reason are data cache misses
 - If objects are neighbors in memory, we can expect performance improvements
 - If objects are not neighbors in memory, we can expect slowdowns
 - If the neighboring pointers do not point to neighboring elements on the heap, we can expect data cache misses
- There is no guarantee that the neighboring pointers will point to neighboring objects in memory
 - As the program becomes bigger and more complex there is less and less chance that this will happen
- Vector of objects is much better for the performance compared to vector of pointers
 - The vector of objects doesn't suffer from data cache misses





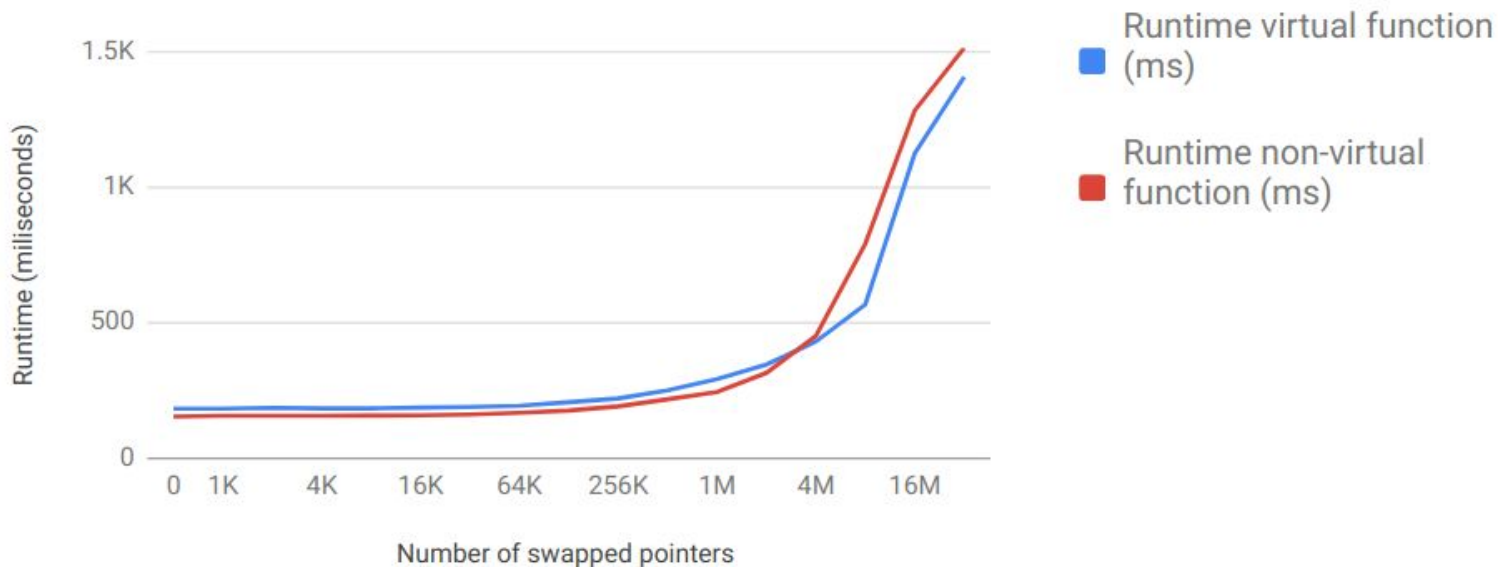
Vector of pointers - experiment

- Vector of objects containing 20 million objects
- Another vector of pointers, pointer at location i points to an object at location i
 - This is the perfect ordering: neighboring pointers point to neighboring objects
- We measure the time needed to iterate through 20 million objects by following the pointers in the vector of pointers
 - There are several iterations of the experiment
 - In each new iteration of the experiment, we shuffle the pointer vector a bit
 - Shuffling slows down the traversal a bit
 - We measure the runtime as a function of number of shuffles
 - One shuffle means swap pointer at position [0] with a pointer at position [rand(0, vector_len)]



Vector of pointers - results

How swapping of pointers in an array influences the speed of access





Vector of pointers - conclusion

- Memory layout is very important for program performance
 - Worst case is 7.5 times slower than the fastest case
- The slowdown isn't related to virtual functions per se
 - The slowdown is related to the memory layout
 - Still, the main reason you want to use the vector of pointers to achieve polymorphism
- Alternatives to vector of pointers:
 - Use `std::variant` with std::visitor``
 - Use `polymorphic_vector` - uses virtual dispatching, but doesn't use pointers. Downside is increased memory consumption → google `polymorphic_vector``
 - Use per type vector (e.g. `boost::base_collection`), a very useful if you don't need a specific ordering in the vector`





Compiler Optimizations

- Compiler knows the address of non-virtual functions at compile time.
 - This means the compiler can inline the non-virtual function and avoid the function call
- Inlining saves a few instructions on the function call, but that is not all
- After inlining, the compiler can perform many other compiler optimizations, e.g:
 - Move loop invariant code outside of the calling loop
 - Use special instructions that can process more than one data at a time in a process called vectorization <- this can increase speed from 2 to 6 times.



Compiler Optimizations - Example

```
void my_loop(int arr_len) {  
    std::vector<double> in;  
    std::vector<double> out;  
  
    ...  
    for (int i = 0; i < out.size(); i++) {  
        out[i] = my_sqrt(in[i]);  
    }  
}
```

```
double my_sqrt(double a) {  
    if (debug && a < 0) {  
        std::cerr << "Value " << a << "negative\n";  
    }  
    return std::sqrt(a);  
}
```

Loop unswitching

```
void my_loop_inlined(int arr_len) {  
    std::vector<double> in;  
    std::vector<double> out;  
  
    ...  
    if (debug) {  
        for (int i = 0; i < out.size(); i++) {  
            if (in[i] < 0) {  
                std::cerr << "Value " << a <<  
                    "negative\n";  
            }  
            out[i] = std::sqrt(in[i]);  
        }  
    } else {  
        for (int i = 0; i < out.size(); i++) {  
            out[i] = std::sqrt(in[i]);  
        }  
    }  
}
```





Compiler Optimizations - experiment

```
class object {
    protected:
        bool m_is_visible;
        unsigned int m_id;
        static unsigned int m_offset;
    public:
        ATTRNOINLINE
        bool is_visible () { return m_is_visible; }
        ATTRNOINLINE
        unsigned int get_id3 () { return m_id + m_offset; };
};
```

```
// Test loop
for (int i = 0; i < arr_len; i++) {
    object* o = pv.get(i);
    if (o->is_visible()) {
        count += o->get_id3();
    }
}
```





Compiler Optimizations - results

- Measured the performance of non-virtual function, inlined and non-inlined.

```
// Test loop
for (int i = 0; i < arr_len; i++) {
    object* o = pv.get(i);
    if (o->is_visible()) {
        count += o->get_id3();
    }
}
```

Vector 20M objects	Non-inlined	Inlined
Runtime	242 ms	136 ms



Compiler Optimizations - conclusion

- Virtual functions inhibit compiler optimizations because they are essentially not inlinable
- A solution for this is ***type based processing***
 - Don't mix the types, each type has its own container and its own loop
 - The compiler can inline small functions and perform the compiler optimizations
 - Already implemented in `boost::base_collection`
 - This approach is applicable if objects in the vector don't have to be sorted
- The benefits of compiler optimization that happen due to inlining are very case dependent
 - Some code profits a lot from compiler optimizations, other not so much
 - Smaller functions in principle benefit more





Jump Destination Guessing

- To speed up computation, modern CPUs do a lot of guessing (technical term is *speculative execution*)
- In the case of virtual function:
 - The CPU guesses which virtual function will get called
 - It starts executing the instructions belonging to the guessed virtual function
- If the guess is correct, this saves time
- If the guess is wrong, the CPU needs to cancel the effect of wrongly executed instructions and start over
 - This costs time



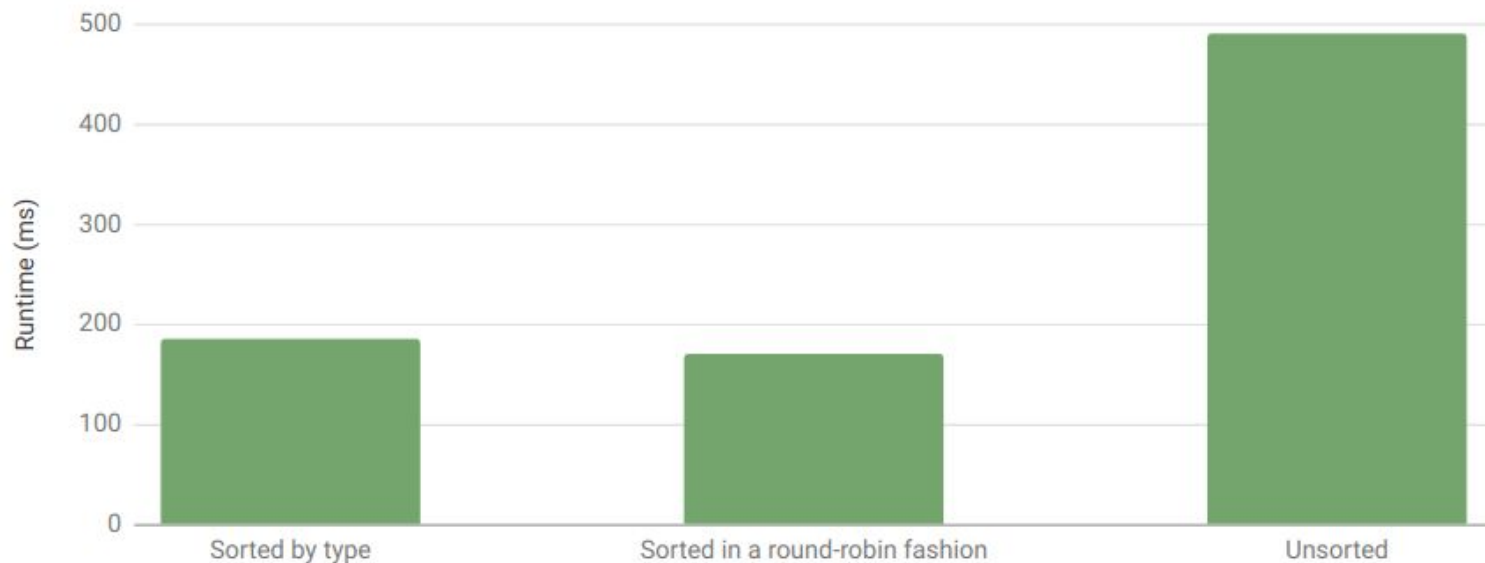
Jump Destination Guessing - experiment

- Three vector of 20 million objects
 - First vector is sorted by type: A, A, A, A, B, B, B, B, C, C, C, C, D, D, D, D
 - Types in vector in predictable fashion: A, B, C, D, A, B, C, D, A, B, C, D, A, B, C, D
 - Types in vector random: B, C, A, C, A, C, B, B, A, C, B, A.
- We measure time needed to call a small virtual function on the three types of vectors



Jump Destination Guessing - results

Performance of virtual functions depending on the sorting type





Jump Destination Guessing - conclusion

- Types sorted in predictable manner -> the CPU can successfully predict the address of the virtual function and this speeds up the computation
- If types are appear randomly , the CPU cannot guess successfully and precious cycles are lost
 - A solution to this is again, type based processing
 - However, type based processing is not always usable
- The effect is mostly pronounced with short virtual functions



Instruction Cache Evictions

- Modern CPUs rely on “getting to know” the instructions they are executing
- The code that has already been executed is *hot* code
 - Its instructions are in the instruction cache
 - Its branch predictors know the outcome of the branch (true/false)
 - Its jump predictors know the target of the jump
- The CPU is faster when executing hot code compared to executing cold code
- The CPU’s memory is limited
 - The code that is currently hot will eventually become cold unless executed frequently
- Virtual functions, especially large virtual functions where each object has a different virtual function, mean that we are switching from one implementation to another
 - The CPU is constantly switching between different implementations and is always running cold code





Instruction Cache Evictions - experiment

- Measuring the effect of instruction cache eviction is the hardest, because it depends on many factors
 - The number of different virtual function implementations - the bigger the number, the slower the code
 - The number of executed instructions in the virtual functions - the bigger the number, the slower the code
 - The size of virtual function correlates to the number of executed instructions, but they are not the same
 - How sorted are the objects in the container (by type)
 - Best case is when they are sorted by type (AAABBBCCDDDD)
 - Worst case is when they are sorted by type in a round robin fashion (ABCDABCDABCD)



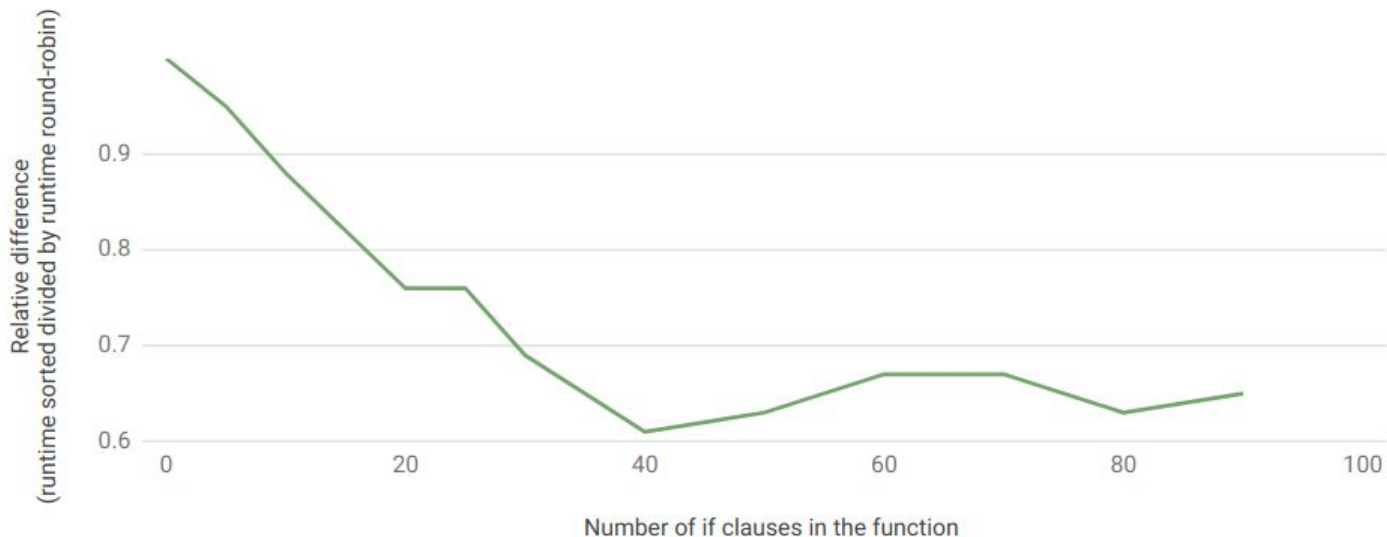
Instruction Cache Evictions - experiment

- Four classes: *rectangle*, *circle*, *line* and *monster*
- Four implementations of *long_virtual_functions*
- The *long_virtual_function* consists of a *for* loop with a large *if/elseif/.../else* inside it
- For measurements we use two vectors (20 million objects)
 - Elements of the vector sorted by type: AAABBBCCDDDD
 - Elements of the vector sorted by type in a round-robin fashion: ABCDABCDABCD
- We change the number of comparisons in a large *if/elseif/.../else* block and compare the time needed to iterate the two vectors



Instruction Cache Evictions - result

Relative performance difference between the sorted and round-robin vector



In the worst case, the same function took 7.5 seconds to execute in the sorted vector, and 12.3 seconds to execute in the round-robin vector



Instruction Cache Eviction - conclusion

- In our example, the cold code was running at the speed of 0.6 of the speed of the fast code
- The phenomenon is not related to the virtual functions themselves
 - E.g, it will happen if each instance has a pointer to a different function
- However, it is most likely to occur with large virtual functions on mixed-type unsorted vectors with many different derived types





Conclusion

- Virtual functions do not incur too much additional cost by themselves
- It is the environment where they run which determines their speed
- The hardware craves predictability: same type, same function, neighboring virtual address
 - When this is true, the hardware run at its fastest
 - It's difficult to achieve this with casual usage of virtual functions
- In game development, they use another paradigm instead of OOP called: data-oriented design
 - One of its major parts is **type based processing**: each vector holds one type only
 - This eliminates all the problems related to virtual functions
 - However, this approach is not applicable everywhere





Conclusion

- If you need to use virtual functions, bear in mind:
 - The number one factor that is responsible for bad performance are data cache misses
 - ***Avoiding vector of pointers on a hot path is a must!***
 - Other factors also play their role, but to a lesser extent
 - With careful design, you can reap most benefit of virtual functions without incurring too much additional cost
- Here are a few ideas to fix your code with virtual functions:
 - Arrangement of objects in memory is very important!
 - Try to make small functions non-virtual!
 - Most overhead of virtual functions comes from small functions, they cost more to call than to execute
 - Try to keep objects in the vector sorted by type





The End

- Questions?
- Interested in C/C++ software performance? Subscribe:
 - Twitter: @johnnysswlab
 - LinkedIn: <https://www.linkedin.com/company/johnnysswlab/>
- Your program is slow and you need help? Contact me!
 - ivica@johnnysswlab.com
 - <https://johnnysswlab.com/consulting/>